Jeffrey A. Cardille
Morgan A. Crowley
David Saah
Nicholas E. Clinton *Editors*

Cloud-Based Remote Sensing with Google Earth Engine

Fundamentals and Applications

Springer

Cloud-Based Remote Sensing with Google Earth Engine

Jeffrey A. Cardille · Morgan A. Crowley · David Saah · Nicholas E. Clinton
Editors

Cloud-Based Remote Sensing with Google Earth Engine

Fundamentals and Applications

Springer

Editors
Jeffrey A. Cardille
Department of Natural Resource Sciences
McGill University
Sainte-Anne-De-Bellevue, QC, Canada

Morgan A. Crowley
Department of Natural Resource Sciences
McGill University
Sainte-Anne-De-Bellevue, QC, Canada

David Saah
University of San Francisco
San Francisco, CA, USA

Nicholas E. Clinton
Google LLC
Mountain View, CA, USA

ISBN 978-3-031-26587-7 ISBN 978-3-031-26588-4 (eBook)
https://doi.org/10.1007/978-3-031-26588-4

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Foreword

In 2010, we introduced Google Earth Engine at the COP16 International Climate Conference. It was the world's first cloud-native, planetary-scale geospatial data analytics platform, designed from the ground-up to democratize access to earth observation data and help accelerate the transformation of "pixels to knowledge" for societal benefit. Looking back, while nothing like it had existed before (e.g., analysis-ready EO data, petapixel-scale processing, easily-accessible from any web browser), it seems almost inevitable that such a platform would be invented. A set of factors had converged in that moment to shift the dominant paradigm of remote sensing from the desktop to the cloud—the release by USGS/NASA of Landsat data for free (millions of scenes and petabytes of data, sitting on tapes going back to 1972), the advent of large-scale publicly available cloud computing such as the Google cloud, and the urgent need for accessible, timely, accurate information about the state of the planet and how it was changing, to support better decision-making and action on critical environmental and humanitarian issues.

In particular, the initial impetus for the Google Earth Engine project came from Brazilian geoscientists in 2008/9, who asked us if Google could help them stop the loss of more than a million acres of Amazon rainforest annually. They had already developed the remote sensing techniques to derive valuable information from daily updating satellite data, but were struggling to manage and process the "brutal" amount of EO data in a timely manner on their desktop computers. Further consultation with leading remote sensing scientists convinced us that this was a ubiquitous challenge as well as a "Google-scale" one, and so we set off to build Earth Engine in close collaboration with the remote sensing community.

Our first step was to partner with the USGS EROS Data Center team in bringing the entire multi-petabyte, multi-decadal Landsat data archive off tapes, and online, for the first time. It took three years to create this online "mirror" for Landsat, but it was worth it. This novel co-location of big Earth observation data in analysis-ready form with hundreds of thousands of CPUs for processing created new possibilities for characterizing our changing planet. Scientists began developing Earth Engine-powered methodologies which mapped, measured, and monitored global landscape change and environmental phenomena at unprecedented resolution, speed, and scale. Early applications included tracking and reducing global deforestation, mapping global surface water resources over time, estimating global crop yields to enhance food security, mapping and mitigating the risks of extreme weather events such as floods and drought, forecasting future risks of vector-borne diseases such as malaria, and even predicting where chimpanzees were likely to build their nests.

Since those early days, it has been exciting to witness and support the Earth Engine user community as it has grown and thrived, now including more than half a million individuals and organizations worldwide. This diverse community of scientists, academics, non-profits, governments, and industries has become increasingly impactful across a range of disciplines, advancing both the "state of the science" as well as the practical, real-world application of global environmental remote sensing. To date, the GEE community has collectively (and creatively) applied Earth Engine to author more than 20,000 scientific publications in journals such as Nature, Science and Remote Sensing of Environment. They have also developed numerous high-impact environmental monitoring applications, including Global Forest Watch, Global Fishing Watch, OpenET, Climate Engine, MapBiomas, SDG661.app, Restor.eco, Earth Map, Tracemark, and more. Every year, we have grown the scale of cloud computation that we serve to support the amazing work of this community, and today Earth Engine provides more than 500 M CPU hours per year (i.e., more than half a billion CPU hours!).

With our announcement last year that Earth Engine is available through Google Cloud Platform as an enterprise-grade service, governments and businesses are now relying upon GEE for mission-critical purposes, such as sustainable natural resource management, climate and disaster risk resilience, and sustainable sourcing of commodities to meet zero-deforestation commitments. In addition, new startups are now applying GEE technology to transform science and EO data into innovative sustainability-focused ideas and enterprises.

So, I am tremendously excited about this book, which has come at the perfect time. With Earth Engine's maturation into such a powerful and flexible environmental monitoring platform, the demand for geospatial data scientists with fluency and expertise in Earth Engine is skyrocketing. We are seeing universities all over the world creating curricula and teaching courses based on Earth Engine. While prior texts for teaching remote sensing reflect the old (desktop) paradigm, modern teachers, learners, and practitioners need a new textbook that is written in the language of cloud computing. A textbook that embeds familiar technical concepts of data structures, image processing and remote sensing in the context and machinery of cloud infrastructure, with its unique ability to process massive, multi-sensor, near real-time datasets at unprecedented speed and scale.

Cloud-Based Remote Sensing with Google Earth Engine: Fundamentals and Applications is exactly the textbook that we need: comprehensive, authoritative and also enjoyable! It deftly and thoroughly covers the fundamental content and material required to go from zero to sixty on Earth Engine. The variety of applications represented can help jump-start and inspire the reader in so many wonderful directions. The associated code samples ensure that you can get hands-on and productive, immediately and throughout.

I am especially delighted that this book was authored by such a diverse, accomplished group of Earth Engine experts, spanning backgrounds, disciplines, geographies, and career-stage (including undergrads to full professors), who voluntarily joined together in a common mission to create this much-needed Earth Engine reference text. They have made a tremendous contribution to our entire global community, aligned with our founding Earth Engine mission to support the greater good, and I am forever grateful.

Looking ahead, this is a time of innovation and evolution in the field of cloud-powered environmental remote sensing, with new satellites and sensors continually launching, producing new types of data (e.g., global atmospheric methane concentrations), while also improving spatial, spectral, and temporal resolution of existing data sources. Meanwhile, the classical remote sensing machine learning toolkit is now expanding to include modern deep learning/AI-based techniques. At Google, and together with the GEE community, we are continuing to push this envelope by advancing Earth Engine's fundamental capabilities and applications. For example, the recent launch of Dynamic World together with WRI introduced the first near real-time, probabilistic, 10 m global landcover mapping dataset based on deep learning/AI techniques, with which the GEE community is already driving further downstream innovations.

It has never been a more exciting, important, and relevant time to join this field and contribute to generating new knowledge about our changing planet. Policymakers and project implementers worldwide are seeking timely, accurate, science-based, and data-driven information to guide wiser decision-making and action on critical environmental and humanitarian issues. I hope that this book will turbo-charge your interest and ability to become proficient in Earth Engine and that you will join our growing community of people around the world dedicated to making it a better place, now and for generations to come. Enjoy!

Rebecca Moore
Director
Google Earth and Earth Engine
Google
Mountain View, USA

Introduction

Welcome to *Cloud-Based Remote Sensing with Google Earth Engine: Fundamentals and Applications*! This book is the product of more than a year of effort from more than a hundred individuals, working in concert to provide this free resource for learning how to use this exciting technology for the public good. The book includes work from professors, undergraduates, master's students, PhD students, postdocs, and independent consultants.

The book is broadly organized into two halves. The first half, **Fundamentals**, is a set of 31 labs designed to take you from being a complete Earth Engine novice to being a quite advanced user. The second half, **Applications**, presents you with a tour of the world of Earth Engine across 24 chapters, showing how it is used in a very wide variety of settings that rely on remote sensing data.

Using several strategies, we have worked hard to ensure a high-quality body of work across 55 chapters, more than 10,000 lines of code, and 250,000 words:

- The text of each chapter has been reviewed for the clarity of the scientific content and instructions, on a minimum of three occasions by people working independently of each other, for more than 350 detailed chapter reviews.
- The code in each chapter has been reviewed at Google for adherence to best practices and subsequently reviewed for consistency with the instructions in the book's text by two Earth Engine experts.
- A professional copy-editing team has worked through the entire book text, ensuring that all chapters have a consistent sound and approach, while preserving the voice of the authors.

Fundamentals

The goal of the Fundamentals half of the book is to introduce users to Earth Engine with a set of sequenced labs that are intended to be done in order. We first illustrate the operation of Earth Engine on individual multi-band images, with techniques that form the core vocabulary for Earth Engine image processing. We then introduce more complex options for data representation, data enhancements, and data interpretation. The tools introduced in those sections build on the core analysis strategies, with important techniques that can be highly valuable for addressing specialized analysis problems.

The Fundamentals half of the book is intended to have two main entry points: at Chaps. 1 and 12. It is comprised of six thematic parts, described below:

Part I: Programming and Remote Sensing Basics

Those who are almost entirely unfamiliar with remote sensing data, or are almost entirely unfamiliar with programming, or both, are encouraged to start at the beginning: Part I, Chap. 1 That chapter assumes that the reader has no programming experience, and begins with the most fundamental first steps: defining variables, printing values, etc.

The rest of Part I teaches the basics of displaying remote sensing data (Chap. 2), gives a brief survey of Earth Engine's very large data catalog (Chap. 3), and presents the vocabulary of remote sensing (Chap. 4), intended for both novices and those familiar with remote sensing but just starting with Earth Engine.

Part II: Interpreting Images

Part II assumes that you are aware of the topics, terms, and methods encountered in Part I. This part illustrates the basic operations that can be done on individual satellite images. Like in Part I, the chapters are arranged in sequence, here with a goal of leading you through how bands of remote sensing images can be manipulated to form indices (Chap. 5), which can be classified using a variety of supervised and unsupervised techniques (Chap. 6), giving results that can be assessed and further adjusted (Chap. 7).

Part III: Advanced Image Processing

Part III presents more advanced image processing techniques that can be accessed in Earth Engine using operations on a single image. These include finding a relationship between two or more bands using regression (Chap. 8), linear combinations of bands to produce the tasseled cap and principal components transformations (Chap. 9), morphological operations on classified images to highlight or de-emphasize spatial characteristics (Chap. 10), and object-based image analysis that groups pixels into spectrally similar spatially contiguous clusters (Chap. 11).

Part IV: Interpreting Image Series

Part IV introduces the analysis of time series in Earth Engine. Using long-term series of imagery can inform change and stability in land-use and land-cover patterns around the world. This series of chapters begins by presenting the *filter*, *map*, *reduce* paradigm (Chap. 12), which is used throughout Earth Engine for manipulating collections of both vector and raster data. The next chapter shows strategies for learning about and visualizing image collections (Chap. 13). Subsequent chapters show pixel-by-pixel calculations, including the aggregation of image characteristics in time (Chap. 14); detection and removal of cloud artifacts (Chap. 15), and the detection of change by comparing images at two dates (Chap. 16). Later chapters involve assessments that summarize a pixel's history, at annual scales (Chap. 17), in harmonically repeating patterns (Chap. 18), and at sub-annual scales (Chap. 19). Long-time series of image classifications can be interpreted for stability and change via Bayesian methods (Chap. 20), and lag effects in time series can be detected (Chap. 21).

Part V: Vectors and Tables

Part V shows how vector data can be used in Earth Engine. This includes uploading vector data and summarizing images within irregular polygons (Chap. 22), converting between raster and vector formats (Chap. 23), and computing zonal statistics efficiently with sophisticated functions (Chap. 24). The chapters conclude with a useful set of advanced vector operations (Chap. 25) and a digitizing tool for drawing features that change across the time span of an image collection (Chap. 26).

Part VI: Advanced Topics

With the foundation provided by the first five parts of the Fundamentals, you will be ready to do more complex work with Earth Engine. Part VI reveals some of these advanced topics, including advanced techniques for better raster visualization (Chap. 27), collaborating with others by sharing scripts and assets (Chap. 28), scaling up in time and space in efficient and effective ways (Chap. 29), creating apps to share to the wider public (Chap. 30), and linking to outside packages (Chap. 31).

Applications

Part of the appeal of such an extensive platform in widespread adoption, and one of the most satisfying aspects of editing this book project, has been exposure to the wide range of applications built on this platform. With thousands of papers now published using Earth Engine, the 24 chapters presented here are not intended to be a complete survey of who uses Earth Engine nor how it is best used. We have worked to find applications and authors who fit the following characteristics: (1) they created vibrant work that can be appreciated by both novices and experts; (2) the work has one or more unique characteristics that can have appeal beyond the scope of the book chapter's application; and (3) the presentation does not seek to be the final word on a subject, but rather as an opening of the subject to others.

Part VII: Human Applications

Earth Engine is used for both large-scale and relatively small-scale interpretations of images and time series of human activity and influence. These include agricultural environments (Chap. 32), the urban built environment (Chaps. 33 and 34), effects on air quality and surface temperature in cities (Chaps. 35 and 36), health and humanitarian activities (Chaps. 37 and 38), and the detection of illegal human activity under cloud cover (Chap. 39).

Part VIII: Aquatic and Hydrological Applications

Earth Engine is also used in the aquatic realm to understand hydrologic patterns across large areas. These include detection of subsurface features like groundwater (Chap. 40) and cover of the sea floor (Chap. 41). Surface water can be detected in satellite series, opening the opportunity to detect rapid changes like floods (Chap. 42) and to map rivers across the globe (Chap. 43). Human influence on hydrological systems can be detected and quantified in patterns of water balance and drought (Chap. 44) and changing patterns of the timing of snow (Chap. 45).

Part IX: Terrestrial Applications

Some of the many terrestrial applications of Earth Engine close out the book. These include the monitoring of active fires using multiple sensors and presented with user-interface apps (Chap. 46). Mangroves are complex systems that have elements of both the terrestrial and aquatic realms; they need to be both mapped (Chap. 47) and monitored for change (Chap. 48). Changes in forests can include both human degradation and deforestation (Chap. 49); the detection of these changes can be aided with the help of multiple sensors (Chap. 50). Analysts often need to use location-specific weather data (Chap. 51) and to create randomly located points for proper statistical analyses (Chap. 52). Rangelands, a major land use worldwide, present subtle changes over long-time periods that require distinctive techniques (Chap. 53). Finally, conservation of natural resources requires understanding both the effect of precipitation changes on area affected by disturbance (Chap. 54) and the effectiveness of protected areas in conserving places as intended (Chap. 55).

Uses of This Book

We have strived to produce a book that can be used by people working in a very wide range of settings: from pre-college students in a classroom, to university courses, to professional development workshops, to individuals working alone. The first seven Fundamentals chapters are tightly sequenced and assume no programming or remote sensing experience; their intention is to build a base of knowledge for novices to succeed. The remaining Fundamentals chapters are sequenced within their larger sections. In their introductions, each Fundamentals chapter lists earlier topics and chapter numbers that form the foundation for understanding that chapter's concepts, under the heading "Assumes you know how to…" However, although the concepts in the chapters do build on each other, each chapter's steps and data stand alone and do not require that you follow every preceding chapter in order to succeed. This should permit you to use individual sections of the book to solve specific problems you encounter. The **Learning Outcomes** index contains listings such as "Attaching user-defined metadata to an image when exporting," "Computing zonal summaries of calculated variables for elements in a Feature Collection," and "Using reducers to implement regression between image bands," among more than 150 other examples. The Applications chapters each stand alone and provide a guide both to the use of Earth Engine for that issue and to the technological details of how that application has been addressed by those authors.

We Want Your Feedback

We would like to get your feedback on chapters of the book. This feedback is useful in two ways: to make your experience better and to help others who will come after.

(1) Helping others: How hard is a chapter? How long did it take? Would you recommend it to someone else? What did you learn? Is something confusing? Has code stopped working? To review a lab, visit https://bit.ly/EEFA-review when you've finished. By doing this, you will keep the book current and provide future book users with your assessment of the strengths and weaknesses of each chapter. Please give feedback!
(2) Helping yourself: What have others already said about a chapter you are planning to do? How long will it take? How interesting is it? You can see what others have said at https://bit.ly/EEFA-reviews-stats.

Acknowledgements

Over 100 chapter authors volunteered their time to make this book a reality. Without compensation, they shared their knowledge, endured rounds of editorial suggestions, and processed multiple chapter reviews by individuals across a range of experience levels. The careful review of chapters was an enormous task undertaken by many people over countless hours: These included Ellen Brock, Florina Richard, Khashayar Azad, Phillip Pichette, Philippe Lizotte, Jake Hart, Natalie Sprenger, Jonah Zoldan, Sheryl Rose Reyes, and anonymous students at McGill University and the University of Toronto.

Copy-editing was provided by the team of Christine Kent and Jose Isaza. Early views of the chapters were edited and prepared with the help of Mark Essig.

This book was made possible in part by funding from SERVIR, a joint initiative of NASA, USAID, and leading geospatial organizations in Asia, Africa, and Latin America. We are grateful for their support and continued dedication to capacity building in the use of Earth observation information, Earth science, and technology.

The book was also made possible through funding from SilvaCarbon, an interagency effort of the US government to build capacity for the measurement, monitoring, and reporting of carbon in forests and other lands. With that support, each chapter's code was standardized and checked for bugs and inefficiencies repeatedly over several months.

The book was also made possible through the funding of a Discovery Grant from the Natural Sciences and Engineering Research Council of Canada. This grant from the people of Canada permitted us to dedicate substantial time to editorial work and overall quality control.

The contents are the responsibility of the authors and editors and do not necessarily reflect the views of Google, NSERC, SERVIR, SilvaCarbon, NASA, USAID, the Government of the United States, or the Government of Canada.

Other Sources

After you finish the tutorials of this book, you might consider some additional resources that could be of interest.

- Ujaval Gandhi's "End to End Earth Engine" materials introduce users to some of the fundamentals discussed here, while going deeper on several topics.
- Samapriya Roy's assembled "Awesome Earth Engine Datasets" addresses the need to curate the many exciting and useful resources that have been made with or for Earth Engine.
- Organizations such as SERVIR, SilvaCarbon, and the World Bank have created online tutorials covering some key Earth Engine topics.

- Earth Engine's impressive documentation includes a suite of tutorials, written by both Googlers and members of the Earth Engine user community, that can explain some key concepts.

Jeffrey A. Cardille
Nicholas E. Clinton
Morgan A. Crowley
David Saah

Contents

Part I

Programming and Remote Sensing Basics

In order to use Earth Engine well, you will need to develop basic skills in remote sensing and programming. The language of this book is JavaScript, and you will begin by learning how to manipulate variables using it. With that base, you will learn about viewing individual satellite images, viewing collections of images in Earth Engine, and how common remote sensing terms are referenced and used in Earth Engine.

JavaScript and the Earth Engine API

1

Ujaval Gandhi

Overview

This chapter introduces the Google Earth Engine application programming interface (API) and the JavaScript syntax needed to use it. You will learn about the Code Editor environment and get comfortable typing, running, and saving scripts. You will also learn the basics of JavaScript language, such as variables, data structures, and functions.

Learning Outcomes

- Familiarity with the Earth Engine Code Editor.
- Familiarity with the JavaScript syntax.
- Ability to use the Earth Engine API functions from the Code Editor.

Assumes you know how to

- Sign up for an Earth Engine account (See the Google documentation for details).
- Access the Earth Engine Code Editor (See the Google documentation for details).

U. Gandhi (✉)
Spatial Thoughts LLP, FF105 Aaradhya, Gala Gymkhana Road, Bopal, Ahmedabad 380058, India
e-mail: ujaval@spatialthoughts.com

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_1

1.1 Introduction to Theory

The Earth Engine API

Google Earth Engine is a cloud-based platform for scientific data analysis. It provides ready-to-use, cloud-hosted datasets and a large pool of servers. One feature that makes Earth Engine particularly attractive is the ability to run large computations very fast by distributing them across a large pool of servers. The ability to efficiently use cloud-hosted datasets and computation is enabled by the Earth Engine API.

An API is a way to communicate with Earth Engine servers. It allows you to specify what computation you would like to do and then to receive the results. The API is designed so that users do not need to worry about *how* the computation is distributed across a cluster of machines, and the results are assembled. Users of the API simply specify what needs to be done. This greatly simplifies the code by hiding the implementation detail from the users. It also makes Earth Engine very approachable for users who are not familiar with writing code.

Earth Engine API is designed to be language agnostic. Google provides official client libraries to use the API from both JavaScript and Python. The API remains largely the same regardless of the programming language you use. The main difference is the syntax used to call the API functions. Once you learn the syntax for programming languages, your code can be adapted easily because they all use the same API functions.

Why JavaScript?

JavaScript may not be the first choice of programming language for many researchers and data scientists, and some may be wondering why this book is based on the JavaScript API instead of Python or R.

The Earth Engine JavaScript API is the most mature and easiest to use when getting started. The Earth Engine platform comes with a web-based Code Editor that allows you to start using the Earth Engine JavaScript API without any installation. It also provides additional functionality to display your results on a map, save your scripts, access documentation, manage tasks, and more. It has a one-click mechanism to share your code with other users—allowing for easy reproducibility and collaboration. In addition, the JavaScript API comes with a user interface library, which allows you to create charts and web-based applications with little effort.

In practice, you do not need to become a JavaScript expert to use Earth Engine. The basic syntax described here should be sufficient. A good tip is that if you find yourself doing something complicated in JavaScript, it might be done much better in Earth Engine. All the important computations in Earth Engine need to use the API functions, and even a basic operation—such as adding two numbers in Earth Engine—should be done using the Earth Engine API.

1.2 Practicum

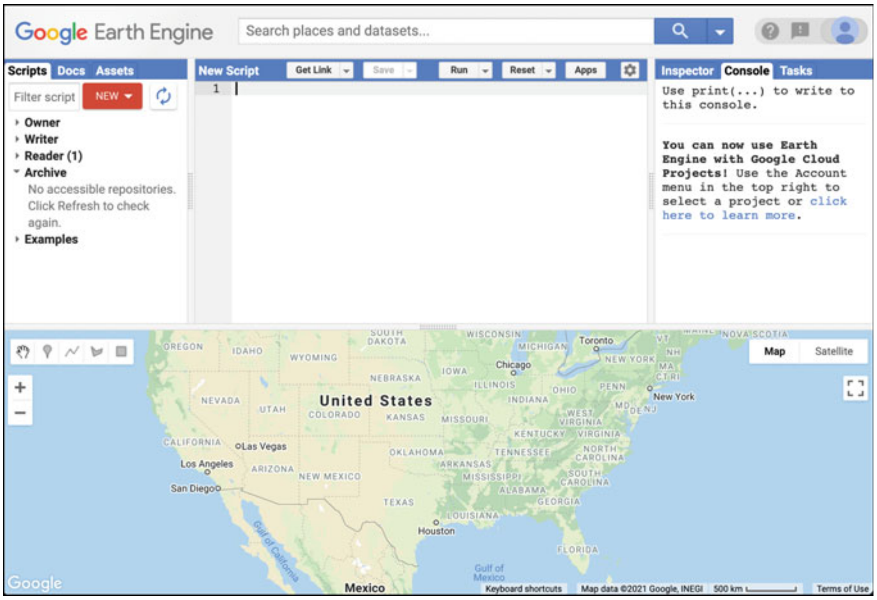
1.2.1 Section 1: Getting Started in the Code Editor

The Code Editor is an integrated development environment for the Earth Engine JavaScript API. It offers an easy way to type, debug, run, and manage code. Once you have followed Google's documentation on registering for an Earth Engine account, you should follow the documentation to open the Code Editor. When you first visit the Code Editor, you will see a screen such as the one shown in Fig. 1.1.

The Code Editor (Fig. 1.1) allows you to type JavaScript code and execute it. When you are first learning a new language and getting used to a new programming environment, it is customary to make a program to display the words "Hello World." This is a fun way to start coding that shows you how to give input to the program and how to execute it. It also shows where the program displays the output. Doing this in JavaScript is quite simple. Copy the following code into the center panel.

```
print('Hello World');
```

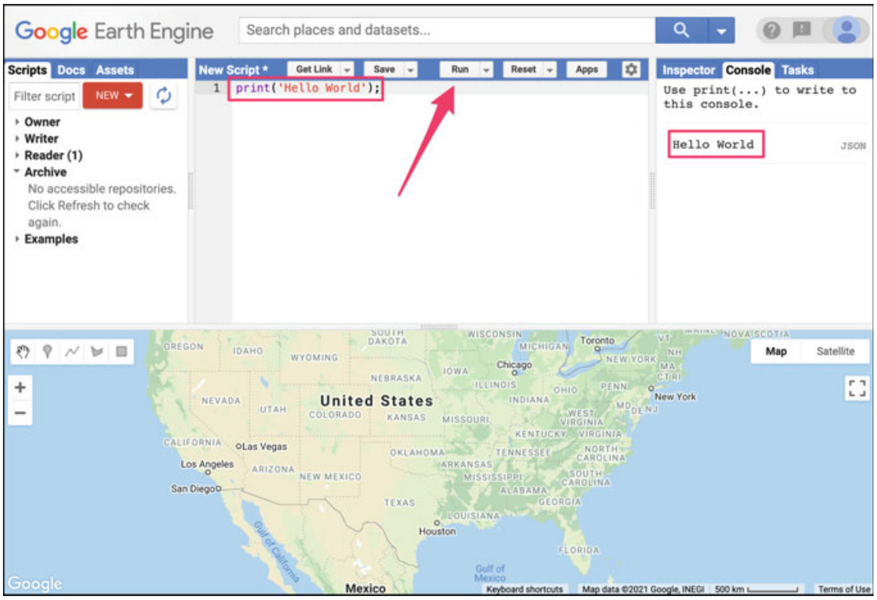
The line of code above uses the JavaScript `print` function to print the text "Hello World" to the screen. Once you enter the code, click the **Run** button. The



No accessible repositories. Click Refresh to check again.

```
Use print(...) to write to this console.

You can now use Earth Engine with Google Cloud Projects! Use the Account menu in the top right to select a project or click here to learn more.
```

Fig. 1.1 Earth Engine code editor



```
print('Hello World');
```

No accessible repositories. Click Refresh to check again.

```
Use print(...) to write to this console.

Hello World
```

Fig. 1.2 Typing and running code

output will be displayed on the upper right-hand panel under the **Console** tab (Fig. 1.2.).

You now know where to type your code, how to run it, and where to look for the output. You just wrote your first Earth Engine script and may want to save it. Click the **Save** button (Fig. 1.3).

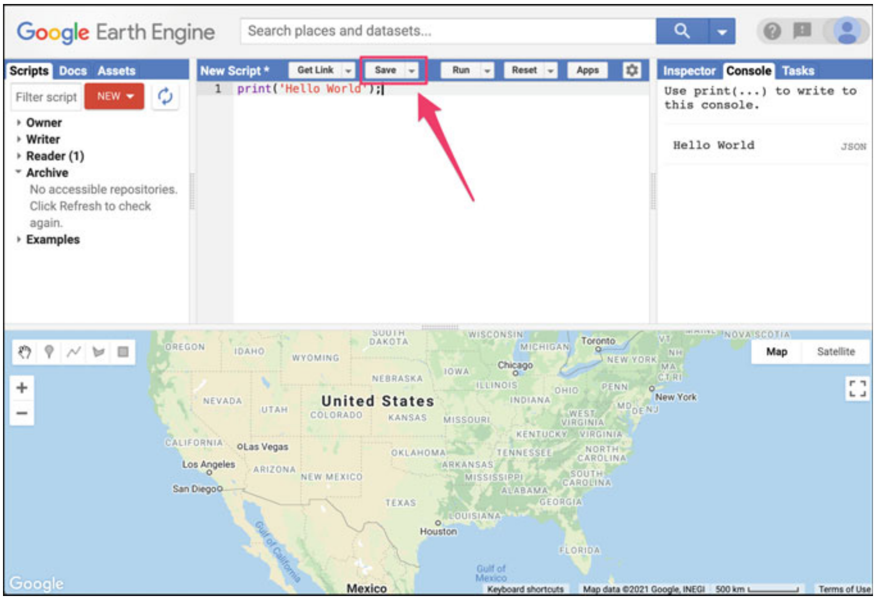
If this is your first time using the Code Editor, you will be prompted to create a *home folder*. This is a folder in the cloud where all your code will be saved. You can pick a name of your choice, but remember that it cannot be changed and will forever be associated with your account. A good choice for the name would be your Google Account username (Fig. 1.4).

Once your home folder is created, you will be prompted to enter a new *repository*. A repository can help you organize and share code. Your account can have multiple repositories, and each repository can have multiple scripts inside it. To get started, you can create a repository named “default” (Fig. 1.5).

Finally, you will be able to save your script inside the newly created repository. Enter the name “hello_world” and click **OK** (Fig. 1.6).

Once the script is saved, it will appear in the script manager panel (Fig. 1.7). The scripts are saved in the cloud and will always be available to you when you open the Code Editor.

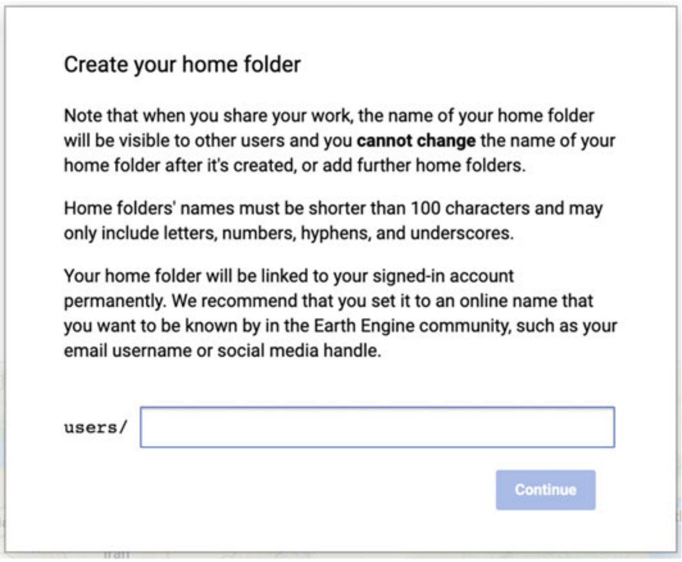
Now you should be familiar with how to create, run, and save your scripts in the Code Editor. You are ready to start learning the basics of JavaScript.


Google Earth Engine
Search places and datasets...
Scripts Docs Assets
Filter script NEW
Owner
Writer
Reader (1)
Archive
No accessible repositories. Click Refresh to check again.
Examples
New Script * Get Link Save Run Reset Apps
Inspector Console Tasks
Map Satellite

```
print('Hello World');
```

```
Use print(...) to write to
this console.

Hello World                JSON
```

Fig. 1.3 Saving a script


Create your home folder

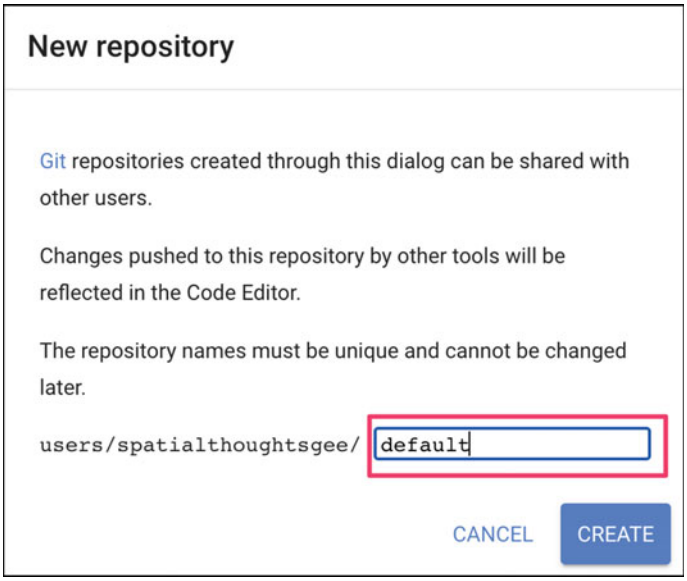
Note that when you share your work, the name of your home folder will be visible to other users and you **cannot change** the name of your home folder after it's created, or add further home folders.

Home folders' names must be shorter than 100 characters and may only include letters, numbers, hyphens, and underscores.

Your home folder will be linked to your signed-in account permanently. We recommend that you set it to an online name that you want to be known by in the Earth Engine community, such as your email username or social media handle.

users/

Fig. 1.4 Creating a home folder



New repository

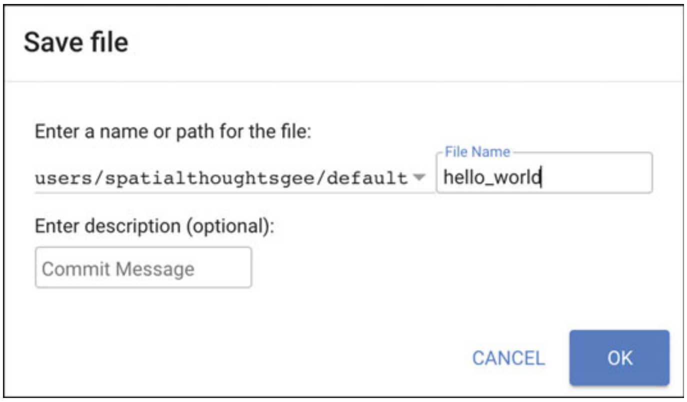
Git repositories created through this dialog can be shared with other users.

Changes pushed to this repository by other tools will be reflected in the Code Editor.

The repository names must be unique and cannot be changed later.

users/spatialthoughtsgee/ default

Fig. 1.5 Creating a new repository



Save file

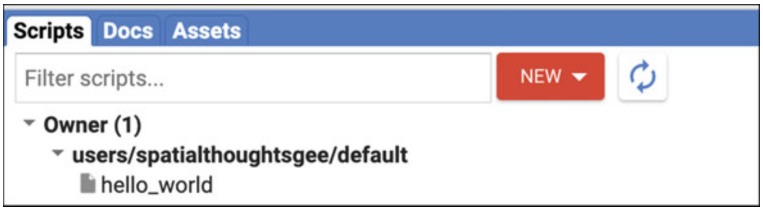
Enter a name or path for the file:

users/spatialthoughtsgee/default File Name: hello_world

Enter description (optional):

Commit Message

Fig. 1.6 Saving a file



Owner (1)
users/spatialthoughtsgee/default
hello_world

Fig. 1.7 Script manager

1.2.2 Section 2: JavaScript Basics

To be able to construct a script for your analysis, you will need to use JavaScript. This section covers the JavaScript syntax and basic data structures. In the sections that follow, you will see more JavaScript code, noted in a distinct font and with shaded background. As you encounter code, paste it into the Code Editor and run the script.

Variables
In a programming language, variables are used to store data values. In JavaScript, a variable is defined using the **var** keyword followed by the name of the variable. The code below assigns the text "San Francisco" to the variable named city. Note that the text string in the code should be surrounded by quotes. You are free to use either ' (single quotes) or " (double quotes), and they must match at the beginning and end of each string. In your programs, it is advisable to be consistent—use either single quotes or double quotes throughout a given script (the code in this book generally uses single quotes for code). Each statement of your script should typically end with a semicolon, although Earth Engine's code editor does not require it.

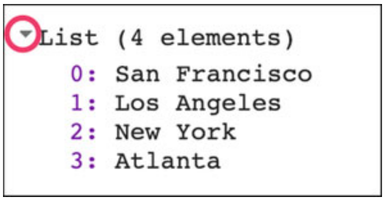
```
var city = 'San Francisco';
```

If you print the variable city, you will get the value stored in the variable (San Francisco) printed in the **Console**.

```
print(city);
```

When you assign a text value, the variable is automatically assigned the type *string*. You can also assign numbers to variables and create variables of type *number*. The following code creates a new variable called population and assigns a number as its value.

```
var population = 873965;
print(population);
```



```
▾List (4 elements)
    0: San Francisco
    1: Los Angeles
    2: New York
    3: Atlanta
```

Fig. 1.8 JavaScript list

Lists

It is helpful to be able to store multiple values in a single variable. JavaScript provides a data structure called a *list* that can hold multiple values. We can create a new list using the square brackets `[]` and adding multiple values separated by a comma.

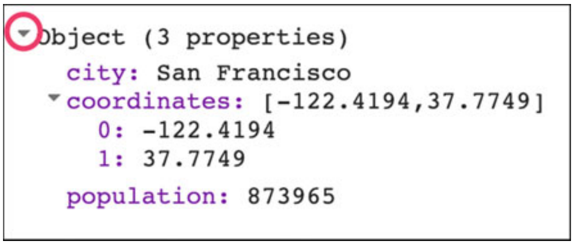
```
var cities = ['San Francisco', 'Los Angeles', 'New York',
'Atlanta'];
print(cities);
```

If you look at the output in the **Console**, you will see "List" with an expander arrow (▹) next to it. Clicking on the arrow will expand the list and show you its content. You will notice that along with the four items in the list, there is a number next to each value. This is the *index* of each item. It allows you to refer to each item in the list using a numeric value that indicates its position in the list (Fig. 1.8).

Objects

Lists allow you to store multiple values in a single container variable. While useful, it is not appropriate to store structured data. It is helpful to be able to refer to each item with its name rather than its position. Objects in JavaScript allow you to store key-value pairs, where each value can be referred to by its key. You can create a dictionary using the curly braces `{}`. The code below creates an object called `cityData` with some information about San Francisco.

Note a few important things about the JavaScript syntax here. First, we can use multiple lines to define the object. Only when we put in the semicolon (`;`) is the command considered complete. This helps format the code to make it more readable. Also note the choice of the variable name `cityData`. The variable contains two words. The first word is in lowercase, and the first letter of the second word is capitalized. This type of naming scheme of joining multiple words into a single variable name is called "camel case." While it is not mandatory to name your variables using this scheme, it is considered a good practice to follow. Functions and parameters in the Earth Engine API follow this convention, so your code will be much more readable if you follow it too.



```
▾Object (3 properties)
   city: San Francisco
  ▾coordinates: [-122.4194,37.7749]
     0: -122.4194
     1: 37.7749
   population: 873965
```

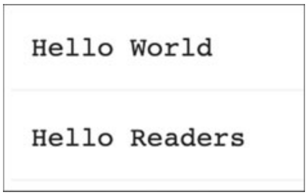
Fig. 1.9 JavaScript object

```
var cityData = {
    'city': 'San Francisco',
    'coordinates': [-122.4194, 37.7749],
    'population': 873965
};
print(cityData);
```

The object will be printed in the **Console**. You can see that instead of a numeric index, each item has a label. This is known as the *key* and can be used to retrieve the value of an item (Fig. 1.9).

Functions
While using Earth Engine, you will need to define your own functions. Functions take user inputs, use them to carry out some computation, and send an output back. Functions allow you to group a set of operations together and repeat the same operations with different parameters without having to rewrite them every time. Functions are defined using the `function` keyword. The code below defines a function called `greet` that takes an input called `name` and returns a greeting with *Hello* prefixed to it. Note that we can call the function with different input, and it generates different outputs with the same code (Fig. 1.10).

```
var greet = function(name) {
    return 'Hello ' + name;
};
print(greet('World'));
print(greet('Readers'));
```



```
Hello World

Hello Readers
```

Fig. 1.10 JavaScript function output

Comments

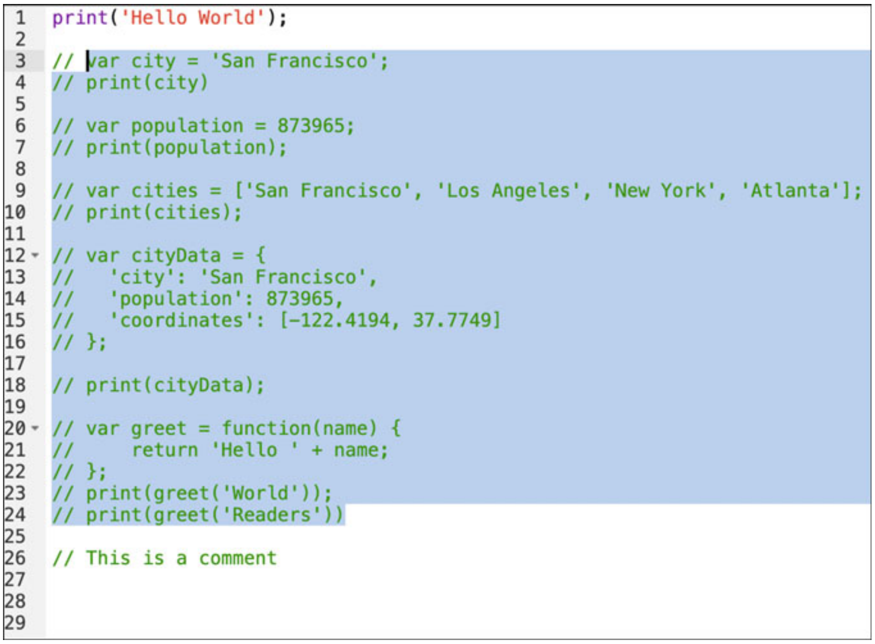
While writing code, it is useful to add a bit of text to explain the code or leave a note for yourself. It is a good programming practice to always add comments in the code explaining each step. In JavaScript, you can prefix any line with two forward slashes // to make it a comment. The text in the comment will be ignored by the interpreter and will not be executed.

```
// This is a comment!
```

The Code Editor also provides a shortcut—*Ctrl* + / on Windows, *Cmd* + / on Mac—to comment or uncomment multiple lines at a time. You can select multiple lines and press the key combination to make them all comments. Press again to reverse the operation. This is helpful when debugging code to stop certain parts of the script from being executed (Fig. 1.11).

Congratulations! You have learned enough JavaScript to be able to use the Earth Engine API. In the next section, you will see how to access and execute Earth Engine API functions using JavaScript.

Code Checkpoint F10a. The book's repository contains a script that shows what your code should look like at this point.



```
print('Hello World');

// var city = 'San Francisco';
// print(city)

// var population = 873965;
// print(population);

// var cities = ['San Francisco', 'Los Angeles', 'New York', 'Atlanta'];
// print(cities);

// var cityData = {
//   'city': 'San Francisco',
//   'population': 873965,
//   'coordinates': [-122.4194, 37.7749]
// };

// print(cityData);

// var greet = function(name) {
//     return 'Hello ' + name;
// };
// print(greet('World'));
// print(greet('Readers'))

// This is a comment



```

Fig. 1.11 Commenting multiple lines

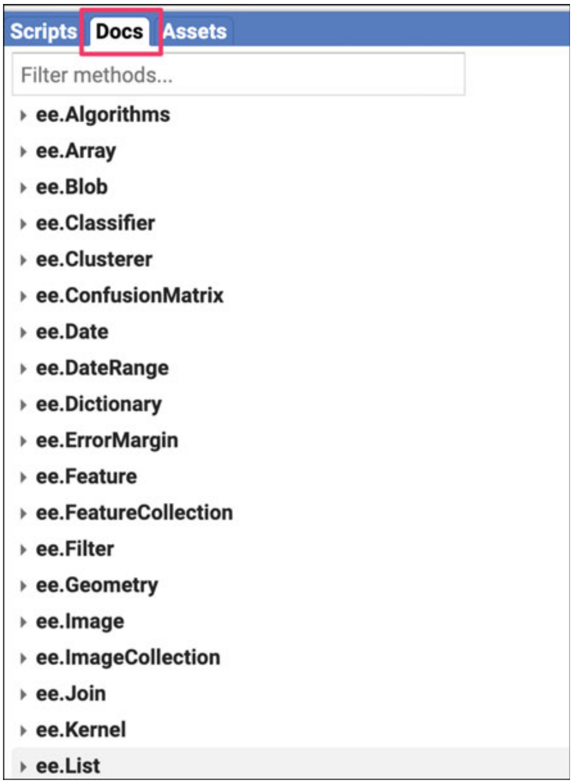
1.2.3 Section 3: Earth Engine API Basics

The Earth Engine API is vast and provides objects and methods to do everything from simple math to advanced algorithms for image processing. In the Code Editor, you can switch to the Docs tab to see the API functions grouped by object types. The API functions have the prefix `ee` (for Earth Engine) (Fig. 1.12).

Let us learn to use the API. Suppose you want to add two numbers, represented by the variables `a` and `b`, as below. Make a new script and enter the following:

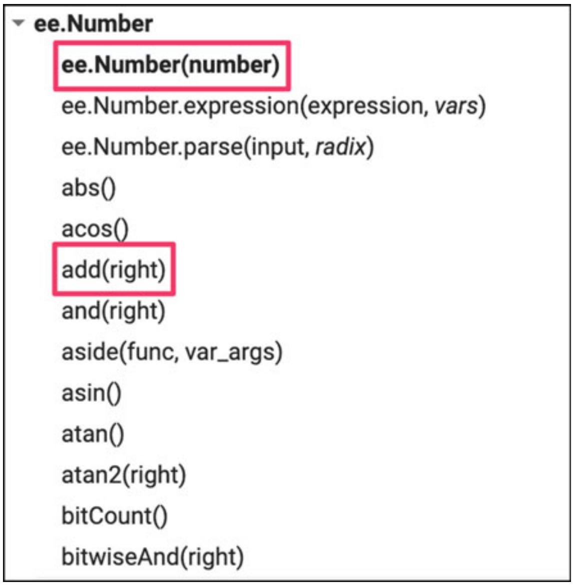
```
var a = 1;
var b = 2;
```

In Sect. 1.2.1, you learned how to store numbers in variables, but not how to do any computation. This is because when you use Earth Engine, you do not do addition using JavaScript operators. For example, you would not write "var $c = a + b$" to add the two numbers. Instead, the Earth Engine API provides you with functions to do this, and it is important that you use the API functions whenever



ee.Algorithms
ee.Array
ee.Blob
ee.Classifier
ee.Clusterer
ee.ConfusionMatrix
ee.Date
ee.DateRange
ee.Dictionary
ee.ErrorMargin
ee.Feature
ee.FeatureCollection
ee.Filter
ee.Geometry
ee.Image
ee.ImageCollection
ee.Join
ee.Kernel
ee.List

Fig. 1.12 Earth Engine API docs



ee.Number
ee.Number(number)
ee.Number.expression(expression, *vars*)
ee.Number.parse(input, *radix*)
abs()
acos()
add(right)
and(right)
aside(func, var_args)
asin()
atan()
atan2(right)
bitCount()
bitwiseAnd(right)

Fig. 1.13 ee.Number module

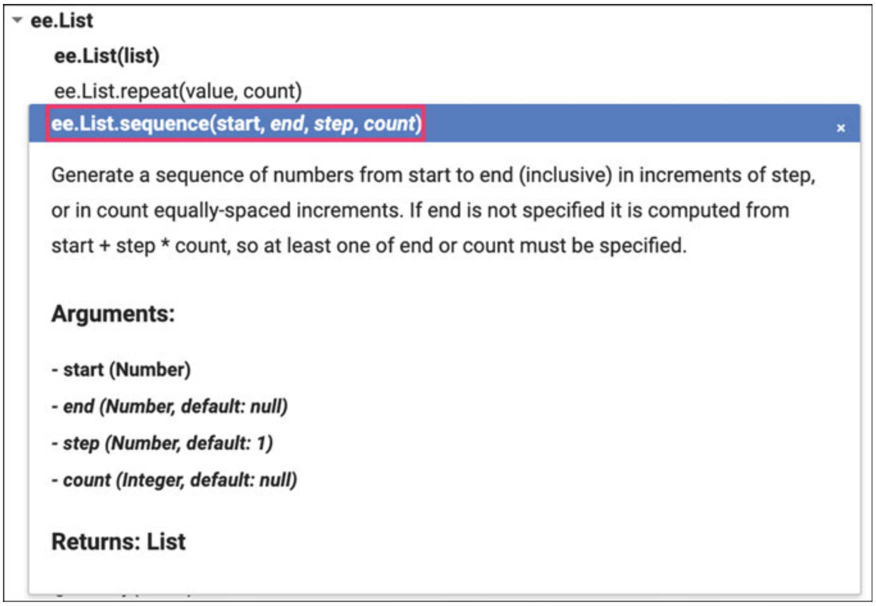
you can. It may seem awkward at first, but using the functions, as we will describe below, will help you avoid timeouts and create efficient code.

Looking at the **Docs** tab, you will find a group of methods that can be called on an ee.Number. Expand it to see the various functions available to work with numbers. You will see the ee.Number function that creates an Earth Engine number object from a value. In the list of functions, there is an add function for adding two numbers. That is what you use to add a and b (Fig. 1.13).

To add a and b, we first create an ee.Number object from variable a with ee.Number(a). And then we can use the add(b) call to add the value of b to it. The following code shows the syntax and prints the result which, of course, is the value 3:

```
var result = ee.Number(a).add(b);
print(result);
```

By now, you may have realized that when learning to program in Earth Engine, you do not need to deeply learn JavaScript or Python—instead, they are ways to access the Earth Engine API. This API is the same whether it is called from JavaScript or Python.


▾ ee.List

ee.List(list)

ee.List.repeat(value, count)

ee.List.sequence(start, *end*, *step*, *count*)

Generate a sequence of numbers from start to end (inclusive) in increments of step, or in count equally-spaced increments. If end is not specified it is computed from start + step * count, so at least one of end or count must be specified.

Arguments:

- start (Number)
- *end (Number, default: null)*
- *step (Number, default: 1)*
- *count (Integer, default: null)*

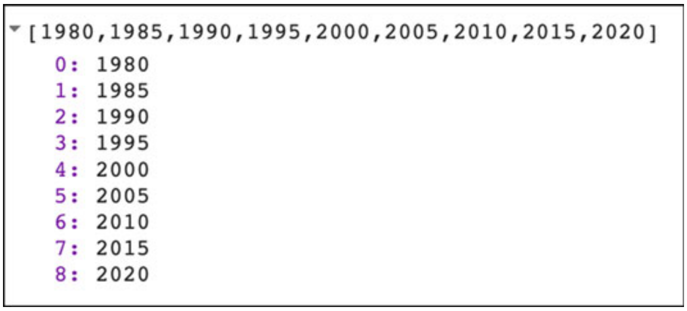
Returns: List

Fig. 1.14 `ee.List.sequence` function

Here is another example to drive this point home. Let us say you are working on a task that requires you to create a list of years from 1980 to 2020 with a five-year interval. If you are faced with this task, the first step is to switch to the **Docs** tab and open the `ee.List` module. Browse through the functions and see if there are any functions that can help. You will notice a function `ee.List.sequence`. Clicking on it will bring up the documentation of the function (Fig. 1.14).

The function `ee.List.sequence` is able to generate a sequence of numbers from a given `start` value to the `end` value. It also has an optional parameter `step` to indicate the increment between each number. We can create a `ee.List` of numbers representing years from 1980 to 2020, counting by 5, by calling this predefined function with the following values: start = 1980, end = 2020, and step = 5.

```
var yearList = ee.List.sequence(1980, 2020, 5);
print(yearList);
```



```
▾[1980,1985,1990,1995,2000,2005,2010,2015,2020]
   0: 1980
   1: 1985
   2: 1990
   3: 1995
   4: 2000
   5: 2005
   6: 2010
   7: 2015
   8: 2020
```

Fig. 1.15 Output of `ee.List.sequence` function

The output printed in the **Console** will show that the variable `yearList` indeed contains the list of years with the correct interval (Fig. 1.15).

You just accomplished a moderately complex programming task with the help of Earth Engine API.

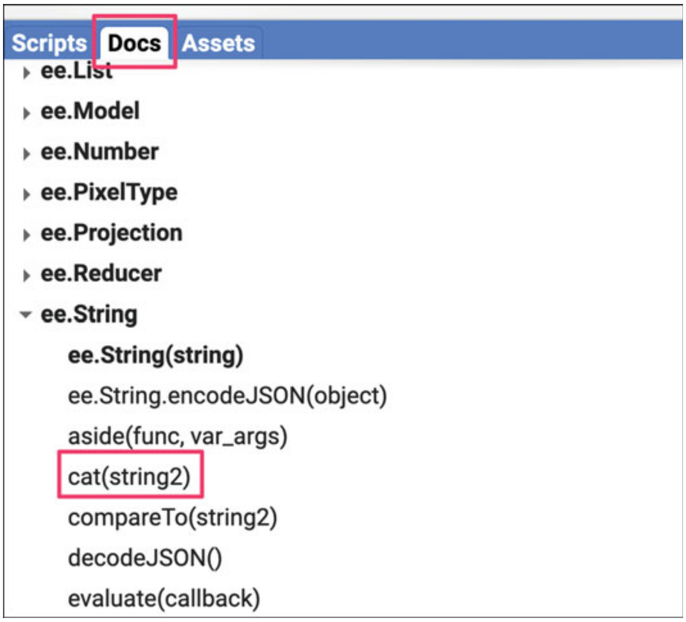
Code Checkpoint F10b. The book's repository contains a script that shows what your code should look like at this point.

1.3 Synthesis

Assignment 1. Suppose you have the following two string variables defined in the code below. Use the Earth Engine API to create a new string variable called `result` by combining these two strings. Print it in the **Console**. The printed value should read "Sentinel2A."

```
var mission = ee.String('Sentinel');
var satellite = ee.String('2A');
```

Hint: Use the `cat` function from the `ee.String` module to "concatenate" (join together) the two strings. You will find more information about all available functions in the **Docs** tab of the Code Editor (Fig. 1.16).



ee.List
ee.Model
ee.Number
ee.PixelType
ee.Projection
ee.Reducer
ee.String
ee.String(string)
ee.String.encodeJSON(object)
aside(func, var_args)
cat(string2)
compareTo(string2)
decodeJSON()
evaluate(callback)

Fig. 1.16 **Docs** tab showing functions in the `ee.String` module

1.4 Conclusion

This chapter introduced the Earth Engine API. You also learned the basics of JavaScript syntax to be able to use the API in the Code Editor environment. We hope you now feel a bit more comfortable starting your journey to become an Earth Engine developer. Regardless of your programming background or familiarity with JavaScript, you have the tools at your disposal to start using the Earth Engine API to build scripts for remote sensing analysis.

2 Exploring Images

Jeff Howarth

Overview
Satellite images are at the heart of Google Earth Engine's power. This chapter teaches you how to inspect and visualize data stored in image bands. We first visualize individual bands as separate map layers and then explore a method to visualize three different bands in a single composite layer. We compare different kinds of composites for satellite bands that measure electromagnetic radiation in the visible and non-visible spectrum. We then explore images that represent more abstract attributes of locations and create a composite layer to visualize change over time.

Learning Outcomes

- Using the Code Editor to load an image.
- Using code to select image bands and visualize them as map layers.
- Understanding true- and false-color composites of images.
- Constructing new multiband images.
- Understanding how additive color works and how to interpret RGB composites.

Assumes you know how to

- Sign up for an Earth Engine account, open the Code Editor, and save your script (Chap. 1).

J. Howarth (✉)
Middlebury College, Middlebury, VT, USA
e-mail: jhowarth@middlebury.edu

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_2

2.1 Practicum

2.1.1 Section 1: Accessing an Image

To begin, you will construct an image with the Code Editor. In the sections that follow, you will see code in a distinct font and with shaded background. As you encounter code, paste it into the center panel of the Code Editor and click **Run**.

First, copy and paste the following:

```
var first_image = ee.Image(
    'LANDSAT/LT05/C02/T1_L2/LT05_118038_20000606');
```

When you click **Run**, Earth Engine will load an image captured by the Landsat 5 satellite on June 6, 2000. You will not yet see any output.

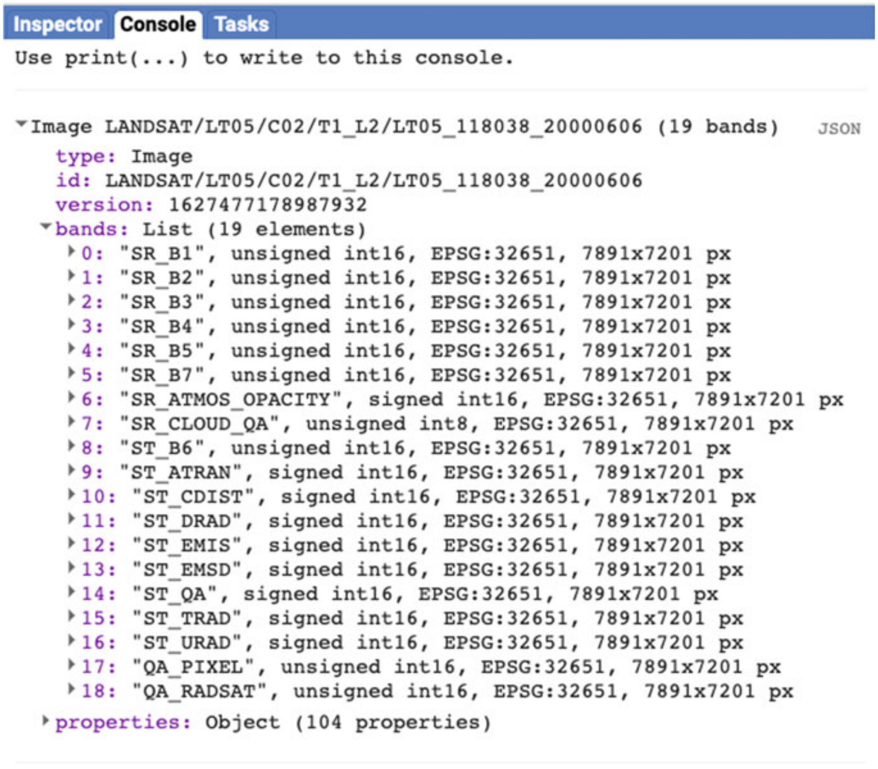
You can explore the image in several ways. To start, you can retrieve *metadata* (descriptive data about the image) by printing the image to the Code Editor's **Console** panel:

```
print(first_image);
```

In the **Console** panel, you may need to click the expander arrows to show the information. You should be able to read that this image consists of 19 different *bands*. For each band, the metadata lists four properties, but for now let us simply note that the first property is a *name* or label for the band enclosed in quotation marks. For example, the name of the first band is “SR_B1” (Fig. 2.1).

A satellite sensor like Landsat 5 measures the magnitude of radiation in different portions of the electromagnetic spectrum. The first six bands in our image (“SR_B1” through “SR_B7”) contain measurements for six different portions of the spectrum. The first three bands measure visible portions of the spectrum or quantities of blue, green, and red lights. The other three bands measure infrared portions of the spectrum that are not visible to the human eye.

An image band is an example of a *raster data model*, a method of storing geographic data in a two-dimensional grid of *pixels*, or *picture elements*. In remote sensing, the value stored by each pixel is often called a *Digital Number* or *DN*. Depending on the sensor, the pixel value or DN can represent a range of possible data values.



```
Use print(...) to write to this console.

▾Image LANDSAT/LT05/C02/T1_L2/LT05_118038_20000606 (19 bands)
   type: Image
   id: LANDSAT/LT05/C02/T1_L2/LT05_118038_20000606
   version: 1627477178987932
  ▾bands: List (19 elements)
    ▸0: "SR_B1", unsigned int16, EPSG:32651, 7891x7201 px
    ▸1: "SR_B2", unsigned int16, EPSG:32651, 7891x7201 px
    ▸2: "SR_B3", unsigned int16, EPSG:32651, 7891x7201 px
    ▸3: "SR_B4", unsigned int16, EPSG:32651, 7891x7201 px
    ▸4: "SR_B5", unsigned int16, EPSG:32651, 7891x7201 px
    ▸5: "SR_B7", unsigned int16, EPSG:32651, 7891x7201 px
    ▸6: "SR_ATMOS_OPACITY", signed int16, EPSG:32651, 7891x7201 px
    ▸7: "SR_CLOUD_QA", unsigned int8, EPSG:32651, 7891x7201 px
    ▸8: "ST_B6", unsigned int16, EPSG:32651, 7891x7201 px
    ▸9: "ST_ATRAN", signed int16, EPSG:32651, 7891x7201 px
    ▸10: "ST_CDIST", signed int16, EPSG:32651, 7891x7201 px
    ▸11: "ST_DRAD", signed int16, EPSG:32651, 7891x7201 px
    ▸12: "ST_EMIS", signed int16, EPSG:32651, 7891x7201 px
    ▸13: "ST_EMSD", signed int16, EPSG:32651, 7891x7201 px
    ▸14: "ST_QA", signed int16, EPSG:32651, 7891x7201 px
    ▸15: "ST_TRAD", signed int16, EPSG:32651, 7891x7201 px
    ▸16: "ST_URAD", signed int16, EPSG:32651, 7891x7201 px
    ▸17: "QA_PIXEL", unsigned int16, EPSG:32651, 7891x7201 px
    ▸18: "QA_RADSAT", unsigned int16, EPSG:32651, 7891x7201 px
  ▸properties: Object (104 properties)
```

Fig. 2.1 Image metadata printed to **Console** panel

Some of this information, like the names of the bands and their dimensions (number of pixels wide by number of pixels tall), we can see in the metadata. Other pieces of information, like the portions of the spectrum measured in each band and the range of possible data values, can be found through the Earth Engine Data Catalog (which is described in the next two chapters) or with other Earth Engine methods. These will be described in more detail later in the book.

2.1.2 Section 2: Visualizing an Image

Now, let us add one of the bands to the map as a *layer* so that we can see it.

```
Map.addLayer(
    first_image, //  dataset to display
    {
        bands: ['SR_B1'], //  band to display
        min: 8000, //  display range
        max: 17000
    },
    'Layer 1' //  name to show in Layer Manager
);
```

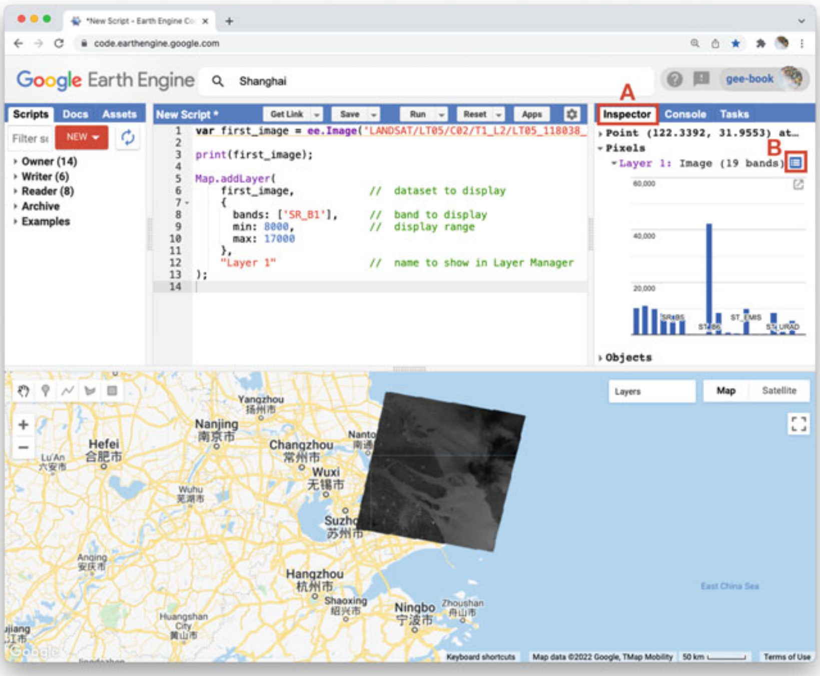
The code here uses the `addLayer` method of the map in the Code Editor. There are four important components of the command above:

1. `first_image`: This is the dataset to display on the map.
2. `bands`: These are the particular bands from the dataset to display on the map. In our example, we displayed a single band named "SR_B1".
3. `min`, `max`: These represent the lower and upper bounds of values from "SR_B1" to display on the screen. By default, the minimum value provided (8000) is mapped to black, and the maximum value provided (17,000) is mapped to white. The values between the minimum and maximum are mapped linearly to grayscale between black and white. Values below 8000 are drawn as black. Values above 17,000 are drawn as white. Together, the bands, min, and max parameters define *visualization parameters*, or instructions for data display.
4. `'Layer 1'`: This is a label for the map layer to display in the Layer Manager. This label appears in the dropdown menu of layers in the upper right of the map.

When you run the code, you might not notice the image displayed unless you pan around and look for it. To do this, click and drag the map toward Shanghai, China. (You can also jump there by typing "Shanghai" into the **Search** panel at the top of the Code Editor, where the prompt says **Search places and datasets…**) Over Shanghai, you should see a small, dark, slightly angled square. Use the zoom tool (the + sign, upper left of map) to increase the zoom level and make the square appear larger.

Can you recognize any features in the image? By comparing it to the standard Google Map that appears under the image (as the *base layer*), you should be able to distinguish the coastline. The water near the shore generally appears a little lighter than the land, except perhaps for a large, light-colored blob on the land in the bottom of the image.

Let us explore this image with the **Inspector** tool. When you click on the **Inspector** tab on the right side of the Code Editor (Fig. 2.2, area *A*), your cursor should now look like crosshairs. When you click on a location in the image, the **Inspector** panel will report data for that location under three categories as follows:



```
var first_image = ee.Image('LANDSAT/LT05/C02/T1_L2/LT05_118038_

print(first_image);

Map.addLayer(
    first_image,          //  dataset to display
    {
      bands: ['SR_B1'],     //  band to display
      min: 8000,            //  display range
      max: 17000
    },
    "Layer 1"               //  name to show in Layer Manager
);
```

```
▸ Point (122.3392, 31.9553) at…
▾ Pixels
  ▾ Layer 1: Image (19 bands)
▸ Objects
```

Fig. 2.2 Image data reported through the **Inspector** panel

- *Point*: data about the location on the map. This includes the geographic location (longitude and latitude) and some data about the map display (zoom level and scale).
- *Pixels***:** data about the pixel in the layer. If you expand this, you will see the name of the map layer, a description of the data source, and a bar chart. In our example, we see that "Layer 1" is drawn from an image dataset that contains 19 bands. Under the layer name, the chart displays the pixel value at the location that you clicked for each band in the dataset. When you hover your cursor over a bar, a panel will pop up to display the band name and "band value" (pixel value). To find the pixel value for "SR_B1", hover the cursor over the first bar on the left. Alternatively, by clicking on the little blue icon to the right of "Layer 1" (Fig. 2.2, area *B*), you will change the display from a bar chart to a dictionary that reports the pixel value for each band.
- *Objects*: data about the source dataset. Here, you will find metadata about the image that looks very similar to what you retrieved earlier when you directed Earth Engine to print the image to the **Console**.

Let us add two more bands to the map.

```
Map.addLayer(
    first_image,
    {
        bands: ['SR_B2'],
        min: 8000,
        max: 17000
    },
    'Layer 2',
    0, //  shown
    1 //  opacity
);

Map.addLayer(
    first_image,
    {
        bands: ['SR_B3'],
        min: 8000,
        max: 17000
    },
    'Layer 3',
    1, //  shown
    0 //  opacity
);
```

In the code above, notice that we included two additional parameters to the `Map.addLayer` call. One parameter controls whether or not the layer is *shown* on the screen when the layer is drawn. It may be either 1 (shown) or 0 (not shown). The other parameter defines the *opacity* of the layer or your ability to "see through" the map layer. The opacity value can range between 0 (transparent) and 1 (opaque).

Do you see how these new parameters influence the map layer displays (Fig. 2.3)? For Layer 2, we set the shown parameter as 0. For Layer 3, we set the opacity parameter as 0. As a result, neither layer is visible to us when we first run the code. We can make each layer visible with controls in the **Layers** manager checklist on the map (at top right). Expand this list and you should see the names that we gave each layer when we added them to the map. Each name sits between a checkbox and an opacity slider. To make Layer 2 visible, click the checkbox (Fig. 2.3, area *A*). To make Layer 3 visible, move the opacity slider to the right (Fig. 2.3, area *B*).

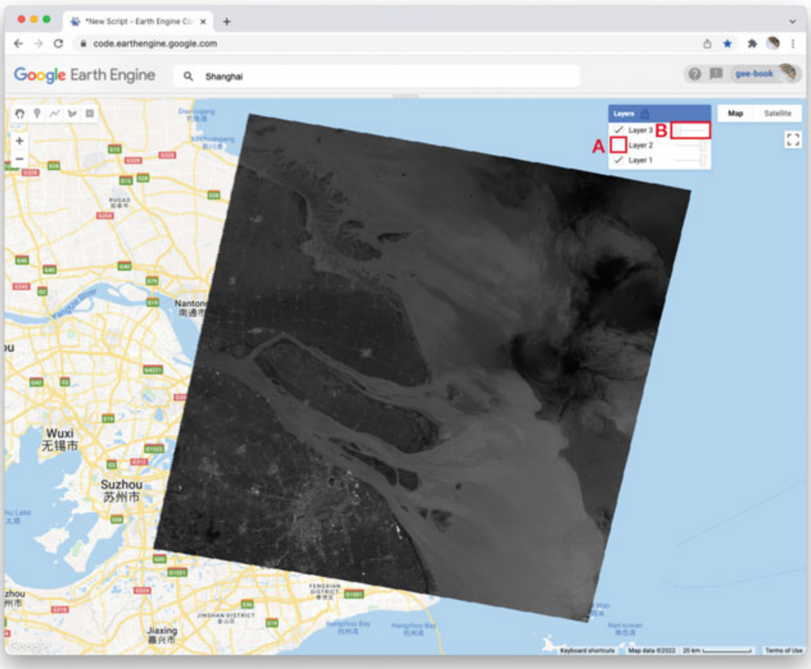


Fig. 2.3 Three bands from the Landsat image, drawn as three different grayscale layers

By manipulating these controls, you should notice that these layers are displayed as a *stack*, meaning one on top of the other. For example, set the opacity for each layer to be 1 by pushing the opacity sliders all the way to the right. Then, make sure that each box is checked next to each layer so that all the layers are shown. Now you can identify which layer is on top of the stack by checking and unchecking each layer. If a layer is on top of another, unchecking the top layer will reveal the layer underneath. If a layer is under another layer in the stack, then unchecking the bottom layer will not alter the display (because the top layer will remain visible). If you try this on our stack, you should see that the list order reflects the stack order, meaning that the layer at the top of the layer list appears on the top of the stack. Now, compare the order of the layers in the list to the sequence of operations in your script. What layer did your script add first and where does this appear in the layering order on the map?

Code Checkpoint F11a. The book's repository contains a script that shows what your code should look like at this point.

2.1.3 Section 3: True-Color Composites

Using the controls in the Layers manager, explore these layers and examine how the pixel values in each band differ. Does Layer 2 (displaying pixel values from the “SR_B2” band) appear generally brighter than Layer 1 (the “SR_B1” band)? Compared with Layer 2, do the ocean waters in Layer 3 (the “SR_B3” band) appear a little darker in the north, but a little lighter in the south?

We can use color to compare these visual differences in the pixel values of each band layer all at once as an *RGB composite***.** This method uses the three primary colors (red, green, and blue) to display each pixel’s values across three bands.

To try this, add this code and run it.

```
Map.addLayer(
    first_image,
    {
        bands: ['SR_B3', 'SR_B2', 'SR_B1'],
        min: 8000,
        max: 17000
    },
    'Natural Color');
```

The result (Fig. 2.4) looks like the world we see and is referred to as a *natural-color composite***,** because it naturally pairs the spectral ranges of the image bands to display colors. Also called a *true-color composite*, this image shows the red spectral band with shades of red, the green band with shades of green, and the blue band with shades of blue. We specified the pairing simply through the order of the bands in the list: *B*3, *B*2, *B*1. Because bands 3, 2, and 1 of Landsat 5 correspond to the real-world colors of red, green, and blue, the image resembles the world that we would see outside the window of a plane or with a low-flying drone.

2.1.4 Section 4: False-Color Composites

As you saw when you printed the band list (Fig. 2.1), a Landsat image contains many more bands than just the three true-color bands. We can make RGB composites to show combinations of any of the bands—even those outside what the human eye can see. For example, band 4 represents the near-infrared band, just outside the range of human vision. Because of its value in distinguishing environmental conditions, this band was included on even the earliest 1970s Landsats. It has different values in coniferous and deciduous forests, for example, and can indicate crop health. To see an example of this, add this code to your script and run it.

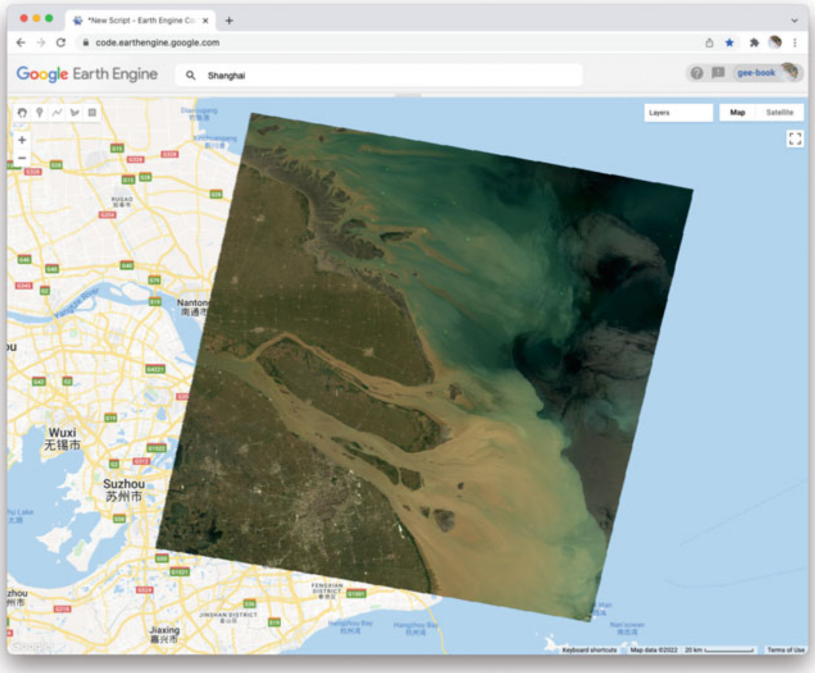


Fig. 2.4 True-color composite

```
Map.addLayer(
    first_image,
    {
        bands: ['SR_B4', 'SR_B3', 'SR_B2'],
        min: 8000,
        max: 17000
    },
    'False Color');
```

In this *false-color composite* (Fig. 2.5), the display colors no longer pair naturally with the bands. This particular example, which is more precisely referred to as a *color-infrared composite*, is a scene that we could not observe with our eyes, but that you can learn to read and interpret. Its meaning can be deciphered logically by thinking through what is passed to the red, green, and blue color channels.

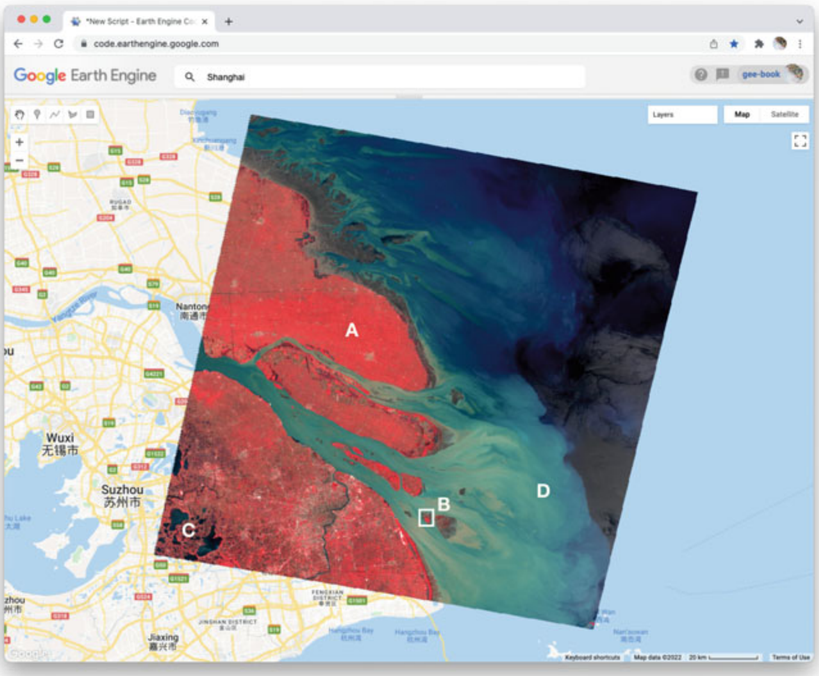
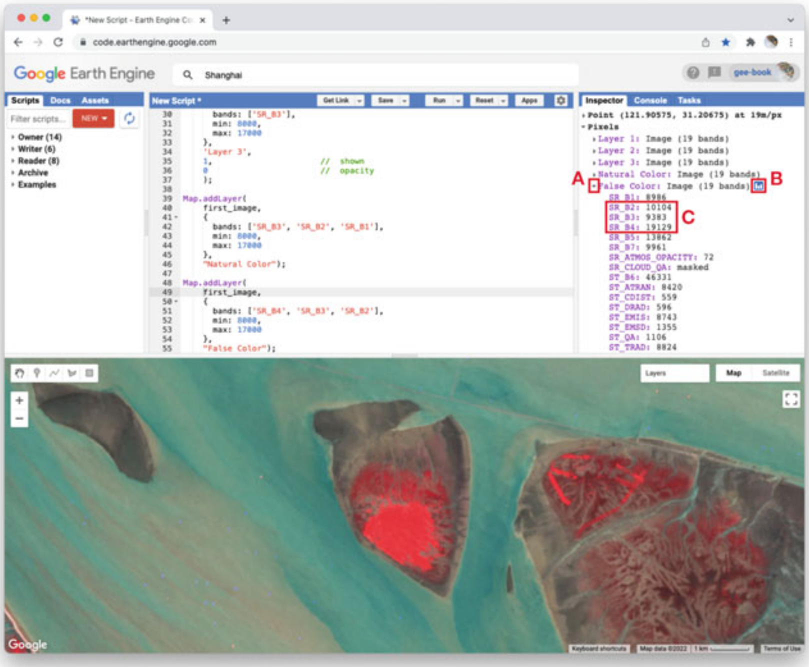


Fig. 2.5 Color-infrared image (a false-color composite)

Notice how the land on the northern peninsula appears bright red (Fig. 2.5, area *A*). This is because for that area, the pixel value of the first band (which is drawing the near-infrared brightness) is much higher relative to the pixel value of the other two bands. You can check this by using the **Inspector** tool. Try zooming into a part of the image with a red patch (Fig. 2.5, area *B*) and clicking on a pixel that appears red. Then, expand the “False Color” layer in the **Inspector** panel (Fig. 2.6, area *A*), click the blue icon next to the layer name (Fig. 2.6, area *B*), and read the pixel value for the three bands of the composite (Fig. 2.6, area *C*). The pixel value for B4 should be much greater than for *B*3 or *B*2.

In the bottom left corner of the image (Fig. 2.5, area *C*), rivers and lakes appear very dark, which means that the pixel value in all three bands is low. However, sediment plumes fanning from the river into the sea appear with blue and cyan tints (Fig. 2.5, area *D*). If they look like primary blue, then the pixel value for the second band (*B*3) is likely higher than the first (*B*4) and third (*B*2) bands. If they appear more like cyan, an additive color, it means that the pixel values of the second and third bands are both greater than the first.



```
        bands: ['SR_B3'],
        min: 8000,
        max: 17000
    },
    'Layer 3',
    1,                       //  shown
    0                        //  opacity
    );

Map.addLayer(
    first_image,
    {
      bands: ['SR_B3', 'SR_B2', 'SR_B1'],
      min: 8000,
      max: 17000
    },
    "Natural Color");

Map.addLayer(
    first_image,
    {
      bands: ['SR_B4', 'SR_B3', 'SR_B2'],
      min: 8000,
      max: 17000
    },
    "False Color");
```

```
▸Point (121.90575, 31.20675) at 19m/px
▾Pixels
  ▸Layer 1: Image (19 bands)
  ▸Layer 2: Image (19 bands)
  ▸Layer 3: Image (19 bands)
  ▸Natural Color: Image (19 bands)
  ▾False Color: Image (19 bands)
      SR_B1: 8986
      SR_B2: 10104
      SR_B3: 9383
      SR_B4: 19129
      SR_B5: 13862
      SR_B7: 9961
      SR_ATMOS_OPACITY: 72
      SR_CLOUD_QA: masked
      ST_B6: 46331
      ST_ATRAN: 8420
      ST_CDIST: 559
      ST_DRAD: 596
      ST_EMIS: 8743
      ST_EMSD: 1355
      ST_QA: 1106
      ST_TRAD: 8824
```

Fig. 2.6 Values of *B*4, *B*3, *B*2 bands for a pixel that appears bright red

In total, the false-color composite provides more contrast than the true-color image for understanding differences across the scene. This suggests that other bands might contain more useful information as well. We saw earlier that our satellite image consisted of 19 bands. Six of these represent different portions of the electromagnetic spectrum, including three beyond the visible spectrum, that can be used to make different false-color composites. Use the code below to explore a composite that shows shortwave infrared, near infrared, and visible green (Fig. 2.7).

```
Map.addLayer(
    first_image,
    {
        bands: ['SR_B5', 'SR_B4', 'SR_B2'],
        min: 8000,
        max: 17000
    },
    'Short wave false color');
```

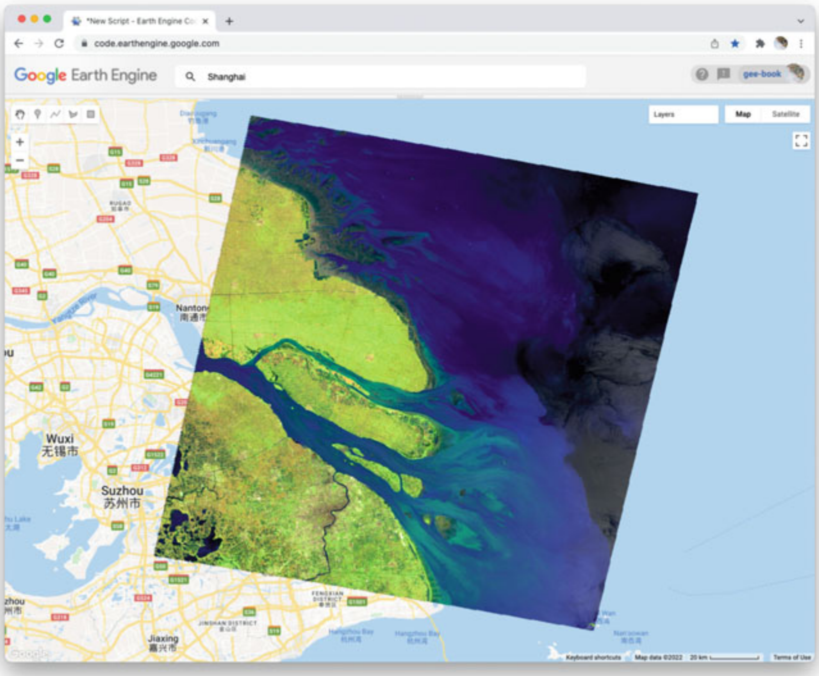


Fig. 2.7 Shortwave infrared false-color composite

To compare the two false-color composites, zoom into the area shown in the two pictures of Fig. 2.8. You should notice that bright red locations in the left composite appear bright green in the right composite. Why do you think that is? Does the image on the right show new distinctions not seen in the image on the left? If so, what do you think they are?

Code Checkpoint F11b. The book's repository contains a script that shows what your code should look like at this point.

2.1.5 Section 5: Additive Color System

Thus far, we have used RGB composites to make a true-color image, in which the colors on the screen match the colors in our everyday world. We also used the same principles to draw two false-color combinations of optical bands collected by

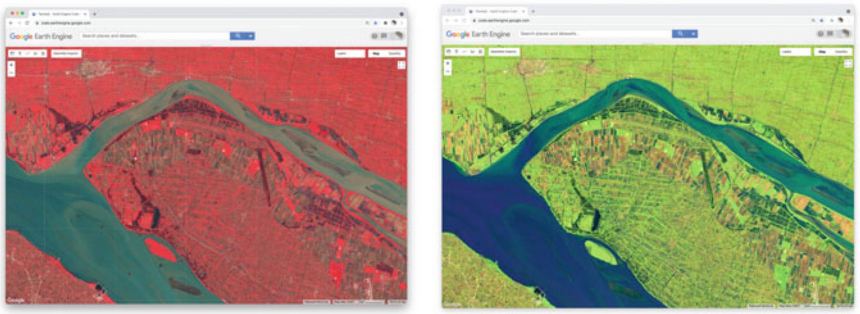


Fig. 2.8 Near-infrared versus shortwave infrared false-color composites

the satellite. To be able to read and interpret information from composite images generally, it is useful to understand the *additive color system*. Views of data in Earth Engine, and indeed everything drawn on a computer screen, use three channels for display (red, green, and blue). The order of the bands in a composite layer determines the *color channel* used to display the DN of pixels. When the DN is higher in one band relative to the other two bands, the pixel will appear tinted with the color channel used to display that band. For example, when the first band is higher relative to the other two bands, the pixel will appear reddish. The intensity of the pixel color will express the magnitude of difference between the DN quantities.

The way that primary colors combine to make new colors in an additive color system can be confusing at first, especially if you learned how to mix colors by painting or printing. When using an additive color system, red combined with green makes yellow, green combined with blue makes cyan, and red combined with blue makes magenta (Fig. 2.9). Combining all three primary colors makes white. The absence of all primary colors makes black. For RGB composites, this means that if the pixel value of two bands is greater than that of the third band, the pixel color will appear tinted as a combined color. For example, when the pixel value of the first and second bands of a composite is higher than that of the third band, the pixel will appear yellowish.

2.1.6 Section 6: Attributes of Locations

So far, we have explored bands as a method for storing data about slices of the electromagnetic spectrum that can be measured by satellites. Now, we will work toward applying the additive color system to bands that store non-optical and more abstract *attributes* of geographic locations.

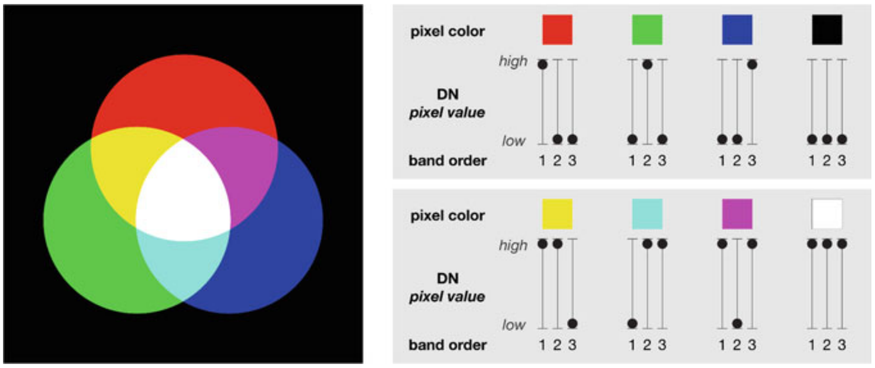


Fig. 2.9 Additive color system

To begin, add this code to your script and run it.

```
var lights93 = ee.Image('NOAA/DMSP-OLS/NIGHTTIME_LIGHTS/F101993');
print('Nighttime lights', lights93);

Map.addLayer(
    lights93,
    {
        bands: ['stable_lights'],
        min: 0,
        max: 63
    },
    'Lights');
```

This code loads an image of global nighttime lights and adds a new layer to the map. Please look at the metadata that we printed to the Console panel. You should see that the image consists of four bands. The code selects the “stable_lights” band to display as a layer to the map. The range of values for display (0–63) represents the minimum and maximum pixel values in this image. As mentioned earlier, you can find this range in the Earth Engine Data Catalog or with other Earth Engine methods. These will be described in more detail in the next few chapters.

The global nighttime lights’ image represents the average brightness of nighttime lights at each pixel for a calendar year. For those of us who have sat by a window in an airplane as it descends to a destination at night, the scene may look vaguely familiar. But, the image is very much an abstraction. It provides us a view of the planet that we would never be able to see from an airplane or even from space. Night blankets the entire planet in darkness. There are no clouds. In the

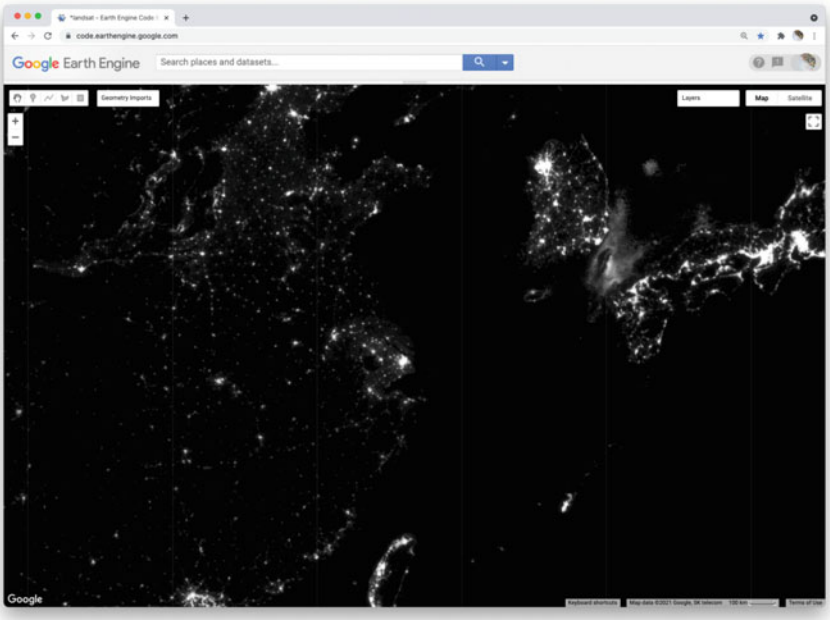


Fig. 2.10 Stable nighttime lights in 1993

“stable lights” band, there are no ephemeral sources of light. Lightning strikes, wildfires, and other transient lights have been removed. It is a layer that aims to answer one question about our planet at one point in time: In 1993, how bright were Earth’s stable, artificial sources of light?

With the zoom controls on the map, you can zoom out to see the bright spot of Shanghai, the large blob of Seoul to the north and east, the darkness of North Korea except for the small dot of Pyongyang, and the dense strips of lights of Japan and the west coast of Taiwan (Fig. 2.10).

2.1.7 Section 7: Abstract RGB Composites

Now, we can use the additive color system to make an RGB composite that compares stable nighttime lights at three different slices of time. Add the code below to your script and run it.

```
var lights03 = ee.Image('NOAA/DMSP-OLS/NIGHTTIME_LIGHTS/F152003')
    .select('stable_lights').rename('2003');

var lights13 = ee.Image('NOAA/DMSP-OLS/NIGHTTIME_LIGHTS/F182013')
    .select('stable_lights').rename('2013');

var changeImage = lights13.addBands(lights03)
    .addBands(lights93.select('stable_lights').rename('1993'));

print('change image', changeImage);

Map.addLayer(
    changeImage,
    {
        min: 0,
        max: 63
    },
    'Change composite');
```

This code does a few things. First, it creates two new images, each representing a different slice of time. For both, we use the `select` method to select a band ("stable_lights") and the `rename` method to change the band name to indicate the year it represents.

Next, the code uses the `addBands` method to create a new, three-band image that we name "changeImage". It does this by taking one image (lights13) as the first band, using another image (lights03) as the second band, and the lights93 image seen earlier as the third band. The third band is given the name "1993" as it is placed into the image.

Finally, the code prints metadata to the **Console** and adds the layer to the map as an RGB composite using `Map.addLayer`. If you look at the printed metadata, you should see under the label "change image" that our image is composed of three bands, with each band named after a year. You should also notice the order of the bands in the image: 2013, 2003, 1993. This order determines the color channels used to represent each slice of time in the composite: 2013 as red, 2003 as green, and 1993 as blue (Fig. 2.11).

We can now read the colors displayed on the layer to interpret different kinds of changes in nighttime lights across the planet over two decades. Pixels that appear white have high brightness in all three years. You can use the **Inspector** panel to confirm this. Click on the **Inspector** panel to change the cursor to a crosshair and then click on a pixel that appears white. Look under the *Pixel* category of the **Inspector** panel for the "Change composite" layer. The pixel value for each band should be high (at or near 63).

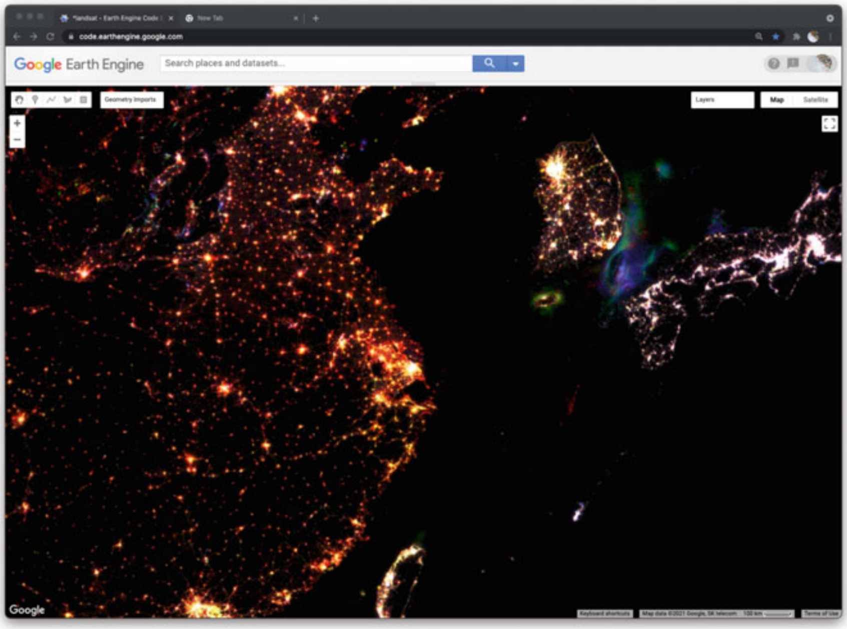


Fig. 2.11 RGB composite of stable nighttime lights (2013, 2003, 1993)

Many clumps of white pixels represent urban cores. If you zoom into Shanghai, you will notice that the periphery of the white-colored core appears yellowish and the terminal edges appear reddish. Yellow represents locations that were bright in 2013 and 2003 but dark in 1993. Red represents locations that appear bright in 2013 but dark in 2003 and 1993. If you zoom out, you will see that this gradient of white core to yellow periphery to red edge occurs around many cities across the planet and shows the global pattern of urban sprawl over the 20-year period.

When you zoom out from Shanghai, you will likely notice that each map layer redraws every time you change the zoom level. In order to explore the change composite layer more efficiently, use the **Layer** manager panel to not show (uncheck) all of the layers except for “Change composite.” Now, the map will respond faster when you zoom and pan because it will only refresh the single-displayed shown layer.

In addition to urban change, the layer also shows changes in resource extraction activities that produce bright lights. Often, these activities produce lights that are stable over the span of a year (and therefore included in the “stable lights” band), but are not sustained over the span of a decade or more. For example, in the Korea Strait (between South Korea and Japan), you can see geographic shifts of fishing fleets that use bright halogen lights to attract squid and other sea creatures toward the water surface and into their nets. Bluish pixels were likely fished more heavily in 1993 and became used less frequently by 2003, while greenish pixels were likely fished more heavily in 2003 and less frequently by 2013 (Fig. 2.11).

Similarly, fossil fuel extraction produces nighttime lights through gas flaring. If you pan to North America (Fig. 2.12), red blobs in Alberta and North Dakota and a red swath in southeastern Texas all represent places where oil and gas extraction was absent in 1993 and 2003 but booming by 2013. Pan over to the Persian Gulf and you will see changes that look like holiday lights with dots of white, red, green, and blue appearing near each other; these distinguish stable and shifting locations of oil production. Blue lights in Syria near the border with Iraq signify the abandonment of oil fields after 1993 (Fig. 2.13). Pan further north and you will see another "holiday lights" display from oil and gas extraction around Surgut, Russia. In many of these places, you can check for oil and gas infrastructure by zooming into a colored spot, making the lights layer not visible, and selecting the **Satellite** base layer (upper right).

As you explore this image, remember to check your interpretations with the **Inspector** panel by clicking on a pixel and reading the pixel value for each band. Refer back to the additive color figure to remember how the color system works. If you practice this, you should be able to read any RGB composite by knowing how colors relate to the relative pixel value of each band. This will empower you

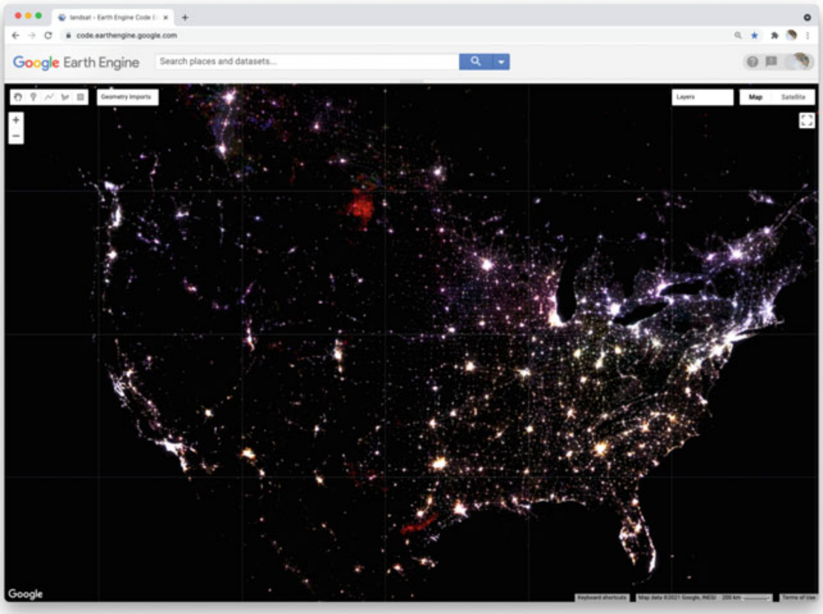


Fig. 2.12 Large red blobs in North Dakota and Texas from fossil fuel extraction in specific years

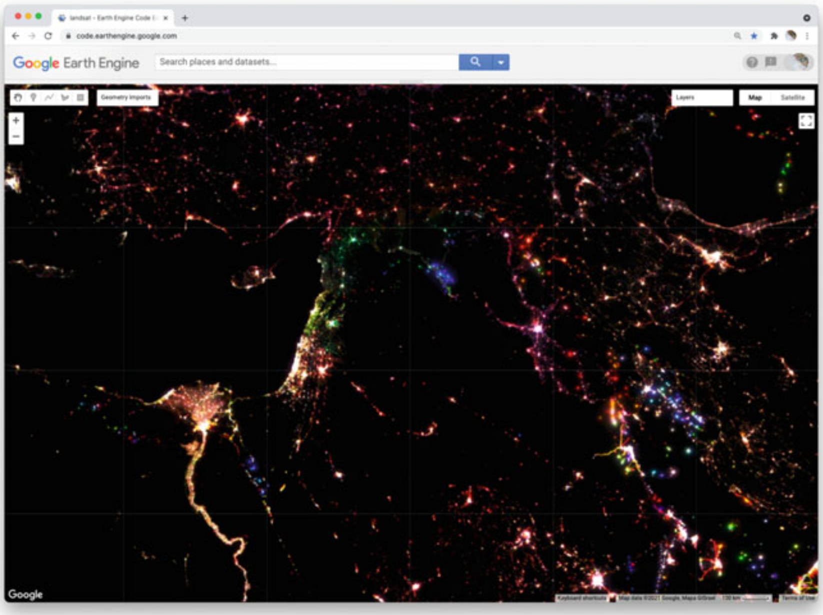


Fig. 2.13 Nighttime light changes in the Middle East

to employ false-color composites as a flexible and powerful method to explore and interpret geographic patterns and changes on Earth's surface.

Code Checkpoint F11c. The book's repository contains a script that shows what your code should look like at this point.

2.2 Synthesis

Assignment 1. Compare and contrast the changes in nighttime lights around Damascus, Syria versus Amman, Jordan. How are the colors for the two cities similar and different? How do you interpret the differences?

Assignment 2. Look at the changes in nighttime lights in the region of Port Harcourt, Nigeria. What kinds of changes do you think these colors signify? What clues in the satellite basemap can you see to confirm your interpretation?

Assignment 3. In the nighttime lights' change composite, we did not specify the three bands to use for our RGB composite. How do you think Earth Engine chose the three bands to display? How do you think Earth Engine determined which band should be shown with the red, green, and blue channels?

Assignment 4. Create a new script to make three composites (natural-color, near-infrared false-color, and shortwave infrared false-color composites) for this image:

```
'LANDSAT/LT05/C02/T1_L2/LT05_022039_20050907'
```

What environmental event do you think the images show? Compare and contrast the natural and false-color composites. What do the false-color composites help you see that is more difficult to decipher in the natural-color composite?

Assignment 5. Create a new script and run this code to view this image over Shanghai:

```
var image =
ee.Image('LANDSAT/LT05/C02/T1_L2/LT05_118038_20000606');

Map.addLayer(
    image,
    {
        bands: ['SR_B1'],
        min: 8000,
        max: 17000
    },
    'Layer 1'
);

Map.addLayer(
    image.select('SR_B1'),
    {
        min: 8000,
        max: 17000
    },
    'Layer 2'
);
```

Inspect Layer 1 and Layer 2 with the **Inspector** panel**.** Describe how the two layers differ and explain why they differ.

2.3 Conclusion

In this chapter, we looked at how an image is composed of one or more bands, where each band stores data about geographic locations as pixel values. We explored different ways of visualizing these pixel values as map layers, including a grayscale display of single bands and RGB composites of three bands. We created natural and false-color composites that use additive color to display information in visible and non-visible portions of the spectrum. We examined additive color as a general system for visualizing pixel values across multiple bands. We then explored how bands and RGB composites can be used to represent more abstract phenomena, including different kinds of change over time.

Survey of Raster Datasets

3

Andréa Puzzi Nicolau, Karen Dyson, David Saah, and Nicholas Clinton

Overview

The purpose of this chapter is to introduce you to the many types of collections of images available in Google Earth Engine. These include sets of individual satellite images, pre-made composites (which merge multiple individual satellite images into one composite image), classified land use and land cover (LULC) maps, weather data, and other types of datasets. If you are new to JavaScript or programming, work through Chaps. 1 and 2 first.

A. P. Nicolau (✉) · K. Dyson · D. Saah
Spatial Informatics Group, Pleasanton, CA, USA
e-mail: apnicolau@sig-gis.com

K. Dyson
e-mail: kdyson@sig-gis.com

D. Saah
e-mail: dssaah@usfca.edu

A. P. Nicolau · K. Dyson
SERVIR-Amazonia, Cali, Colombia

K. Dyson
Dendrolytics, Seattle, WA, USA

D. Saah
University of San Francisco, San Francisco, CA, USA

N. Clinton
Google LLC, Mountain View, CA, USA
e-mail: nclinton@google.com

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_3

Learning Outcomes

- Accessing and viewing sets of images in Earth Engine.
- Extracting single scenes from collections of images.
- Applying visualization parameters in Earth Engine to visualize an image.

Assumes you know how to

- Sign up for an Earth Engine account, open the Code Editor, and save your script. (Chap. 1)
- Locate the Earth Engine **Inspector** and **Console** tabs and understand their purposes (Chap. 1).
- Use the **Inspector** tab to assess pixel values (Chap. 2).

3.1 Introduction to Theory

The previous chapter introduced you to images, one of the core building blocks of remotely sensed imagery in Earth Engine. In this chapter, we will expand on this concept of images by introducing *image collections*. Image collections in Earth Engine organize many different images into one larger data storage structure. Image collections include information about the location, date collected, and other properties of each image, allowing you to sift through the `ImageCollection` for the exact image characteristics needed for your analysis.

3.2 Practicum

There are many different types of image collections available in Earth Engine. These include collections of individual satellite images, pre-made composites that combine multiple images into one blended image, classified LULC maps, weather data, and other non-optical datasets. Each one of these is useful for different types of analyses. For example, one recent study examined the drivers of wildfires in Australia (Sulova and Jokar 2021). The research team used the European Center for Medium-Range Weather Forecast Reanalysis (ERA5) dataset produced by the European Center for Medium-Range Weather Forecasts (ECMWF) and is freely available in Earth Engine. We will look at this dataset later in the chapter.

3.2.1 Section 1: Image Collections: An Organized Set of Images

If you have no already done so, you add the book's code repository to the Code Editor by entering https://code.earthengine.google.com/?accept_repo=projects/gee-edu/book 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 http://bit.ly/EEFA-repo-help for help.

You saw some of the basic ways to interact with an individual `ee.Image` in Chap. 2. However, depending on how long a remote sensing platform has been in operation, there may be thousands or millions of images collected of Earth. In Earth Engine, these are organized into an `ImageCollection`, a specialized data type that has specific operations available in the Earth Engine API. Like individual images, they can be viewed with `Map.addLayer`.

You will learn to work with image collections in complex ways later in the book, particularly in Part IV. For now, we will show you how to view and work with their most basic attributes and use these skills to view some of the major types of image collections in Earth Engine. This chapter will give a brief tour of the Earth Engine Data Catalog, which contains decades of satellite imagery and much more. We will view some of the different types of datasets in the following sections, including climate and weather data, digital elevation models and other terrain data, land cover, cropland, satellite imagery, and others.

View an Image Collection

The Landsat program from NASA and the United States Geological Survey (USGS) has launched a sequence of Earth observation satellites, named Landsat 1, 2, etc. Landsats have been returning images since 1972, making that collection of images the longest continuous satellite-based observation of the Earth's surface. We will now view images and basic information about one of the image collections that is still growing: collections of scenes taken by the Operational Land Imager aboard Landsat 8, which was launched in 2013. Copy and paste the following code into the center panel and click Run. While the enormous image catalog is accessed, it could take a couple of minutes to see the result in the Map area. You may note individual "scenes" being drawn, which equate to the way that the Landsat program partitions Earth into "paths" and "rows". If it takes more than a couple of minutes to see the images, try zooming in to a specific area to speed up the process.

```
/////
// View an Image Collection
/////

// Import the Landsat 8 Raw Collection.
var landsat8 = ee.ImageCollection('LANDSAT/LC08/C02/T1');

// Print the size of the Landsat 8 dataset.
print('The size of the Landsat 8 image collection is:',
landsat8
.size());

// Try to print the image collection.
// WARNING! Running the print code immediately below
produces an error because
// the Console can not print more than 5000 elements.
print(landsat8);

// Add the Landsat 8 dataset to the map as a mosaic. The
collection is
// already chronologically sorted, so the most recent pixel
is displayed.
Map.addLayer(landsat8,
    {
        bands: ['B4', 'B3', 'B2'],
        min: 5000,
        max: 15000
    },
    'Landsat 8 Image Collection');
```

First, let us examine the map output (Fig. 3.1).

Notice the high amount of cloud cover and the "layered" look. Zoom out if needed. This is because Earth Engine is drawing each of the images that make up the `ImageCollection` one on top of the other. The striped look is the

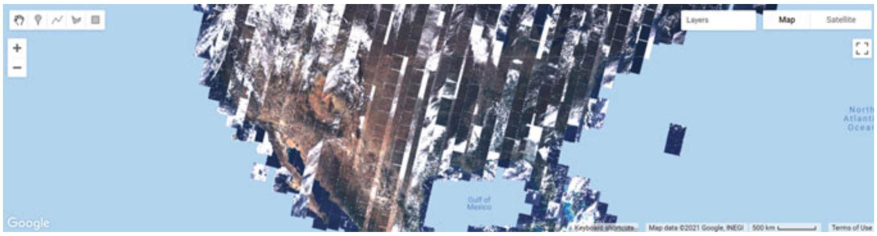
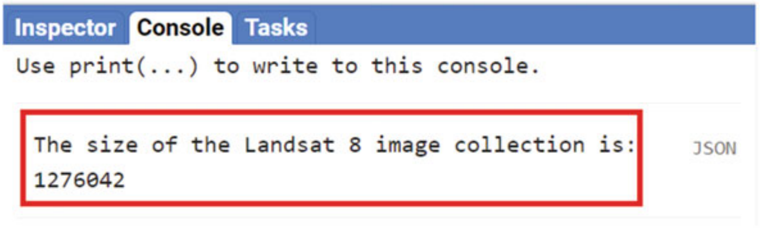
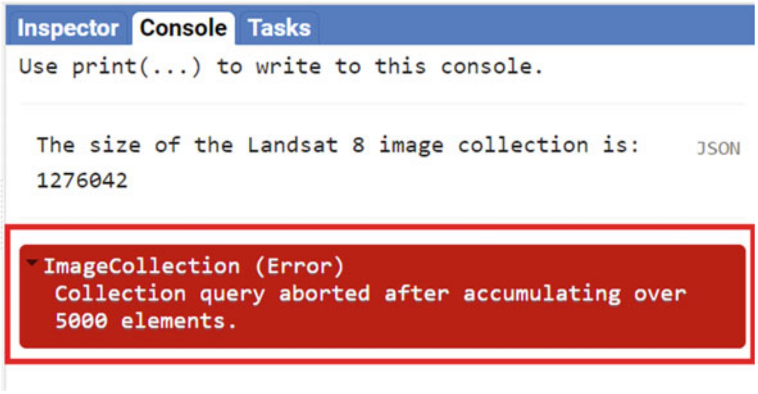


Fig. 3.1 USGS Landsat 8 collection 2 Tier 1 Raw Scenes collection



```
Use print(...) to write to this console.

The size of the Landsat 8 image collection is:
1276042
```

Fig. 3.2 Size of the entire Landsat 8 collection. Note that this number is constantly growing



```
Use print(...) to write to this console.

The size of the Landsat 8 image collection is:
1276042

ImageCollection (Error)
 Collection query aborted after accumulating over
 5000 elements.
```

Fig. 3.3 Error encountered when trying to print the names and information to the screen for too many elements

result of how the satellite collects imagery. The overlaps between images and the individual nature of the images mean that these are not quite ready for analysis; we will address this issue in future chapters.

Now examine the printed size on the **Console**. It will indicate that there are more than a million images in the dataset (Fig. 3.2). If you return to this lab in the future, the number will be even larger, since this active collection is continually growing as the satellite gathers more imagery. For the same reason, Fig. 3.1 might look slightly different on your map because of this.

Note that printing the `ImageCollection` returned an error message (Fig. 3.3), because calling `print` on an `ImageCollection` will write the name of every image in the collection to the **Console**. This is the result of an intentional safeguard within Earth Engine. We do not want to see a million image names printed to the **Console**!

Code Checkpoint F12a. The book's repository contains a script that shows what your code should look like at this point.

Edit your code to comment out the last two code commands you have written. This will remove the call to `Map.addLayer` that drew every image, and will remove the `print` statement that demanded more than 5000 elements. This will speed up your code in subsequent sections. As described in Chap. 1, placing two forward slashes (`//`) at the beginning of a line will make it into a comment, and any commands on that line will not be executed.

Filtering Image Collections

The `ImageCollection` data type in Earth Engine has multiple approaches to filtering, which helps to pinpoint the exact images you want to view or analyze from the larger collection.

Filter by Date

One of the filters is `filterDate`, which allows us to narrow down the date range of the `ImageCollection`. Copy the following code to the center panel (paste it after the previous code you had):

```
// Filter an Image Collection
/////

// Filter the collection by date.
var landsatWinter = landsat8.filterDate('2020-12-01',
'2021-03-01');

Map.addLayer(landsatWinter,
    {
        bands: ['B4', 'B3', 'B2'],
        min: 5000,
        max: 15000
    },
    'Winter Landsat 8');

print('The size of the Winter Landsat 8 image collection
is:',
    landsatWinter.size());
```

Examine the mapped `landsatWinter` (Fig. 3.4). As described in Chap. 2, the 5000 and the 15000 values in the visualization parameters of the `Map.addLayer` function of the code above refer to the minimum and maximum of the range of display values.

Now, look at the size of the winter Landsat 8 collection. The number is significantly lower than the number of images in the entire collection. This is the result of filtering the dates to three months in the winter of 2020–2021.

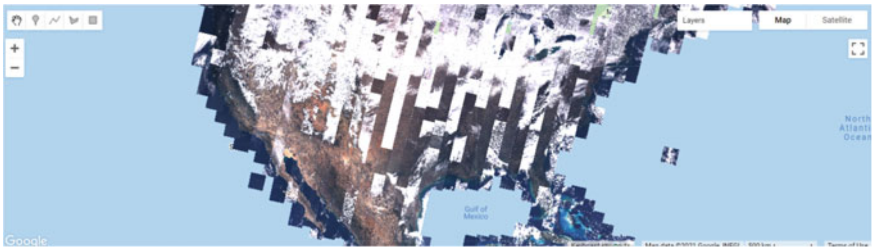


Fig. 3.4 Landsat 8 winter collection

Filter by Location

A second frequently used filtering tool is `filterBounds`. This filter is based on a location—for example, a point, polygon, or other geometry. Copy and paste the code below to filter and add to the map the winter images from the Landsat 8 Image Collection to a point in Minneapolis, Minnesota, USA. Note below the `Map.addLayer` function to add the `pointMN` to the map with an empty dictionary `{}` for the `visParams` argument. This only means that we are not specifying visualization parameters for this element, and it is being added to the map with the default parameters.

```
// Create an Earth Engine Point object.
var pointMN = ee.Geometry.Point([-93.79, 45.05]);

// Filter the collection by location using the point.
var landsatMN = landsatWinter.filterBounds(pointMN);
Map.addLayer(landsatMN,
    {
        bands: ['B4', 'B3', 'B2'],
        min: 5000,
        max: 15000
    },
    'MN Landsat 8');

// Add the point to the map to see where it is.
Map.addLayer(pointMN, {}, 'Point MN');

print('The size of the Minneapolis Winter Landsat 8 image
collection is: ',
    landsatMN.size());
```

If we uncheck the Winter Landsat 8 layer under **Layers**, we can see that only images that intersect our point have been selected (Fig. 3.5). Zoom in or out as needed. Note the printed size of the Minneapolis winter collection—we only have seven images.

Selecting the First Image

The final operation we will explore is the `first` function. This selects the first image in an `ImageCollection`. This allows us to place a single image on the screen for inspection. Copy and paste the code below to select and view the first image of the Minneapolis Winter Landsat 8 Image Collection. In this case, because the images are stored in time order in the `ImageCollection`, it will select the earliest image in the set.

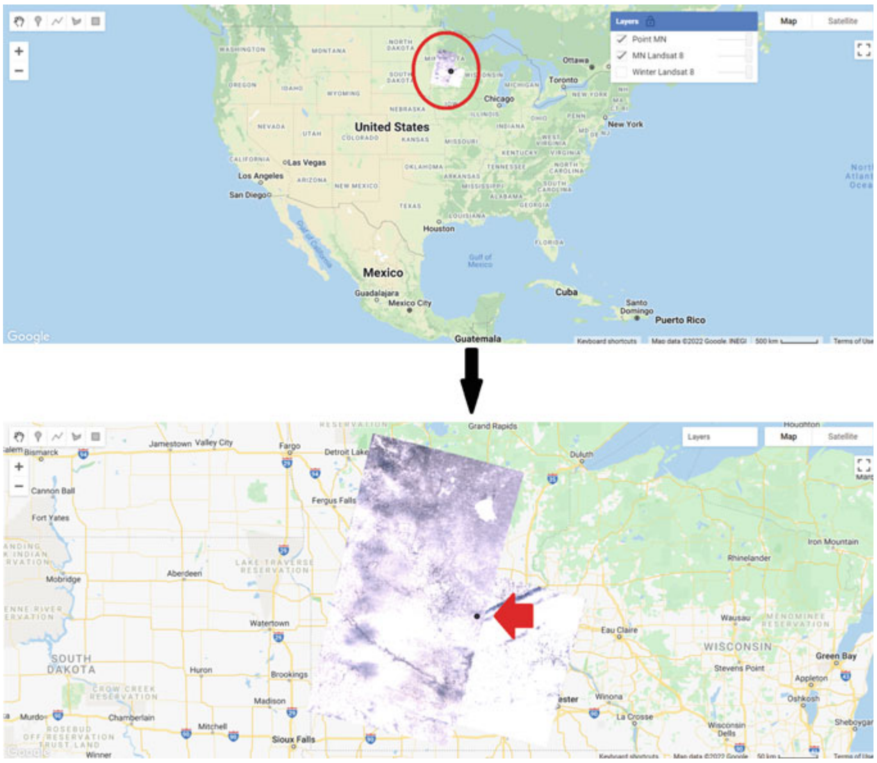


Fig. 3.5 Minneapolis winter collection filtered by bounds. The first still represents the map without zoom applied. The collection is shown inside the red circle. The second still represents the map after zoom was applied to the region. The red arrow indicates the point (in black) used to filter by bounds

```
// Select the first image in the filtered collection.
var landsatFirst = landsatMN.first();

// Display the first image in the filtered collection.
Map.centerObject(landsatFirst, 7);
Map.addLayer(landsatFirst,
    {
        bands: ['B4', 'B3', 'B2'],
        min: 5000,
        max: 15000
    },
    'First Landsat 8');
```

The `first` command takes our stack of location-filtered images and selects the first image. When the layer is added to the Map area, you can see that only one image is returned—remember to uncheck the other layers to be able to visualize the full image (Fig. 3.6). We used the `Map.centerObject` to center the map on the `landsatFirst` image with a zoom level of 7 (zoom levels go from 0 to 24).

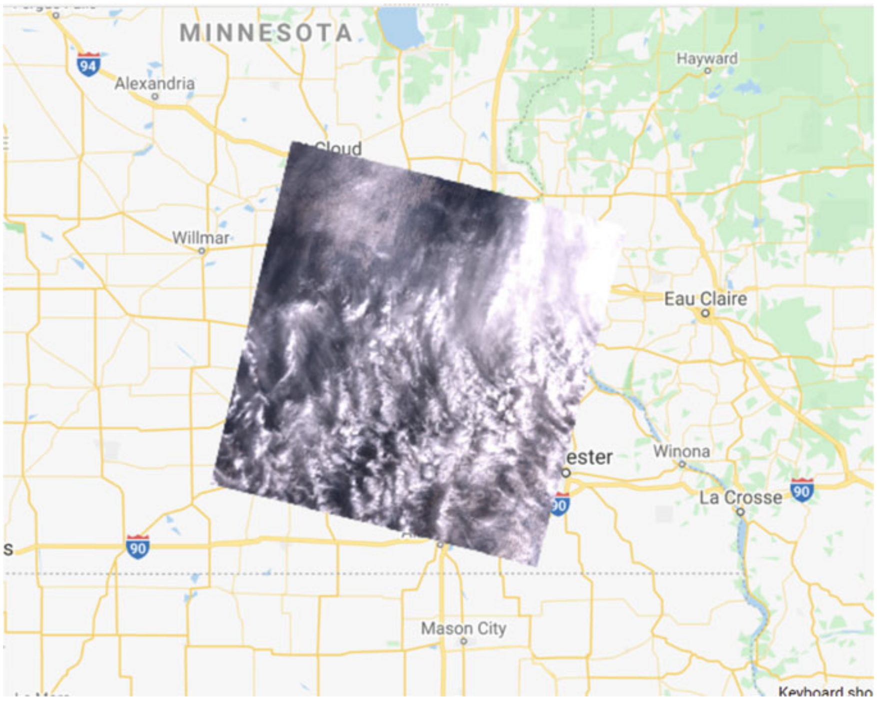


Fig. 3.6 First Landsat image from the filtered set

Code Checkpoint F12b. The book's repository contains a script that shows what your code should look like at this point.

Now that we have the tools to examine different image collections, we will explore other datasets. Save your script for your own future use, as outlined in Chap. 1. Then, refresh the Code Editor to begin with a new script for the next section.

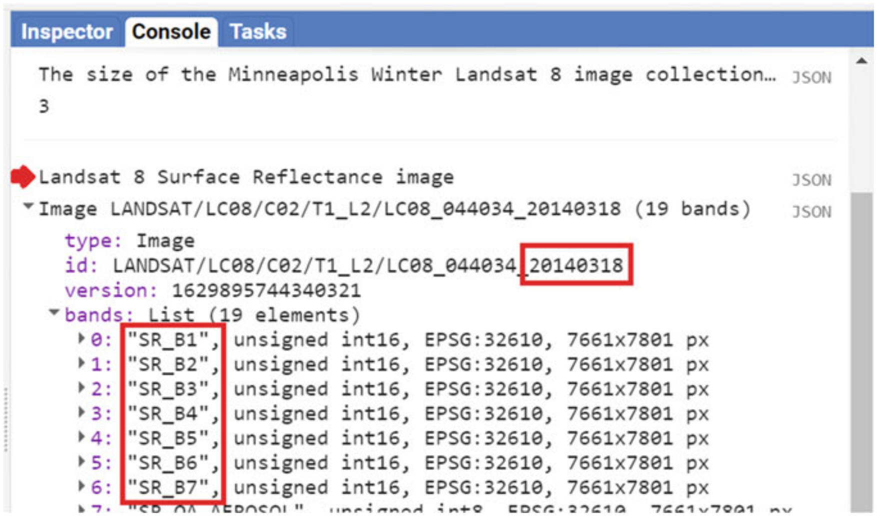
3.2.2 Section 2: Collections of Single Images

When learning about image collections in the previous section, you worked with the Landsat 8 raw image dataset. These raw images have some important corrections already done for you. However, the raw images are only one of several image collections produced for Landsat 8. The remote sensing community has developed additional imagery corrections that help increase the accuracy and consistency of analyses. The results of each of these different imagery processing paths are stored in a distinct `ImageCollection` in Earth Engine.

Among the most prominent of these is the `ImageCollection` meant to minimize the effect of the atmosphere between Earth's surface and the satellite. The view from satellites is made imprecise by the need for light rays to pass through the atmosphere, even on the clearest day. There are two important ways the atmosphere obscures a satellite's view: by affecting the amount of sunlight that strikes the Earth and by altering electromagnetic energy on its trip from its reflection at Earth's surface to the satellite's receptors.

Unraveling those effects is called atmospheric correction, a highly complex process whose details are beyond the scope of this book. Thankfully, in addition to the raw images from the satellite, each image for Landsat and certain other sensors is automatically treated with the most up-to-date atmospheric correction algorithms, producing a product referred to as a "surface reflectance" `ImageCollection`. The surface reflectance estimates the ratio of upward radiance at the Earth's surface to downward radiance at the Earth's surface, imitating what the sensor would have seen if it were hovering a few feet above the ground.

Let us examine one of these datasets meant to minimize the effects of the atmosphere between Earth's surface and the satellite. Copy and paste the code below to import and filter the Landsat 8 surface reflectance data (`landsat8SR`) by date and to a point over San Francisco, California, USA (`pointSF`). We use the `first` function to select the first image—a single image from March 18, 2014. By printing the `landsat8SRimage` image on the **Console**, and accessing its metadata (see Chap. 2), we see that the band names differ from those in the raw image (Fig. 3.7). Here, they have the form "`SR_B*`" as in "Surface Reflectance Band *", where `*` is the band number. We can also check the date of the image by looking at the image "id" (Fig. 3.7). This has the value "20140318", a string indicating that the image was from March 18, 2014.



```
The size of the Minneapolis Winter Landsat 8 image collection…
3

Landsat 8 Surface Reflectance image
▾Image LANDSAT/LC08/C02/T1_L2/LC08_044034_20140318 (19 bands)
   type: Image
   id: LANDSAT/LC08/C02/T1_L2/LC08_044034_20140318
   version: 1629895744340321
  ▾bands: List (19 elements)
    ▸0: "SR_B1", unsigned int16, EPSG:32610, 7661x7801 px
    ▸1: "SR_B2", unsigned int16, EPSG:32610, 7661x7801 px
    ▸2: "SR_B3", unsigned int16, EPSG:32610, 7661x7801 px
    ▸3: "SR_B4", unsigned int16, EPSG:32610, 7661x7801 px
    ▸4: "SR_B5", unsigned int16, EPSG:32610, 7661x7801 px
    ▸5: "SR_B6", unsigned int16, EPSG:32610, 7661x7801 px
    ▸6: "SR_B7", unsigned int16, EPSG:32610, 7661x7801 px
```

Fig. 3.7 Landsat 8 surface reflectance image bands and date

```
/////
// Collections of single images - Landsat 8 Surface Reflectance
/////

// Create and Earth Engine Point object over San Francisco.
var pointSF = ee.Geometry.Point([-122.44, 37.76]);

// Import the Landsat 8 Surface Reflectance collection.
var landsat8SR = ee.ImageCollection('LANDSAT/LC08/C02/T1_L2');

// Filter the collection and select the first image.
var landsat8SRimage = landsat8SR.filterDate('2014-03-18',
        '2014-03-19')
     .filterBounds(pointSF)
     .first();

print('Landsat 8 Surface Reflectance image', landsat8SRimage);
```

Copy and paste the code below to add this image to the map with adjusted R, G, and B bands in the “bands” parameter for true-color display (see Chap. 2).

```
// Center map to the first image.
Map.centerObject(landsat8SRimage, 8);

// Add first image to the map.
Map.addLayer(landsat8SRimage,
    {
        bands: ['SR_B4', 'SR_B3', 'SR_B2'],
        min: 7000,
        max: 13000
    },
    'Landsat 8 SR');
```

Compare this image (Fig. 3.8) with the raw Landsat 8 images from the previous section (Fig. 3.6). Zoom in and out and pan the screen as needed. What do you notice? Save your script but don’t start a new one—we will keep adding code to this script.

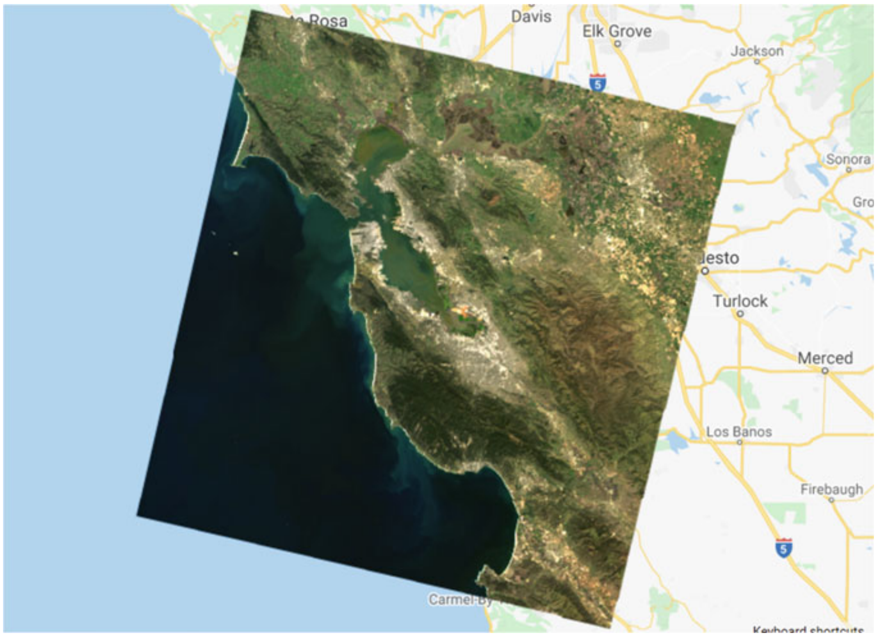


Fig. 3.8 Landsat 8 surface reflectance scene from March 18, 2014

Code Checkpoint F12c. The book's repository contains a script that shows what your code should look like at this point.

3.2.3 Section 3: Pre-made Composites

Pre-made composites take individual images from image collections across a set area or time period and assemble them into a single layer. This can be done for many different datasets, including satellite images (e.g., MODIS, Landsat, Sentinel), climatological information, forest or vegetation information, and more.

For example, image collections may have multiple images in one location, as we saw in our "filter by location" example above. Some of the images might have a lot of cloud cover or other atmospheric artifacts that make the imagery quality poor. Other images might be very high quality, because they were taken on sunny days when the satellite was flying directly overhead. The compositing process takes all of these different images, picks the best ones, and then stitches them together into a single layer. The compositing period can differ for different datasets and goals; for example, you may encounter daily, monthly, and/or yearly composites. To do this manually is more advanced (see, for example, Chap. 15); however, with the pre-made composites available in Earth Engine, some of that complex work has been done for you.

MODIS Daily True-Color Imagery

We will explore two examples of composites made with data from the MODIS sensors, a pair of sensors aboard the Terra and Aqua satellites. On these complex sensors, different MODIS bands produce data at different spatial resolutions. For the visible bands, the lowest common resolution is 500 m (red and NIR are 250 m).

Let us use the code below to import the MCD43A4.006 MODIS Nadir BRDF-Adjusted Reflectance Daily 500 m dataset and view a recent image. This dataset is produced daily based on a 16-day retrieval period, choosing the best representative pixel from the 16-day period. The 16-day period covers about eight days on either side of the nominal compositing date, with pixels closer to the target date given a higher priority.

```
/////
// Pre-made composites
/////

// Import a MODIS dataset of daily BRDF-corrected
reflectance.
var modisDaily = ee.ImageCollection('MODIS/006/MCD43A4');

// Filter the dataset to a recent date.
var modisDailyRecent = modisDaily.filterDate('2021-11-01');

// Add the dataset to the map.
var modisVis = {
    bands: [
        'Nadir_Reflectance_Band1',
        'Nadir_Reflectance_Band4',
        'Nadir_Reflectance_Band3'
    ],
    min: 0,
    max: 4000
};
Map.addLayer(modisDailyRecent, modisVis, 'MODIS Daily
Composite');
```

Uncheck the other layer ("Landsat 8 SR"), zoom out (e.g., country-scale), and pan around the image (Fig. 3.9). Notice how there are no clouds in the image, but there are some pixels with no data (Fig. 3.10). These are persistently cloudy areas that have no clear pixels in the particular period chosen.

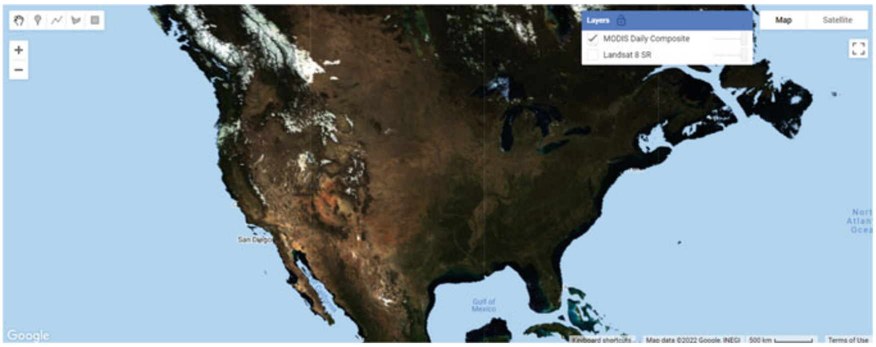


Fig. 3.9 MODIS Daily true-color image

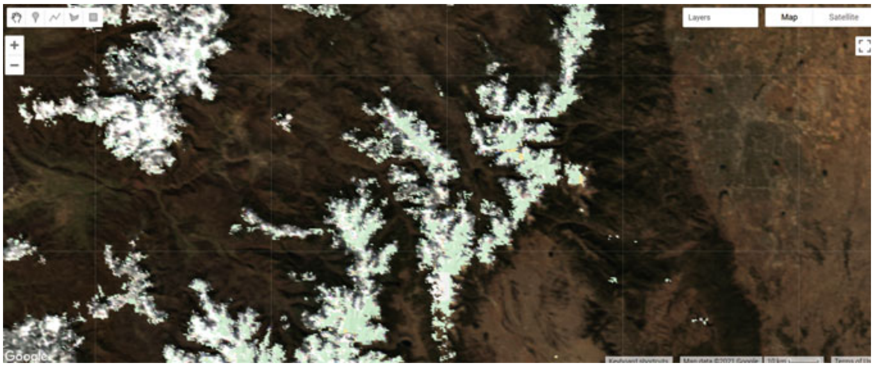


Fig. 3.10 Examples of gaps on the MODIS Daily true-color image near Denver, Colorado, USA

MODIS Monthly Burned Areas

Some of the MODIS bands have proven useful in determining where fires are burning and what areas they have burned. A monthly composite product for burned areas is available in Earth Engine. Copy and paste the code below.

```
// Import the MODIS monthly burned areas dataset.
var modisMonthly = ee.ImageCollection('MODIS/006/MCD64A1');

// Filter the dataset to a recent month during fire season.
var modisMonthlyRecent = modisMonthly.filterDate('2021-08-01');

// Add the dataset to the map.
Map.addLayer(modisMonthlyRecent, {}, 'MODIS Monthly Burn');
```

Uncheck the other layers, and then pan and zoom around the map. Areas that have burned in the past month will show up as red (Fig. 3.11). Can you see where fires burned areas of California, USA? In Southern and Central Africa? Northern Australia?

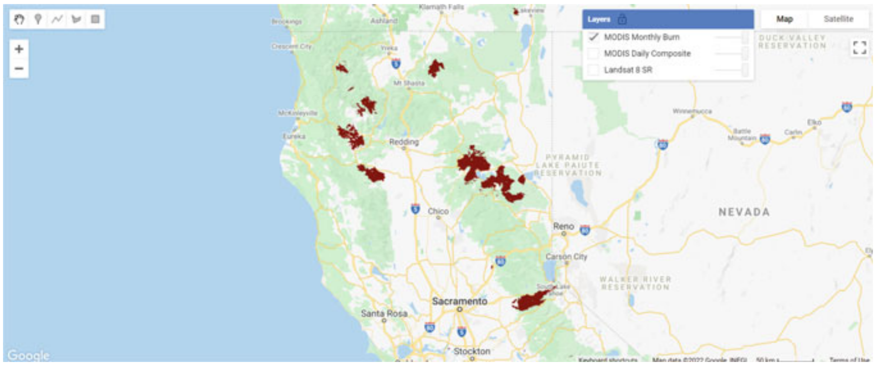


Fig. 3.11 MODIS Monthly Burn image over California

Code Checkpoint F12d. The book's repository contains a script that shows what your code should look like at this point.

Save your script and start a new one by refreshing the page.

3.2.4 Section 4: Other Satellite Products

Satellites can also collect information about the climate, weather, and various compounds present in the atmosphere. These satellites leverage portions of the electromagnetic spectrum and how different objects and compounds reflect when hit with sunlight in various wavelengths. For example, methane (CH_4) reflects the 760 nm portion of the spectrum. Let us take a closer look at a few of these datasets.

Methane

The European Space Agency makes available a methane dataset from Sentinel-5 in Earth Engine. Copy and paste the code below to add to the map methane data from the first time of collection on November 28, 2018. We use the `select` function (See Chap. 2) to select the methane-specific band of the dataset. We also introduce values for a new argument for the visualization parameters of `Map.addLayer`: We use a color `palette` to display a single band of an image in color. Here, we chose varying colors from black for the minimum value to red for the maximum value. Values in between will have the color in the order outlined by the `palette` parameter (a list of string colors: black, blue, purple, cyan, green, yellow, red).

```
/////
// Other satellite products
/////

// Import a Sentinel-5 methane dataset.
var methane =
ee.ImageCollection('COPERNICUS/S5P/OFFL/L3_CH4');

// Filter the methane dataset.
var methane2018 = methane.select(
        'CH4_column_volume_mixing_ratio_dry_air')
    .filterDate('2018-11-28', '2018-11-29')
    .first();

// Make a visualization for the methane data.
var methaneVis = {
    palette: ['black', 'blue', 'purple', 'cyan', 'green',
        'yellow', 'red'
    ],
    min: 1770,
    max: 1920
};

// Center the Map.
Map.centerObject(methane2018, 3);

// Add the methane dataset to the map.
Map.addLayer(methane2018, methaneVis, 'Methane');
```

Notice the different levels of methane over the African continent (Fig. 3.12).

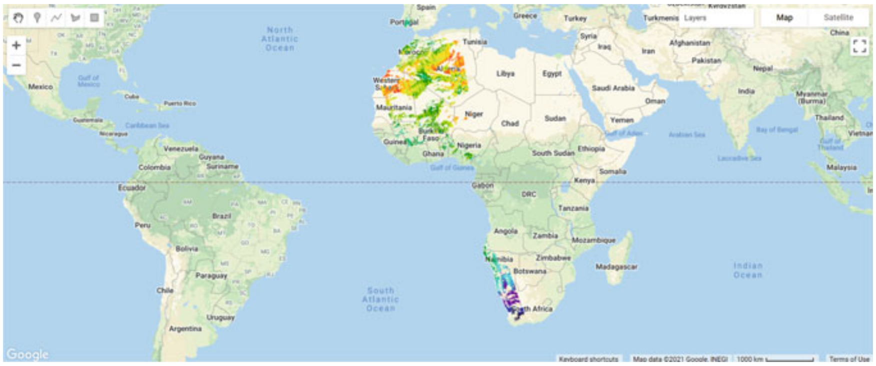


Fig. 3.12 Methane levels over the African continent on November 28, 2018

Weather and Climate Data
Many weather and climate datasets are available in Earth Engine. One of these is the European Center for Medium-Range Weather Forecast Reanalysis (ERA5) dataset used by Sulova and Jokar (2021). Copy and paste the code below to add the January 2018 monthly data to the map.

```
// Import the ERA5 Monthly dataset
var era5Monthly = ee.ImageCollection('ECMWF/ERA5/MONTHLY');

// Filter the dataset
var era5MonthlyTemp =
era5Monthly.select('mean_2m_air_temperature')
    .filterDate('2018-01-01', '2019-01-31')
    .first();

// Add the ERA dataset to the map.
Map.addLayer(era5MonthlyTemp,
    {
        palette: ['yellow', 'red'],
        min: 260,
        max: 320
    },
    'ERA5 Max Monthly Temp');
```

Examine some of the temperatures in this image (Fig. 3.13) by using the **Inspector** tool (see Chap. 2). Pan and zoom out if needed. The units are in Kelvin, which is Celsius plus 273.15°.

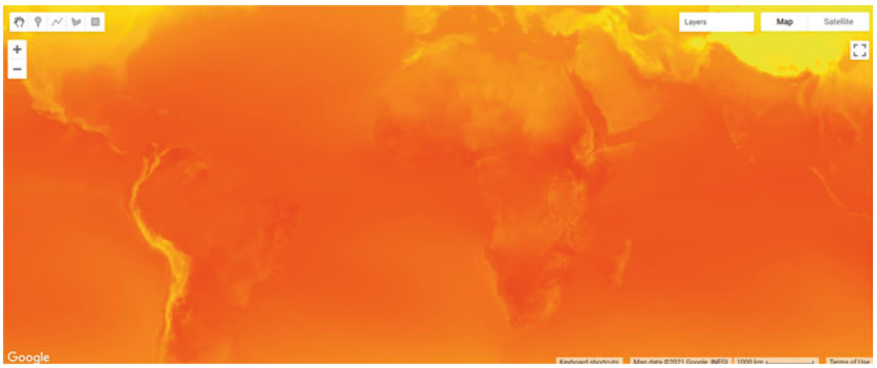


Fig. 3.13 ERA5 maximum monthly temperature, January 2018

Code Checkpoint F12e. The book's repository contains a script that shows what your code should look like at this point.

Save your script and start a new one by refreshing the page.

3.2.5 Section 5: Pre-classified Land Use and Land Cover

Another type of dataset available in Earth Engine is LULC maps that have already been classified. Instead of showing how the Earth's surface looks—that is, the visible and other electromagnetic spectrum reflectance detected by satellites—these datasets take satellite imagery and use it to assign a label to each pixel on Earth's surface. For example, categories might include vegetation, bare soil, built environment (pavement, buildings), and water.

Let us take a closer look at two of these datasets.

ESA WorldCover

The European Space Agency (ESA) provides a global land cover map for the year 2020 based on Sentinel-1 and Sentinel-2 data. WorldCover uses 11 different land cover classes including built-up, cropland, open water, and mangroves. Copy and paste the code below to add this image to the map. In this dataset, the band `'Map'` already contains a palette color associated with the 11 land cover class values.

```
/////
// Pre-classified Land Use Land Cover
/////

// Import the ESA WorldCover dataset.
var worldCover =
ee.ImageCollection('ESA/WorldCover/v100').first();

// Center the Map.
Map.centerObject(worldCover, 3);

// Add the worldCover layer to the map.
Map.addLayer(worldCover, {
    bands: ['Map']
}, 'WorldCover');
```

Examine the WorldCover land cover classification (Fig. 3.14). Compare it with some of the satellite imagery we have explored in previous sections. Chapter 4 shows how to determine the meaning of the colors and values in a dataset like this.

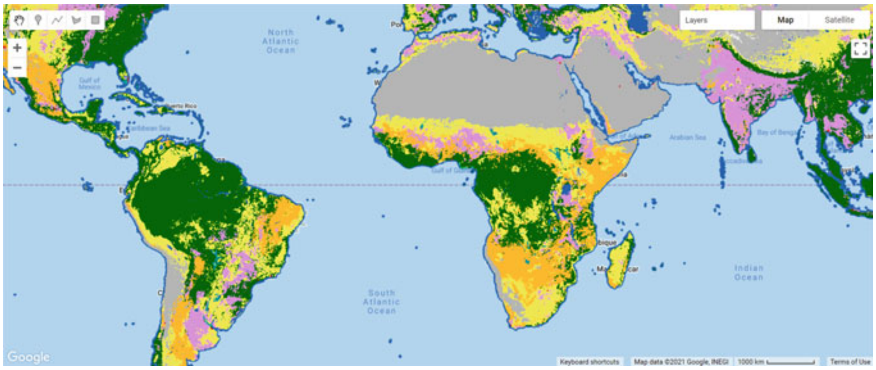


Fig. 3.14 ESA's 2020 WorldCover map

Global Forest Change

Another land cover product that has been pre-classified for you and is available in Earth Engine is the Global Forest Change dataset. This analysis was conducted between 2000 and 2020. Unlike the WorldCover dataset, this dataset focuses on the percent of tree cover across the Earth's surface in a base year of 2000, and how that has changed over time. Copy and paste the code below to visualize the tree cover in 2000. Note that in the code below we define the visualization parameters as a variable `treeCoverViz` instead of having its calculation done within the `Map.addLayer` function.

```
// Import the Hansen Global Forest Change dataset.
var globalForest = ee.Image(
    'UMD/hansen/global_forest_change_2020_v1_8');

// Create a visualization for tree cover in 2000.
var treeCoverViz = {
    bands: ['treecover2000'],
    min: 0,
    max: 100,
    palette: ['black', 'green']
};

// Add the 2000 tree cover image to the map.
Map.addLayer(globalForest, treeCoverViz, 'Hansen 2000 Tree
Cover');
```

Notice how areas with high tree cover (e.g., the Amazon) are greener and areas with low tree cover are darker (Fig. 3.15). In case you see an error on the **Console**

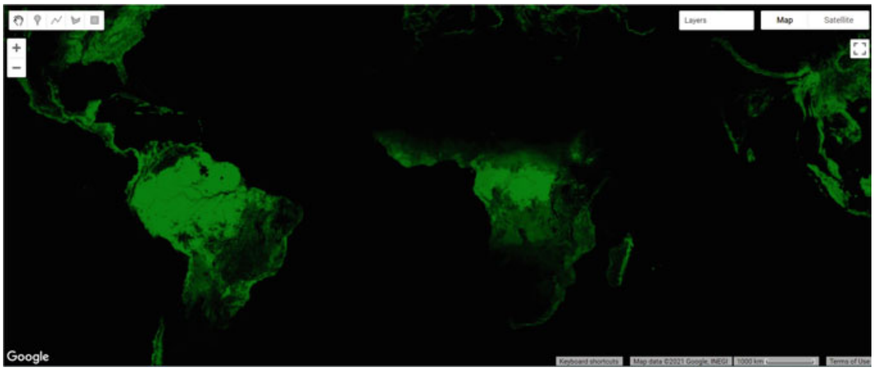


Fig. 3.15 Global Forest Change 2000 tree cover layer

such as “Cannot read properties of null”, do not worry. Sometimes Earth Engine will show these transient errors, but they will not affect the script in any way.

Copy and paste the code below to visualize the tree cover loss over the past 20 years.

```
// Create a visualization for the year of tree loss over
the past 20 years.
var treeLossYearViz = {
    bands: ['lossyear'],
    min: 0,
    max: 20,
    palette: ['yellow', 'red']
};

// Add the 2000-2020 tree cover loss image to the map.
Map.addLayer(globalForest, treeLossYearViz, '2000-2020 Year
of Loss');
```

Leave the previous 2000 tree cover layer checked and analyze the loss layer on top of it—yellow, orange, and red areas (Fig. 3.16). Pan and zoom around the map. Where has there been recent forest loss (which is shown in red)?

Code Checkpoint F12f. The book’s repository contains a script that shows what your code should look like at this point.

Save your script and start a new one.

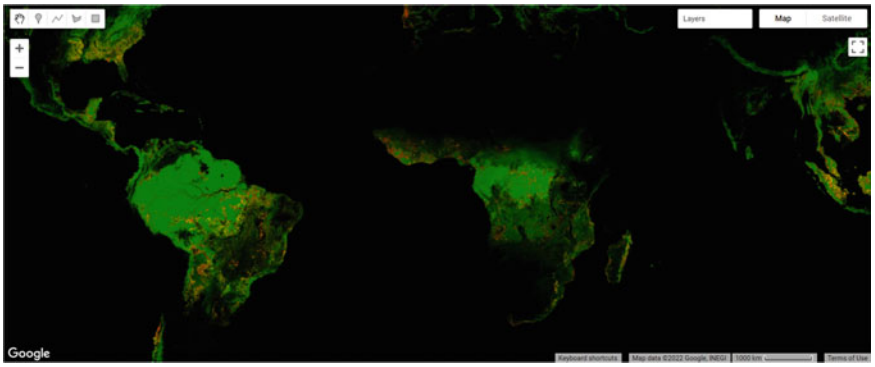


Fig. 3.16 Global Forest Change 2000–2020 tree cover loss (yellow–red) and 2000 tree cover (black-green)

3.2.6 Section 6: Other Datasets

There are many other types of datasets in the Earth Engine Data Catalog that you can explore and use for your own analyses. These include global gridded population counts, terrain, and geophysical data. Let us explore two of these datasets now.

Gridded Population Count

The Gridded Population of the World dataset estimates human population for each grid cell across the entire Earth’s surface. Copy and paste the code below to add the 2000 population count layer. We use a predefined palette `populationPalette`, which is a list of six-digit strings of hexadecimal values representing additive RGB colors (as first seen in Chap. 2). Lighter colors correspond to lower population count, and darker colors correspond to higher population count.

```
/////
// Other datasets
/////

// Import and filter a gridded population dataset.
var griddedPopulation = ee.ImageCollection(
        'CIESIN/GPWv411/GPW_Population_Count')
    .first();

// Predefined palette.
var populationPalette = [
    'ffffe7',
    '86a192',
    '509791',
    '307296',
    '2c4484',
    '000066'
];

// Center the Map.
Map.centerObject(griddedPopulation, 3);

// Add the population data to the map.
Map.addLayer(griddedPopulation,
    {
        min: 0,
        max: 1200,
        'palette': populationPalette
    },
    'Gridded Population');
```

Pan around the image (Fig. 3.17). What happens when you change the minimum and maximum values in the visualization? As described in Chap. 2, the minimum and maximum values represent the range of values of the dataset. Identify a location of interest to you—may be an area near your current location, or your hometown. If you click on the **Inspector** tab, you should be able to find the population count (Fig. 3.18).

Digital Elevation Models

Digital elevation models (DEMs) use airborne and satellite instruments to estimate the elevation of each location. Earth Engine has both local and global DEMs available. One of the global DEMs available is the NASADEM dataset, a DEM produced from a NASA mission. Copy and paste the code below to import the dataset and visualize the elevation band.

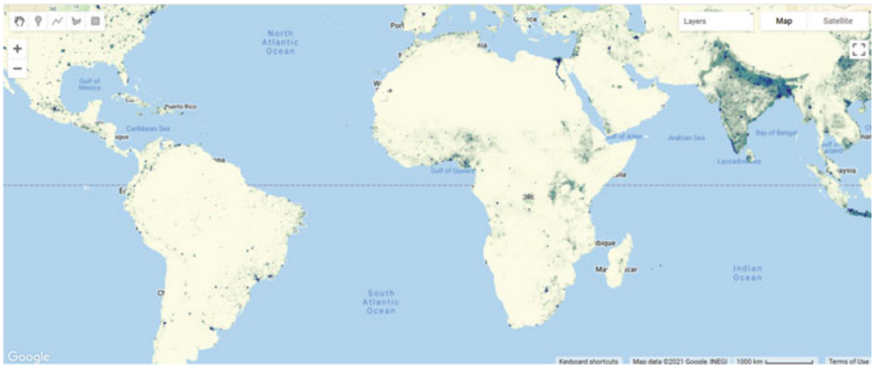
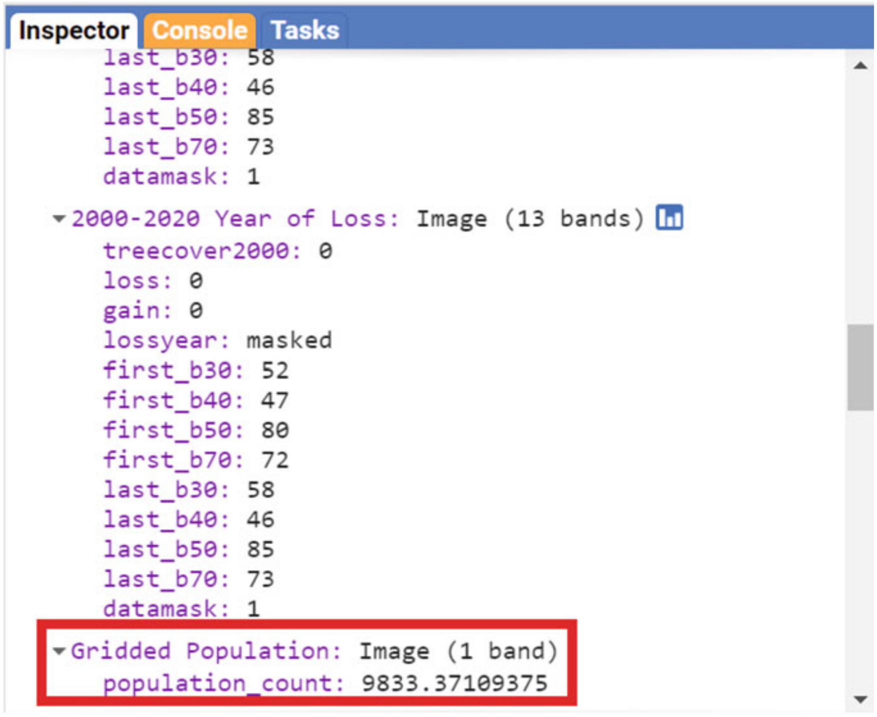


Fig. 3.17 Gridded population count of 2000



```
    last_b30: 58
    last_b40: 46
    last_b50: 85
    last_b70: 73
    datamask: 1
▾2000-2020 Year of Loss: Image (13 bands)
    treecover2000: 0
    loss: 0
    gain: 0
    lossyear: masked
    first_b30: 52
    first_b40: 47
    first_b50: 80
    first_b70: 72
    last_b30: 58
    last_b40: 46
    last_b50: 85
    last_b70: 73
    datamask: 1
▾Gridded Population: Image (1 band)
    population_count: 9833.37109375
```

Fig. 3.18 2000 population count for a point near Rio de Janeiro, Brazil

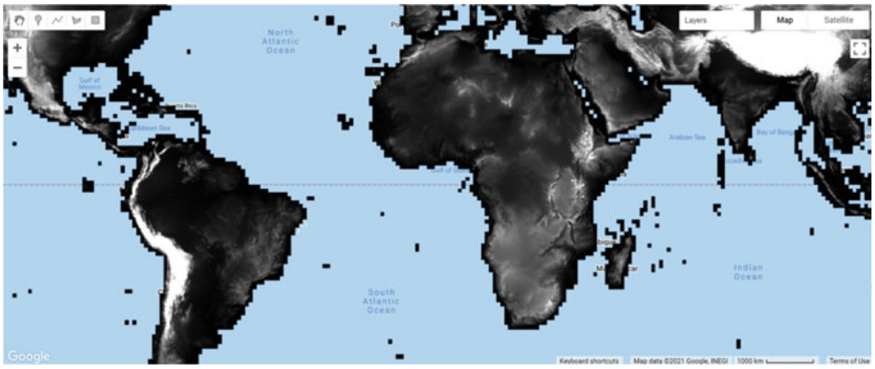


Fig. 3.19 NASADEM elevation

```
// Import the NASA DEM Dataset.
var nasaDEM = ee.Image('NASA/NASADEM_HGT/001');

// Add the elevation layer to the map.
Map.addLayer(nasaDEM, {
    bands: ['elevation'],
    min: 0,
    max: 3000
}, 'NASA DEM');
```

Uncheck the population layer and zoom in to examine the patterns of topography (Fig. 3.19). Can you see where a mountain range is located? Where is a river located? Try changing the minimum and maximum in order to make these features more visible. Save your script.

Code Checkpoint F12g. The book's repository contains a script that shows what your code should look like at this point.

Take a moment to look through all of the different layers that we have explored so far. You can open your scripts one at a time or in different tabs, or even by copying the code into one single script. Turn the layers on and off, pan around, and zoom in and out accordingly to visualize the different datasets on the map.

3.3 Synthesis

Assignment 1. Explore the Earth Engine Data Catalog and find a dataset that is near your location. To do this, you can type keywords into the search bar, located above the Earth Engine code. Import a dataset into your workspace and filter the dataset to a single image. Then, print the information of the image into the **Console**

and add the image to the map, either using three selected bands or a custom palette for one band.

3.4 Conclusion

In this chapter, we introduced image collections in Earth Engine and learned how to apply multiple types of filters to image collections to identify multiple or a single image for use. We also explored a few of the many different image collections available in the Earth Engine Data Catalog. Understanding how to find, access, and filter image collections is an important step in learning how to perform spatial analyses in Earth Engine.

References

Chander G, Huang C, Yang L et al (2009a) Developing consistent Landsat data sets for large area applications: the MRLC 2001 protocol. IEEE Geosci Remote Sens Lett 6:777–781. https://doi.org/10.1109/LGRS.2009.2025244

Chander G, Markham BL, Helder DL (2009b) Summary of current radiometric calibration coefficients for Landsat MSS, TM, ETM+, and EO-1 ALI sensors. Remote Sens Environ 113:893–903. https://doi.org/10.1016/j.rse.2009.01.007

Hansen MC, Potapov PV, Moore R et al (2013) High-resolution global maps of 21st-century forest cover change. Science 342:850–853. https://doi.org/10.1126/science.1244693

Sulova A, Arsanjani JJ (2021) Exploratory analysis of driving force of wildfires in Australia: an application of machine learning within Google Earth Engine. Remote Sens 13:1–23. https://doi.org/10.3390/rs13010010

4 The Remote Sensing Vocabulary

Karen Dyson, Andréa Puzzi Nicolau, David Saah, and Nicholas Clinton

Overview
The purpose of this chapter is to introduce some of the principal characteristics of remotely sensed images and how they can be examined in Earth Engine. We discuss spatial resolution, temporal resolution, and spectral resolution, along with how to access important image metadata. You will be introduced to image data from several sensors aboard various satellite platforms. At the completion of the chapter, you will be able to understand the difference between remotely sensed datasets based on these characteristics and how to choose an appropriate dataset for your analysis based on these concepts.

K. Dyson · A. P. Nicolau · D. Saah
Spatial Informatics Group, Pleasanton, CA, USA
e-mail: kdyson@sig-gis.com

A. P. Nicolau
e-mail: apnicolau@sig-gis.com

K. Dyson · A. P. Nicolau
SERVIR-Amazonia, Cali, Colombia

K. Dyson
Dendrolytics, Seattle, WA, USA

D. Saah (✉)
University of San Francisco, San Francisco, CA, USA
e-mail: dssaah@usfca.edu

N. Clinton
Google LLC, Mountain View, CA, USA
e-mail: nclinton@google.com

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_4

Learning Outcomes

- Understanding spatial, temporal, and spectral resolution.
- Navigating the Earth Engine **Console** to gather information about a digital image, including resolution and other data documentation.

Assumes you know how to

- Navigate among Earth Engine result tabs (Chap. 1).
- Visualize images with a variety of false-color band combinations (Chap. 2).

4.1 Introduction to Theory

Images and image collections form the basis of many remote sensing analyses in Earth Engine. There are many different types of satellite imagery available to use in these analyses, but not every dataset is appropriate for every analysis. To choose the most appropriate dataset for your analysis, you should consider multiple factors. Among these are the resolution of the dataset—including the spatial, temporal, and spectral resolutions—as well as how the dataset was created and its quality.

The resolution of a dataset can influence the granularity of the results, the accuracy of the results, and how long it will take the analysis to run, among other things. For example, spatial resolution, which you will learn more about in Sect. 4.2.1, indicates the amount of Earth's surface area covered by a single pixel. One recent study compared the results of a land use classification (the process by which different areas of the Earth's surface are classified as forest, urban areas, etc.) and peak total suspended solids (TSS) loads using two datasets with different spatial resolutions. One dataset had pixels representing 900 m^2 of the Earth's surface, and the other represented 1 m^2. The higher resolution dataset (1 m^2) had higher accuracy for the land use classification and better predicted TSS loads for the full study area. On the other hand, the lower resolution dataset was less costly and required less analysis time (Fisher et al. 2018).

Temporal and spectral resolutions can also strongly affect analysis outcomes. In the Practicum that follows, we will showcase each of these types of resolution, along with key metadata types. We will also show you how to find more information about the characteristics of a given dataset in Earth Engine.

4.2 Practicum

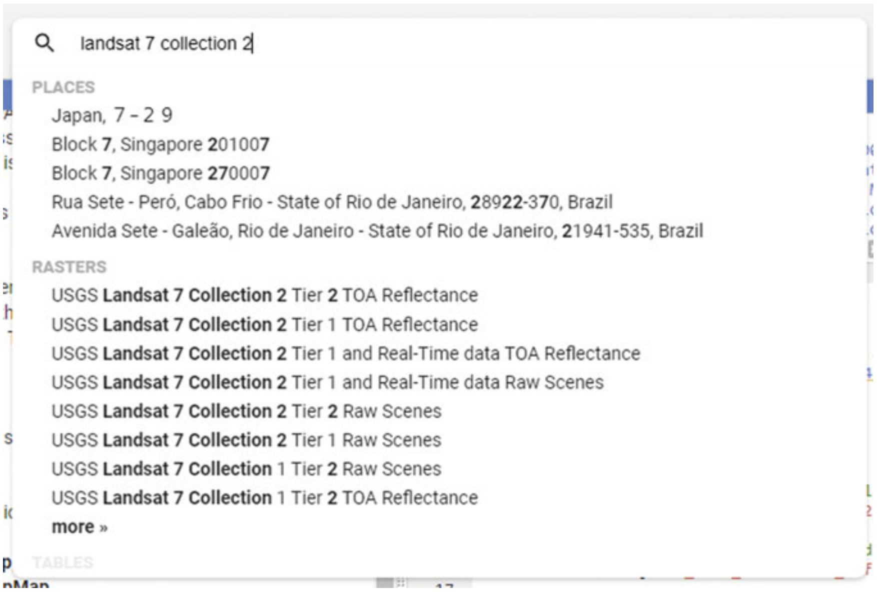
4.2.1 Section 1: Searching for and Viewing Image Collection Information

Earth Engine's search bar can be used to find imagery and to locate important information about datasets in Earth Engine. Let us use the search bar, located above the Earth Engine code, to find out information about the Landsat 7 Collection 2 Raw Scenes. First, type "Landsat 7 collection 2" into the search bar (Fig. 4.1). Without hitting Enter, matches to that search term will appear.

Now, click on **USGS Landsat 7 Collection 2 Tier 1 Raw Scenes**. A new inset window will appear (Fig. 4.2).

The inset window has information about the dataset, including a description, bands that are available, image properties, and terms of use for the data across the top. Click on each of these tabs and read the information provided. While you may not understand all of the information right now, it will set you up for success in future chapters.

On the left-hand side of this window, you will see a range of dates when the data are available, a link to the dataset provider's webpage, and a collection snippet. This collection snippet can be used to import the dataset by pasting it into your script, as you did in previous chapters. You can also use the large **Import** button to import the dataset into your current workspace. In addition, if you click on the **See example** link, Earth Engine will open a new code window with a snippet of

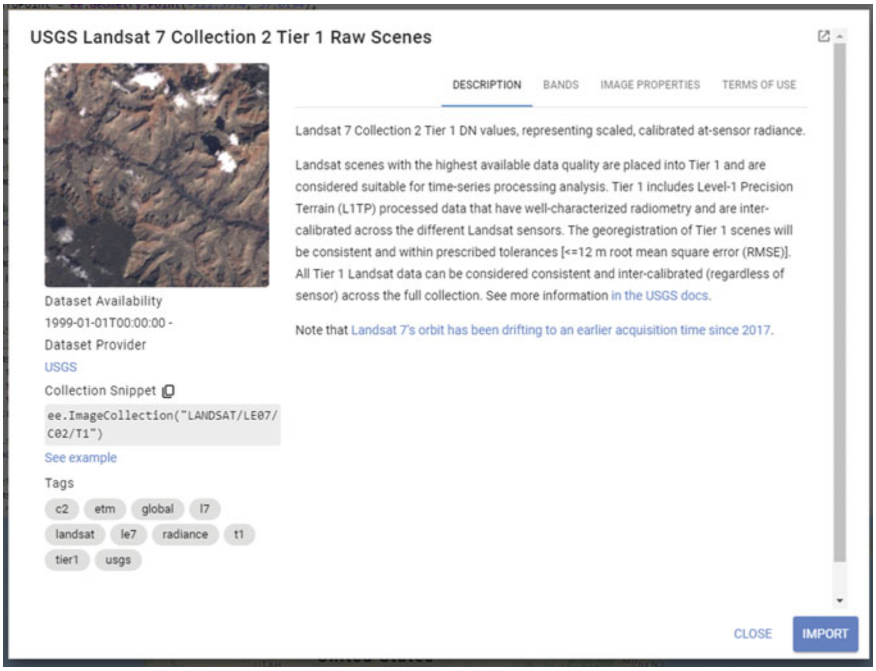

landsat 7 collection 2

PLACES
Japan, 7 - 2 9
Block 7, Singapore 201007
Block 7, Singapore 270007
Rua Sete - Peró, Cabo Frio - State of Rio de Janeiro, 28922-370, Brazil
Avenida Sete - Galeão, Rio de Janeiro - State of Rio de Janeiro, 21941-535, Brazil

RASTERS
USGS Landsat 7 Collection 2 Tier 2 TOA Reflectance
USGS Landsat 7 Collection 2 Tier 1 TOA Reflectance
USGS Landsat 7 Collection 2 Tier 1 and Real-Time data TOA Reflectance
USGS Landsat 7 Collection 2 Tier 1 and Real-Time data Raw Scenes
USGS Landsat 7 Collection 2 Tier 2 Raw Scenes
USGS Landsat 7 Collection 2 Tier 1 Raw Scenes
USGS Landsat 7 Collection 1 Tier 2 Raw Scenes
USGS Landsat 7 Collection 1 Tier 2 TOA Reflectance
more »

TABLES

Fig. 4.1 Searching for Landsat 7 in the search bar


USGS Landsat 7 Collection 2 Tier 1 Raw Scenes

Landsat 7 Collection 2 Tier 1 DN values, representing scaled, calibrated at-sensor radiance.

Landsat scenes with the highest available data quality are placed into Tier 1 and are considered suitable for time-series processing analysis. Tier 1 includes Level-1 Precision Terrain (L1TP) processed data that have well-characterized radiometry and are inter-calibrated across the different Landsat sensors. The georegistration of Tier 1 scenes will be consistent and within prescribed tolerances [<=12 m root mean square error (RMSE)]. All Tier 1 Landsat data can be considered consistent and inter-calibrated (regardless of sensor) across the full collection. See more information in the USGS docs.

Note that Landsat 7's orbit has been drifting to an earlier acquisition time since 2017.

Dataset Availability
1999-01-01T00:00:00 -
Dataset Provider
USGS
Collection Snippet

```
ee.ImageCollection("LANDSAT/LE07/C02/T1")
```

See example
Tags
c2 etm global l7 landsat le7 radiance t1 tier1 usgs

Fig. 4.2 Inset window with information about the Landsat 7 dataset

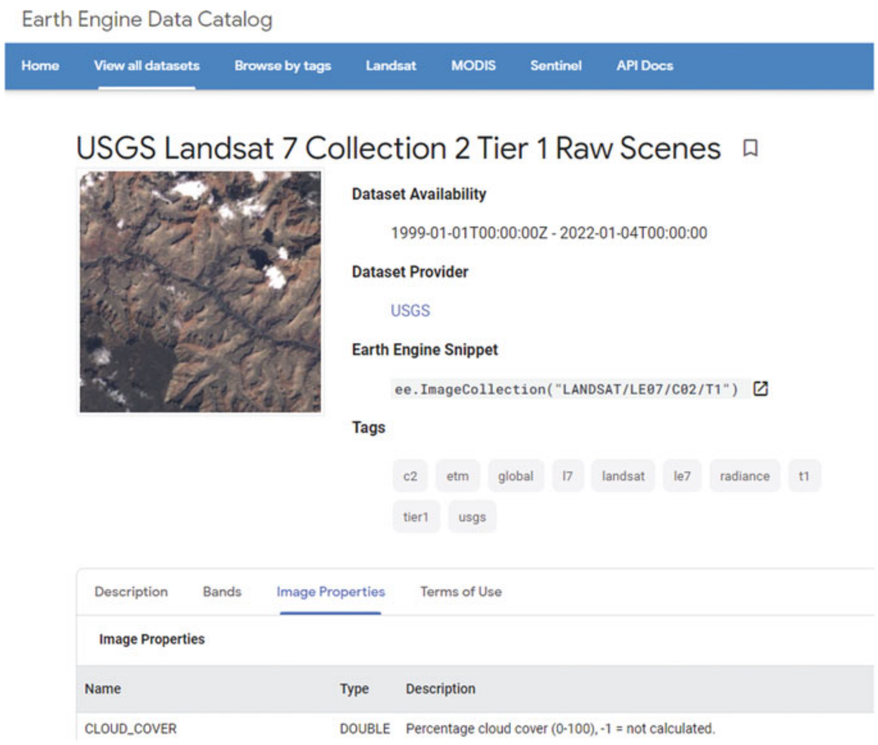
code that shows code using the dataset. Code snippets like this can be very helpful when learning how to use a dataset that is new to you.

For now, click on the small "pop out" button in the upper-right corner of the window. This will open a new window with the same information (Fig. 4.3); you can keep this new window open and use it as a reference as you proceed.

Switch back to your code window. Your "Landsat 7 collection 2" search term should still be in the search bar. This time, click the "Enter" key or click on the search magnifying glass icon. This will open a **Search results** inset window (Fig. 4.4).

This more complete search results' inset window contains short descriptions about each of the datasets matching your search, to help you choose which dataset you want to use. Click on the **Open in Catalog** button to view these search results in the Earth Engine Data Catalog (Fig. 4.5). Note that you may need to click **Enter** in the data catalog search bar with your phrase to bring up the results in this new window.

Now that we know how to view this information, let us dive into some important remote sensing terminology.



USGS Landsat 7 Collection 2 Tier 1 Raw Scenes

Dataset Availability

1999-01-01T00:00:00Z - 2022-01-04T00:00:00

Dataset Provider

USGS

Earth Engine Snippet

```
ee.ImageCollection("LANDSAT/LE07/C02/T1")
```

Tags

c2 etm global l7 landsat le7 radiance t1 tier1 usgs

Image Properties

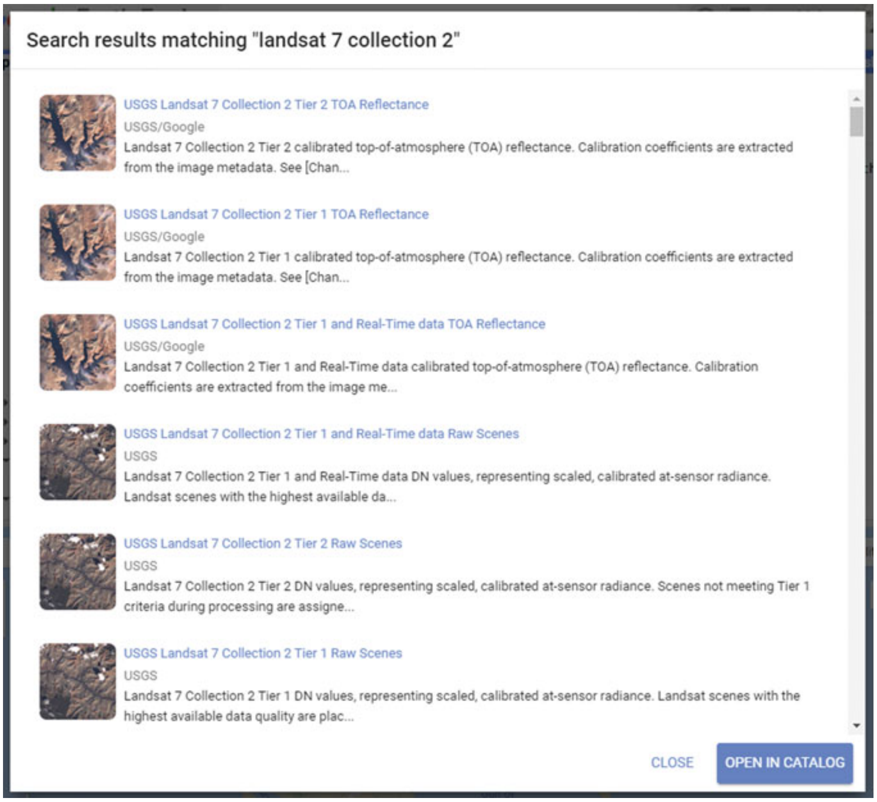
Name	Type	Description
CLOUD_COVER	DOUBLE	Percentage cloud cover (0-100), -1 = not calculated.

Fig. 4.3 Data Catalog page for Landsat 7 with information about the dataset

4.2.2 Section 2: Spatial Resolution

Spatial resolution relates to the amount of Earth's surface area covered by a single pixel. It is typically referred to in linear units, for a single side of a square pixel: for example, we typically say that Landsat 7 has “30 m” color imagery. This means that each pixel is 30 m to a side, covering a total area of 900 m^2 of the Earth's surface. Spatial resolution is often interchangeably also referred to as the *scale*, as will be seen in this chapter when we print that value. The spatial resolution of a given dataset greatly affects the appearance of images and the information in them, when you are viewing them on Earth's surface.

Next, we will visualize data from multiple sensors that capture data at different spatial resolutions, to compare the effect of different pixel sizes on the information and detail in an image. We will be selecting a single image from each `ImageCollection` to visualize. To view the image, we will draw them each as a color-IR image, a type of false-color image (described in detail in Chap. 2) that uses the infrared, red, and green bands. As you move through this portion of the Practicum, zoom in and out to see differences in the pixel size and the image size.


Search results matching "landsat 7 collection 2"

USGS Landsat 7 Collection 2 Tier 2 TOA Reflectance
USGS/Google
Landsat 7 Collection 2 Tier 2 calibrated top-of-atmosphere (TOA) reflectance. Calibration coefficients are extracted from the image metadata. See [Chan...

USGS Landsat 7 Collection 2 Tier 1 TOA Reflectance
USGS/Google
Landsat 7 Collection 2 Tier 1 calibrated top-of-atmosphere (TOA) reflectance. Calibration coefficients are extracted from the image metadata. See [Chan...

USGS Landsat 7 Collection 2 Tier 1 and Real-Time data TOA Reflectance
USGS/Google
Landsat 7 Collection 2 Tier 1 and Real-Time data calibrated top-of-atmosphere (TOA) reflectance. Calibration coefficients are extracted from the image me...

USGS Landsat 7 Collection 2 Tier 1 and Real-Time data Raw Scenes
USGS
Landsat 7 Collection 2 Tier 1 and Real-Time data DN values, representing scaled, calibrated at-sensor radiance. Landsat scenes with the highest available da...

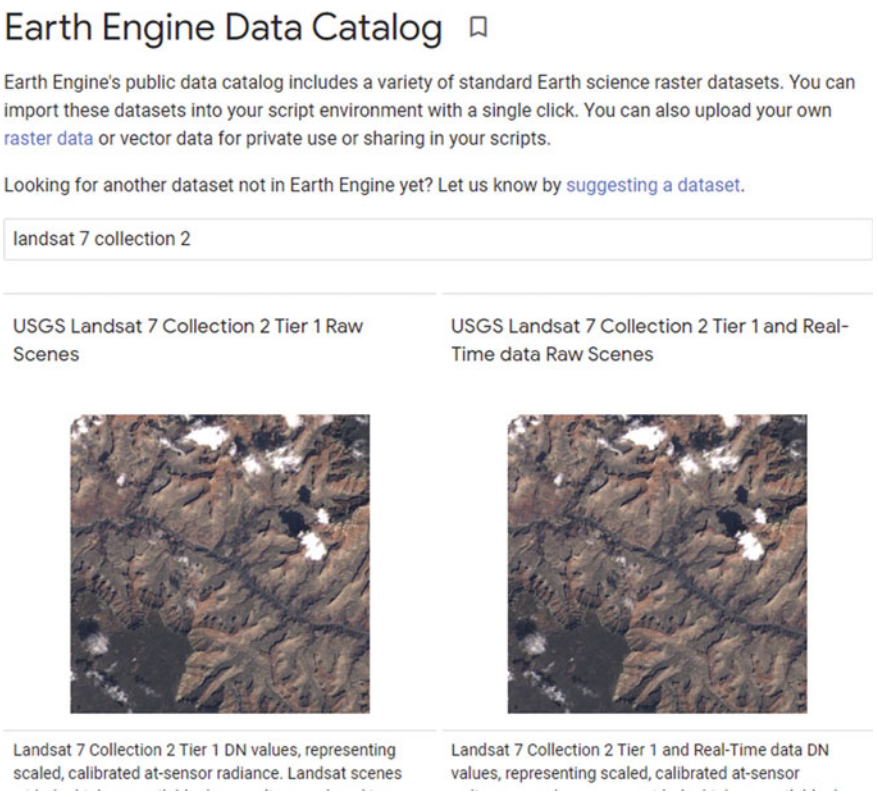
USGS Landsat 7 Collection 2 Tier 2 Raw Scenes
USGS
Landsat 7 Collection 2 Tier 2 DN values, representing scaled, calibrated at-sensor radiance. Scenes not meeting Tier 1 criteria during processing are assigne...

USGS Landsat 7 Collection 2 Tier 1 Raw Scenes
USGS
Landsat 7 Collection 2 Tier 1 DN values, representing scaled, calibrated at-sensor radiance. Landsat scenes with the highest available data quality are plac...

Fig. 4.4 Search results matching “Landsat 7 collection 2”

MODIS (on the Aqua and Terra satellites)
As discussed in Chap. 3, the common resolution collected by MODIS for the infrared, red, and green bands is 500 m. This means that each pixel is 500 m on a side, with a pixel thus representing 0.25 km^2 of area on the Earth’s surface.

Use the following code to center the map on the San Francisco airport at a zoom level of 16.



Earth Engine Data Catalog

Earth Engine's public data catalog includes a variety of standard Earth science raster datasets. You can import these datasets into your script environment with a single click. You can also upload your own raster data or vector data for private use or sharing in your scripts.

Looking for another dataset not in Earth Engine yet? Let us know by suggesting a dataset.

landsat 7 collection 2

USGS Landsat 7 Collection 2 Tier 1 Raw Scenes

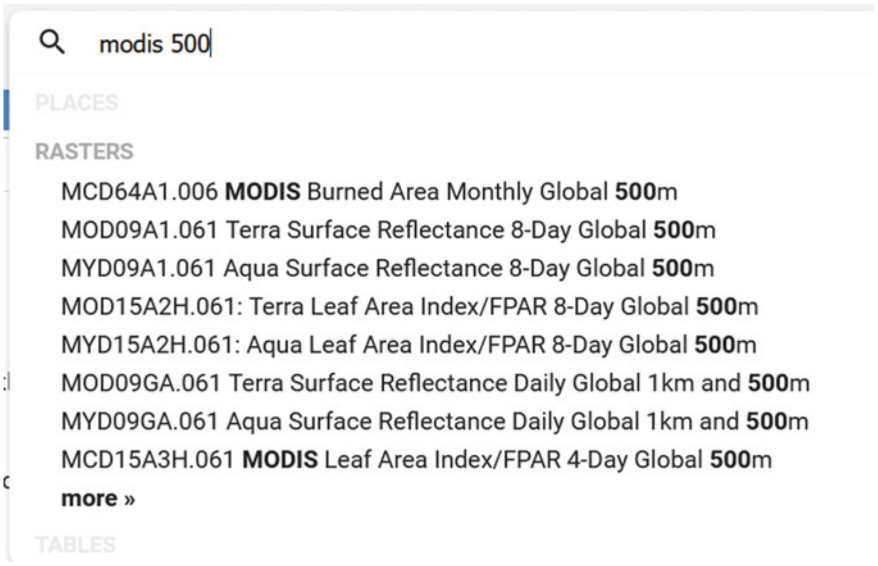
Landsat 7 Collection 2 Tier 1 DN values, representing scaled, calibrated at-sensor radiance. Landsat scenes

USGS Landsat 7 Collection 2 Tier 1 and Real-Time data Raw Scenes

Landsat 7 Collection 2 Tier 1 and Real-Time data DN values, representing scaled, calibrated at-sensor

Fig. 4.5 Earth Engine Data Catalog results for the "Landsat 7 collection 2" search term

```
//////
// Explore spatial resolution
//////

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

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



modis 500

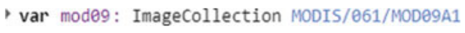
PLACES

RASTERS

MCD64A1.006 **MODIS** Burned Area Monthly Global **500**m
MOD09A1.061 Terra Surface Reflectance 8-Day Global **500**m
MYD09A1.061 Aqua Surface Reflectance 8-Day Global **500**m
MOD15A2H.061: Terra Leaf Area Index/FPAR 8-Day Global **500**m
MYD15A2H.061: Aqua Leaf Area Index/FPAR 8-Day Global **500**m
MOD09GA.061 Terra Surface Reflectance Daily Global 1km and **500**m
MYD09GA.061 Aqua Surface Reflectance Daily Global 1km and **500**m
MCD15A3H.061 **MODIS** Leaf Area Index/FPAR 4-Day Global **500**m
more »

TABLES

Fig. 4.6 Using the search bar for the MODIS dataset



```
▸ var mod09: ImageCollection MODIS/061/MOD09A1
```

Fig. 4.7 Rename the imported MODIS dataset

Let us use what we learned in the previous section to search for, get information about, and import the MODIS data into our Earth Engine workspace. Start by searching for “MODIS 500” in the Earth Engine search bar (Fig. 4.6).

Use this to import the “MOD09A1.061 Terra Surface Reflectance 8-day Global 500 m” `ImageCollection`. A default name for the import appears at the top of your script; change the name of the import to `mod09` (Fig. 4.7).

When exploring a new dataset, you can find the names of bands in images from that set by reading the summary documentation, known as the metadata, of the dataset. In this dataset, the three bands for a color-IR image are “sur_refl_b02” (infrared), “sur_refl_b01” (red), and “sur_refl_b04” (green).

```
// MODIS
// Get an image from your imported MODIS MYD09GA
collection.
var modisImage = mod09.filterDate('2020-02-01', '2020-03-
01').first();

// Use these MODIS bands for near infrared, red, and green,
respectively.
var modisBands = ['sur_refl_b02', 'sur_refl_b01',
'sur_refl_b04'];

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

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

In your map window, you should now see something like this (Fig. 4.8).

You might be surprised to see that the pixels, which are typically referred to as "square", are shown as parallelograms. The shape and orientation of pixels are controlled by the "projection" of the dataset, as well as the projection we are

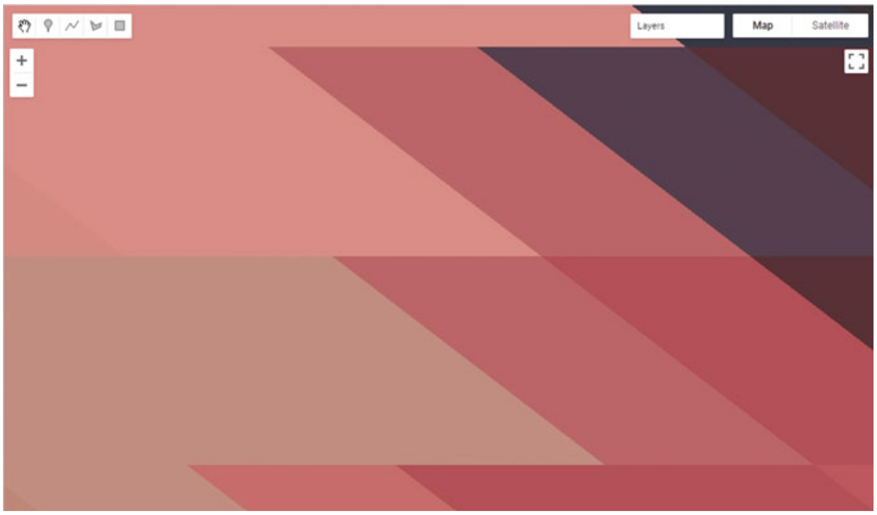


Fig. 4.8 Viewing the MODIS image of the San Francisco airport

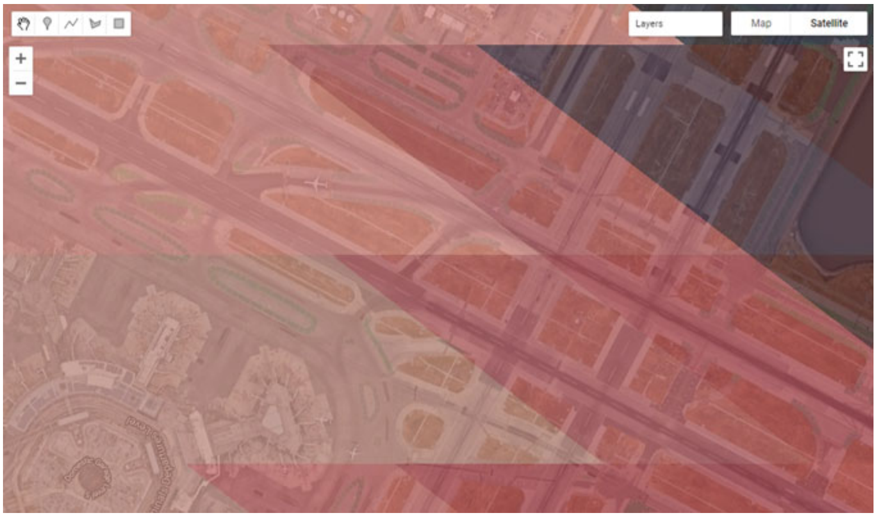


Fig. 4.9 Using transparency to view the MODIS pixel size in relation to high-resolution imagery of the San Francisco airport

viewing them in. Most users do not have to be very concerned about different projections in Earth Engine, which automatically transfers data between different coordinate systems as it did here. For more details about projections in general and their use in Earth Engine, you can consult the official documentation.

Let us view the size of pixels with respect to objects on the ground. Turn on the satellite basemap to see high-resolution data for comparison by clicking on **Satellite** in the upper-right corner of the map window. Then, decrease the layer's opacity: set the opacity in the **Layers** manager using the layer's slider (see Chap. 2). The result will look like Fig. 4.9.

Print the size of the pixels (in meters) by running this code:

```
// Get the scale of the data from the NIR band's
projection:
var modisScale = modisImage.select('sur_refl_b02')
    .projection().nominalScale();

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

In that call, we used the `nominalScale` function here after accessing the projection information from the MODIS NIR band. That function extracts the spatial resolution from the projection information, in a format suitable to be printed to the screen. The nominalScale function returns a value just under the stated 500 m resolution due to the sinusoidal projection of MODIS data and the distance of the pixel from nadir—that is, where the satellite is pointing directly down at the Earth’s surface.

TM (on early Landsat satellites)

Thematic Mapper (TM) sensors were flown aboard Landsat 4 and 5. TM data have been processed to a spatial resolution of 30 m, and were active from 1982 to 2012. Search for “Landsat 5 TM” and import the result called “USGS Landsat 5 TM Collection 2 Tier 1 Raw Scenes”. In the same way you renamed the MODIS collection, rename the import `tm`. In this dataset, the three bands for a color-IR image are called “B4” (infrared), “B3” (red), and “B2” (green). Let us now visualize TM data over the airport and compare it with the MODIS data. Note that we can either define the visualization parameters as a variable (as in the previous code snippet) or place them in curly braces in the `Map.addLayer` function (as in this code snippet).

When you run this code, the TM image will display. Notice how many more pixels are displayed on your screen when compared to the MODIS image (Fig. 4.10).

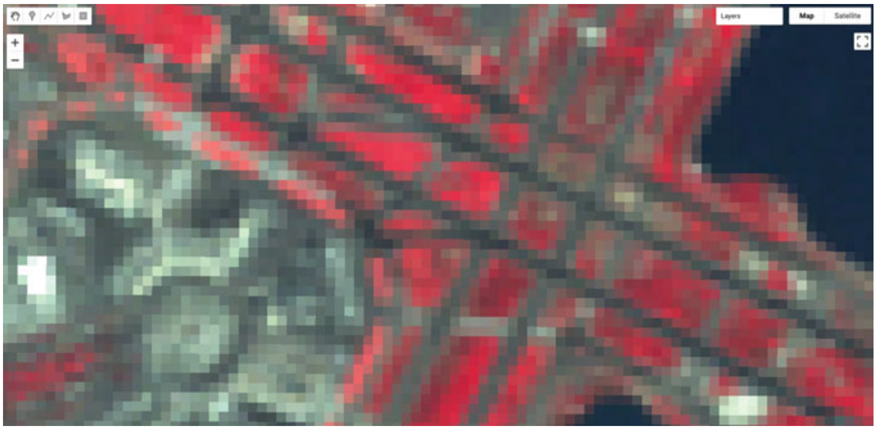


Fig. 4.10 Visualizing the TM imagery from the Landsat 5 satellite

```
// TM
// Filter TM imagery by location and date.
var tmImage = tm
    .filterBounds(Map.getCenter())
    .filterDate('1987-03-01', '1987-08-01')
    .first();

// Display the TM image as a false color composite.
Map.addLayer(tmImage, {
    bands: ['B4', 'B3', 'B2'],
    min: 0,
    max: 100
}, 'TM');
```

As we did for the MODIS data, let us check the scale. The scale is expressed in meters:

```
// Get the scale of the TM data from its projection:
var tmScale = tmImage.select('B4')
    .projection().nominalScale();

print('TM NIR scale:', tmScale);
```

MSI (on the Sentinel-2 satellites)
The MultiSpectral Instrument (MSI) flies aboard the Sentinel-2 satellites, which are operated by the European Space Agency. The red, green, blue, and near-infrared bands are captured at 10 m resolution, while other bands are captured at 20 and 30 m. The Sentinel-2A satellite was launched in 2015 and the 2B satellite was launched in 2017.

Search for “Sentinel 2 MSI” in the search bar, and add the “Sentinel-2 MSI: MultiSpectral Instrument, Level-1C” dataset to your workspace. Name it msi. In this dataset, the three bands for a color-IR image are called “B8” (infrared), “B4” (red), and “B3” (green).

```
// MSI
// Filter MSI imagery by location and date.
var msiImage = msi
    .filterBounds(Map.getCenter())
    .filterDate('2020-02-01', '2020-04-01')
    .first();

// Display the MSI image as a false color composite.
Map.addLayer(msiImage, {
    bands: ['B8', 'B4', 'B3'],
    min: 0,
    max: 2000
}, 'MSI');
```

Compare the MSI imagery with the TM and MODIS imagery, using the opacity slider. Notice how much more detail you can see on the airport terminal and surrounding landscape. The 10 m spatial resolution means that each pixel covers approximately 100 m^2 of the Earth's surface, a much smaller area than the TM imagery (900 m^2) or the MODIS imagery (0.25 km^2) (Fig. 4.11).

The extent of the MSI image displayed is also smaller than that for the other instruments we have looked at. Zoom out until you can see the entire San Francisco Bay. The MODIS image covers the entire globe, the TM image covers the entire San Francisco Bay and the surrounding area south toward Monterey, while the MSI image captures a much smaller area (Fig. 4.12).

Check the scale of the MSI instrument (in meters):

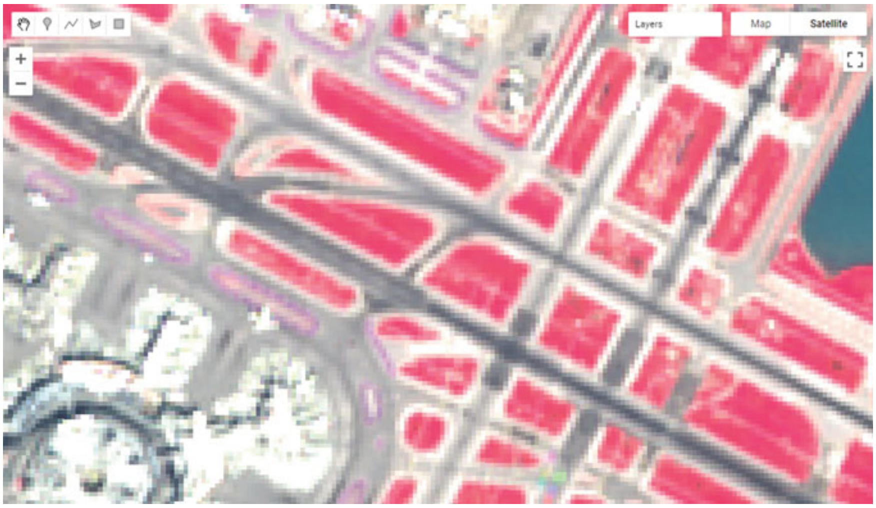


Fig. 4.11 Visualizing the MSI imagery

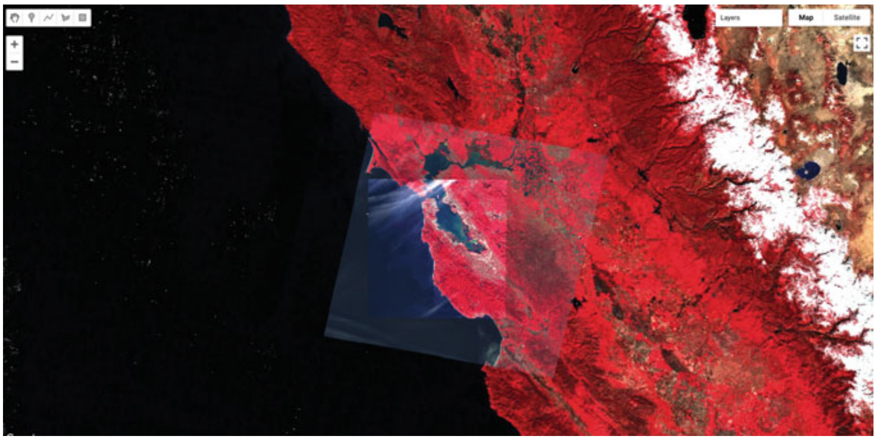


Fig. 4.12 Visualizing the image size for the MODIS, Landsat 5 (TM instrument), and Sentinel-2 (MSI instrument) datasets

```
// Get the scale of the MSI data from its projection:
var msiScale = msiImage.select('B8')
    .projection().nominalScale();
print('MSI scale:', msiScale);
```

NAIP

The National Agriculture Imagery Program (NAIP) is a US government program to acquire imagery over the continental USA using airborne sensors. Data are collected for each state approximately every three years. The imagery has a spatial resolution of 0.5–2 m, depending on the state and the date collected.

Search for “naip” and import the dataset for “NAIP: National Agriculture Imagery Program”. Name the import `naip`. In this dataset, the three bands for a color-IR image are called “N” (infrared), “R” (red), and “G” (green).

```
// NAIP
// Get NAIP images for the study period and region of
interest.
var naipImage = naip
    .filterBounds(Map.getCenter())
    .filterDate('2018-01-01', '2018-12-31')
    .first();

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

The NAIP imagery is even more spatially detailed than the Sentinel-2 MSI imagery. However, we can see that our one NAIP image does not totally cover the San Francisco airport. If you like, zoom out to see the boundaries of the NAIP image as we did for the Sentinel-2 MSI imagery (Fig. 4.13).

And get the scale, as we did before.

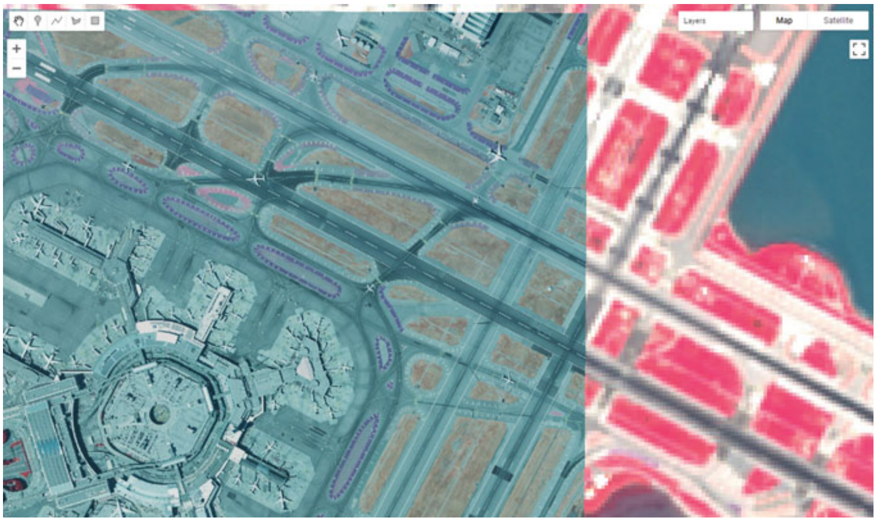


Fig. 4.13 NAIP color-IR composite over the San Francisco airport

```
// Get the NAIP resolution from the first image in the
mosaic.
var naipScale = naipImage.select('N')
    .projection().nominalScale();

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

Each of the datasets we have examined has a different spatial resolution. By comparing the different images over the same location in space, you have seen the differences between the large pixels of MODIS, the medium-sized pixels of TM (Landsat 5) and MSI (Sentinel-2), and the small pixels of the NAIP. Datasets with large-sized pixels are also called “coarse resolution”, those with medium-sized pixels are also called “moderate resolution”, and those with small-sized pixels are also called “fine resolution”.

Code Checkpoint F13a. The book’s repository contains a script that shows what your code should look like at this point.

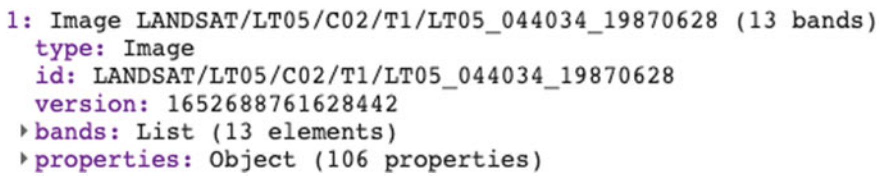
4.2.3 Section 3: Temporal Resolution

Temporal resolution refers to the revisit time or temporal cadence of a particular sensor’s image stream. Revisit time is the number of days between sequential visits of the satellite to the same location on the Earth’s surface. Think of this as the frequency of pixels in a time series at a given location.

Landsat

The Landsat satellites 5 and later are able to image a given location every 16 days. Let us use our existing `tm` dataset from Landsat 5. To see the time series of images at a location, you can filter an `ImageCollection` to an area and date range of interest and then `print` it. For example, to see the Landsat 5 images for three months in 1987, run the following code:

```
/////
// Explore Temporal Resolution
/////
// Use Print to see Landsat revisit time
print('Landsat-5 series:', tm
    .filterBounds(Map.getCenter())
    .filterDate('1987-06-01', '1987-09-01'));
```



```
1: Image LANDSAT/LT05/C02/T1/LT05_044034_19870628 (13 bands)
  type: Image
  id: LANDSAT/LT05/C02/T1/LT05_044034_19870628
  version: 1652688761628442
 ▸bands: List (13 elements)
 ▸properties: Object (106 properties)
```

Fig. 4.14 Landsat image name and feature properties

```
// Create a chart to see Landsat 5's 16 day revisit time.
var tmChart = ui.Chart.image.series({
    imageCollection: tm.select('B4').filterDate('1987-06-01',
       '1987-09-01'),
    region: sfoPoint
}).setSeriesNames(['NIR']);
```

Expand the features' property of the printed `ImageCollection` in the **Console** output to see a `List` of all the images in the collection. Observe that the date of each image is part of the filename (e.g., LANDSAT/LT05/C02/T1/LT05_044034_19870628) (Fig. 4.14).

However, viewing this list does not make it easy to see the temporal resolution of the dataset. We can use Earth Engine's plotting functionality to visualize the temporal resolution of different datasets. For each of the different temporal resolutions, we will create a per-pixel chart of the NIR band that we mapped previously. To do this, we will use the `ui.Chart.image.series` function.

The `ui.Chart.image.series` function requires you to specify a few things in order to calculate the point to chart for each time step. First, we filter the `ImageCollection` (you can also do this outside the function and then specify the ImageCollection directly). We select the B4 (near-infrared) band and then select three months by using `filterDate` on the `ImageCollection`. Next, we need to specify the location to chart; this is the region argument. We will use the `sfoPoint` variable we defined earlier.

```
// Create a chart to see Landsat 5's 16 day revisit time.
var tmChart = ui.Chart.image.series({
    imageCollection: tm.select('B4').filterDate('1987-06-
01',
        '1987-09-01'),
    region: sfoPoint
}).setSeriesNames(['NIR']);
```

By default, this function creates a trend line. It is difficult to see precisely when each image was collected, so let us create a specialized chart style that adds points for each observation.

```
// Define a chart style that will let us see the individual
dates.
var chartStyle = {
    hAxis: {
        title: 'Date'
    },
    vAxis: {
        title: 'NIR Mean'
    },
    series: {
        0: {
            lineWidth: 3,
            pointSize: 6
        }
    },
};

// Apply custom style properties to the chart.
tmChart.setOptions(chartStyle);

// Print the chart.
print('TM Chart', tmChart);
```

When you print the chart, it will have a point each time an image was collected by the TM instrument (Fig. 4.15). In the **Console**, you can move the mouse over the different points and see more information. Also note that you can expand the chart using the button in the upper-right-hand corner. We will see many more examples of charts, particularly in the chapters in Part IV.

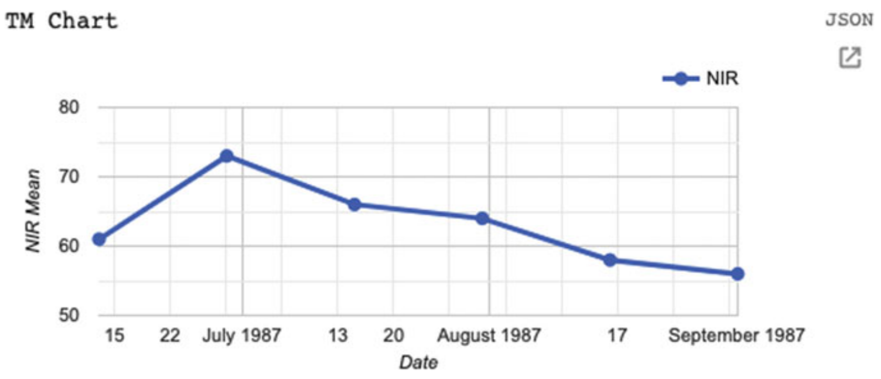


Fig. 4.15 A chart showing the temporal cadence or temporal resolution of the Landsat 5 TM instrument at the San Francisco airport

Sentinel-2
The Sentinel-2 program's two satellites are in coordinated orbits, so that each spot on Earth gets visited about every 5 days. Within Earth Engine, images from these two sensors are pooled in the same dataset. Let us create a chart using the MSI instrument dataset we have already imported.

```
// Sentinel-2 has a 5 day revisit time.
var msiChart = ui.Chart.image.series({
    imageCollection: msi.select('B8').filterDate('2020-06-
01',
        '2020-09-01'),
    region: sfoPoint
}).setSeriesNames(['NIR']);

// Apply the previously defined custom style properties to
the chart.
msiChart.setOptions(chartStyle);

// Print the chart.
print('MSI Chart', msiChart);
```

Compare this Sentinel-2 graph (Fig. 4.16) with the Landsat graph you just produced (Fig. 4.15). Both cover a period of six months, yet there are many more points through time for the Sentinel-2 satellite, reflecting the greater temporal resolution.

Code Checkpoint F13b. The book's repository contains a script that shows what your code should look like at this point.

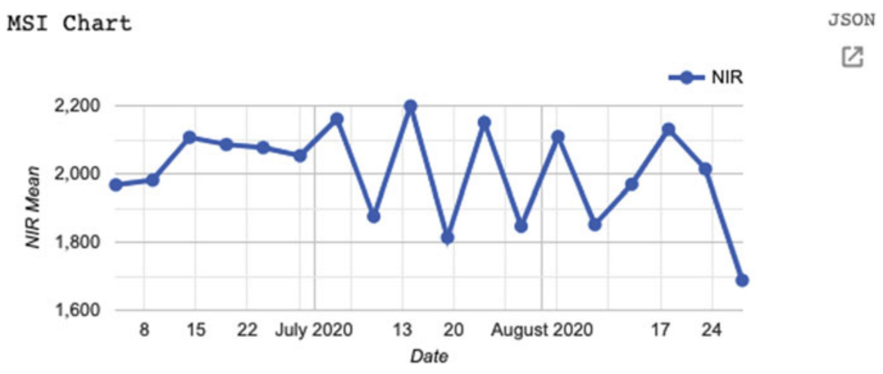


Fig. 4.16 A chart showing the temporal cadence or temporal resolution of the Sentinel-2 MSI instrument at the San Francisco airport

4.2.4 Section 4: Spectral Resolution

Spectral resolution refers to the number and width of spectral bands in which the sensor takes measurements. You can think of the width of spectral bands as the wavelength intervals for each band. A sensor that measures radiance in multiple bands is called a *multispectral* sensor (generally 3–10 bands), while a sensor with many bands (possibly hundreds) is called a *hyperspectral* sensor; however, these are relative terms without universally accepted definitions.

Let us compare the multispectral MODIS instrument with the hyperspectral Hyperion sensor aboard the EO-1 satellite, which is also available in Earth Engine.

MODIS
There is an easy way to check the number of bands in an image:

```
/////
// Explore spectral resolution
/////

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

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

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

Note that not all of the bands are spectral bands. As we did with the temporal resolution, let us graph the spectral bands to examine the spectral resolution. If you ever have questions about what the different bands in the band list are, remember that you can find this information by visiting the dataset information page in Earth Engine or the data or satellite's webpage.

```
// Graph the MODIS spectral bands (bands 11-17).

// Select only the reflectance bands of interest.
var reflectanceImage = modisImage.select(
    'sur_refl_b01',
    'sur_refl_b02',
    'sur_refl_b03',
    'sur_refl_b04',
    'sur_refl_b05',
    'sur_refl_b06',
    'sur_refl_b07'
);
```

As before, we will customize the chart to make it easier to read.

```
// Define an object of customization parameters for the
chart.
var options = {
    title: 'MODIS spectrum at SFO',
    hAxis: {
        title: 'Band'
    },
    vAxis: {
        title: 'Reflectance'
    },
    legend: {
        position: 'none'
    },
    pointSize: 3
};
```

And create a chart using the `ui.Chart.image.regions` function.

```
// Make the chart.
var modisReflectanceChart = ui.Chart.image.regions({
    image: reflectanceImage,
    regions: sfoPoint
}).setOptions(options);

// Display the chart.
print(modisReflectanceChart);
```

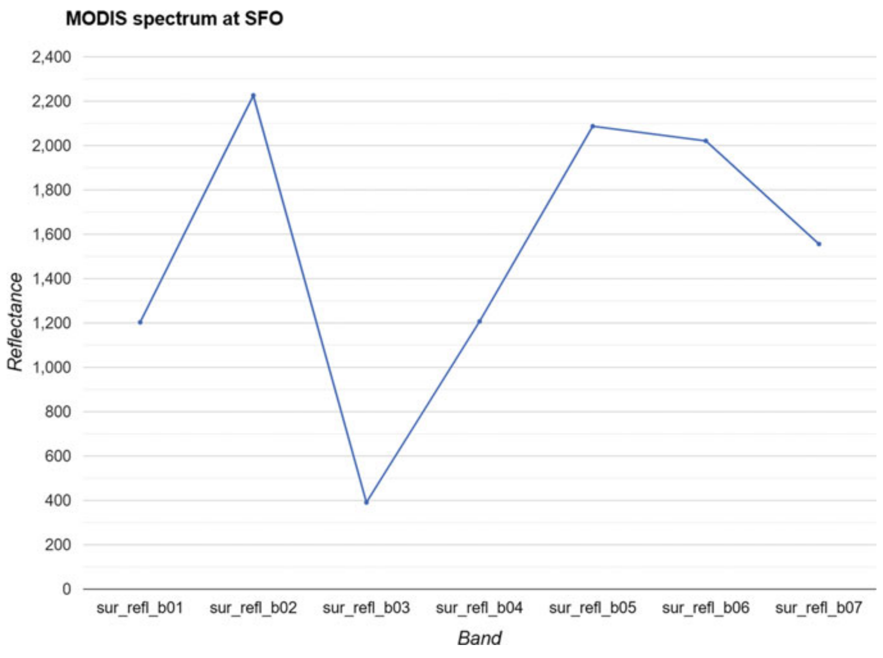


Fig. 4.17 Plot of TOA reflectance for MODIS

The resulting chart is shown in Fig. 4.17. Use the expand button in the upper right to see a larger version of the chart than the one printed to the **Console**.

EO-1

Now, let us compare MODIS with the EO-1 satellite's hyperspectral sensor. Search for "eo-1" and import the "EO-1 Hyperion Hyperspectral Imager" dataset. Name it `eo1`. We can look at the number of bands from the EO-1 sensor.

```
// Get the EO-1 band names as a ee.List
var eo1Image = eo1
    .filterDate('2015-01-01', '2016-01-01')
    .first();

// Extract the EO-1 band names.
var eo1Bands = eo1Image.bandNames();

// Print the list of band names.
print('EO-1 bands:', eo1Bands);
```

Examine the list of bands that are printed in the **Console**. Notice how many more bands the hyperspectral instrument provides.

Now let us create a reflectance chart as we did with the MODIS data.

```
// Create an options object for our chart.
var optionsEO1 = {
    title: 'EO1 spectrum',
    hAxis: {
        title: 'Band'
    },
    vAxis: {
        title: 'Reflectance'
    },
    legend: {
        position: 'none'
    },
    pointSize: 3
};

// Make the chart and set the options.
var eo1Chart = ui.Chart.image.regions({
    image: eo1Image,
    regions: ee.Geometry.Point([6.10, 81.12])
}).setOptions(optionsEO1);

// Display the chart.
print(eo1Chart);
```

The resulting chart is shown in Fig. 4.18. There are so many bands that their names only appear as "…"!

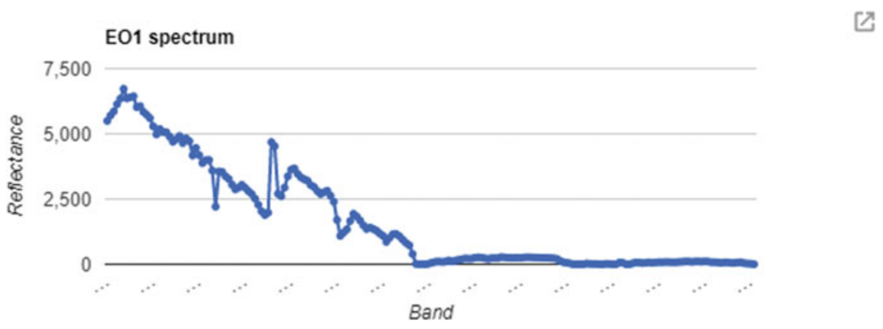


Fig. 4.18 Plot of TOA reflectance for EO-1 as displayed in the **Console**. Note the button to expand the plot in the upper-right-hand corner

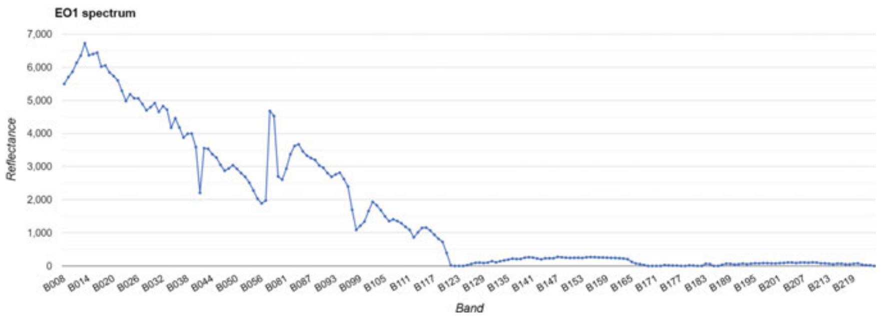


Fig. 4.19 Expanded plot of TOA reflectance for EO-1

If we click on the expand icon in the top right corner of the chart, it is a little easier to see the band identifiers, as shown in Fig. 4.19.

Compare this hyperspectral instrument chart with the multispectral chart we plotted above for MODIS.

Code Checkpoint F13c. The book's repository contains a script that shows what your code should look like at this point.

4.2.5 Section 5: Per-Pixel Quality

As you saw above, an image consists of many bands. Some of these bands contain spectral responses of Earth's surface, including the NIR, red, and green bands we examined in the spectral resolution section. What about the other bands? Some of these other bands contain valuable information, like pixel-by-pixel quality-control data.

For example, Sentinel-2 has a QA60 band, which contains the surface reflectance quality assurance information. Let us map it to inspect the values.

```
/////
// Examine pixel quality
/////

// Sentinel Quality Visualization.
var msiCloud = msi
    .filterBounds(Map.getCenter())
    .filterDate('2019-12-31', '2020-02-01')
    .first();
```

Use the **Inspector** tool to examine some of the values. You may see values of 0 (black), 1024 (gray), and 2048 (white). The QA60 band has values of 1024 for opaque clouds and 2048 for cirrus clouds. Compare the false-color image with the QA60 band to see these values. More information about how to interpret these complex values is given in Chap. 15, which explains the treatment of clouds.

Code Checkpoint F13d. The book's repository contains a script that shows what your code should look like at this point.

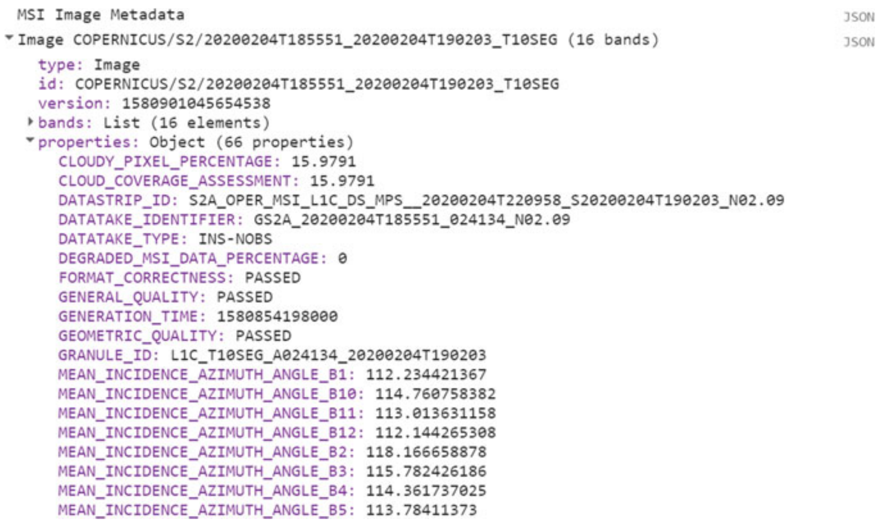
4.2.6 Section 6: Metadata

In addition to band imagery and per-pixel quality flags, Earth Engine allows you to access substantial amounts of metadata associated with an image. This can all be easily printed to the **Console** for a single image.

Let us examine the metadata for the Sentinel-2 MSI.

```
/////
// Metadata
/////
print('MSI Image Metadata', msiImage);
```

Examine the object you have created in the **Console** (Fig. 4.20). Expand the image name, then the `properties` object.



```
MSI Image Metadata                                                              JSON
▾Image COPERNICUS/S2/20200204T185551_20200204T190203_T10SEG (16 bands)          JSON
    type: Image
    id: COPERNICUS/S2/20200204T185551_20200204T190203_T10SEG
    version: 1580901045654538
   ▸bands: List (16 elements)
   ▾properties: Object (66 properties)
      CLOUDY_PIXEL_PERCENTAGE: 15.9791
      CLOUD_COVERAGE_ASSESSMENT: 15.9791
      DATASTRIP_ID: S2A_OPER_MSI_L1C_DS_MPS__20200204T220958_S20200204T190203_N02.09
      DATATAKE_IDENTIFIER: GS2A_20200204T185551_024134_N02.09
      DATATAKE_TYPE: INS-NOBS
      DEGRADED_MSI_DATA_PERCENTAGE: 0
      FORMAT_CORRECTNESS: PASSED
      GENERAL_QUALITY: PASSED
      GENERATION_TIME: 1580854198000
      GEOMETRIC_QUALITY: PASSED
      GRANULE_ID: L1C_T10SEG_A024134_20200204T190203
      MEAN_INCIDENCE_AZIMUTH_ANGLE_B1: 112.234421367
      MEAN_INCIDENCE_AZIMUTH_ANGLE_B10: 114.760758382
      MEAN_INCIDENCE_AZIMUTH_ANGLE_B11: 113.013631158
      MEAN_INCIDENCE_AZIMUTH_ANGLE_B12: 112.144265308
      MEAN_INCIDENCE_AZIMUTH_ANGLE_B2: 118.166658878
      MEAN_INCIDENCE_AZIMUTH_ANGLE_B3: 115.782426186
      MEAN_INCIDENCE_AZIMUTH_ANGLE_B4: 114.361737025
      MEAN_INCIDENCE_AZIMUTH_ANGLE_B5: 113.78411373
```

Fig. 4.20 Checking the "CLOUDY_PIXEL_PERCENTAGE" property in the metadata for Sentinel-2

The first entry is the CLOUDY_PIXEL_PERCENTAGE information. Distinct from the cloudiness flag attached to every pixel, this is an image-level summary assessment of the overall cloudiness in the image. In addition to viewing the value, you might find it useful to print it to the screen, for example, or to record a list of cloudiness values in a set of images. Metadata properties can be extracted from an image's properties using the get function and printed to the **Console**.

```
// Image-level Cloud info
var msiCloudiness = msiImage.get('CLOUDY_PIXEL_PERCENTAGE');

print('MSI CLOUDY_PIXEL_PERCENTAGE:', msiCloudiness);
```

Code Checkpoint F13e. The book's repository contains a script that shows what your code should look like at this point.

4.3 Synthesis

Assignment 1. Recall the plots of spectral resolution we created for MODIS and EO-1. Create a plot of spectral resolution for one of the other sensors described in this chapter. What are the bands called? What wavelengths of the electromagnetic spectrum do they correspond to?

Assignment 2. Recall how we extracted the spatial resolution and saved it to a variable. In your code, set the following variables to the scales of the bands shown in Table 4.1.

Assignment 3. Make this point in your code: `ee.Geometry.Point([-122.30144, 37.80215])`. How many `MYD09A1` images are there in 2017 at this point? Set a variable called `mod09ImageCount` with that value, and print it. How many Sentinel-2 MSI surface reflectance images are there in 2017 at this point? Set a variable called `msiImageCount` with that value, and print it.

Table 4.1 Three datasets and bands to use

Dataset	Band	Variable name
MODIS MYD09A1	sur_refl_b01	modisB01Scale
Sentinel-2 MSI	B5	msiB5Scale
NAIP	R	naipScale

4.4 Conclusion

A good understanding of the characteristics of your images is critical to your work in Earth Engine and the chapters going forward. You now know how to observe and query a variety of remote sensing datasets and can choose among them for your work. For example, if you are interested in change detection, you might require a dataset with spectral resolution including near-infrared imagery and a fine temporal resolution. For analyses at a continental scale, you may prefer data with a coarse spatial scale, while analyses for specific forest stands may benefit from a very fine spatial scale.

Reference

Fisher JRB, Acosta EA, Dennedy-Frank PJ et al (2018) Impact of satellite imagery spatial resolution on land use classification accuracy and modeled water quality. Remote Sens Ecol Conserv 4:137–149. https://doi.org/10.1002/rse2.61

Part II
Interpreting Images

Now that you know how images are viewed and what kinds of images exist in Earth Engine, how do we manipulate them? To gain the skills of interpreting images, you'll work with bands, combining values to form indices and masking unwanted pixels. Then, you'll learn some of the techniques available in Earth Engine for classifying images and interpreting the results.

5 Image Manipulation: Bands, Arithmetic, Thresholds, and Masks

Karen Dyson, Andréa Puzzi Nicolau, David Saah, and Nicholas Clinton

Overview
Once images have been identified in Earth Engine, they can be viewed in a wide array of band combinations for targeted purposes. For users who are already versed in remote sensing concepts, this chapter shows how to do familiar tasks on this platform; for those who are entirely new to such concepts, it introduces the idea of band combinations.

K. Dyson · A. P. Nicolau · D. Saah
Spatial Informatics Group, Pleasanton, California, USA
e-mail: kdyson@sig-gis.com

A. P. Nicolau
e-mail: apnicolau@sig-gis.com

K. Dyson
Dendrolytics, Seattle, Washington, USA

K. Dyson · A. P. Nicolau
SERVIR-Amazonia, Cali, Colombia

D. Saah (✉)
University of San Francisco, San Fransisco, California, USA
e-mail: dssaah@usfca.edu

N. Clinton
Google LLC, Inc, Mountain View, California, USA
e-mail: nclinton@google.com

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_5

Learning Outcomes

- Understanding what spectral indices are and why they are useful.
- Being introduced to a range of example spectral indices used for a variety of purposes.

Assumes you know how to

- Import images and image collections, filter, and visualize (Part I).

5.1 Introduction to Theory

Spectral indices are based on the fact that different objects and land covers on the Earth's surface reflect different amounts of light from the Sun at different wavelengths. In the visible part of the spectrum, for example, a healthy green plant reflects a large amount of green light while absorbing blue and red light—which is why it appears green to our eyes. Light also arrives from the Sun at wavelengths outside what the human eye can see, and there are large differences in reflectances between living and nonliving land covers and between different types of vegetation, both in the visible and outside the visible wavelengths. We visualized this earlier, in Chaps. 2 and 4 when we mapped color-infrared images (Fig. 5.1).

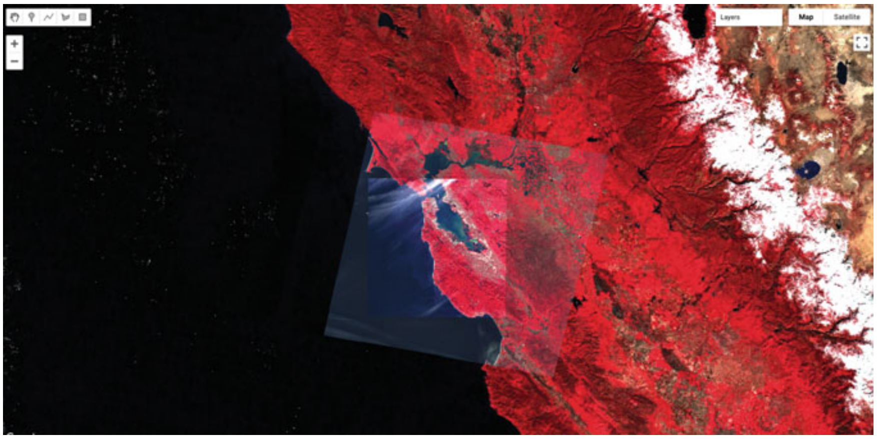


Fig. 5.1 Mapped color-IR images from multiple satellite sensors that we mapped in Chap. 4. The near-infrared spectrum is mapped as red, showing where there are high amounts of healthy vegetation

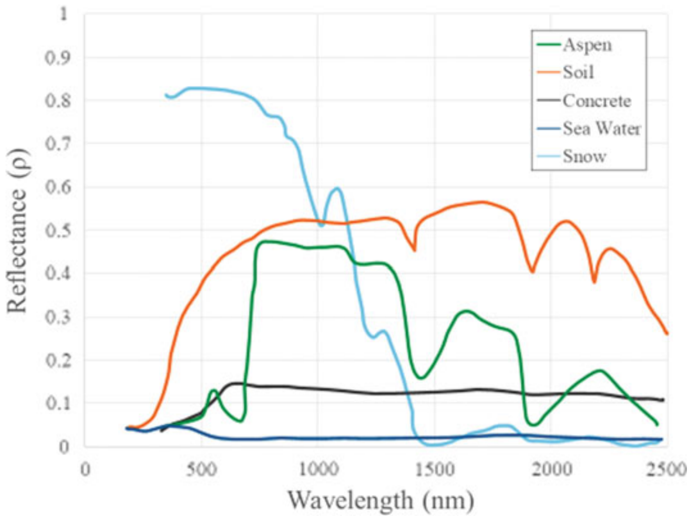


Fig. 5.2 Graph of the amount of reflectance for different objects on the Earth's surface at different wavelengths in the visible and infrared portions of the electromagnetic spectrum. 1 μm (μm) = 1000 nm (nm)

If we graph the amount of light (reflectance) at different wavelengths that an object or land cover reflects, we can visualize this more easily (Fig. 5.2). For example, look at the reflectance curves for soil and water in the graph below. Soil and water both have relatively low reflectance at wavelengths around 300 nm (ultraviolet and violet light). Conversely, at wavelengths above 700 nm (red and infrared light), soil has relatively high reflectance, while water has very low reflectance. Vegetation, meanwhile, generally reflects large amounts of near-infrared light, relative to other land covers.

Spectral indices use math to express how objects reflect light across multiple portions of the spectrum as a single number. Indices combine multiple bands, often with simple operations of subtraction and division, to create a single value across an image that is intended to help to distinguish particular land uses or land covers of interest. Using Fig. 5.2, you can imagine which wavelengths might be the most informative for distinguishing among a variety of land covers. We will explore a variety of calculations made from combinations of bands in the following sections.

Indices derived from satellite imagery are used as the basis of many remote sensing analyses. Indices have been used in thousands of applications, from detecting anthropogenic deforestation to examining crop health. For example, the growth of economically important crops such as wheat and cotton can be monitored throughout the growing season: Bare soil reflects more red wavelengths, whereas growing crops reflect more of the near-infrared (NIR) wavelengths. Thus, calculating a ratio of these two bands can help monitor how well crops are growing (Jackson and Huete 1991).

5.2 Practicum

5.2.1 Section 1: Band Arithmetic in Earth Engine

Many indices can be calculated using band arithmetic in Earth Engine. Band arithmetic is the process of adding, subtracting, multiplying, or dividing two or more bands from an image. Here, we will first do this manually and then show you some more efficient ways to perform band arithmetic in Earth Engine.

Arithmetic Calculation of NDVI

The red and near-infrared bands provide a lot of information about vegetation due to vegetation's high reflectance in these wavelengths. Take a look at Fig. 5.2 and note, in particular, that vegetation curves (graphed in green) have relatively high reflectance in the NIR range (approximately 750–900 nm). Also note that vegetation has low reflectance in the red range (approximately 630–690 nm), where sunlight is absorbed by chlorophyll. This suggests that if the red and near-infrared bands could be combined, they would provide substantial information about vegetation.

Soon after the launch of Landsat 1 in 1972, analysts worked to devise a robust single value that would convey the health of vegetation along a scale of −1 to 1. This yielded the NDVI, using the formula:

$$\mathrm{NDVI} = \frac{\mathrm{NIR} - \mathrm{red}}{\mathrm{NIR} + \mathrm{red}}, \tag{5.1}$$

where *NIR* and *red* refer to the brightness of each of those two bands. As seen in Chaps. 2 and 3, this brightness might be conveyed in units of reflectance, radiance, or digital number (DN); the NDVI is intended to give nearly equivalent values across platforms that use these wavelengths. The general form of this equation is called a "normalized difference"—the numerator is the "difference" and the denominator "normalizes" the value. Outputs for NDVI vary between −1 and 1. High amounts of green vegetation have values around 0.8–0.9. Absence of green leaves gives values near 0, and water gives values near −1.

To compute the NDVI, we will introduce Earth Engine's implementation of *band arithmetic*. Cloud-based band arithmetic is one of the most powerful aspects of Earth Engine, because the platform's computers are optimized for this type of heavy processing. Arithmetic on bands can be done even at planetary scale very quickly—an idea that was out of reach before the advent of cloud-based remote sensing. Earth Engine automatically partitions calculations across a large number of computers as needed and assembles the answer for display.

As an example, let us examine an image of San Francisco (Fig. 5.3).

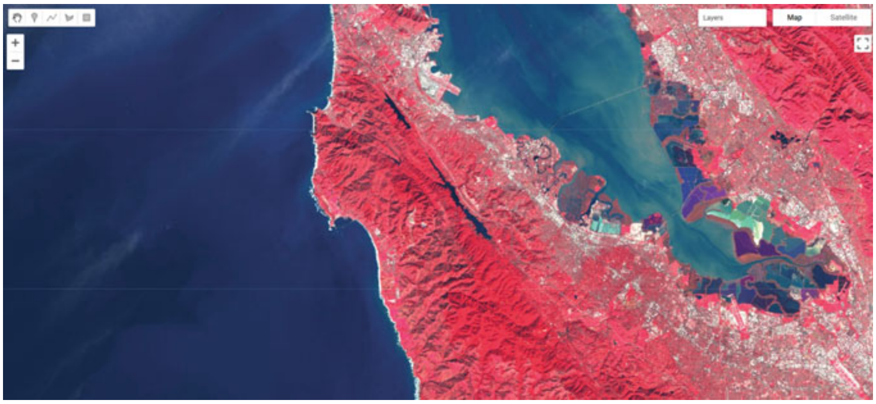


Fig. 5.3 False-color Sentinel-2 imagery of San Francisco and surroundings

```
/////
// Band Arithmetic
/////

// Calculate NDVI using Sentinel 2

// Import and filter imagery by location and date.
var sfoPoint = ee.Geometry.Point(-122.3774, 37.6194);
var sfoImage = ee.ImageCollection('COPERNICUS/S2')
    .filterBounds(sfoPoint)
    .filterDate('2020-02-01', '2020-04-01')
    .first();

// Display the image as a false color composite.
Map.centerObject(sfoImage, 11);
Map.addLayer(sfoImage, {
    bands: ['B8', 'B4', 'B3'],
    min: 0,
    max: 2000
}, 'False color');
```

The simplest mathematical operations in Earth Engine are the `add`, `subtract`, `multiply`, and `divide` methods. Let us select the near-infrared and red bands and use these operations to calculate NDVI for our image.

```
// Extract the near infrared and red bands.
var nir = sfoImage.select('B8');
var red = sfoImage.select('B4');

// Calculate the numerator and the denominator using
subtraction and addition respectively.
var numerator = nir.subtract(red);
var denominator = nir.add(red);

// Now calculate NDVI.
var ndvi = numerator.divide(denominator);

// Add the layer to our map with a palette.
var vegPalette = ['red', 'white', 'green'];
Map.addLayer(ndvi, {
    min: -1,
    max: 1,
    palette: vegPalette
}, 'NDVI Manual');
```

Examine the resulting index, using the **Inspector** to pick out pixel values in areas of vegetation and non-vegetation if desired (Fig. 5.4).

Using these simple arithmetic tools, you can build almost any index or develop and visualize your own. Earth Engine allows you to quickly and easily calculate and display the index across a large area.

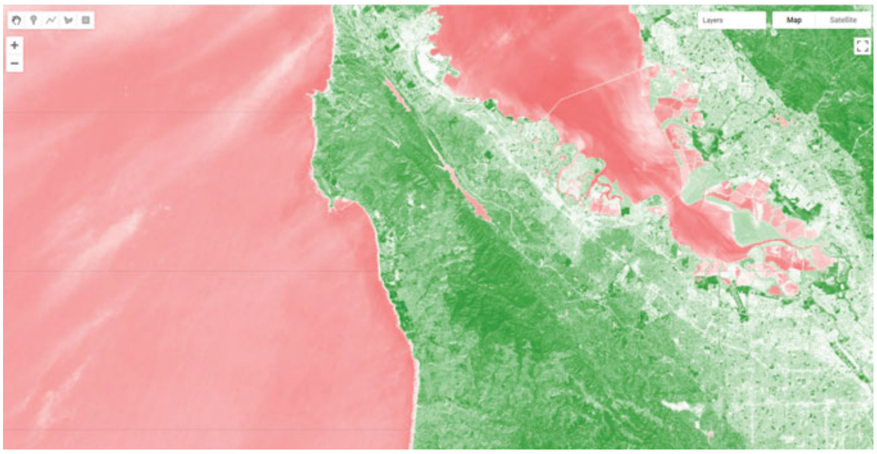


Fig. 5.4 NDVI calculated using Sentinel-2. Remember that outputs for NDVI vary between − 1 and 1. High amounts of green vegetation have values around 0.8–0.9. Absence of green leaves gives values near 0, and water gives values near − 1

Single-Operation Computation of Normalized Difference for NDVI
Normalized differences like NDVI are so common in remote sensing that Earth Engine provides the ability to do that particular sequence of subtraction, addition, and division in a single step, using the `normalizedDifference` method. This method takes an input image, along with bands you specify, and creates a normalized difference of those two bands. The NDVI computation previously created with band arithmetic can be replaced with one line of code:

```
// Now use the built-in normalizedDifference function to
achieve the same outcome.
var ndviND = sfoImage.normalizedDifference(['B8', 'B4']);
Map.addLayer(ndviND, {
    min: -1,
    max: 1,
    palette: vegPalette
}, 'NDVI normalizedDiff');
```

Note that the order in which you provide the two bands to `normalizedDifference` is important. We use *B*8, the near-infrared band, as the first parameter, and the red band *B*4 as the second. If your two computations of NDVI do not look identical when drawn to the screen, check to make sure that the order you have for the NIR and red bands is correct.

Using Normalized Difference for NDWI
As mentioned, the normalized difference approach is used for many different indices. Let us apply the same `normalizedDifference` method to another index.

The Normalized Difference Water Index (NDWI) was developed by Gao (1996) as an index of vegetation water content. The index is sensitive to changes in the liquid content of vegetation canopies. This means that the index can be used, for example, to detect vegetation experiencing drought conditions or differentiate crop irrigation levels. In dry areas, crops that are irrigated can be differentiated from natural vegetation. It is also sometimes called the Normalized Difference Moisture Index (NDMI). NDWI is formulated as follows:

$$\mathrm{NDWI} = \frac{\mathrm{NIR} - \mathrm{SWIR}}{\mathrm{NIR} + \mathrm{SWIR}}, \tag{5.2}$$

where NIR is near-infrared, centered near 860 nm (0.86 μm), and SWIR is shortwave infrared, centered near 1240 nm (1.24 μm).

Compute and display NDWI in Earth Engine using the `normalizedDifference` method. Remember that for Sentinel-2, *B*8 is the NIR band and *B*11 is the SWIR band (refer to Chaps. 2 and 4 to find information about imagery bands).

```
// Use normalizedDifference to calculate NDWI
var ndwi = sfoImage.normalizedDifference(['B8', 'B11']);
var waterPalette = ['white', 'blue'];
Map.addLayer(ndwi, {
    min: -0.5,
    max: 1,
    palette: waterPalette
}, 'NDWI');
```

Examine the areas of the map that NDVI identified as having a lot of vegetation. Notice which are more blue. This is vegetation that has higher water content (Fig. 5.5).

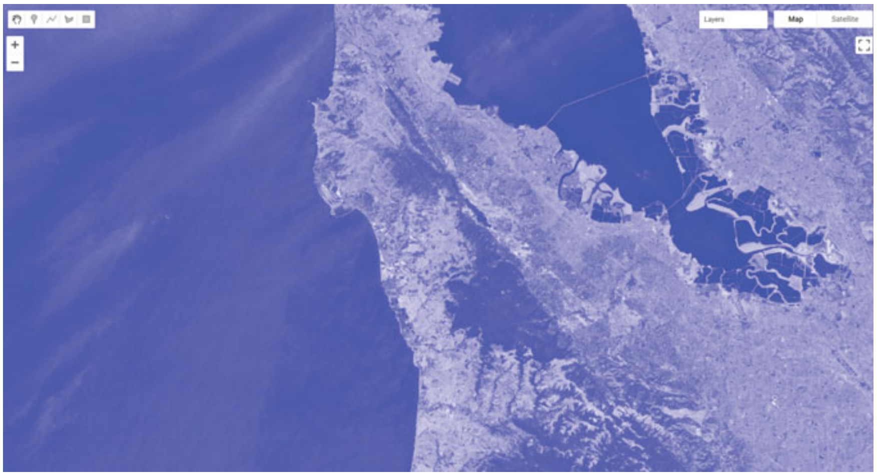


Fig. 5.5 NDWI displayed for Sentinel-2 over San Francisco

Code Checkpoint F20a. The book's repository contains a script that shows what your code should look like at this point.

5.2.2 Section 2: Thresholding, Masking, and Remapping Images

The previous section in this chapter discussed how to use band arithmetic to manipulate images. Those methods created new continuous values by combining bands within an image. This section uses logical operators to categorize band or index values to create a categorized image.

Implementing a Threshold

Implementing a threshold uses a number (the threshold value) and logical operators to help us partition the variability of images into categories. For example, recall our map of NDVI. High amounts of vegetation have NDVI values near 1 and non-vegetated areas are near 0. If we want to see what areas of the map have vegetation, we can use a threshold to generalize the NDVI value in each pixel as being either "no vegetation" or "vegetation". That is a substantial simplification, to be sure, but can help us to better comprehend the rich variation on the Earth's surface. This type of categorization may be useful if, for example, we want to look at the proportion of a city that is vegetated. Let us create a Sentinel-2 map of NDVI near Seattle, Washington, USA. Enter the code below in a new script (Fig. 5.6).

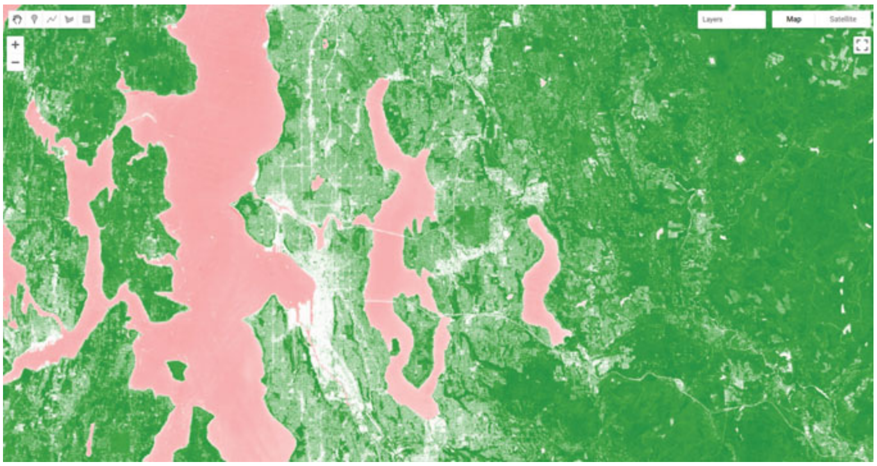


Fig. 5.6 NDVI image of Sentinel-2 imagery over Seattle, Washington, USA

```
// Create an NDVI image using Sentinel 2.
var seaPoint = ee.Geometry.Point(-122.2040, 47.6221);
var seaImage = ee.ImageCollection('COPERNICUS/S2')
    .filterBounds(seaPoint)
    .filterDate('2020-08-15', '2020-10-01')
    .first();

var seaNDVI = seaImage.normalizedDifference(['B8', 'B4']);

// And map it.
Map.centerObject(seaPoint, 10);
var vegPalette = ['red', 'white', 'green'];
Map.addLayer(seaNDVI,
    {
        min: -1,
        max: 1,
        palette: vegPalette
    },
    'NDVI Seattle');
```

Inspect the image. We can see that vegetated areas are darker green, while non-vegetated locations are white and water is pink. If we use the **Inspector** to query our image, we can see that parks and other forested areas have an NDVI over about 0.5. Thus, it would make sense to define areas with NDVI values greater than 0.5 as forested and those below that threshold as not forested.

Now, let us define that value as a threshold and use it to threshold our vegetated areas.

```
// Implement a threshold.
var seaVeg = seaNDVI.gt(0.5);

// Map the threshold.
Map.addLayer(seaVeg,
    {
        min: 0,
        max: 1,
        palette: ['white', 'green']
    },
    'Non-forest vs. Forest');
```

The `gt` method is from the family of Boolean operators—that is, `gt` is a function that performs a test in each pixel and returns the value 1 if the test evaluates to true, and 0 otherwise. Here, for every pixel in the image, it tests whether the

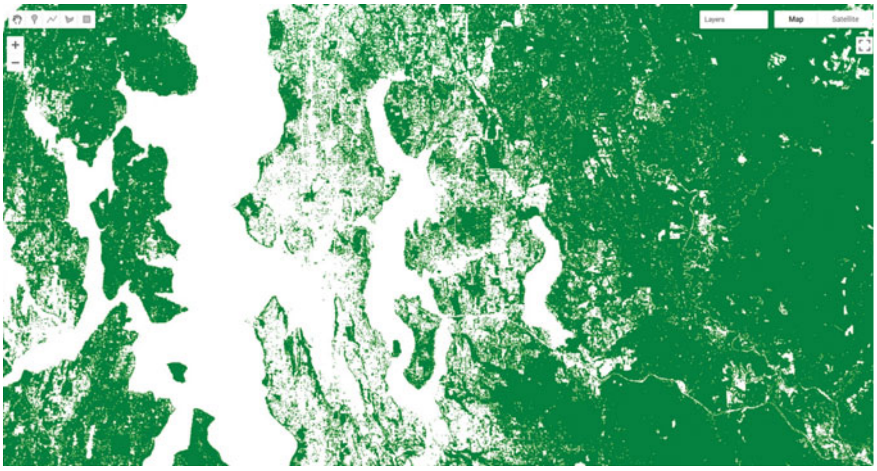


Fig. 5.7 Thresholded forest and non-forest image based on NDVI for Seattle, Washington, USA

NDVI value is greater than 0.5. When this condition is met, the layer `seaVeg` gets the value 1. When the condition is false, it receives the value 0 (Fig. 5.7).

Use the **Inspector** tool to explore this new layer. If you click on a green location, that NDVI should be greater than 0.5. If you click on a white pixel, the NDVI value should be equal to or less than 0.5.

Other operators in this Boolean family include less than (`lt`), less than or equal to (`lte`), equal to (`eq`), not equal to (`neq`), and greater than or equal to (`gte`) and more.

Building Complex Categorizations with .where

A binary map classifying NDVI is very useful. However, there are situations where you may want to split your image into more than two bins. Earth Engine provides a tool, the `where` method, that conditionally evaluates to true or false within each pixel depending on the outcome of a test. This is analogous to an *if* statement seen commonly in other languages. However, to perform this logic when programming for Earth Engine, we avoid using the JavaScript *if* statement. Importantly, JavaScript *if* commands are not calculated on Google's servers and can create serious problems when running your code—in effect, the servers try to ship all of the information to be executed to your own computer's browser, which is very underequipped for for such enormous tasks. Instead, we use the `where` clause for conditional logic.

Suppose instead of just splitting the forested areas from the non-forested areas in our NDVI, we want to split the image into likely water, non-forested, and forested areas. We can use `where` and thresholds of −0.1 and 0.5. We will start by creating an image using `ee.Image`. We then clip the new image so that it covers the same area as our `seaNDVI` layer (Fig. 5.8).

```
// Implement .where.
// Create a starting image with all values = 1.
var seaWhere = ee.Image(1)
    // Use clip to constrain the size of the new image.
    .clip(seaNDVI.geometry());

// Make all NDVI values less than -0.1 equal 0.
seaWhere = seaWhere.where(seaNDVI.lte(-0.1), 0);

// Make all NDVI values greater than 0.5 equal 2.
seaWhere = seaWhere.where(seaNDVI.gte(0.5), 2);

// Map our layer that has been divided into three classes.
Map.addLayer(seaWhere,
    {
        min: 0,
        max: 2,
        palette: ['blue', 'white', 'green']
    },
    'Water, Non-forest, Forest');
```

There are a few interesting things to note about this code that you may not have seen before. First, we are not defining a new variable for each `where` call. As a result, we can perform many `where` calls without creating a new variable each time and needing to keep track of them. Second, when we created the starting image, we set the value to 1. This means that we could easily set the bottom and top values with one `where` clause each. Finally, while we did not do it here, we can combine multiple `where` clauses using `and` and `or`. For example, we could identify pixels with an intermediate level of NDVI using `seaNDVI.gte(−0.1).and(seaNDVI.lt(0.5))`.

Masking Specific Values in an Image

Masking an image is a technique that removes specific areas of an image—those covered by the mask—from being displayed or analyzed. Earth Engine allows you to both view the current mask and update the mask (Fig. 5.9).

```
// Implement masking.
// View the seaVeg layer's current mask.
Map.centerObject(seaPoint, 9);
Map.addLayer(seaVeg.mask(), {}, 'seaVeg Mask');
```

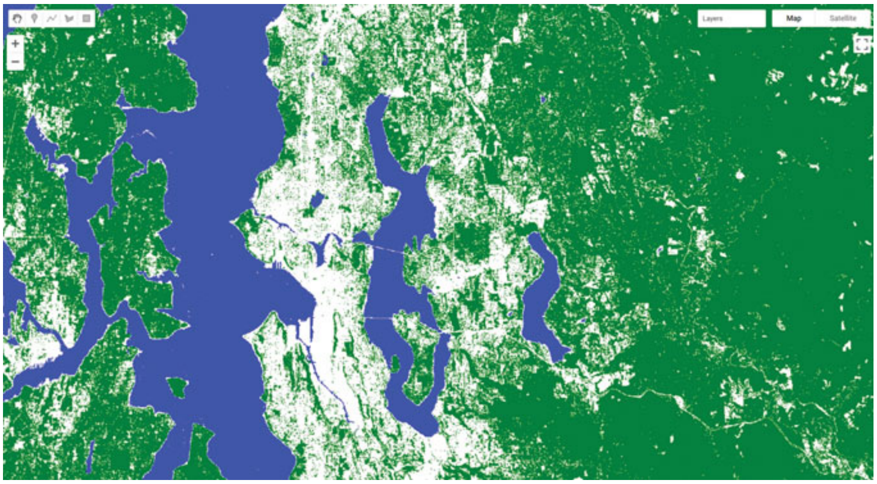


Fig. 5.8 Thresholded water, forest, and non-forest image based on NDVI for Seattle, Washington, USA

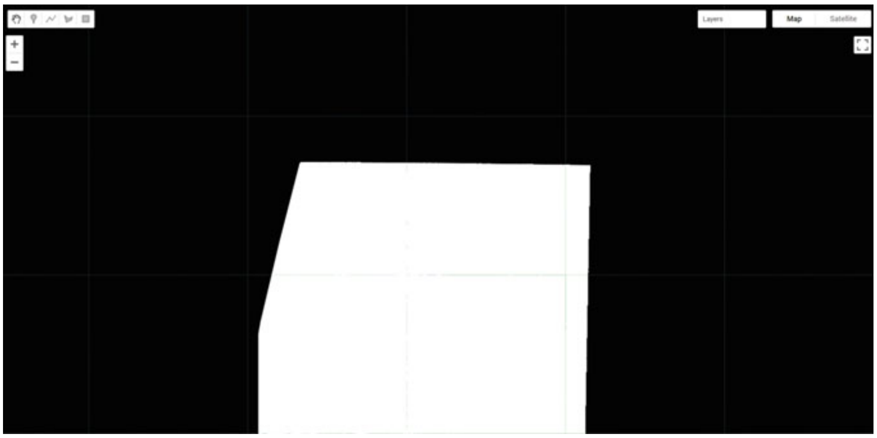


Fig. 5.9 Existing mask for the `seaVeg` layer we created previously

You can use the **Inspector** to see that the black area is masked and the white area has a constant value of 1. This means that data values are mapped and available for analysis within the white area only.

Now suppose we only want to display and conduct analyses in the forested areas. Let us mask out the non-forested areas from our image. First, we create a binary mask using the equals (`eq`) method.

```
// Create a binary mask of non-forest.
var vegMask = seaVeg.eq(1);
```

In making a mask, you set the values you want to see and analyze to be a number greater than 0. The idea is to set unwanted values to get the value of 0. Pixels that had 0 values become masked out (in practice, they do not appear on the screen at all) once we use the `updateMask` method to add these values to the existing mask.

```
// Update the seaVeg mask with the non-forest mask.
var maskedVeg = seaVeg.updateMask(vegMask);

// Map the updated Veg layer
Map.addLayer(maskedVeg,
    {
        min: 0,
        max: 1,
        palette: ['green']
    },
    'Masked Forest Layer');
```

Turn off all of the other layers. You can see how the `maskedVeg` layer now has masked out all non-forested areas (Fig. 5.10).

Map the updated mask for the layer and you can see why this is (Fig. 5.11).

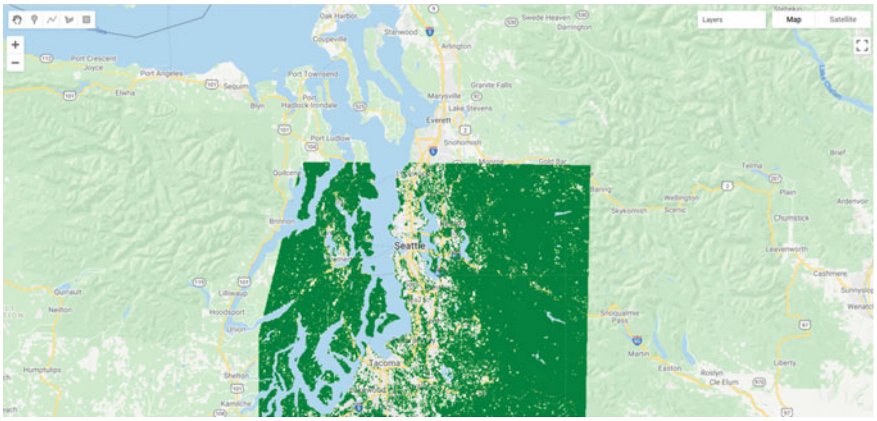


Fig. 5.10 Updated mask now displays only the forested areas. Non-forested areas are masked out and transparent

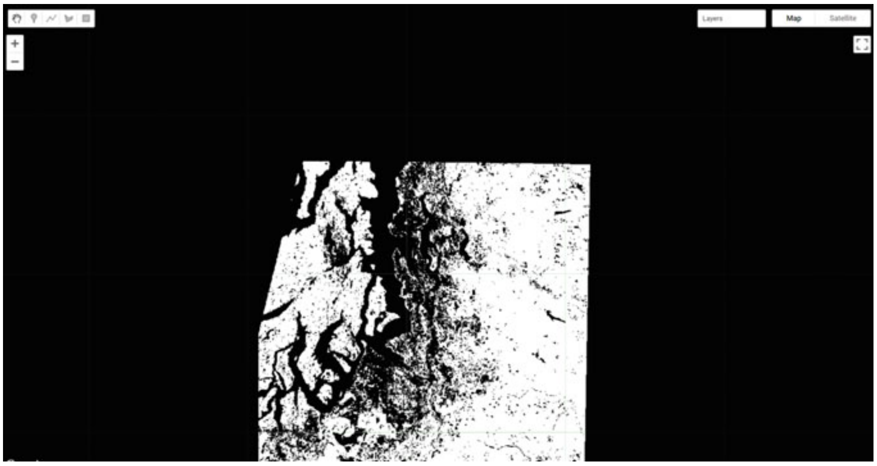


Fig. 5.11 Updated mask. Areas of non-forest are now masked out as well (black areas of the image)

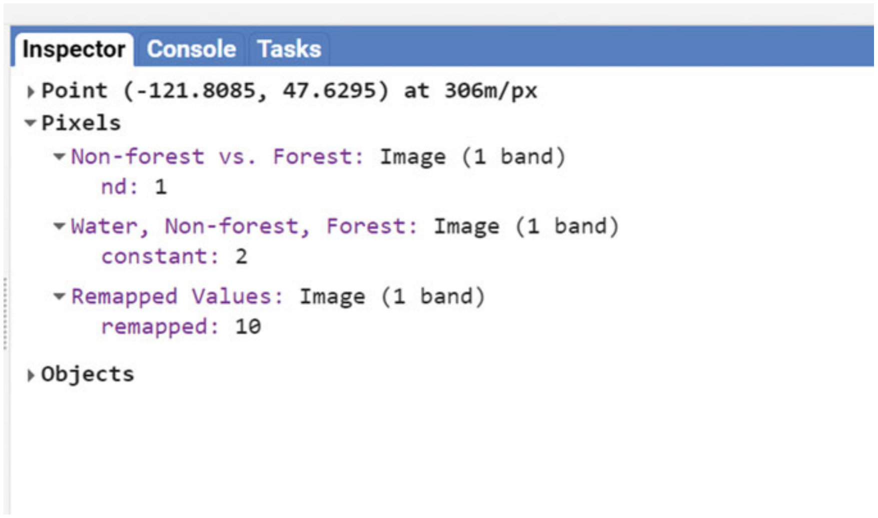
```
// Map the updated mask
Map.addLayer(maskedVeg.mask(), {}, 'maskedVeg Mask');
```

Remapping Values in an Image

Remapping takes specific values in an image and assigns them a different value. This is particularly useful for categorical datasets, including those you read about in Chap. 3 and those we have created earlier in this chapter.

Let us use the `remap` method to change the values for our `seaWhere` layer. Note that since we are changing the middle value to be the largest, we will need to adjust our palette as well.

```
// Implement remapping.
// Remap the values from the seaWhere layer.
var seaRemap = seaWhere.remap([0, 1, 2], // Existing values.
    [9, 11, 10]); // Remapped values.

Map.addLayer(seaRemap,
    {
        min: 9,
        max: 11,
        palette: ['blue', 'green', 'white']
    },
    'Remapped Values');
```



```
▸Point (-121.8085, 47.6295) at 306m/px
▾Pixels
  ▾Non-forest vs. Forest: Image (1 band)
     nd: 1
  ▾Water, Non-forest, Forest: Image (1 band)
     constant: 2
  ▾Remapped Values: Image (1 band)
     remapped: 10
▸Objects
```

Fig. 5.12 For forested areas, the remapped layer has a value of 10, compared with the original layer, which has a value of 2. You may have more layers in your **Inspector**

Use the inspector to compare values between our original `seaWhere` (displayed as Water, Non-Forest, Forest) and the `seaRemap`, marked as "Remapped Values". Click on a forested area and you should see that the Remapped Values should be 10, instead of 2 (Fig. 5.12).

Code Checkpoint F20b. The book's repository contains a script that shows what your code should look like at this point.

5.3 Synthesis

Assignment 1. In addition to vegetation indices and other land cover indices, you can use properties of different soil types to create geological indices (Drury 1987). The Clay Minerals Ratio (CMR) is one of these (Nath et al. 2019). This index highlights soils containing clay and alunite, which absorb radiation in the SWIR portion (2.0–2.3 μm) of the spectrum.

$$\mathrm{CMR} = \frac{\mathrm{SWIR\,1}}{\mathrm{SWIR\,2}}.$$

SWIR 1 should be in the 1.55–1.75 μm range, and SWIR 2 should be in the 2.08–2.35 μm range. Calculate and display CMR at the following point: `ee.Geometry.Point(−100.543, 33.456)`. Do not forget to use `Map.centerObject`.

We have selected an area of Texas known for its clay soils. Compare this with an area without clay soils (for example, try an area around Seattle or Tacoma, Washington, USA). Note that this index will also pick up roads and other paved areas.

Assignment 2. Calculate the Iron Oxide Ratio, which can be used to detect hydrothermally altered rocks (e.g., from volcanoes) that contain iron-bearing sulfides which have been oxidized (Segal 1982).

Here is the formula:

$$\mathrm{IOR} = \frac{\mathrm{Red}}{\mathrm{Blue}}.$$

Red should be the 0.63–0.69 µm spectral range and blue the 0.45–0.52 µm. Using Landsat 8, you can also find an interesting area to map by considering where these types of rocks might occur.

Assignment 3. Calculate the Normalized Difference Built-Up Index (NDBI) for the `sfoImage` used in this chapter.

The NDBI was developed by Zha et al. (2003) to aid in differentiating urban areas (e.g., densely clustered buildings and roads) from other land cover types. The index exploits the fact that urban areas, which generally have a great deal of impervious surface cover, reflect SWIR very strongly. If you like, refer back to Fig. 5.2.

The formula is:

$$\mathrm{NDBI} = \frac{\mathrm{SWIR} - \mathrm{NIR}}{\mathrm{SWIR} + \mathrm{NIR}}.$$

Using what we know about Sentinel-2 bands, compute NDBI and display it.

Bonus: Note that, NDBI is the negative of NDWI computed earlier. We can prove this by using the JavaScript *reverse* method to reverse the palette used for NDWI in Earth Engine. This method reverses the order of items in the JavaScript list. Create a new palette for NDBI using the reverse method and display the map. As a hint, here is code to use the reverse method.

```
var barePalette = waterPalette.reverse();
```

5.4 Conclusion

In this chapter, you learned how to select multiple bands from an image and calculate indices. You also learned about thresholding values in an image, slicing them into multiple categories using thresholds. It is also possible to work with one set of class numbers and remap them quickly to another set. Using these techniques, you have some of the basic tools of image manipulation. In subsequent chapters, you will encounter more complex and specialized image manipulation techniques, including pixel-based image transformations (Chap. 9), neighborhood-based image transformations (Chap. 10), and object-based image analysis (Chap. 11).

References

Drury SA (1987) Image interpretation in geology
Gao BC (1996) NDWI—a normalized difference water index for remote sensing of vegetation liquid water from space. Remote Sens Environ 58:257–266. https://doi.org/10.1016/S0034-4257(96)00067-3
Jackson RD, Huete AR (1991) Interpreting vegetation indices. Prev Vet Med 11:185–200. https://doi.org/10.1016/S0167-5877(05)80004-2
Nath B, Niu Z, Mitra AK (2019) Observation of short-term variations in the clay minerals ratio after the 2015 Chile great earthquake (8.3 Mw) using Landsat 8 OLI data. J Earth Syst Sci 128:1–21. https://doi.org/10.1007/s12040-019-1129-2
Segal D (1982) Theoretical basis for differentiation of ferric-iron bearing minerals, using Landsat MSS data. In: Proceedings of symposium for remote sensing of environment, 2nd thematic conference on remote sensing for exploratory geology, Fort Worth, TX, pp 949–951
Zha Y, Gao J, Ni S (2003) Use of normalized difference built-up index in automatically mapping urban areas from TM imagery. Int J Remote Sens 24(3):583–594

Interpreting an Image: Classification

6

Andréa Puzzi Nicolau, Karen Dyson, David Saah, and Nicholas Clinton

Overview
Image classification is a fundamental goal of remote sensing. It takes the user from viewing an image to labeling its contents. This chapter introduces readers to the concept of classification and walks users through the many options for image classification in Earth Engine. You will explore the processes of training data collection, classifier selection, classifier training, and image classification.

Learning Outcomes

- Running a classification in Earth Engine.
- Understanding the difference between supervised and unsupervised classification.

A. P. Nicolau · K. Dyson · D. Saah (✉)
Spatial Informatics Group, Pleasanton, California, USA
e-mail: dssaah@usfca.edu

A. P. Nicolau
e-mail: apnicolau@sig-gis.com

K. Dyson
e-mail: kdyson@sig-gis.com

A. P. Nicolau · K. Dyson
SERVIR-Amazonia, Cali, Colombia

K. Dyson
Dendrolytics, Seattle, Washington, USA

D. Saah
University of San Francisco, San Francisco, California, USA

N. Clinton
Google LLC, Mountain View, California, USA
e-mail: nclinton@google.com

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_6

- Learning how to use Earth Engine geometry drawing tools.
- Learning how to collect sample data in Earth Engine.
- Learning the basics of the hexadecimal numbering system.

Assumes you know how to

- Import images and image collections, filter, and visualize (Part I).
- Understand bands and how to select them (Chaps. 3 and 5).

6.1 Introduction to Theory

Classification is addressed in a broad range of fields, including mathematics, statistics, data mining, machine learning, and more. For a deeper treatment of classification, interested readers may see some of the following suggestions: Witten et al. (2011), Hastie et al. (2009), Goodfellow et al. (2016), Gareth et al. (2013), Géron (2019), Müller and Guido (2016), or Witten et al. (2005). In classification, the target variable is categorical or discrete, meaning it has a finite number of categories. Examples include predicting whether an email is spam or not, classifying images into different objects, or determining whether a customer will churn or not. On the other hand, regression predicts continuous variables, where the target variable can take on any value within a range. Examples include predicting a person's income, the price of a house, or the temperature. Chapter 8 covers the concept and application of image regression.

In remote sensing, image classification is an attempt to categorize all pixels in an image into a finite number of labeled land cover and/or land use classes. The resulting classified image is a simplified thematic map derived from the original image (Fig. 6.1). Land cover and land use information is essential for many environmental and socioeconomic applications, including natural resource management, urban planning, biodiversity conservation, agricultural monitoring, and carbon accounting.

Image classification techniques for generating land cover and land use information have been in use since the 1980s (Li et al. 2014). Here, we will cover the concepts of pixel-based supervised and unsupervised classifications, testing out different classifiers. Chapter 11 covers the concept and application of object-based classification.

6.2 Practicum

It is important to define land use and land cover. Land cover relates to the physical characteristics of the surface: Simply put, it documents whether an area of the Earth's surface is covered by forests, water, impervious surfaces, etc. Land use

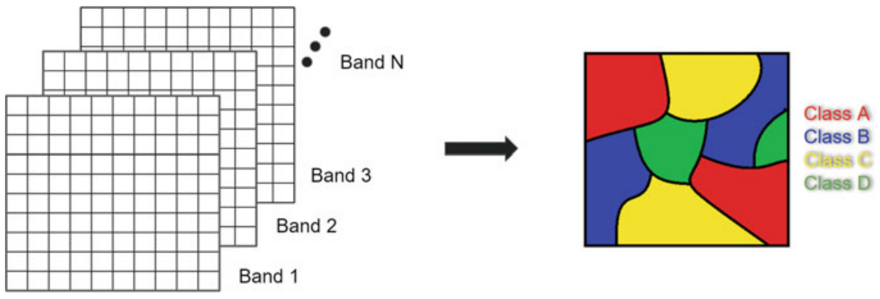


Fig. 6.1 Image classification concept

refers to how this land is being used by people. For example, herbaceous vegetation is considered a land cover but can indicate different land uses: The grass in a pasture is an agricultural land use, whereas the grass in an urban area can be classified as a park.

6.2.1 Section 1: Supervised Classification

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.

Supervised classification uses a training dataset with known labels and representing the spectral characteristics of each land cover class of interest to "supervise" the classification. The overall approach of a supervised classification in Earth Engine is summarized as follows:

1. Get a scene.
2. Collect training data.
3. Select and train a classifier using the training data.
4. Classify the image using the selected classifier.

We will begin by creating training data manually, based on a clear Landsat image (Fig. 6.2). Copy the code block below to define your Landsat 8 scene variable and add it to the map. We will use a point in Milan, Italy, as the center of the area for our image classification.

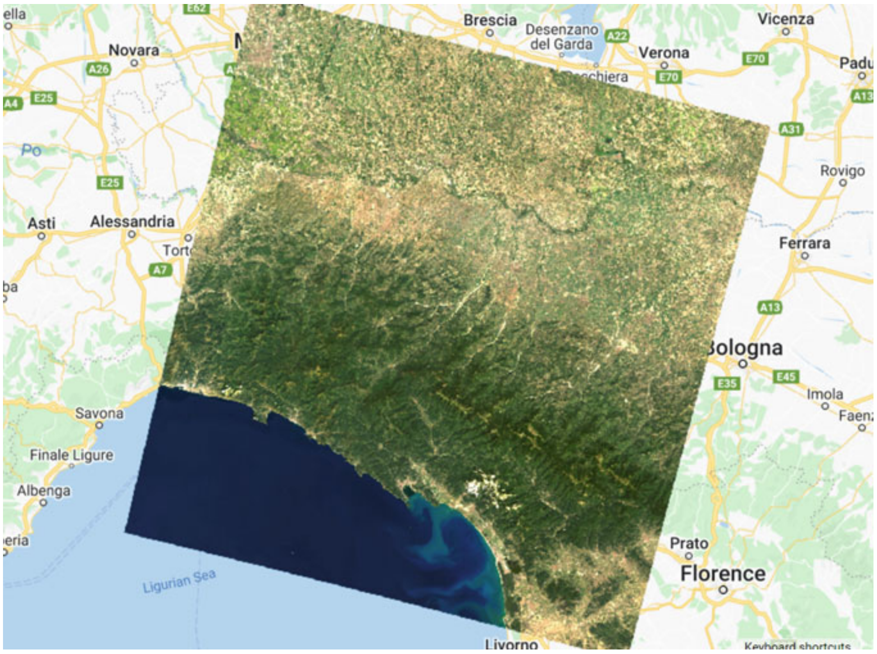


Fig. 6.2 Landsat image

```
// Create an Earth Engine Point object over Milan.
var pt = ee.Geometry.Point([9.453, 45.424]);

// Filter the Landsat 8 collection and select the least
cloudy image.
var landsat = ee.ImageCollection('LANDSAT/LC08/C02/T1_L2')
    .filterBounds(pt)
    .filterDate('2019-01-01', '2020-01-01')
    .sort('CLOUD_COVER')
    .first();

// Center the map on that image.
Map.centerObject(landsat, 8);

// Add Landsat image to the map.
var visParams = {
    bands: ['SR_B4', 'SR_B3', 'SR_B2'],
    min: 7000,
    max: 12000
};
Map.addLayer(landsat, visParams, 'Landsat 8 image');
```

Using the **Geometry Tools**, we will create points on the Landsat image that represent land cover classes of interest to use as our training data. We'll need to do two things: (1) identify where each land cover occurs on the ground, and (2) label the points with the proper class number. For this exercise, we will use the classes and codes given in Table 6.1.

In the **Geometry Tools**, click on the marker option (Fig. 6.3). This will create a point geometry which will show up as an import named "geometry." Click on the gear icon to configure this import.

We will start by collecting forest points, so name the import `forest`. Import it as a `FeatureCollection`, and then click + **Property**. Name the new property "class" and give it a value of 0 (Fig. 6.4). We can also choose a color to represent this class. For a forest class, it is natural to choose a green color. You can choose the color you prefer by clicking on it, or, for more control, you can use a hexadecimal value.

Hexadecimal values are used throughout the digital world to represent specific colors across computers and operating systems. They are specified by six values arranged in three pairs, with one pair each for the red, green, and blue brightness values. If you're unfamiliar with hexadecimal values, imagine for a moment that colors were specified in pairs of base 10 numbers instead of pairs of base 16. In that case, a bright pure red value would be "990000"; a bright pure green value would be "009900"; and a bright pure blue value would be "000099". A value like "501263" would be a mixture of the three colors, not especially bright, having roughly equal amounts of blue and red, and much less green: A color that would be a shade of purple. To create numbers in the hexadecimal system, which might

Table 6.1 Land cover classes

Class	Class code
Forest	0
Developed	1
Water	2
Herbaceous	3

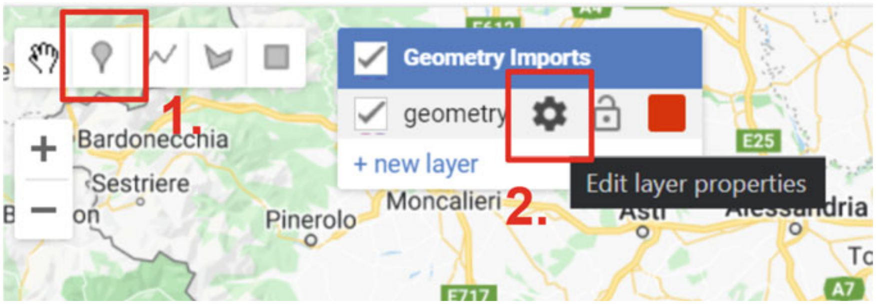
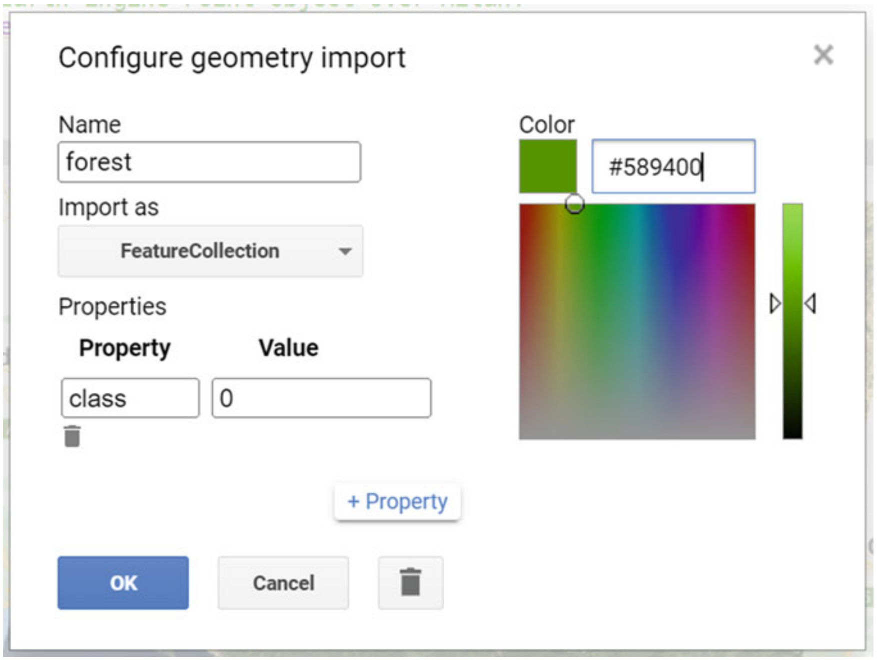


Fig. 6.3 Creating a new layer in the **Geometry Imports**


Configure geometry import

Name: forest

Import as: FeatureCollection

Properties

Property	Value
class	0

Color: #589400

Fig. 6.4 Edit geometry layer properties

feel entirely natural if humans had evolved to have 16 fingers, sixteen “digits” are needed: A base 16 counter goes 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, *A*, *B*, *C*, *D*, *E*, *F*, then 10, 11, and so on. Given that counting framework, the number “FF” is like “99” in base 10: the largest two-digit number. The hexadecimal color used for coloring the letters of the word `FeatureCollection` in this book, a color with roughly equal amounts of blue and red, and much less green, is “7F1FA2”.

Returning to the coloring of the `forest` points, the hexadecimal value “589400” is a little bit of red, about twice as much green, and no blue: The deep green is shown in Fig. 6.4. Enter that value, with or without the “#” in front and click **OK** after finishing the configuration.

Now, in the **Geometry Imports**, we will see that the import has been renamed `forest`. Click on it to activate the drawing mode (Fig. 6.5) in order to start collecting `forest` points.

Now, start collecting points over forested areas (Fig. 6.6). Zoom in and out as needed. You can use the satellite basemap to assist you, but the basis of your collection should be the Landsat image. Remember that the more points you collect, the more the classifier will learn from the information you provide. For now, let us set a goal to collect 25 points per class. Click **Exit** next to **Point drawing** (Fig. 6.5) when finished.

Repeat the same process for the other classes by creating new layers (Fig. 6.7). Don’t forget to import using the `FeatureCollection` option as mentioned

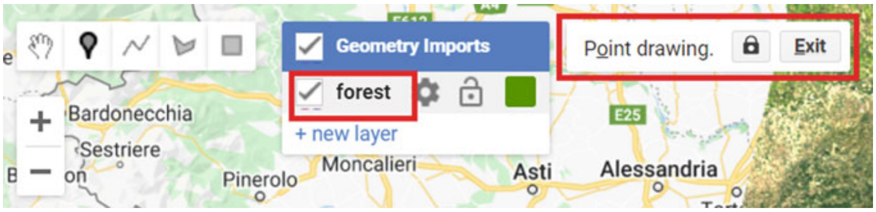


Fig. 6.5 Activate forest layer to start collection

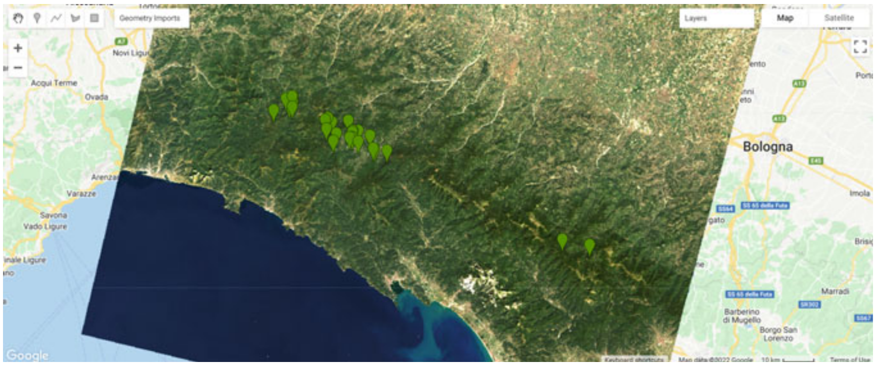


Fig. 6.6 Forest points

above. For the `developed` class, collect points over urban areas. For the `water` class, collect points over the Ligurian Sea and also look for other bodies of water, like rivers. For the `herbaceous` class, collect points over agricultural fields. Remember to set the "class" property for each class to its corresponding code (see Table 6.1) and click **Exit** once you finalize collecting points for each class as mentioned above. We will be using the following hexadecimal colors for the other classes: #FF0000 for `developed`, #1A11FF for `water`, and #D0741E for `herbaceous`.

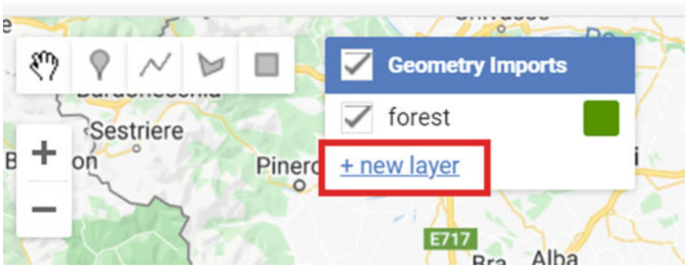


Fig. 6.7 New layer option in **Geometry Imports**

You should now have four FeatureCollection imports named forest, developed, water, and herbaceous (Fig. 6.8).

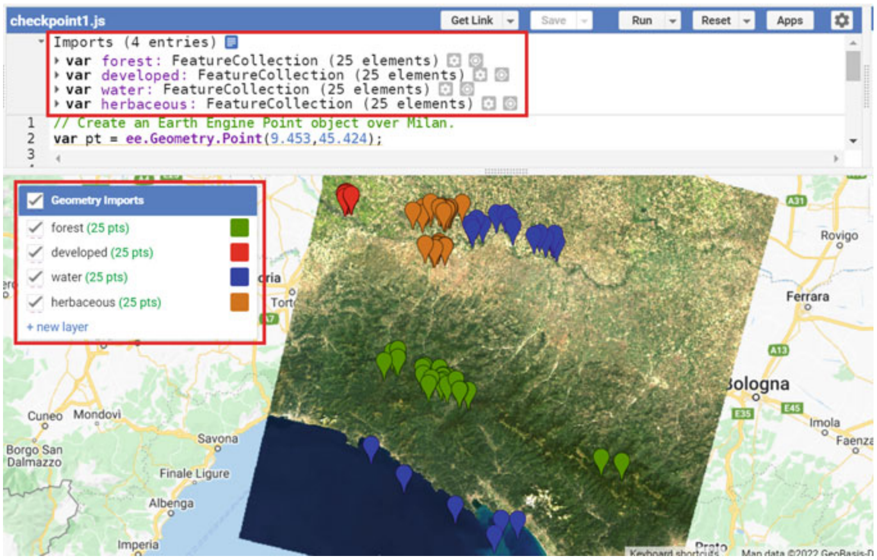
Code Checkpoint F21a. The book's repository contains a script that shows what your code should look like at this point.

If you wish to have the exact same results demonstrated in this chapter from now on, continue beginning with this Code Checkpoint. If you use the points collected yourself, the results may vary from this point forward.

The next step is to combine all the training feature collections into one. Copy and paste the code below to combine them into one FeatureCollection called trainingFeatures. Here, we use the flatten method to avoid having a collection of feature collections—we want individual features within our FeatureCollection.

```
// Combine training feature collections.
var trainingFeatures = ee.FeatureCollection([
    forest, developed, water, herbaceous
]).flatten();
```

Note: Alternatively, you could use an existing set of reference data. For example, the European Space Agency (ESA) WorldCover dataset is a global map of land use and land cover derived from ESA's Sentinel-2 imagery at 10 m resolution. With existing datasets, we can randomly place points on pixels classified



```
Imports (4 entries)
▸ var forest: FeatureCollection (25 elements)
▸ var developed: FeatureCollection (25 elements)
▸ var water: FeatureCollection (25 elements)
▸ var herbaceous: FeatureCollection (25 elements)
// Create an Earth Engine Point object over Milan.
var pt = ee.Geometry.Point(9.453,45.424);
```

Geometry Imports	
✓ forest (25 pts)	🟩
✓ developed (25 pts)	🟥
✓ water (25 pts)	🟦
✓ herbaceous (25 pts)	🟧
+ new layer	

Fig. 6.8 Example of training points

as the classes of interest (if you are curious, you can explore the Earth Engine documentation to learn about the `ee.Image.stratifiedSample` and the `ee.FeatureCollection.randomPoints` methods). The drawback is that these global datasets will not always contain the specific classes of interest for your region, or may not be entirely accurate at the local scale. Another option is to use samples that were collected in the field (e.g., GPS points). In Chap. 22, you will see how to upload your own data as Earth Engine assets.

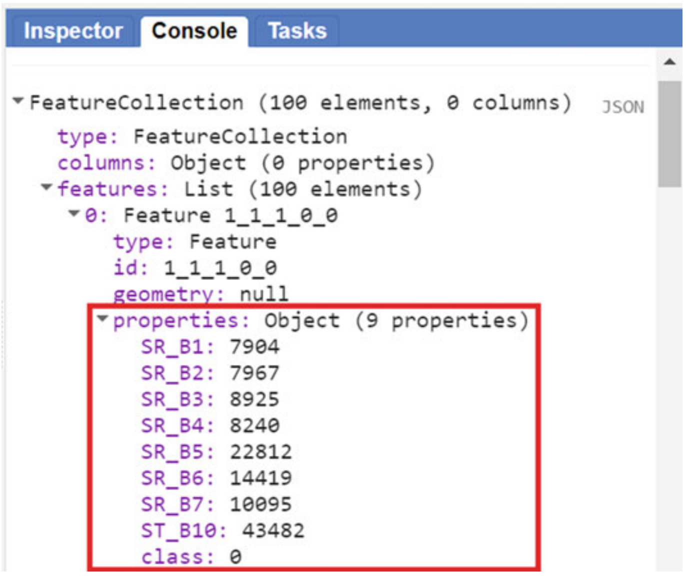
In the combined `FeatureCollection`, each `Feature` point should have a property called "class." The class values are consecutive integers from 0 to 3 (you could verify that this is true by printing `trainingFeatures` and checking the properties of the features).

Now that we have our training points, copy and paste the code below to extract the band information for each class at each point location. First, we define the prediction bands to extract different spectral and thermal information from different bands for each class. Then, we use the `sampleRegions` method to sample the information from the Landsat image at each point location. This method requires information about the `FeatureCollection` (our reference points), the property to extract ("class"), and the pixel scale (in meters).

```
// Define prediction bands.
var predictionBands = [
    'SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6',
'SR_B7','ST_B10'
];

// Sample training points.
var classifierTraining = landsat.select(predictionBands)
    .sampleRegions({
        collection: trainingFeatures,
        properties: ['class'],
        scale: 30
    });
```

You can check whether the `classifierTraining` object extracted the properties of interest by printing it and expanding the first feature. You should see the band and class information (Fig. 6.9).

Now we can choose a classifier. The choice of classifier is not always obvious, and there are many options from which to pick—you can quickly expand the `ee.Classifier` object under **Docs** to get an idea of how many options we have for image classification. Therefore, we will be testing different classifiers and comparing their results. We will start with a Classification and Regression Tree (CART) classifier, a well-known classification algorithm (Fig. 6.10) that has been around for decades.



```
▾FeatureCollection (100 elements, 0 columns)
   type: FeatureCollection
   columns: Object (0 properties)
  ▾features: List (100 elements)
    ▾0: Feature 1_1_1_0_0
       type: Feature
       id: 1_1_1_0_0
       geometry: null
      ▾properties: Object (9 properties)
         SR_B1: 7904
         SR_B2: 7967
         SR_B3: 8925
         SR_B4: 8240
         SR_B5: 22812
         SR_B6: 14419
         SR_B7: 10095
         ST_B10: 43482
         class: 0
```

Fig. 6.9 Example of extracted band information for one point of class 0 (forest)

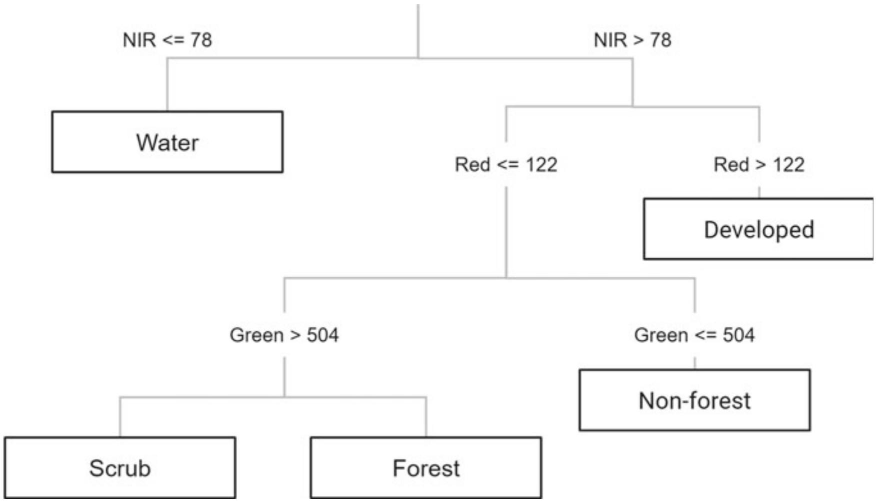


Fig. 6.10 Example of a decision tree for satellite image classification. Values and classes are hypothetical

Copy and paste the code below to instantiate a CART classifier (ee.Classifier.smileCart) and train it.

```
//////////////// CART Classifier ///////////////////

// Train a CART Classifier.
var classifier = ee.Classifier.smileCart().train({
    features: classifierTraining,
    classProperty: 'class',
    inputProperties: predictionBands
});
```

Essentially, the classifier contains the mathematical rules that link labels to spectral information. If you print the variable classifier and expand its properties, you can confirm the basic characteristics of the object (bands, properties, and classifier being used). If you print classifier.explain, you can find a property called “tree” that contains the decision rules.

After training the classifier, copy and paste the code below to classify the Landsat image and add it to the **Map**.

```
// Classify the Landsat image.
var classified =
landsat.select(predictionBands).classify(classifier);

// Define classification image visualization parameters.
var classificationVis = {
    min: 0,
    max: 3,
    palette: ['589400', 'ff0000', '1a11ff', 'd0741e']
};

// Add the classified image to the map.
Map.addLayer(classified, classificationVis, 'CART
classified');
```

Note that, in the visualization parameters, we define a palette parameter which in this case represents colors for each pixel value (0–3, our class codes). We use the same hexadecimal colors used when creating our training points for each class. This way, we can associate a color with a class when visualizing the classified image in the **Map**.

Inspect the result: Activate the Landsat composite layer and the satellite basemap to overlay with the classified images (Fig. 6.11). Change the layers’ transparency to inspect some areas. What do you notice? The result might not look very satisfactory in some areas (e.g., confusion between developed and

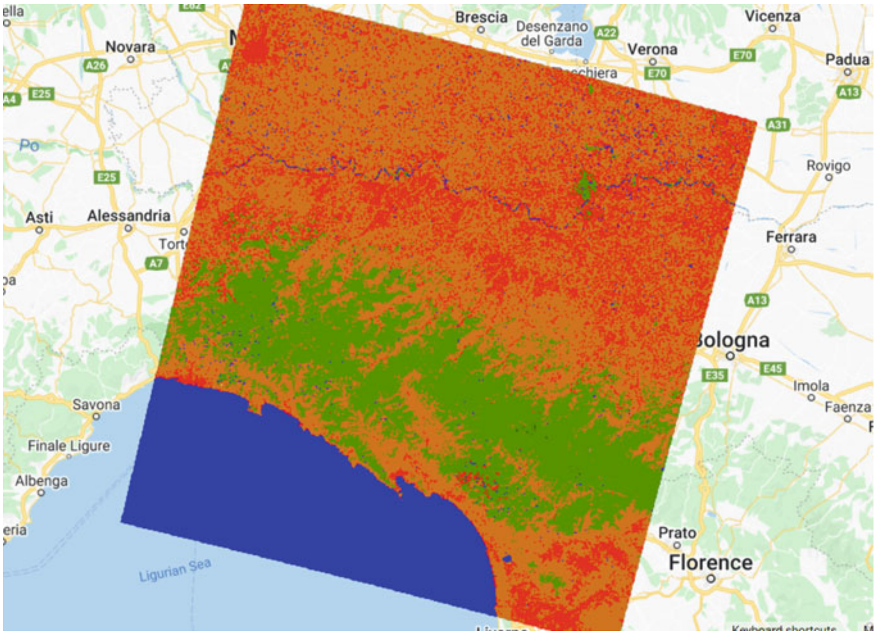


Fig. 6.11 CART classification

herbaceous classes). Why do you think this is happening? There are a few options to handle misclassification errors:

- **Collect more training data**: We can try incorporating more points to have a more representative sample of the classes.
- **Tune the model**: Classifiers typically have "hyperparameters," which are set to default values. In the case of classification trees, there are ways to tune the number of leaves in the tree, for example. Tuning models is addressed in Chap. 7.
- **Try other classifiers**: If a classifier's results are unsatisfying, we can try some of the other classifiers in Earth Engine to see if the result is better or different.
- **Expand the collection location**: It is good practice to collect points across the entire image and not just focus on one location. Also, look for pixels of the same class that show variability (e.g., for the developed class, building rooftops look different than house rooftops; for the herbaceous class, crop fields show distinctive seasonality/phenology).
- **Add more predictors**: We can try adding spectral indices to the input variables; this way, we are feeding the classifier new, unique information about each class. For example, there is a good chance that a vegetation index specialized for detecting vegetation health (e.g., NDVI) would improve the developed versus herbaceous classification.

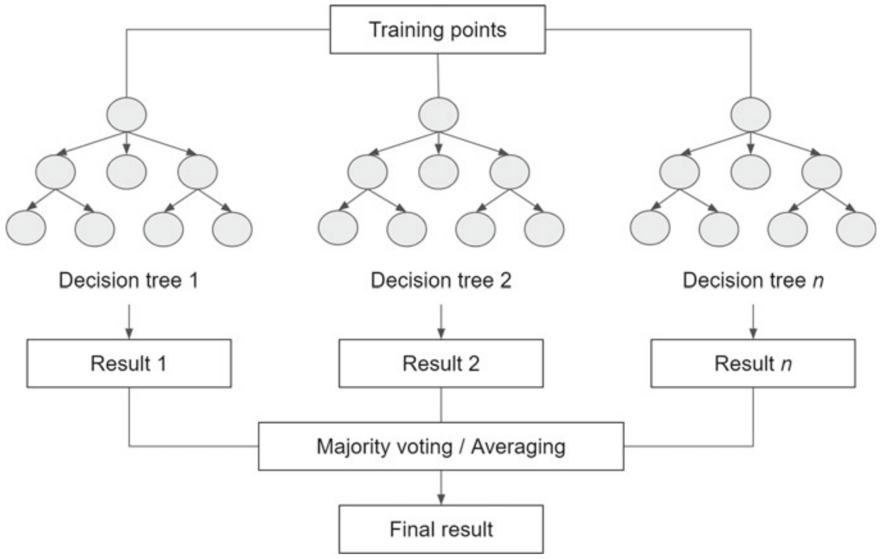


Fig. 6.12 General concept of Random Forest

For now, we will try another supervised learning classifier that is widely used: Random Forest (RF). The RF algorithm (Breiman 2001; Pal 2005) builds on the concept of decision trees, but adds strategies to make them more powerful. It is called a “forest” because it operates by constructing a multitude of decision trees (Fig. 6.12). As mentioned previously, a decision tree creates the rules which are used to make decisions. A Random Forest will randomly choose features and make observations, build a forest of decision trees and then use the full set of trees to estimate the class. It is a great choice when you do not have a lot of insight about the training data.

Copy and paste the code below to train the RF classifier (`ee.Classifier.smileRandomForest`) and apply the classifier to the image. The RF algorithm requires, as its argument, the number of trees to build. We will use 50 trees.

```
/////////////// Random Forest Classifier
/////////////////////

// Train RF classifier.
var RFclassifier =
ee.Classifier.smileRandomForest(50).train({
    features: classifierTraining,
    classProperty: 'class',
    inputProperties: predictionBands
});

// Classify Landsat image.
var RFclassified =
landsat.select(predictionBands).classify(
    RFclassifier);

// Add classified image to the map.
Map.addLayer(RFclassified, classificationVis, 'RF
classified');
```

Note that in the `ee.Classifier.smileRandomForest` documentation (**Docs** tab), there is a `seed` (random number) parameter. Setting a `seed` allows you to exactly replicate your model each time you run it. Any number is acceptable as a seed.

Inspect the result (Fig. 6.13). How does this classified image differ from the CART one? Is the classifications better or worse? Zoom in and out and change the transparency of layers as needed. In Chap. 7, you will see more systematic ways to assess what is better or worse, based on accuracy metrics.

Code Checkpoint F21b. The book's repository contains a script that shows what your code should look like at this point.

6.2.2 Section 2: Unsupervised Classification

In an unsupervised classification, we have the opposite process of supervised classification. Spectral classes are grouped first and then categorized into clusters. Therefore, in Earth Engine, these classifiers are `ee.Clusterer` objects. They are "self-taught" algorithms that do not use a set of labeled training data (i.e., they are "unsupervised"). You can think of it as performing a task that you have not experienced before, starting by gathering as much information as possible. For example, imagine learning a new language without knowing the basic grammar, learning only by watching a TV series in that language, listening to examples, and finding patterns.

Similar to the supervised classification, unsupervised classification in Earth Engine has this workflow:

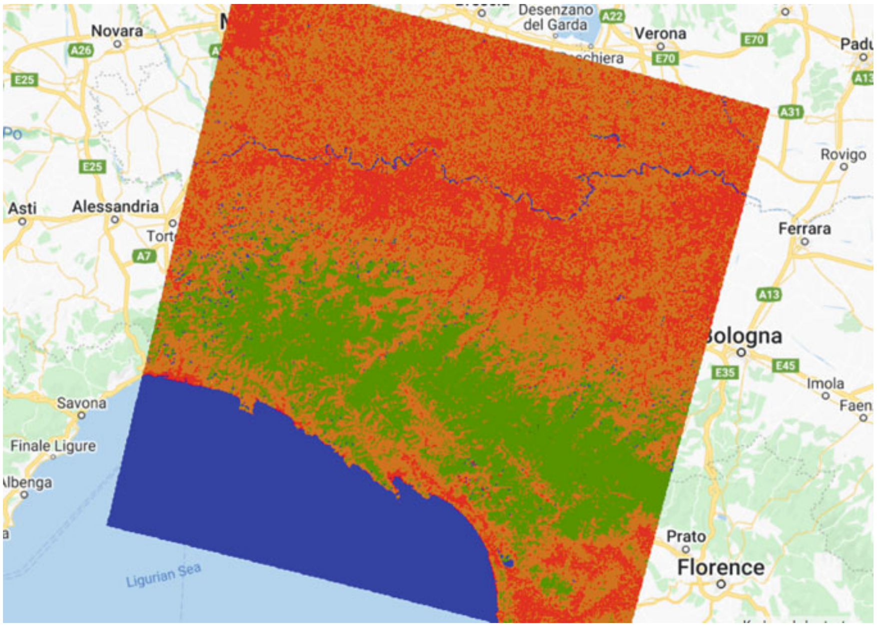


Fig. 6.13 Random forest classified image

1. Assemble features with numeric properties in which to find clusters (training data).
2. Select and instantiate a clusterer.
3. Train the clusterer with the training data.
4. Apply the clusterer to the scene (classification).
5. Label the clusters.

In order to generate training data, we will use the `sample` method, which randomly takes samples from a region (unlike `sampleRegions`, which takes samples from predefined locations). We will use the image's footprint as the region by calling the `geometry` method. Additionally, we will define the number of pixels (`numPixels`) to sample—in this case, 1000 pixels—and define a `tileScale` of 8 to avoid computation errors due to the size of the region. Copy and paste the code below to sample 1000 pixels from the Landsat image. You should add to the same script as before to compare supervised versus unsupervised classification results at the end.

```
//////////////// Unsupervised classification
////////////////

// Make the training dataset.
var training = landsat.sample({
    region: landsat.geometry(),
    scale: 30,
    numPixels: 1000,
    tileScale: 8
});
```

Now we can instantiate a clusterer and train it. As with the supervised algorithms, there are many unsupervised algorithms to choose from. We will use the *k*-means clustering algorithm, which is a commonly used approach in remote sensing. This algorithm identifies groups of pixels near each other in the spectral space (image *x* bands) by using an iterative regrouping strategy. We define a number of clusters, *k,* and then the method randomly distributes that number of seed points into the spectral space. A large sample of pixels is then grouped into its closest seed, and the mean spectral value of this group is calculated. That mean value is akin to a center of mass of the points and is known as the centroid. Each iteration recalculates the class means and reclassifies pixels with respect to the new means. This process is repeated until the centroids remain relatively stable, and only a few pixels change from class to class on subsequent iterations (Fig. 6.14).

Copy and paste the code below to request four clusters, the same number as for the supervised classification, in order to directly compare them.

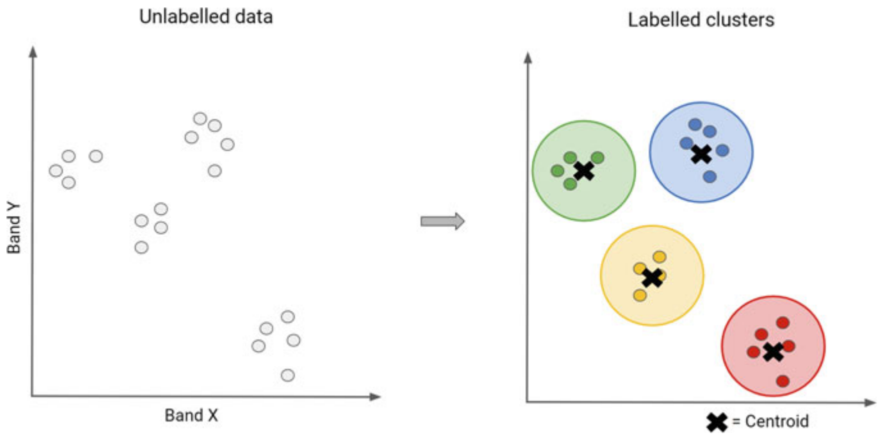


Fig. 6.14 *K*-means visual concept

```
// Instantiate the clusterer and train it.
var clusterer =
ee.Clusterer.wekaKMeans(4).train(training);
```

Now copy and paste the code below to apply the clusterer to the image and add the resulting classification to the **Map** (Fig. 6.15). Note that we are using a method called `randomVisualizer` to assign colors for the visualization. We are not associating the unsupervised classes with the color palette we defined earlier in the supervised classification. Instead, we are assigning random colors to the classes, since we do not yet know which of the unsupervised classes best corresponds to each of the named classes (e.g., `forest`, `herbaceous`). Note that the colors in Fig. 6.15 might not be the same as you see on your **Map,** since they are assigned randomly.

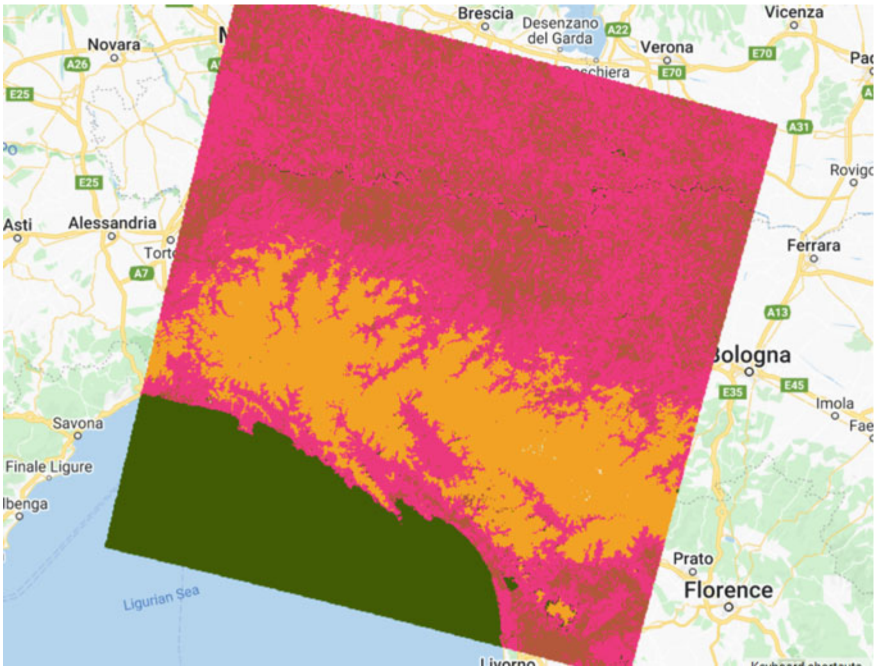


Fig. 6.15 *K*-means classification

```
// Cluster the input using the trained clusterer.
var Kclassified = landsat.cluster(clusterer);

// Display the clusters with random colors.
Map.addLayer(Kclassified.randomVisualizer(), {},
    'K-means classified - random colors');
```

Inspect the results. How does this classification compare to the previous ones? If preferred, use the **Inspector** to check which classes were assigned to each pixel value ("cluster" band) and change the last line of your code to apply the same palette used for the supervised classification results (see Code Checkpoint below for an example).

Another key point of classification is the accuracy assessment of the results. This will be covered in Chap. 7.

Code Checkpoint F21c. The book's repository contains a script that shows what your code should look like at this point.

6.3 Synthesis

Test if you can improve the classifications by completing the following assignments.

Assignment 1. For the supervised classification, try collecting more points for each class. The more points you have, the more spectrally represented the classes are. It is good practice to collect points across the entire composite and not just focus on one location. Also look for pixels of the same class that show variability. For example, for the water class, collect pixels in parts of rivers that vary in color. For the developed class, collect pixels from different rooftops.

Assignment 2. Add more predictors. Usually, the more spectral information you feed the classifier, the easier it is to separate classes. Try calculating and incorporating a band of NDVI or the Normalized Difference Water Index (Chap. 5) as a predictor band. Does this help the classification? Check for `developed` areas that were being classified as `herbaceous` or vice versa.

Assignment 3. Use more trees in the Random Forest classifier. Do you see any improvements compared to 50 trees? Note that the more trees you have, the longer it will take to compute the results, and that more trees might not always mean better results.

Assignment 4. Increase the number of samples that are extracted from the composite in the unsupervised classification. Does that improve the result?

Assignment 5. Increase the number *k* of clusters for the *k*-means algorithm. What would happen if you tried 10 classes? Does the classified map result in meaningful classes?

Assignment 6. Test other clustering algorithms. We only used *k*-means; try other options under the `ee.Clusterer` object.

6.4 Conclusion

Classification algorithms are key for many different applications because they allow you to predict categorical variables. You should now understand the difference between supervised and unsupervised classification and have the basic knowledge on how to handle misclassifications. By being able to map the landscape for land use and land cover, we will also be able to monitor how it changes (Part IV).

References

Breiman L (2001) Random forests. Mach Learn 45:5–32. https://doi.org/10.1023/A:1010933404324
Gareth J, Witten D, Hastie T, Tibshirani R (2013) An introduction to statistical learning. Springer
Géron A (2019) Hands-on machine learning with Scikit-Learn, Keras and TensorFlow: concepts, tools, and techniques to build intelligent systems. O'Reilly Media, Inc.
Goodfellow I, Bengio Y, Courville A (2016) Deep learning. MIT Press
Hastie T, Tibshirani R, Friedman JH (2009) The elements of statistical learning: data mining, inference, and prediction. Springer
Li M, Zang S, Zhang B et al (2014) A review of remote sensing image classification techniques: the role of spatio-contextual information. Eur J Remote Sens 47:389–411. https://doi.org/10.5721/EuJRS20144723
Müller AC, Guido S (2016) Introduction to machine learning with python: a guide for data scientists. O'Reilly Media, Inc.
Pal M (2005) Random forest classifier for remote sensing classification. Int J Remote Sens 26:217–222. https://doi.org/10.1080/01431160412331269698
Witten IH, Frank E, Hall MA, et al (2005) Practical machine learning tools and techniques. In: Data mining, p 4

Accuracy Assessment: Quantifying Classification Quality

7

Andréa Puzzi Nicolau, Karen Dyson, David Saah, and Nicholas Clinton

Overview
This chapter will enable you to assess the accuracy of an image classification. You will learn about different metrics and ways to quantify classification quality in Earth Engine. Upon completion, you should be able to evaluate whether your classification needs improvement and know how to proceed when it does.

Learning Outcomes

- Learning how to perform accuracy assessment in Earth Engine.
- Understanding how to generate and read a confusion matrix.
- Understanding overall accuracy and the kappa coefficient.

A. P. Nicolau · K. Dyson · D. Saah (✉)
Spatial Informatics Group, Pleasanton, CA, USA
e-mail: dssaah@usfca.edu

A. P. Nicolau
e-mail: apnicolau@sig-gis.com

K. Dyson
e-mail: kdyson@sig-gis.com

A. P. Nicolau · K. Dyson
SERVIR-Amazonia, Cali, Colombia

K. Dyson
Dendrolytics, Seattle, WA, USA

D. Saah
University of San Francisco, San Francisco, CA, USA

N. Clinton
Google LLC, Mountain View, CA, USA
e-mail: nclinton@google.com

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_7

- Understanding the difference between user's and producer's accuracy and the difference between omission and commission errors.

Assumes you know how to

- Create a graph using `ui.Chart` (Chap. 4).
- Perform a supervised Random Forest image classification (Chap. 6).

7.1 Introduction to Theory

Any map or remotely sensed product is a generalization or model that will have inherent errors. Products derived from remotely sensed data used for scientific purposes and policymaking require a quantitative measure of accuracy to strengthen the confidence in the information generated (Foody 2002; Strahler et al. 2006; Olofsson et al. 2014). Accuracy assessment is a crucial part of any classification project, as it measures the degree to which the classification agrees with another data source that is considered to be accurate, ground-truth data (i.e., "reality").

The history of accuracy assessment reveals increasing detail and rigor in the analysis, moving from a basic visual appraisal of the derived map (Congalton 1994; Foody 2002) to the definition of best practices for sampling and response designs and the calculation of accuracy metrics (Foody 2002; Stehman 2013; Olofsson et al. 2014; Stehman and Foody 2019). The confusion matrix (also called the "error matrix") (Stehman 1997) summarizes key accuracy metrics used to assess products derived from remotely sensed data.

7.2 Practicum

In Chap. 6, we asked whether the classification results were satisfactory. In remote sensing, the quantification of the answer to that question is called accuracy assessment. In the classification context, accuracy measurements are often derived from a confusion matrix.

In a thorough accuracy assessment, we think carefully about the sampling design, the response design, and the analysis (Olofsson et al. 2014). Fundamental protocols are taken into account to produce scientifically rigorous and transparent estimates of accuracy and area, which requires robust planning and time. In a standard setting, we would calculate the number of samples needed for measuring accuracy (sampling design). Here, we will focus mainly on the last step, analysis, by examining the confusion matrix and learning how to calculate the accuracy metrics. This will be done by partitioning the existing data into training and testing sets.

7.2.1 Quantifying Classification Accuracy Through a Confusion Matrix

To illustrate some of the basic ideas about classification accuracy, we will revisit the data and location of part of Chap. 6, where we tested different classifiers and classified a Landsat image of the area around Milan, Italy. We will name this dataset 'data'. This variable is a FeatureCollection with features containing the "class" values (Table 7.1) and spectral information of four land cover/land use classes: forest, developed, water, and herbaceous (see Figs. 6.8 and 6.9 for a refresher). We will also define a variable, predictionBands, which is a list of bands that will be used for prediction (classification)—the spectral information in the data variable.

The first step is to partition the set of known values into training and testing sets in order to have something for the classifier to predict over that it has not been shown before (the testing set), mimicking unseen data that the model might see in the future. We add a column of random numbers to our FeatureCollection using the randomColumn method. Then, we filter the features into about 80% for training and 20% for testing using ee.Filter. Copy and paste the code below to partition the data and filter features based on the random number.

```
// Import the reference dataset.
var data = ee.FeatureCollection(
    'projects/gee-book/assets/F2-2/milan_data');

// Define the prediction bands.
var predictionBands = [
    'SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6',
'SR_B7','ST_B10','ndvi', 'ndwi'
];

// Split the dataset into training and testing sets.
var trainingTesting = data.randomColumn();
var trainingSet = trainingTesting
    .filter(ee.Filter.lessThan('random', 0.8));
var testingSet = trainingTesting
    .filter(ee.Filter.greaterThanOrEquals('random', 0.8));
```

Table 7.1 Land cover classes

Class	Class value
Forest	0
Developed	1
Water	2
Herbaceous	3

Table 7.2 Confusion matrix for a binary classification where the classes are “positive” and “negative”

		Actual values	
		Positive	Negative
Predicted values	Positive	TP (true positive)	FP (false positive)
	Negative	FN (false negative)	TN (true negative)

Note that `randomColumn` creates pseudorandom numbers in a deterministic way. This makes it possible to generate a reproducible pseudorandom sequence by defining the `seed` parameter (Earth Engine uses a seed of 0 by default). In other words, given a starting value (i.e., the seed), `randomColumn` will always provide the same sequence of pseudorandom numbers.

Copy and paste the code below to train a Random Forest classifier with 50 decision trees using the `trainingSet`.

```
// Train the Random Forest Classifier with the
trainingSet.
var RFclassifier =
ee.Classifier.smileRandomForest(50).train({
    features: trainingSet,
    classProperty: 'class',
    inputProperties: predictionBands
});
```

Now, let us discuss what a confusion matrix is. A confusion matrix describes the quality of a classification by comparing the predicted values to the actual values. A simple example is a confusion matrix for a binary classification into the classes “positive” and “negative,” as given in Table 7.2.

In Table 7.2, the columns represent the actual values (the truth), while the rows represent the predictions (the classification). “True positive” (TP) and “true negative” (TN) mean that the classification of a pixel matches the truth (e.g., a water pixel correctly classified as water). “False positive” (FP) and “false negative” (FN) mean that the classification of a pixel does not match the truth (e.g., a non-water pixel incorrectly classified as water).

- TP: classified as positive, and the actual class is positive
- FP: classified as positive, and the actual class is negative
- FN: classified as negative, and the actual class is positive
- TN: classified as negative, and the actual class is negative.

We can extract some statistical information from a confusion matrix. Let us look at an example to make this clearer. Table 7.3 is a confusion matrix for a sample

Table 7.3 Confusion matrix for a binary classification where the classes are "positive" (forest) and "negative" (non-forest)

		Actual values	
		Positive	Negative
Predicted values	Positive	307	18
	Negative	14	661

of 1000 pixels for a classifier that identifies whether a pixel is forest (positive) or non-forest (negative), a binary classification.

In this case, the classifier correctly identified 307 forest pixels, wrongly classified 18 non-forest pixels as forest, correctly identified 661 non-forest pixels, and wrongly classified 14 forest pixels as non-forest. Therefore, the classifier was correct 968 times and wrong 32 times. Let's calculate the main accuracy metrics for this example.

The overall accuracy tells us what proportion of the reference data was classified correctly and is calculated as the total number of correctly identified pixels divided by the total number of pixels in the sample.

$$\text{Overall Accuracy} = (\text{TP} + \text{TN})/\text{Sample size}$$

In this case, the overall accuracy is 96.8%, calculated using $(307 + 661)/1000$.

Two other important accuracy metrics are the producer's accuracy and the user's accuracy, also referred to as the "recall" and the "precision," respectively. Importantly, these metrics quantify aspects of per-class accuracy.

The producer's accuracy is the accuracy of the map from the point of view of the map maker (the "producer") and is calculated as the number of correctly identified pixels of a given class divided by the total number of pixels actually in that class. The producer's accuracy for a given class tells us the proportion of the pixels in that class that were classified correctly.

$$\text{Producer's accuracy of the Forest(Positive)class } = \text{TP}/(\text{TP} + \text{FN})$$

$$\text{Producer's accuracy of the Non - Forest(Negative)class} = \text{TN}/(\text{TN} + \text{FP})$$

In this case, the producer's accuracy for the forest class is 95.6%, which is calculated using $307/(307 + 14)$. The producer's accuracy for the non-forest class is 97.3%, which is calculated from $661/(661 + 18)$.

The user's accuracy (also called the "consumer's accuracy") is the accuracy of the map from the point of view of a map user and is calculated as the number of correctly identified pixels of a given class divided by the total number of pixels claimed to be in that class. The user's accuracy for a given class tells us the proportion of the pixels identified on the map as being in that class that are actually in that class on the ground.

$$\text{User's accuracy of the Forest (Positive)class} = \text{TP}/(\text{TP} + \text{FP})$$

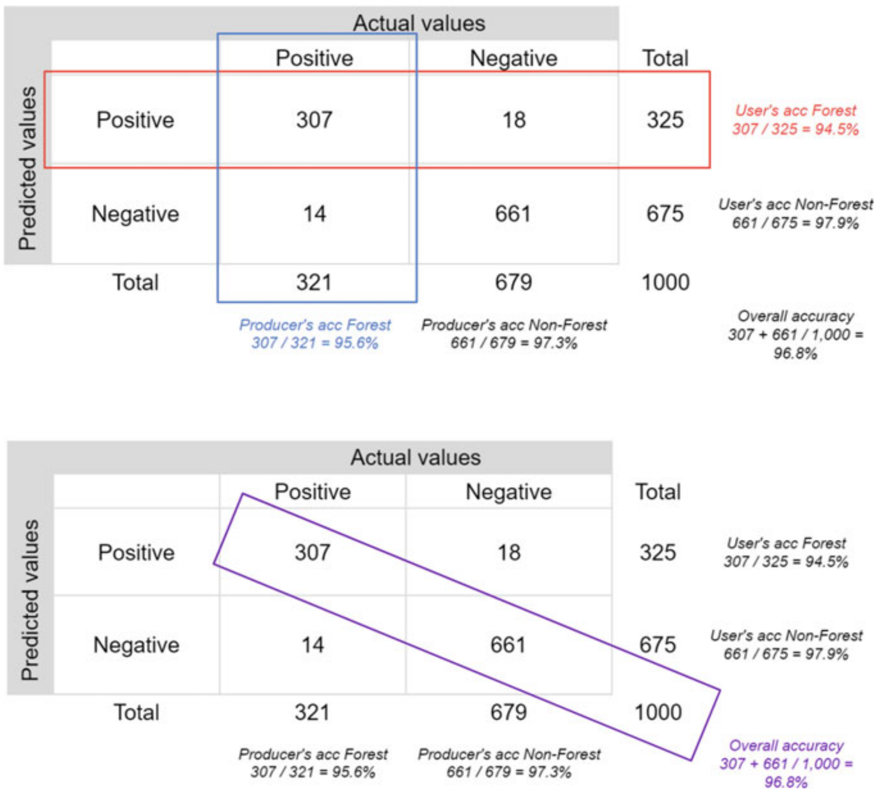
$$\text{User's accuracy of the Non - Forest(Negative)class} = \text{TN}/(\text{TN} + \text{FN})$$

In this case, the user's accuracy for the forest class is 94.5%, which is calculated using 307/(307+18). The user's accuracy for the non-forest class is 97.9%, which is calculated from 661/(661 + 14).

Figure 7.1 helps visualize the rows and columns that are used to calculate each accuracy.

It is very common to talk about two types of error when addressing remote sensing classification accuracy: omission errors and commission errors. Omission errors refer to the reference pixels that were left out of (omitted from) the correct class in the classified map. In a two-class system, an error of omission in one class will be counted as an error of commission in another class. Omission errors are complementary to the producer's accuracy.

$$\text{Omission error} = 100\% - \text{Producer's accuracy}$$



Predicted values	Actual values: Positive	Actual values: Negative	Total	
Positive	307	18	325	User's acc Forest 307 / 325 = 94.5%
Negative	14	661	675	User's acc Non-Forest 661 / 675 = 97.9%
Total	321	679	1000	
	Producer's acc Forest 307 / 321 = 95.6%	Producer's acc Non-Forest 661 / 679 = 97.3%		Overall accuracy 307 + 661 / 1,000 = 96.8%

Predicted values	Actual values: Positive	Actual values: Negative	Total	
Positive	307	18	325	User's acc Forest 307 / 325 = 94.5%
Negative	14	661	675	User's acc Non-Forest 661 / 675 = 97.9%
Total	321	679	1000	
	Producer's acc Forest 307 / 321 = 95.6%	Producer's acc Non-Forest 661 / 679 = 97.3%		Overall accuracy 307 + 661 / 1,000 = 96.8%

Fig. 7.1 Confusion matrix for a binary classification where the classes are "positive" (forest) and "negative" (non-forest), with accuracy metrics

Commission errors refer to the class pixels that were erroneously classified in the map and are complementary to the user's accuracy.

$$\text{Commission error} = 100\% - \text{User's accuracy}$$

Finally, another commonly used accuracy metric is the kappa coefficient, which evaluates how well the classification performed as compared to random. The value of the kappa coefficient can range from − 1 to 1: A negative value indicates that the classification is worse than a random assignment of categories would have been; a value of 0 indicates that the classification is no better or worse than random; and a positive value indicates that the classification is better than random.

$$\text{Kappa Coefficient} = \frac{\text{observed accuracy} - \text{chance agreement}}{1 - \text{chance agreement}}$$

The chance agreement is calculated as the sum of the product of row and column totals for each class, and the observed accuracy is the overall accuracy. Therefore, for our example, the kappa coefficient is 0.927.

$$\text{Kappa Coefficient} = \frac{0.968 - [(0.321x0.325) + (0.679x0.675)]}{1 - [(0.321x0.325) + (0.679x0.675)]} = 0.927$$

Now, let's go back to the script. In Earth Engine, there are API calls for these operations. Note that our confusion matrix will be a 4×4 table, since we have four different classes.

Copy and paste the code below to classify the `testingSet` and get a confusion matrix using the method `errorMatrix`. Note that the classifier automatically adds a property called "classification," which is compared to the "class" property of the reference dataset.

```
// Now, to test the classification (verify model's accuracy),
// we classify the testingSet and get a confusion matrix.
var confusionMatrix = testingSet.classify(RFclassifier)
    .errorMatrix({
        actual: 'class',
        predicted: 'classification'
    });
```

Copy and paste the code below to print the confusion matrix and accuracy metrics. Expand the confusion matrix object to inspect it. The entries represent the number of pixels. Items on the diagonal represent correct classification. Items off the diagonal are misclassifications, where the class in row *i* is classified as column *j* (values from 0 to 3 correspond to our class codes: forest, developed, water, and herbaceous, respectively). Also expand the producer's accuracy, user's accuracy (consumer's accuracy), and kappa coefficient objects to inspect them.

```
// Print the results.
print('Confusion matrix:', confusionMatrix);
print('Overall Accuracy:', confusionMatrix.accuracy());
print('Producers Accuracy:',
confusionMatrix.producersAccuracy());
print('Consumers Accuracy:',
confusionMatrix.consumersAccuracy());
print('Kappa:', confusionMatrix.kappa());
```

How is the classification accuracy? Which classes have higher accuracy compared to the others? Can you think of any reasons why? (Hint: Check where the errors in these classes are in the confusion matrix—i.e., being committed and omitted.)

Code Checkpoint F22a. The book's repository contains a script that shows what your code should look like at this point.

7.2.2 Hyperparameter Tuning

We can also assess how the number of trees in the Random Forest classifier affects the classification accuracy. Copy and paste the code below to create a function that charts the overall accuracy versus the number of trees used. The code tests from 5 to 100 trees at increments of 5, producing Fig. 7.2. (Do not worry too much about fully understanding each item at this stage of your learning. If you want to find out how these operations work, you can see more in Chaps. 12 and 13).

```
// Hyperparameter tuning.
var numTrees = ee.List.sequence(5, 100, 5);

var accuracies = numTrees.map(function(t) {
    var classifier = ee.Classifier.smileRandomForest(t)
        .train({
            features: trainingSet,
            classProperty: 'class',
            inputProperties: predictionBands
        });
    return testingSet
        .classify(classifier)
        .errorMatrix('class', 'classification')
        .accuracy();
});

print(ui.Chart.array.values({
    array: ee.Array(accuracies),
    axis: 0,
    xLabels: numTrees
}).setOptions({
    hAxis: {
        title: 'Number of trees'
    },
    vAxis: {
        title: 'Accuracy'
    },
    title: 'Accuracy per number of trees'
}));
```

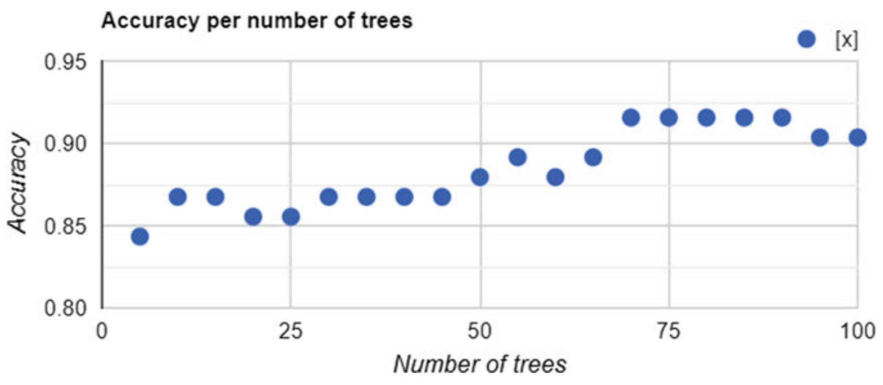


Fig. 7.2 Chart showing accuracy per number of random forest trees

Code Checkpoint F22b. The book's repository contains a script that shows what your code should look like at this point.

7.2.3 Spatial Autocorrelation

We might also want to ensure that the samples from the training set are uncorrelated with the samples from the testing set. This might result from the spatial autocorrelation of the phenomenon being predicted. One way to exclude samples that might be correlated in this manner is to remove samples that are within some distance to any other sample. In Earth Engine, this can be accomplished with a spatial join. The following Code Checkpoint replicates Sect. 7.2.1 but with a spatial join that excludes training points that are less than 1000 m distant from testing points.

Code Checkpoint F22c. The book's repository contains a script that shows what your code should look like at this point.

7.3 Synthesis

Assignment 1. Based on Sect. 7.2.1, test other classifiers (e.g., a Classification and Regression Tree or Support Vector Machine classifier) and compare the accuracy results with the Random Forest results. Which model performs better?

Assignment 2. Try setting a different seed in the `randomColumn` method and see how that affects the accuracy results. You can also change the split between the training and testing sets (e.g., 70/30 or 60/40).

7.4 Conclusion

You should now understand how to calculate how well your classifier is performing on the data used to build the model. This is a useful way to understand how a classifier is performing, because it can help indicate which classes are performing better than others. A poorly modeled class can sometimes be improved by, for example, collecting more training points for that class.

Nevertheless, a model may work well on training data but work poorly in locations randomly chosen in the study area. To understand a model's behavior on testing data, analysts employ protocols required to produce scientifically rigorous and transparent estimates of the accuracy and area of each class in the study region. We will not explore those practices in this chapter, but if you are interested, there are tutorials and papers available online that can guide you through the process. Links to some of those tutorials can be found in the "For Further Reading" section of this book.

References

Congalton R (1994) Accuracy assessment of remotely sensed data: future needs and directions. In: Proceedings of Pecora 12 land information from space-based systems, pp 385–388

Foody GM (2002) Status of land cover classification accuracy assessment. Remote Sens Environ 80:185–201. https://doi.org/10.1016/S0034-4257(01)00295-4

Olofsson P, Foody GM, Herold M et al (2014) Good practices for estimating area and assessing accuracy of land change. Remote Sens Environ 148:42–57. https://doi.org/10.1016/j.rse.2014.02.015

Stehman SV (1997) Selecting and interpreting measures of thematic classification accuracy. Remote Sens Environ 62:77–89. https://doi.org/10.1016/S0034-4257(97)00083-7

Stehman SV (2013) Estimating area from an accuracy assessment error matrix. Remote Sens Environ 132:202–211. https://doi.org/10.1016/j.rse.2013.01.016

Stehman SV, Foody GM (2019) Key issues in rigorous accuracy assessment of land cover products. Remote Sens Environ 231:111199. https://doi.org/10.1016/j.rse.2019.05.018

Strahler AH, Boschetti L, Foody GM et al (2006) Global land cover validation: recommendations for evaluation and accuracy assessment of global land cover maps. Eur Communities, Luxemb 51:1–60

Part III
Advanced Image Processing

Once you understand the basics of processing images in Earth Engine, this part will present some of the more advanced processing tools available for treating individual images. These include creating regressions among image bands, transforming images with pixel-based and neighborhood-based techniques, and grouping individual pixels into objects that can then be classified.

Interpreting an Image: Regression

8

Karen Dyson, Andréa Puzzi Nicolau, David Saah, and Nicholas Clinton

Overview

This chapter introduces the use of regression to interpret imagery. Regression is one of the fundamental tools you can use to move from viewing imagery to analyzing it. In the present context, regression means predicting a numeric variable for a pixel instead of a categorical variable, such as a class label.

Learning Outcomes

- Learning about Earth Engine reducers.
- Understanding the difference between regression and classification.
- Using reducers to implement regression between image bands.
- Evaluating regression model performance visually and numerically.

K. Dyson (✉) · D. Saah
Spatial Informatics Group, Pleasanton, California, USA
e-mail: kdyson@sig-gis.com

D. Saah
e-mail: dsaah@usfca.edu

K. Dyson · A. P. Nicolau
SERVIR-Amazonia, Cali, Colombia
e-mail: apnicolau@sig-gis.com

K. Dyson
Dendrolytics, Seattle, Washington, USA

D. Saah
University of San Francisco, San Francisco, California, USA

N. Clinton
Google LLC, Mountain View, USA
e-mail: nclinton@google.com

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_8

Assumes you know how to

- Import images and image collections, filter, and visualize (Part I).
- Perform basic image analysis: select bands, compute indices, and create masks (Part II).
- Understand the characteristics of Landsat data and MODIS data (Chaps. 3, and 4).
- Use `normalizedDifference` to calculate vegetation indices (Chap. 5).
- Interpret variability of the NDVI, the Normalized Difference Vegetation Index (Chap. 5).
- Have an understanding of regression.

8.1 Introduction to Theory

If you have already used regression in other contexts, this approach will be familiar to you. Regression in remote sensing uses the same basic concepts as regression in other contexts. We determine the strength and characteristics of the relationship between the dependent variable and one or more independent variables to better understand or forecast the dependent variable. In a remote-sensing context, these dependent variables might be the likelihood of a natural disaster or a species occurrence, while independent variables might be bands, indices, or other raster datasets like tree height or soil type.

Importantly, we use regression for numeric dependent variables. Classification, which is covered in Chap. 6, is used for categorical dependent variables, such as land cover or land use.

Regression in remote sensing is a very powerful tool that can be used to answer many important questions across large areas. For example, in China, researchers used a random forest regression algorithm to estimate above-ground wheat biomass based on vegetation indices (Zhou et al. 2016). In the United States, researchers used regression to estimate bird species richness based on lidar measurements of forest canopy (Goetz et al. 2007). In Kenya, researchers estimated tree species diversity using vegetation indices including the tasseled cap transformation and simple and multivariate regression analysis (Maeda et al. 2014).

8.2 Practicum

In general, regression in remote sensing follows a standard set of steps.

1. Data about known instances of the dependent variable are collected. This might be known landslide areas, known areas of species occurrence, etc.

2. The independent variables are defined and selected.
3. A regression is run to generate an equation that describes the relationship between dependent and independent variables.
4. Optional: Using this equation and the independent variable layers, you create a map of the dependent variable over the entire area.

While this set of steps remains consistent across regressions, there are multiple types of regression functions to choose from, based on the properties of your dependent and independent variables, and how many independent variables you have. The choice of regression type in remote sensing follows the same logic as in other contexts. There are many other excellent online resources and textbooks to help you better understand regression and make your choice of regression method. Here, we will focus on running regression in Google Earth Engine and some of the different options available to you.

8.2.1 Reducers

A key goal of working with remote-sensing data is summarizing over space, time, and band information to find relationships. Earth Engine provides a large set of summary techniques for remote-sensing data, which fall under the collective term *reducers*.

An example of a reducer of a set of numbers is the median, which finds the middle number in a set, reducing the complexity of the large set to a representative estimate. That reducing operation is implemented in Earth Engine with `ee.Reducer.median`., or for this particularly common operation, with the shorthand operation `median`.

Reducers can operate on a pixel-by-pixel basis or with awareness of a pixel's surroundings. An example of a reducer of a spatial object is computing the median elevation in a neighborhood, which can be implemented in Earth Engine by embedding the call to `ee.Reducer.median` inside a `reduceRegion` function. The `reduceRegion` call directs Earth Engine to analyze pixels that are grouped across an area, allowing an area like a city block or park boundary to be considered as a unified unit of interest. Reducers can be used to summarize many forms of data in Earth Engine, including image collections, images, lists, and feature collections. They are encountered throughout the rest of this book in a wide variety of settings.

Reducers can be relatively simple, like the median, or more complex. Imagine a 1000-person sample of height and weight of individuals: a linear regression would reduce the 2000 assembled values to just two: the slope and the intercept. Of course the relationship might be weak or strong in a given set; the bigger idea is that many operations of simplifying and summarizing large amounts of data can be conceptualized using this idea and terminology of *reducers*.

In this chapter, we will illustrate reducers through their use in regressions of bands in a single image and revisit regressions using analogous operations on time series in other parts of the book.

8.3 Section 1: Linear Fit

The simplest regression available in Earth Engine is implemented through the reducer `linearFit`. This function is a least squares estimate of a linear function with one independent variable and one dependent variable. This regression equation is written $Y = \alpha + \beta X + \epsilon$, where α is the intercept of the line and β is the slope, Y is the dependent variable, X is the independent variable, and ϵ is the error.

Suppose the goal is to estimate percent tree cover in each Landsat pixel, based on known information about Turin, Italy. Below, we will define a geometry that is a rectangle around Turin and name it `Turin`.

```
// Define a Turin polygon.
var Turin = ee.Geometry.Polygon(
    [
        [
            [7.455553918110218, 45.258245019259036],
            [7.455553918110218, 44.71237367431335],
            [8.573412804828967, 44.71237367431335],
            [8.573412804828967, 45.258245019259036]
        ]
    ], null, false);

// Center on Turin
Map.centerObject(Turin, 9);
```

We need to access data to use as our dependent variable. For this example, we will use the “MOD44B.006 Terra Vegetation Continuous Fields Yearly Global 250 m” dataset. Add the code below to your script.

```
var mod44b = ee.ImageCollection('MODIS/006/MOD44B');
```

Note: If the path to that ImageCollection were to change, you would get an error when trying to access it. If this happens, you can search for "vegetation continuous MODIS" to find it. Then, you could import it and change its name to mod44b. This has the same effect as the one line above.

Next, we will access that part of the ImageCollection that represents the global percent tree cover at 250 m resolution from 2020. We'll access the image from that time period using the filterDate command, convert that single image to an Image type using the first command, clip it to Turin, and select the appropriate band. We then print information about the new image percentTree2020 to the **Console**.

```
/////
// Start Linear Fit
/////

// Put together the dependent variable by filtering the
// ImageCollection to just the 2020 image near Turin and
// selecting the percent tree cover band.
var percentTree2020 = mod44b
    .filterDate('2020-01-01', '2021-01-01')
    .first()
    .clip(Turin)
    .select('Percent_Tree_Cover');

// You can print information to the console for inspection.
print('2020 Image', percentTree2020);

Map.addLayer(percentTree2020, {
    max: 100
}, 'Percent Tree Cover');
```

Now we need to access data to use as our independent variables. For this example, we will use the "USGS Landsat 8 Collection 2 Tier 1 and Real-Time Data Raw Scenes." Add the code below to your script.

```
var landsat8_raw =
ee.ImageCollection('LANDSAT/LC08/C02/T1_RT');
```

Note: If the path to that ImageCollection changes, you will get an error when trying to access it. If this happens, you can search for “Landsat 8 raw” to find it. Then, import it and change its name to landsat8_raw.

We also need to filter this collection by date and location. We will filter the collection to a clear (cloud-free) date in 2020 and then filter by location.

```
// Put together the independent variable.
var landsat8filtered = landsat8_raw
    .filterBounds(Turin.centroid({
        'maxError': 1
    }))
    .filterDate('2020-04-01', '2020-4-30')
    .first();

print('Landsat8 filtered', landsat8filtered);

// Display the L8 image.
var visParams = {
    bands: ['B4', 'B3', 'B2'],
    max: 16000
};
Map.addLayer(landsat8filtered, visParams, 'Landsat 8
Image');
```

Using the centroid function will select images that intersect with the center of our Turin geometry, instead of anywhere in the geometry.

Note that we are using a single image for the purposes of this exercise, but in practice you will almost always need to filter an image collection to the boundaries of your area of interest and then create a composite. In that case, you would use a compositing Earth Engine algorithm to get a cloud-free composite of Landsat imagery. If you’re interested in learning more about working with clouds, please see Chap. 15.

Use the bands of the Landsat image to calculate NDVI, which we will use as the independent variable:

```
// Calculate NDVI which will be the independent variable.
var ndvi = landsat8filtered.normalizedDifference(['B5',
'B4']);
```

Now we begin to assemble our data in Earth Engine into the correct format. First, use the addBands function to create an image with two bands: first, the image ndvi, which will act as the independent variable; and second, the image percentTree2020 created earlier.

```
// Create the training image.
var trainingImage = ndvi.addBands(percentTree2020);
print('training image for linear fit', trainingImage);
```

Now we can set up and run the regression, using the linear fit reducer over our geometry and print the results. Since building the regression model requires assembling points from around the image for consideration, it is implemented using reduceRegion rather than reduce (see also Part V). We need to include the scale variable (here 30 m, which is the resolution of Landsat).

```
// Independent variable first, dependent variable second.
// You need to include the scale variable.
var linearFit = trainingImage.reduceRegion({
    reducer: ee.Reducer.linearFit(),
    geometry: Turin,
    scale: 30,
    bestEffort: true
});

// Inspect the results.
print('OLS estimates:', linearFit);
print('y-intercept:', linearFit.get('offset'));
print('Slope:', linearFit.get('scale'));
```

Note the parameter bestEffort in the request to ee.Reducer.linearFit. What does it mean? If you had tried to run this without the bestEffort: **true** argument, you would most likely get an error: "Image.reduceRegion: Too many pixels in the region." This means that the number of pixels involved has surpassed Earth Engine's default maxPixels limit of 10 million. The reason is the complexity of the regression we are requesting. If you've ever done a regression using, say, 100 points, you may have seen or made a scatter plot with the 100 points on it; now imagine a scatter plot with more than 10 million points to envision the scale of what is being requested. Here, the limitation is not Earth Engine's computing capacity, but rather it is more like a notification that the code is calling for an especially large computation, and that a novice user may not be entirely intending to do something of that complexity. The error text points to several ways to resolve this issue. First, we could increase maxPixels. Second, we could aggregate at a lower resolution (e.g., increase scale from 30 to 50 m). Third, we could set the bestEffort parameter to **true** as we do here, which directs the reducer to use the highest resolution for scale without going above maxPixels. Fourth, we could reduce the area for the region (that is, make the study area around Turin smaller). Finally, we could randomly sample the image stack and use that sample for the regression.

Finally, let's apply the regression to the whole NDVI area, which is larger than our Turin boundary polygon. The calculation is done with an expression, which will be explained further in Chap. 9.

```
// Create a prediction based on the linearFit model.
var predictedTree = ndvi.expression(
    'intercept + slope * ndvi', {
        'ndvi': ndvi.select('nd'),
        'intercept': ee.Number(linearFit.get('offset')),
        'slope': ee.Number(linearFit.get('scale'))
    });

print('predictedTree', predictedTree);

// Display the results.
Map.addLayer(predictedTree, {
    max: 100
}, 'Predicted Percent Tree Cover');
```

Your estimates based on NDVI are a higher resolution than the MODIS data because Landsat is 30 m resolution (Fig. 8.1). Notice where your estimates agree with the MODIS data and where they disagree, if anywhere. In particular, look at areas of agriculture. Since NDVI doesn't distinguish between trees and other vegetation, our model will estimate that agricultural areas have higher tree cover percentages than the MODIS data (you can use the **Inspector** tool to verify this).

Code Checkpoint F30a. The book's repository contains a script that shows what your code should look like at this point.

8.3.1 Section 2: Linear Regression

The linear regression reducer allows us to increase the number of dependent and independent variables. Let's revisit our model of percent cover and try to improve on our linear fit attempt by using the linear regression reducer with more independent variables.

We will use our percentTree dependent variable.

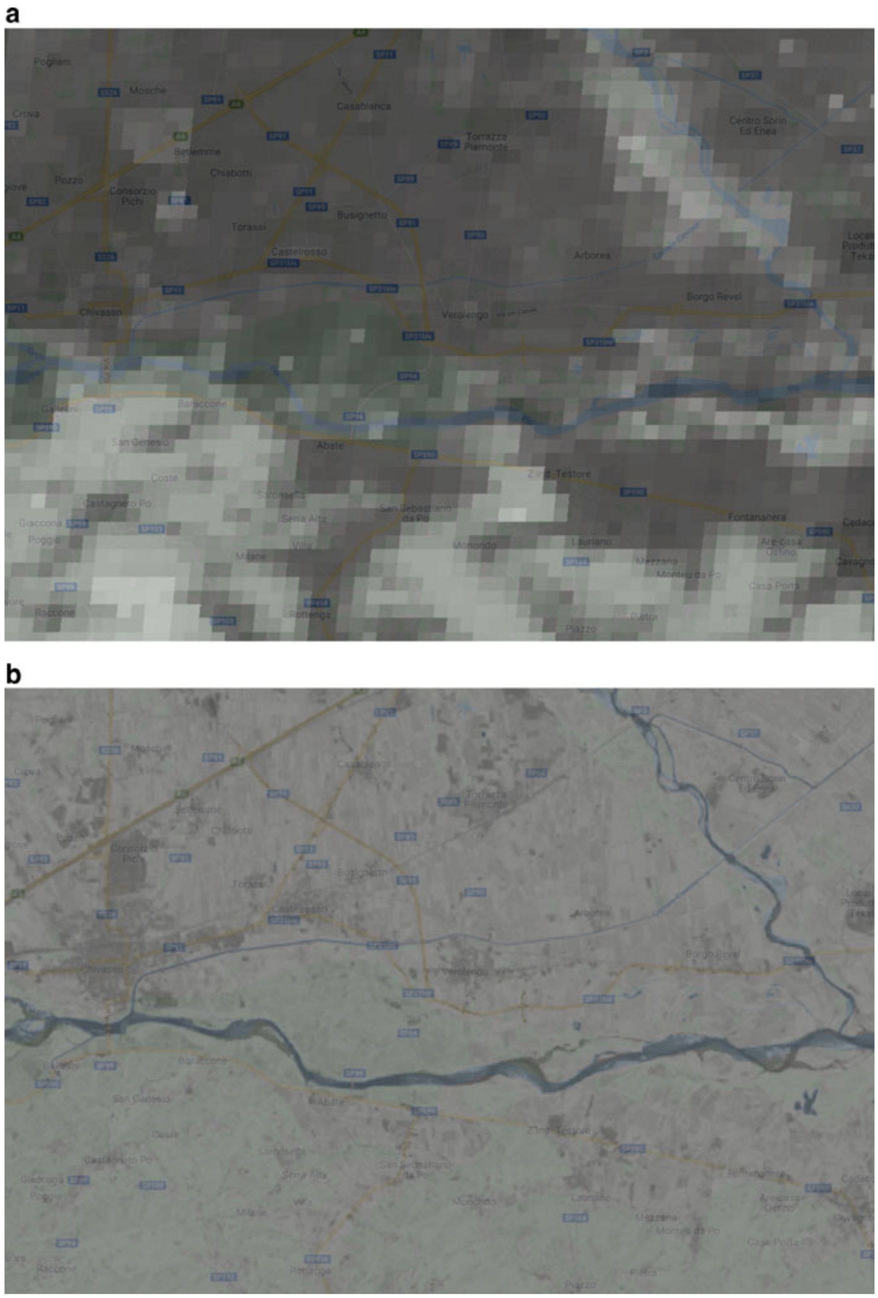


Fig. 8.1 Estimates of the same area based on MODIS (**a**) and NDVI (**b**)

For our independent variable, let's revisit our Landsat 8 image collection. Instead of using only NDVI, let's use multiple bands. Define these bands to select:

```
//////
// Start Linear Regression
//////

// Assemble the independent variables.
var predictionBands = ['B1', 'B2', 'B3', 'B4', 'B5', 'B6',
'B7','B10',  'B11'
];
```

Now let's assemble the data for the regression. For the constant term needed by the linear regression, we use the `ee.Image` function to create an image with the value (1) at every pixel. We then stack this constant with the Landsat prediction bands and the dependent variable (percent tree cover).

```
// Create the training image stack for linear regression.
var trainingImageLR = ee.Image(1)
    .addBands(landsat8filtered.select(predictionBands))
    .addBands(percentTree2020);

print('Linear Regression training image:',
trainingImageLR);
```

Inspect your training image to make sure it has multiple layers: a constant term, your independent variables (the Landsat bands), and your dependent variable.

Now we'll implement the `ee.Reducer.linearRegression` reducer function to run our linear regression. The reducer needs to know how many *X*, or independent variables, and the number of *Y*, or dependent variables, you have. It expects your independent variables to be listed first. Notice that we have used `reduceRegion` rather than just `reduce`. This is because we are reducing over the Turin geometry, meaning that the reducer's activity is across multiple pixels comprising an area, rather than "downward" through a set of bands or an `ImageCollection`. As mentioned earlier, you will likely get an error that there are too many pixels included in your regression if you do not use the `bestEffort` variable.

```
// Compute ordinary least squares regression coefficients
using
// the linearRegression reducer.
var linearRegression = trainingImageLR.reduceRegion({
    reducer: ee.Reducer.linearRegression({
        numX: 10,
        numY: 1
    }),
    geometry: Turin,
    scale: 30,
    bestEffort: true
});
```

There is also a robust linear regression reducer (`ee.Reducer.robustLinearRegression`). This linear regression approach uses regression residuals to de-weight outliers in the data following (O'Leary 1990).

Let's inspect the results. We'll focus on the results of `linearRegression`, but the same applies to inspecting the results of the `robustLinearRegression` reducer.

```
// Inspect the results.
print('Linear regression results:', linearRegression);
```

The output is an object with two properties. First is a list of coefficients (in the order specified by the inputs list) and second is the root-mean-square residual.

To apply the regression coefficients to make a prediction over the entire area, we will first turn the output coefficients into an image, then perform the requisite math.

```
// Extract the coefficients as a list.
var coefficients =
ee.Array(linearRegression.get('coefficients'))
    .project([0])
    .toList();

print('Coefficients', coefficients);
```

This code extracts the coefficients from our linear regression and converts them to a list. When we first extract the coefficients, they are in a matrix (essentially 10 lists of 1 coefficient). We use `project` to remove one of the dimensions from the matrix so that they are in the correct data format for subsequent steps.

Now we will create the predicted tree cover based on the linear regression. First, we create an image stack starting with the constant (1) and use `addBands` to add the prediction bands. Next, we multiply each of the coefficients by their respective *X* (independent variable band) using `multiply` and then sum all of these using `ee.Reducer.sum` in order to create the estimate of our dependent variable. Note that we are using the `rename` function to rename the band (if you don't rename the band, it will have the default name "sum" from the reducer function).

```
// Create the predicted tree cover based on linear
regression.
var predictedTreeLR = ee.Image(1)
    .addBands(landsat8filtered.select(predictionBands))
    .multiply(ee.Image.constant(coefficients))
    .reduce(ee.Reducer.sum())
    .rename('predictedTreeLR')
    .clip(landsat8filtered.geometry());

Map.addLayer(predictedTreeLR, {
    min: 0,
    max: 100
}, 'LR prediction');
```

Carefully inspect this result using the satellite imagery provided by Google and the input Landsat data (Fig. 8.2). Does the model predict high forest cover in forested areas? Does it predict low forest cover in unforested areas, such as urban areas and agricultural areas? Note where the model is making mistakes. Are there any patterns? For example, look at the mountainous slopes. One side has a high value and the other side has a low value for predicted forest cover. However, it appears that neither side of the mountain has vegetation above the treeline. This suggests a fault with the model having to do with the aspect of the terrain (the compass direction in which the terrain surface faces) and some of the bands used.

When you encounter model performance issues that you find unsatisfactory, you may consider adding or subtracting independent variables, testing other regression functions, collecting more training data, or all of the above.

Code Checkpoint F30b. The book's repository contains a script that shows what your code should look like at this point.

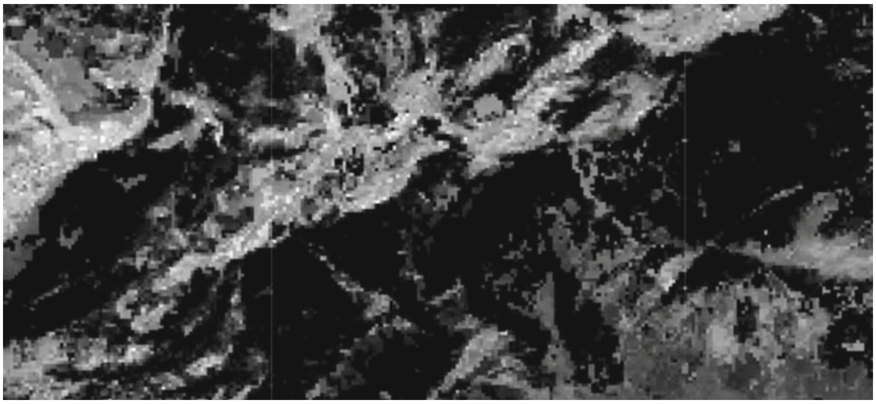


Fig. 8.2 Examining the results of the linear regression model. Note that the two sides of this mountain have different forest prediction values, despite being uniformly forested. This suggests there might be a fault with the model having to do with the aspect

8.3.2 Section 3: Nonlinear Regression

Earth Engine also allows the user to perform nonlinear regression. Nonlinear regression allows for nonlinear relationships between the independent and dependent variables. Unlike linear regression, which is implemented via reducers, this nonlinear regression function is implemented by the classifier library. The classifier library also includes supervised and unsupervised classification approaches that were discussed in Chap. 6 as well as some of the tools used for accuracy assessment in Chap. 7.

For example, a classification and regression tree (CART; see Breiman et al. 2017) is a machine learning algorithm that can learn nonlinear patterns in your data. Let's reuse our dependent variables and independent variables (Landsat prediction bands) from above to train the CART in regression mode.

For CART we need to have our input data as a feature collection. For more on feature collections, see the chapters in Part V. Here, we first need to create a training data set based on the independent and dependent variables we used for the linear regression section. We will not need the constant term.

```
/////
// Start Non-linear Regression
/////
// Create the training data stack.
var trainingImageCART =
ee.Image(landsat8filtered.select(predictionBands))
    .addBands(percentTree2020);
```

Now sample the image stack to create the feature collection training data needed. Note that we could also have used this approach in previous regressions instead of the bestEffort approach we did use.

```
// Sample the training data stack.
var trainingData = trainingImageCART.sample({
    region: Turin,
    scale: 30,
    numPixels: 1500,
    seed: 5
});

// Examine the CART training data.
print('CART training data', trainingData);
```

Now run the regression. The CART regression function must first be trained using the trainingData. For the train function, we need to provide the features to use to train (trainingData), the name of the dependent variable (classProperty), and the name of the independent variables (inputProperties). Remember that you can find these band names if needed by inspecting the trainingData item.

```
// Run the CART regression.
var cartRegression = ee.Classifier.smileCart()
    .setOutputMode('REGRESSION')
    .train({
        features: trainingData,
        classProperty: 'Percent_Tree_Cover',
        inputProperties: predictionBands
    });
```

Now we can use this object to make predictions over the input imagery and display the result:

```
// Create a prediction of tree cover based on the CART
regression.
var cartRegressionImage =
landsat8filtered.select(predictionBands)
    .classify(cartRegression, 'cartRegression');

Map.addLayer(cartRegressionImage, {
    min: 0,
    max: 100
}, 'CART regression');
```

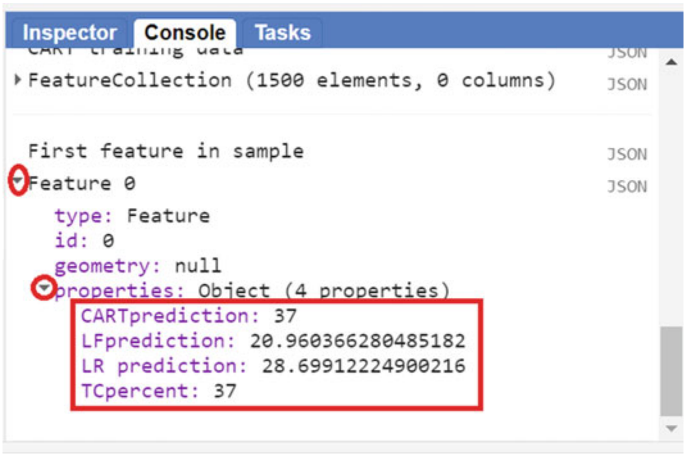
Turn on the satellite imagery from Google and examine the output of the CART regression using this imagery and the Landsat 8 imagery.

8.3.3 Section 4: Assessing Regression Performance Through RMSE

A standard measure of performance for regression models is the root-mean-square error (RMSE), or the correlation between known and predicted values. The RMSE is calculated as follows:

$$\text{RMSE} = \sqrt{\sum \frac{(\text{Predicted}_i - \text{Actual}_i)^2}{n}} \tag{8.1}$$

To assess the performance, we will create a sample from the percent tree cover layer and from each regression layer (the predictions from the linear fit, the linear regression, and CART regression). First, using the `ee.Image.cat` function, we will concatenate all of the layers into one single image where each band of this image contains the percent tree cover and the predicted values from the different regressions. We use the `rename` function (Chap. 2) to rename the bands to meaningful names (tree cover percentage and each model). Then we will extract 500 sample points (the "*n*" from the equation above) from the single image using the `sample` function to calculate the RMSE for each regression model. We print the first feature from the sample collection to visualize its properties—values from the percent tree cover and regression models at that point (Fig. 8.3), as an example.



```
CART training data                                    JSON
▸ FeatureCollection (1500 elements, 0 columns)        JSON

First feature in sample                               JSON
▾ Feature 0                                           JSON
    type: Feature
    id: 0
    geometry: null
  ▾ properties: Object (4 properties)
      CARTprediction: 37
      LFprediction: 20.960366280485182
      LR prediction: 28.69912224900216
      TCpercent: 37
```

Fig. 8.3 First point from the sample FeatureCollection showing the actual tree cover percentage and the regression predictions. These values may differ from what you see

```
/////
// Calculating RMSE to assess model performance
/////

// Concatenate percent tree cover image and all
predictions.
var concat = ee.Image.cat(percentTree2020,
        predictedTree,
        predictedTreeLR,
        cartRegressionImage)
    .rename([
        'TCpercent',
        'LFprediction',
        'LRprediction',
        'CARTprediction'
    ]);

// Sample pixels
var sample = concat.sample({
    region: Turin,
    scale: 30,
    numPixels: 500,
    seed: 5
});

print('First feature in sample', sample.first());
```

In Earth Engine, the RMSE can be calculated by steps. We first define a function to calculate the squared difference between the predicted value and the actual value. We do this for each regression result.

```
// First step: This function computes the squared
difference between
// the predicted percent tree cover and the known percent
tree cover
var calculateDiff = function(feature) {
    var TCpercent = ee.Number(feature.get('TCpercent'));
    var diffLFsq = ee.Number(feature.get('LFprediction'))
        .subtract(TCpercent).pow(2);
    var diffLRsq = ee.Number(feature.get('LRprediction'))
        .subtract(TCpercent).pow(2);
    var diffCARTsq =
ee.Number(feature.get('CARTprediction'))
        .subtract(TCpercent).pow(2);

    // Return the feature with the squared difference set
to a 'diff' property.
    return feature.set({
        'diffLFsq': diffLFsq,
        'diffLRsq': diffLRsq,
        'diffCARTsq': diffCARTsq
    });
};
```

Now, we can apply this function to our sample and use the `reduceColumns` function to take the mean value of the squared differences and then calculate the square root of the mean value.

```
// Second step: Calculate RMSE for population of difference
pairs.
var rmse = ee.List([ee.Number(
    // Map the difference function over the collection.
    sample.map(calculateDiff)
    // Reduce to get the mean squared difference.
    .reduceColumns({
        reducer: ee.Reducer.mean().repeat(3),
        selectors: ['diffLFsq', 'diffLRsq',
            'diffCARTsq'
        ]
    }).get('mean')
    // Flatten the list of lists.
)]).flatten().map(function(i) {
    // Take the square root of the mean square differences.
    return ee.Number(i).sqrt();
});

// Print the result.
print('RMSE', rmse);
```

Note the following (do not worry too much about fully understanding each item at this stage of your learning; just keep in mind that this code block calculates the RMSE value):

- We start by casting an `ee.List` since we are calculating three different values—which is also the reason to cast `ee.Number` at the beginning. Also, it is not possible to map a function to a `ee.Number` object—another reason why we need the `ee.List`.
- Since we have three predicted values we used `repeat(3)` for the `reducer` parameter of the `reduceColumns` function—i.e., we want the mean value for each of the `selectors` (the squared differences).
- The direct output of `reduceColumns` is a dictionary, so we need to use `get('mean')` to get these specific key values.
- At this point, we have a "list of lists," so we use `flatten` to dissolve it into one list
- Finally, we map the function to calculate the square root of each mean value for the three results.
- The RMSE values are in the order of the `selectors`; thus, the first value is the linear fit RMSE, followed by the linear regression RMSE, and finally the CART RMSE.

Inspect the RMSE values on the **Console**. Which regression performed best? The lower the RMSE, the better the result.

Code Checkpoint F30c. The book's repository contains a script that shows what your code should look like at this point.

8.4 Synthesis

Assignment 1. Examine the CART output you just created. How does the nonlinear CART output compare with the linear regression we ran previously? Use the inspect tool to check areas of known forest and non-forest (e.g., agricultural and urban areas). Do the forested areas have a high predicted percent tree cover (will display as white)? Do the non-forested areas have a low predicted percent tree cover (will display as black)? Do the alpine areas have low predicted percent tree cover, or do they have the high/low pattern based on aspect seen in the linear regression?

Assignment 2. Revisit our percent tree cover regression examples. In these, we used a number of bands from Landsat 8, but there are other indices, transforms, and datasets that we could use for our independent variables.

For this assignment, work to improve the performance of this regression. Consider adding or subtracting independent variables, testing other regression functions, using more training data (a larger geometry for Turin), or all of the above. Don't forget to document each of the steps you take. What model settings and inputs are you using for each model run?

To improve the model, think about what you know about tree canopies and tree canopy cover. What spectral signature do the deciduous trees of this region have? What indices are useful for detecting trees? How do you distinguish trees from other vegetation? If the trees in this region are deciduous, what time frame would be best to use for the Landsat 8 imagery? Consider seasonality—how do these forests look in summer versus winter?

Practice your visual assessment skills. Ask critical questions about the results of your model iterations. Why do you think one model is better than another?

8.5 Conclusion

Regression is a fundamental tool that you can use to analyze remote-sensing imagery. Regression specifically allows you to analyze and predict numeric dependent variables, while classification allows for the analysis and prediction of categorical variables (see Chap. 6). Regression analysis in Earth Engine is flexible and implemented via reducers (linear regression approaches) and via classifiers (nonlinear regression approaches). Other forms of regression will be discussed in Chap. 18 and the chapters that follow.

References

Breiman L, Friedman JH, Olshen RA, Stone CJ (2017) Classification and regression trees. Routledge

Goetz S, Steinberg D, Dubayah R, Blair B (2007) Laser remote sensing of canopy habitat heterogeneity as a predictor of bird species richness in an eastern temperate forest, USA. Remote Sens Environ 108:254–263. https://doi.org/10.1016/j.rse.2006.11.016

Maeda EE, Heiskanen J, Thijs KW, Pellikka PKE (2014) Season-dependence of remote sensing indicators of tree species diversity. Remote Sens Lett 5:404–412. https://doi.org/10.1080/2150704X.2014.912767

O'Leary DP (1990) Robust regression computation using iteratively reweighted least squares. SIAM J Matrix Anal Appl 11:466–480. https://doi.org/10.1137/0611032

Zhou X, Zhu X, Dong Z et al (2016) Estimation of biomass in wheat using random forest regression algorithm and remote sensing data. Crop J 4:212–219. https://doi.org/10.1016/j.cj.2016.01.008

Advanced Pixel-Based Image Transformations

9

Karen Dyson, Andréa Puzzi Nicolau, Nicholas Clinton, and David Saah

Overview

Using bands and indices is often not sufficient for obtaining high-quality image classifications. This chapter introduces the idea of more complex pixel-based band manipulations that can extract more information for analysis, building on what was presented in Part I and Part II. We will first learn how to manipulate images with expressions and then move on to more complex linear transformations that leverage matrix algebra.

Learning Outcomes

- Understanding what linear transformations are and why pixel-based image transformations are useful.

K. Dyson · A. P. Nicolau · D. Saah
Spatial Informatics Group, Pleasanton, CA, USA
e-mail: kdyson@sig-gis.com

A. P. Nicolau
e-mail: apnicolau@sig-gis.com

K. Dyson · A. P. Nicolau
SERVIR-Amazonia, Cali, Colombia

N. Clinton
Google LLC, Inc, Mountain View, CA, USA
e-mail: nclinton@google.com

D. Saah (✉)
University of San Francisco, San Francisco, CA, USA
e-mail: dssaah@usfca.edu

K. Dyson
Dendrolytics, Seattle, WA, USA

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_9

- Learning how to use expressions for band manipulation.
- Being introduced to some of the most common types of linear transformations.
- Using arrays and functions in Earth Engine to apply linear transformations to images.

Assumes you know how to

- Import images and image collections, filter, and visualize (Part I).
- Perform basic image analysis: select bands, compute indices, create masks (Part II).
- Use drawing tools to create points, lines, and polygons (Chap. 6).
- Understand basic operations of matrices.

9.1 Introduction to Theory

Image transformations are essentially complex calculations among image bands that can leverage matrix algebra and more advanced mathematics. In return for their greater complexity, they can provide larger amounts of information in a few variables, allowing for better classification (Chap. 6), time-series analysis (Chaps. 17 through 21), and change detection (Chap. 16) results. They are particularly useful for difficult use cases, such as classifying images with heavy shadow or high values of greenness and distinguishing faint signals from forest degradation.

In this chapter, we explore linear transformations, which are linear combinations of input pixel values. This approach is pixel-based—that is, each pixel in the remote sensing image is treated separately.

We introduce here some of the best-established linear transformations used in remote sensing (e.g., tasseled cap transformations) along with some of the newest (e.g., spectral unmixing). Researchers are continuing to develop new applications of these methods. For example, when used together, spectral unmixing and time-series analysis (Chaps. 17 through 21.) are effective at detecting and monitoring tropical forest degradation (Bullock et al. 2020, Souza et al, 2013). As forest degradation is notoriously hard to monitor and also responsible for significant carbon emissions, this represents an important step forward. Similarly, using tasseled cap transformations alongside classification approaches allowed researchers to accurately map cropping patterns with high spatial and thematic resolution (Rufin et al. 2019).

9.2 Practicum

In this practicum, we will first learn how to manipulate images with expressions and then move on to more complex linear transformations that leverage matrix algebra. In Earth Engine, these types of linear transformations are applied by treating pixels as arrays of band values. An array in Earth Engine is a list of lists, and by using arrays, you can define matrices (i.e., two-dimensional arrays), which are the basis of linear transformations. Earth Engine uses the word "axis" to refer to what are commonly called the rows (axis 0) and columns (axis 1) of a matrix.

9.2.1 Section 1: Manipulating Images with Expressions

Arithmetic Calculation of EVI

The Enhanced Vegetation Index (EVI) is designed to minimize saturation and other issues with NDVI, an index discussed in detail in Chap. 5 (Huete et al. 2002). In areas of high chlorophyll (e.g., rainforests), EVI does not saturate (i.e., reach maximum value) the same way that NDVI does, making it easier to examine variation in the vegetation in these regions. The generalized equation for calculating EVI is

$$\mathrm{EVI} = G \times \frac{(\mathrm{NIR} - \mathrm{Red})}{(\mathrm{NIR} + C1 \times \mathrm{RED} - C2 \times \mathrm{Blue} + L)} \tag{9.1}$$

where G, $C1$, $C2$, and L are constants. You do not need to memorize these values, as they have been determined by other researchers and are available online for you to look up. For Sentinel-2, the equation is

$$\mathrm{EVI} = 2.5 \times \frac{(B8 - B4)}{(B8 + 6 \times B4 - 7.5 \times B2 + 1)} \tag{9.2}$$

Using the basic arithmetic, we learned previously in 5, let us calculate and then display the EVI for the Sentinel-2 image. We will need to extract the bands and then divide by `10,000` to account for the scaling in the dataset. You can find out more by navigating to the dataset information.

```
// Import and filter imagery by location and date.
var sfoPoint = ee.Geometry.Point(-122.3774, 37.6194);

var sfoImage = ee.ImageCollection('COPERNICUS/S2')
    .filterBounds(sfoPoint)
    .filterDate('2020-02-01', '2020-04-01')
    .first();
Map.centerObject(sfoImage, 11);

// Calculate EVI using Sentinel 2

// Extract the bands and divide by 10,000 to account for
scaling done.
var nirScaled = sfoImage.select('B8').divide(10000);
var redScaled = sfoImage.select('B4').divide(10000);
var blueScaled = sfoImage.select('B2').divide(10000);

// Calculate the numerator, note that order goes from left
to right.
var numeratorEVI =
(nirScaled.subtract(redScaled)).multiply(2.5);

// Calculate the denominator.
var denomClause1 = redScaled.multiply(6);
var denomClause2 = blueScaled.multiply(7.5);
var denominatorEVI = nirScaled.add(denomClause1)
    .subtract(denomClause2).add(1);

// Calculate EVI and name it.
var EVI =
numeratorEVI.divide(denominatorEVI).rename('EVI');

// And now map EVI using our vegetation palette.
var vegPalette = ['red', 'white', 'green'];
var visParams = {min: -1, max: 1, palette: vegPalette};
    Map.addLayer(EVI, visParams, 'EVI');
```

Using an Expression to Calculate EVI
The EVI code works (Fig. 9.1), but creating a large number of variables and explicitly calling addition, subtraction, multiplication, and division can be confusing and introduces the chance for errors. In these circumstances, you can create a function to make the steps more robust and easily repeatable. In another simple strategy outlined below, Earth Engine has a way to define an expression to achieve the same result.

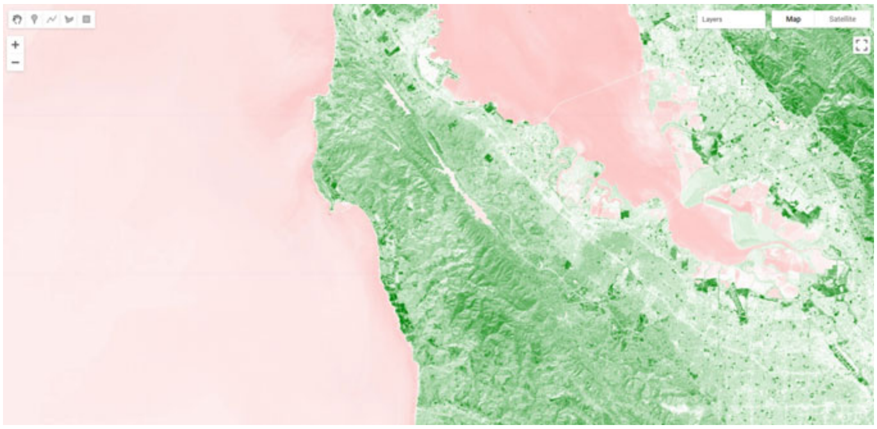


Fig. 9.1 EVI displayed for sentinel-2 over San Francisco

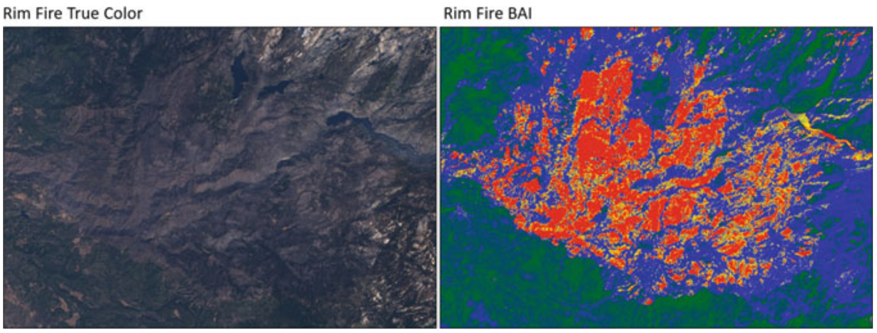


Fig. 9.2 Comparison of true color and BAI for the rim fire

```
// Calculate EVI.
var eviExpression = sfoImage.expression({
    expression: '2.5 * ((NIR - RED) / (NIR + 6 * RED - 7.5
* BLUE + 1))',
    map: { // Map between variables in the expression and
images.
        'NIR': sfoImage.select('B8').divide(10000),
        'RED': sfoImage.select('B4').divide(10000),
        'BLUE': sfoImage.select('B2').divide(10000)
    }
});

// And now map EVI using our vegetation palette.
Map.addLayer(eviExpression, visParams, 'EVI Expression');
```

The expression is defined first as a string using human readable names. We then define these names by selecting the proper bands.

Code Checkpoint F31a. The book's repository contains a script that shows what your code should look like at this point.

Using an Expression to Calculate BAI

Now that we have seen how expressions work, let us use an expression to calculate another index. Martin (1998) developed the Burned Area Index (BAI) to assist in the delineation of burn scars and assessment of burn severity. It relies on fires leaving ash and charcoal; fires that do not create ash or charcoal and old fires where the ash and charcoal have been washed away or covered will not be detected well. BAI computes the spectral distance of each pixel to a spectral reference point that burned areas tend to be similar to. Pixels that are far away from this reference (e.g., healthy vegetation) will have a very small value while pixels that are close to this reference (e.g., charcoal from fire) will have very large values.

$$\mathrm{BAI} = \frac{1}{\left((\rho c_r - \mathrm{Red})^2 + (\rho c_{\mathrm{nir}} - \mathrm{NIR})^2\right)} \tag{9.3}$$

There are two constants in this equation: ρc_r is a constant for the red band, equal to 0.1; and ρc_{nir} is for the NIR band, equal to 0.06.

To examine burn indices, load an image from 2013 showing the Rim Fire in the Sierra Nevada, California mountains. We will use Landsat 8 to explore this fire. Enter the code below in a new script.

```
// Examine the true-color Landsat 8 images for the 2013 Rim
Fire.
var burnImage =
ee.ImageCollection('LANDSAT/LC08/C02/T1_TOA')
    .filterBounds(ee.Geometry.Point(-120.083, 37.850))
    .filterDate('2013-09-15', '2013-09-27')
    .sort('CLOUD_COVER')
    .first();

Map.centerObject(ee.Geometry.Point(-120.083, 37.850), 11);

var rgbParams = {
    bands: ['B4', 'B3', 'B2'],
    min: 0,
    max: 0.3
};
Map.addLayer(burnImage, rgbParams, 'True-Color Burn
Image');
```

Examine the true-color display of this image. Can you spot the fire? If not, the BAI may help. As with EVI, use an expression to compute BAI in Earth Engine, using the equation above and what you know about Landsat 8 bands:

```
// Calculate BAI.
var bai = burnImage.expression(
    '1.0 / ((0.1 - RED)**2 + (0.06 - NIR)**2)', {
        'NIR': burnImage.select('B5'),
        'RED': burnImage.select('B4'),
    });
```

Display the result.

```
// Display the BAI image.
var burnPalette = ['green', 'blue', 'yellow', 'red'];
Map.addLayer(bai, {
    min: 0,
    max: 400,
    palette: burnPalette
}, 'BAI');
```

The burn area should be more obvious in the BAI visualization (Fig. 9.1, right panel). Note that the minimum and maximum values here are larger than what we have used for Landsat. At any point, you can inspect a layer's bands using what you have already learned to see the minimum and maximum values, which will give you an idea of what to use here.

Code Checkpoint F31b. The book's repository contains a script that shows what your code should look like at this point.

9.2.2 Section 2: Manipulating Images with Matrix Algebra

Now that we have covered expressions, let us turn our attention to linear transformations that leverage matrix algebra.

Tasseled Cap Transformation

The first of these is the tasseled cap (TC) transformation. TC transformations are a class of transformations which, when graphed, look like a wooly hat with a tassel. The most common implementation is used to maximize the separation between different growth stages of wheat, an economically important crop. As wheat grows, the field progresses from bare soil, to green plant development, to yellow plant

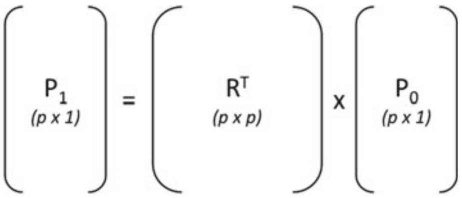


Fig. 9.3 Visualization of the matrix multiplication used to transform the original vector of band values (p_0) for a pixel to the rotated values (p_1) for that same pixel

ripening, to field harvest. Separating these stages for many fields over a large area was the original purpose of the tasseled cap transformation.

Based on observations of agricultural land covers in the combined near-infrared and red spectral space, Kauth and Thomas (1976) devised a rotational transform of the form:

$$\boldsymbol{p}_1 = \boldsymbol{R}^T \boldsymbol{p}_0 \tag{9.4}$$

where $\boldsymbol{p}_0$ is the original $p \times 1$ pixel vector (a stack of the p band values for that specific pixel as an array), and the matrix $\boldsymbol{R}$ is an orthonormal basis of the new space in which each column is orthogonal to one another (therefore, $\boldsymbol{R}^T$ is its transpose), and the output $\boldsymbol{p}_1$ is the rotated stack of values for that pixel (Fig. 9.3).

Kauth and Thomas found $\boldsymbol{R}$ by defining the first axis of their transformed space to be parallel to the soil line in Fig. 9.4. The first column was chosen to point along the major axis of soils and the values derived from Landsat imagery at a given point in Illinois, USA. The second column was chosen to be orthogonal to the first column and point toward what they termed "green stuff," i.e., green vegetation. The third column is orthogonal to the first two and points toward the "yellow stuff," e.g., ripening wheat and other grass crops. The final column is orthogonal to the first three and is called "nonesuch" in the original derivation—that is, akin to noise.

The $\boldsymbol{R}$ matrix has been derived for each of the Landsat satellites, including Landsat 5 (Crist 1985), Landsat 7, (Huang et al. 2002) Landsat 8 (Baig et al. 2014), and others. We can implement this transform in Earth Engine with arrays. Specifically, let us create a new script and make an array of TC coefficients for Landsat 5's Thematic Mapper (TM) instrument:

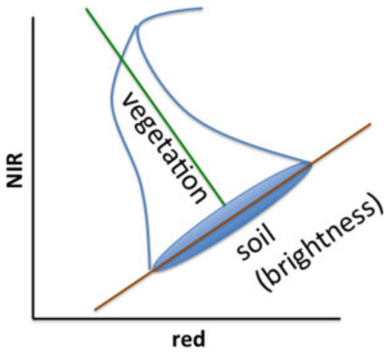


Fig. 9.4 Visualization of the tasseled cap transformation. This is a graph of two dimensions of a higher dimensional space (one for each band). The NIR and red bands represent two dimensions of p_0, while the vegetation and soil brightness represent two dimensions of p_1. You can see that there is a rotation caused by R^T

```
/////
// Manipulating images with matrices
/////

// Begin Tasseled Cap example.
var landsat5RT = ee.Array([
    [0.3037, 0.2793, 0.4743, 0.5585, 0.5082, 0.1863],
    [-0.2848, -0.2435, -0.5436, 0.7243, 0.0840, -0.1800],
    [0.1509, 0.1973, 0.3279, 0.3406, -0.7112, -0.4572],
    [-0.8242, 0.0849, 0.4392, -0.0580, 0.2012, -0.2768],
    [-0.3280, 0.0549, 0.1075, 0.1855, -0.4357, 0.8085],
    [0.1084, -0.9022, 0.4120, 0.0573, -0.0251, 0.0238]
]);

print('RT for Landsat 5', landsat5RT);
```

Note that the structure we just made is a list of six lists, which is then converted to an Earth Engine `ee.Array` object. The six-by-six array of values corresponds to the linear combinations of the values of the six non-thermal bands of the TM instrument: bands 1–5 and 7. To examine how Earth Engine ingests the array, view the output of the `print` function to display the array in the **Console**. You can explore how the different elements of the array match with how the array were defined using `ee.Array`.

The next steps of this lab center on the small town of Odessa in eastern Washington, USA. You can search for “Odessa, WA, USA” in the search bar. We use the state abbreviation here because this is how Earth Engine displays it. The search will take you to the town and its surroundings, which you can explore with the **Map** or **Satellite** options in the upper right part of the display. In the code below, we will define a point in Odessa and center the display on it to view the results at a good zoom level.

Since these coefficients are for the TM sensor at satellite reflectance (top of atmosphere), we will access a less-cloudy Landsat 5 scene. We will access the collection of Landsat 5 images, filter them, then sort by increasing cloud cover, and take the first one.

```
// Define a point of interest in Odessa, Washington, USA.
var point = ee.Geometry.Point([-118.7436019417829,
47.18135755009023]);
Map.centerObject(point, 10);

// Filter to get a cloud free image to use for the TC.
var imageL5 = ee.ImageCollection('LANDSAT/LT05/C02/T1_TOA')
    .filterBounds(point)
    .filterDate('2008-06-01', '2008-09-01')
    .sort('CLOUD_COVER')
    .first();

//Display the true-color image.
var trueColor = {
    bands: ['B3', 'B2', 'B1'],
    min: 0,
    max: 0.3
};
Map.addLayer(imageL5, trueColor, 'L5 true color');
```

To do the matrix multiplication, first convert the input image from a multi-band image (where for each band, each pixel stores a single value) to an *array image*. An array image is a higher-dimension image in which each pixel stores an array of values for a band. (Array images are encountered and discussed in more detail in part IV.) You will use bands 1–5 and 7 and the `toArray` function:

```
var bands = ['B1', 'B2', 'B3', 'B4', 'B5', 'B7'];

// Make an Array Image, with a one dimensional array per
pixel.
// This is essentially a list of values of length 6,
// one from each band in variable 'bands.'
var arrayImage1D = imageL5.select(bands).toArray();

// Make an Array Image with a two dimensional array per
pixel,
// of dimensions 6x1. This is essentially a one column
matrix with
// six rows, with one value from each band in 'bands.'
// This step is needed for matrix multiplication (p0).
var arrayImage2D = arrayImage1D.toArray(1);
```

The 1 refers to the columns (the "first" axis in Earth Engine) to create a 6 row by 1 column array for p_0 (Fig. 9.3).

Next, we complete the matrix multiplication of the tasseled cap linear transformation using the matrixMultiply function, then convert the result back to a multi-band image using the arrayProject and arrayFlatten functions:

```
//Multiply RT by p0.
var tasselCapImage = ee.Image(landsat5RT)
    // Multiply the tasseled cap coefficients by the array
    // made from the 6 bands for each pixel.
    .matrixMultiply(arrayImage2D)
    // Get rid of the extra dimensions.
    .arrayProject([0])
    // Get a multi-band image with TC-named bands.
    .arrayFlatten(
        [
            ['brightness', 'greenness', 'wetness',
'fourth', 'fifth',
                'sixth'
            ]
        ]);
```

Finally, display the result:

```
var vizParams = {
    bands: ['brightness', 'greenness', 'wetness'],
    min: -0.1,
    max: [0.5, 0.1, 0.1]
};
Map.addLayer(tasselCapImage, vizParams, 'TC components');
```

This maps `brightness` to red, `greenness` to green, and `wetness` to blue. Your resulting layer will contain a high amount of contrast (Fig. 9.5). Water appears blue, healthy irrigated crops are the bright circles, and drier crops are red. We have chosen this area near Odessa because it is naturally dry, and the irrigated crops make the patterns identified by the tasseled cap transformation particularly striking.

If you would like to see how the array image operations work, you can consider building `tasselCapImage`, one step at a time. You can assign the result of `matrixMultiply` operation to its own variable then map the result. Then, do the `arrayProject` command on that new variable into a second new image and map that result. Then, do the `arrayFlatten` call on that result to produce

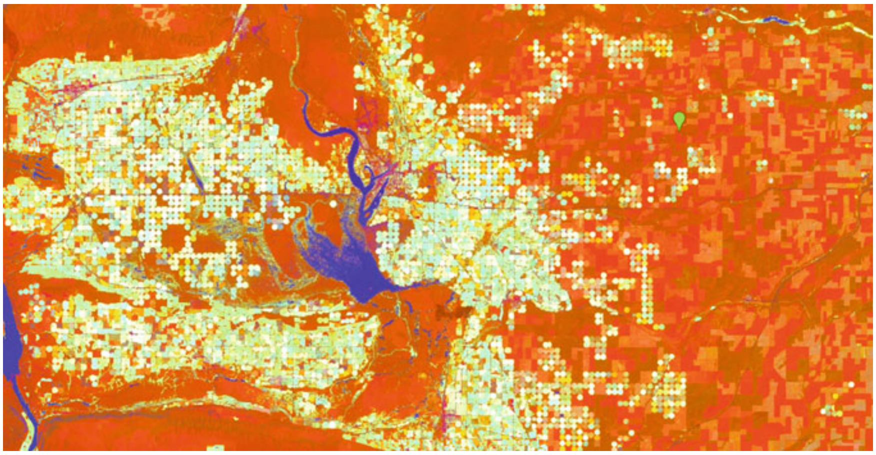


Fig. 9.5 Output of the tasseled cap transformation. Water appears blue, green irrigated crops are the bright circles, and dry crops are red

tasselCapImage as before. You can then use the **Inspector** tool to view these details of how the data is processed as tasselCapImage is built.

Principal Component Analysis

Like the TC transform, the principal component analysis (PCA) transform is a rotational transform. PCA is an orthogonal linear transformation—essentially, it mathematically transforms the data into a new coordinate system where all axes are orthogonal. The first axis, also called a coordinate, is calculated to capture the largest amount of variance of the dataset, the second captures the second-greatest variance, and so on.

Because these are calculated to be orthogonal, the principal components are uncorrelated. PCA can be used as a dimension reduction tool, as most of the variation in a dataset with n axes can be captured in $n - x$ axes. This is a very brief explanation; if you want to learn more about PCA and how it works, there are many excellent statistical texts and online resources on the subject.

To demonstrate the practical application of PCA applied to an image, import the Landsat 8 TOA image, and name it imageL8. First, we will convert it to an array image:

```
// Begin PCA example.

// Select and map a true-color L8 image.
var imageL8 = ee.ImageCollection('LANDSAT/LC08/C02/T1_TOA')
    .filterBounds(point)
    .filterDate('2018-06-01', '2018-09-01')
    .sort('CLOUD_COVER')
    .first();

var trueColorL8 = {
    bands: ['B4', 'B3', 'B2'],
    min: 0,
    max: 0.3
};
Map.addLayer(imageL8, trueColorL8, 'L8 true color');

// Select which bands to use for the PCA.
var PCAbands = ['B2', 'B3', 'B4', 'B5', 'B6', 'B7', 'B10',
'B11'];

// Convert the Landsat 8 image to a 2D array for the later
matrix
// computations.
var arrayImage = imageL8.select(PCAbands).toArray();
```

In the next step, use the reduceRegion method and the ee.Reducer.covariance function to compute statistics (in this case the covariance of bands) for the image.

```
// Calculate the covariance using the reduceRegion method.
var covar = arrayImage.reduceRegion({
    reducer: ee.Reducer.covariance(),
    maxPixels: 1e9
});

// Extract the covariance matrix and store it as an array.
var covarArray = ee.Array(covar.get('array'));
```

Note that the result of the reduction is an object with one property, array, that stores the covariance matrix. We use the ee.Array.get function to extract the covariance matrix and store it as an array.

Now that we have a covariance matrix based on the image, we can perform an eigen analysis to compute the eigenvectors that we will need to perform the PCA. To do this, we will use the eigen function. Again, if these terms are unfamiliar to you, we suggest one of the many excellent statistics textbooks or online resources. Compute the eigenvectors and eigenvalues of the covariance matrix:

```
//Compute and extract the eigenvectors
var eigens = covarArray.eigen();
```

The eigen function outputs both eigenvectors and the eigenvalues. Since we need the eigenvectors for the PCA, we can use the slice function for arrays to extract them. The eigenvectors are stored in the 0th position of the 1-axis.

```
var eigenVectors = eigens.slice(1, 1);
```

Now, we perform matrix multiplication using these eigenVectors and the arrayImage we created earlier. This is the same process that we used with the tasseled cap components. Each multiplication results in a principal component.

```
// Perform matrix multiplication
var principalComponents = ee.Image(eigenVectors)
    .matrixMultiply(arrayImage.toArray(1));
```

Finally, convert back to a multi-band image and display the first principal component (`pc1`):

```
var pcImage = principalComponents
    // Throw out an unneeded dimension, [[]] -> [].
    .arrayProject([0])
    // Make the one band array image a multi-band image, []
-> image.
    .arrayFlatten([
        ['pc1', 'pc2', 'pc3', 'pc4', 'pc5', 'pc6', 'pc7',
'pc8']
    ]);

// Stretch this to the appropriate scale.
Map.addLayer(pcImage.select('pc1'), {}, 'pc1');
```

When first displayed, the PC layer will be all black. Use the layer manager to stretch the result in greyscale by hovering over **Layers**, then **PC**, and then clicking the gear icon next to **PC**. Note how the range (minimum and maximum values) changes based on the stretch you choose.

What do you observe? Try displaying some of the other principal components. How do they differ from each other? What do you think each band is capturing? Hint: You will need to recreate the stretch for each principal component you try to map.

Look at what happens when you try to display '`pc1`', '`pc3`', and '`pc4`', for example, in a three-band display. Because the values of each principal component band differ substantially, you might see a gradation of only one color in your output. To control the display of multiple principal component bands together, you will need to use lists in order to specify the `min` and `max` values individually for each principal component band.

Once you have determined which bands you would like to plot, input the `min` and `max` values for each band, making sure they are in the correct order.

```
//The min and max values will need to change if you map
different bands or locations.
var visParamsPCA = {
    bands: ['pc1', 'pc3', 'pc4'],
    min: [-455.09, -2.206, -4.53],
    max: [-417.59, -1.3, -4.18]
};

Map.addLayer(pcImage, visParamsPCA, 'PC_multi');
```

Examine the PCA map (Fig. 9.6). Unlike with the tasseled cap transformation, PCA does not have defined output axes. Instead, each axis dynamically captures some aspect of the variation within the dataset (if this does not make sense to you, please review an online resource on the statistical theory behind PCA). Thus, the mapped PCA may differ substantially based on where you have performed the PCA and which bands you are mapping.

Code Checkpoint F31c. The book's repository contains a script that shows what your code should look like at this point.

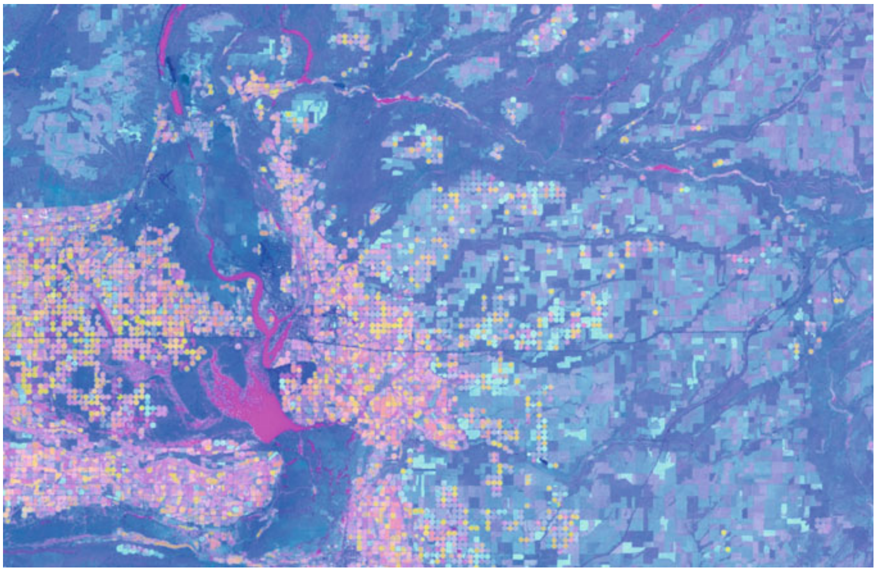


Fig. 9.6 Output of the PCA transformation near Odessa, Washington, USA

9.2.3 Section 3: Spectral Unmixing

If we think about a single pixel in our dataset—a 30×30 m space corresponding to a Landsat pixel, for instance—it is likely to represent multiple physical objects on the ground. As a result, the spectral signature for the pixel is a mixture of the “pure” spectra of each object existing in that space. For example, consider a Landsat pixel of forest. The spectral signature of the pixel is a mixture of trees, understory, shadows cast by the trees, and patches of soil visible through the canopy.

The linear spectral unmixing model is based on this assumption (Schultz et al. 2016, Souza 2005). The pure spectra, called endmembers, are from land cover classes such as water, bare land, and vegetation. These endmembers represent the spectral signature of pure spectra from ground features, such as only bare ground. The goal is to solve the following equation for $\boldsymbol{f}$, the $P \times 1$ vector of endmember fractions in the pixel:

$$p = Sf \tag{9.5}$$

$\boldsymbol{S}$ is a $B \times P$ matrix in which B is the number of bands and the columns are P pure endmember spectra and $\boldsymbol{p}$ is the $B \times 1$ pixel vector when there are B bands (Fig. 9.7). We know **p,** and we can define the endmember spectra to get $\boldsymbol{S}$ such that we can solve for $\boldsymbol{f}$.

We will use the Landsat 8 image for this exercise. In this example, the number of bands (B) is six.

```
// Specify which bands to use for the unmixing.
var unmixImage = imageL8.select(['B2', 'B3', 'B4', 'B5',
'B6', 'B7']);
```

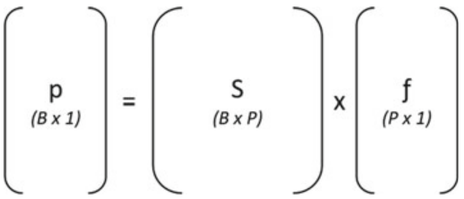


Fig. 9.7 Visualization of the matrix multiplication used to transform the original vector of band values (p) for a pixel to the endmember values ($\boldsymbol{f}$) for that same pixel

The first step is to define the endmembers such that we can define ***S***. We will do this by computing the mean spectra in polygons delineated around regions of pure land cover.

Zoom the map to a location with homogeneous areas of bare land, vegetation, and water (an airport can be used as a suitable location). Visualize the Landsat 8 image as a false color composite:

```
// Use a false color composite to help define polygons of
'pure' land cover.
Map.addLayer(imageL8, {
    bands: ['B5', 'B4', 'B3'],
    min: 0.0,
    max: 0.4
}, 'false color');
```

For faster rendering, you may want to comment out the previous layers you added to the map.

In general, the way to do this is to draw polygons around areas of pure land cover in order to define the spectral signature of these land covers. If you would like to do this on your own, here is how. Using the geometry drawing tools, make three new layers (thus, $P = 3$) by selecting the polygon tool and then clicking + **new layer**. In the first layer, digitize a polygon around pure bare land; in the second layer, make a polygon of pure vegetation; in the third layer, make a water polygon. Name the imports `bare`, `water`, and `veg`, respectively. You will need to use the settings (gear icon) to rename the geometries.

You can also use this code to specify predefined areas of bare, water, and vegetation. This will only work for this example.

```
// Define polygons of bare, water, and vegetation.
var bare = /* color: #d63000 */ ee.Geometry.Polygon(
        [
            [
                [-119.29158963591193, 47.204453926034134],
                [-119.2919222982978, 47.20372502078616],
                [-119.29054893881415, 47.20345532330602],
                [-119.29017342955207, 47.20414049800489]
            ]
        ]),
    water = /* color: #98ff00 */ ee.Geometry.Polygon(
        [
            [
                [-119.42904610218152, 47.22253398528318],
                [-119.42973274768933, 47.22020224831784],
                [-119.43299431385144, 47.21390604625894],
                [-119.42904610218152, 47.21326472446865],
                [-119.4271149116908, 47.21868656429651],
                [-119.42608494342907, 47.2217470355224]
            ]
        ]),
    veg = /* color: #0b4a8b */ ee.Geometry.Polygon(
        [
            [
                [-119.13546041722502, 47.04929418944858],
                [-119.13752035374846, 47.04929418944858],
                [-119.13966612096037, 47.04765665820436],
                [-119.13777784581389, 47.04408900535686]
            ]
        ]);
```

Check the polygons you made or imported by charting mean spectra in them using `ui.Chart.image.regions`.

```
//Print a chart.
var lcfeatures = ee.FeatureCollection([
    ee.Feature(bare, {label: 'bare'}),
    ee.Feature(water, {label: 'water'}),
    ee.Feature(veg, {label: 'vegetation'})
]);

print(
  ui.Chart.image.regions({
  image: unmixImage,
  regions: lcfeatures,
  reducer: ee.Reducer.mean(),
  scale: 30,
  seriesProperty: 'label',
  xLabels: [0.48, 0.56, 0.65, 0.86, 1.61, 2.2]
})
.setChartType('LineChart')
.setOptions({
  title: 'Image band values in 3 regions',
  hAxis: {
    title: 'Wavelength'
  },
  vAxis: {
    title: 'Mean Reflectance'
  }
}));
```

The xLabels line of code takes the mean of each polygon (feature) at the spectral midpoint of each of the six bands. The numbers (`[0.48, 0.56, 0.65, 0.86, 1.61, 2.2]`) represent these spectral midpoints. Your chart should look something like Fig. 9.8.

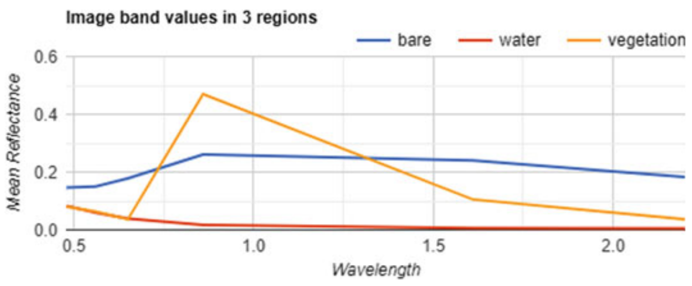


Fig. 9.8 Mean of the pure land cover reflectance for each band

Use the reduceRegion method to compute the mean values within the polygons you made, for each of the bands. Note that the return value of reduceRegion is a Dictionary of numbers summarizing values within the polygons, with the output indexed by band name.

Get the means as a List by calling the values function after computing the mean. Note that values return the results in alphanumeric order sorted by the keys. This works because $B2 - B7$ are already alphanumerically sorted, but it will not work in cases when they are not already sorted. In those cases, please specify the list of band names so that you get them in a known order first.

```
// Get the means for each region.
var bareMean = unmixImage
    .reduceRegion(ee.Reducer.mean(), bare, 30).values();
var waterMean = unmixImage
    .reduceRegion(ee.Reducer.mean(), water, 30).values();
var vegMean = unmixImage
    .reduceRegion(ee.Reducer.mean(), veg, 30).values();
```

Each of these three lists represents a mean spectrum vector, which is one of the columns for our $\boldsymbol{S}$ matrix defined above. Stack the vectors into a 6×3 Array of endmembers by concatenating them along the 1-axis (columns):

```
// Stack these mean vectors to create an Array.
var endmembers = ee.Array.cat([bareMean, vegMean,
waterMean], 1);
print(endmembers);
```

Use print if you would like to view your new matrix.

As we have done in the previous sections, we will now convert the 6-band input image into an image in which each pixel is a 1D vector (toArray), then into an image in which each pixel is a 6×1 matrix (toArray(1)). This creates $\boldsymbol{p}$ so that we can solve the equation above for each pixel.

```
// Convert the 6-band input image to an image array.
var arrayImage = unmixImage.toArray().toArray(1);
```

Now that we have everything in place, for each pixel, we solve the equation for $\boldsymbol{f}$:

```
// Solve for f.
var unmixed = ee.Image(endmembers).matrixSolve(arrayImage);
```

For this task, we use the `matrixSolve` function. This function solves for x in the equation $A * x = B$. Here, A is our matrix $\boldsymbol{S}$ and B is the matrix $\boldsymbol{p}$.

Finally, convert the result from a two-dimensional array image into a one-dimensional array image (`arrayProject`), and then into a zero-dimensional, more familiar multi-band image (`arrayFlatten`). This is the same approach we used in the previous sections. The three bands correspond to the estimates of bare, vegetation, and water fractions in $\boldsymbol{f}$:

```
// Convert the result back to a multi-band image.
var unmixedImage = unmixed
    .arrayProject([0])
    .arrayFlatten([
        ['bare', 'veg', 'water']
    ]);
```

Display the result where bare is red, vegetation is green, and water is blue (the `addLayer` call expects bands in order, RGB). Use either code or the layer visualization parameter tool to achieve this. Your resulting image should look like Fig. 9.9.

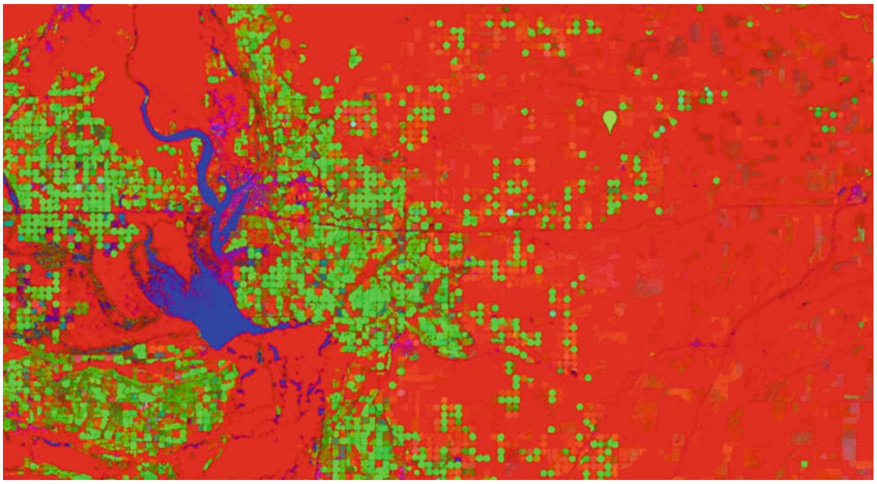


Fig. 9.9 Result of the spectral unmixing example

```
Map.addLayer(unmixedImage, {}, 'Unmixed');
```

9.2.4 Section 4: The Hue, Saturation, Value Transform

Whereas the other three transforms we have discussed will transform the image based on spectral signatures from the original image, the hue, saturation, and value (HSV) transform is a color transform of the RGB color space.

Among many other things, it is useful for pan-sharpening, a process by which a higher-resolution panchromatic image is combined with a lower-resolution multi-band raster. This involves converting the multi-band raster RGB to HSV color space, swapping the panchromatic band for the value band, then converting back to RGB. Because the value band describes the brightness of colors in the original image, this approach leverages the higher resolution of the panchromatic image.

For example, let us pan-sharpen the Landsat 8 scene we have been working with in this chapter. In Landsat 8, the panchromatic band is 15 m resolution, while the RGB bands are 30 m resolution. We use the `rgbToHsv` function here—it is such a common transform that there is a built-in function for it.

```
// Begin HSV transformation example

// Convert Landsat 8 RGB bands to HSV color space
var hsv = imageL8.select(['B4', 'B3', 'B2']).rgbToHsv();

Map.addLayer(hsv, {
    max: 0.4
}, 'HSV Transform');
```

Next, we convert the image back to RGB space after substituting the panchromatic band for the value band, which appears third in the HSV image. We do this by first concatenating the different image bands using the `ee.Image.cat` function and then by converting to RGB.

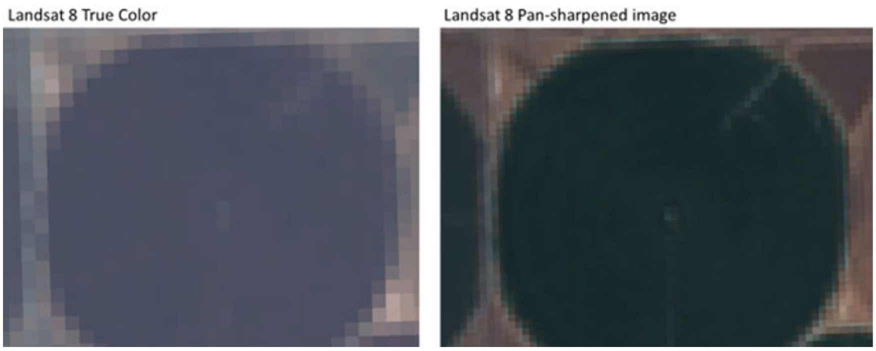


Fig. 9.10 Results of the pan-sharpening process (right) compared with the original true-color image (left)

```
// Convert back to RGB, swapping the image panchromatic
band for the value.
var rgb = ee.Image.cat([
    hsv.select('hue'),
    hsv.select('saturation'),
    imageL8.select(['B8'])
]).hsvToRgb();

Map.addLayer(rgb, {
    max: 0.4
}, 'Pan-sharpened');
```

In Fig. 9.10, compare the pan-sharpened image to the original true-color image. What do you notice? Is it easier to interpret the image following pan-sharpening?

Code Checkpoint F31d. The book's repository contains a script that shows what your code should look like at this point.

9.3 Synthesis

Assignment 1. Write an expression to calculate the normalized burn ratio thermal (NBRT) index for the Rim Fire Landsat 8 image (`burnImage`).

NBRT was developed based on the idea that burned land has low NIR reflectance (less vegetation), high SWIR reflectance (from ash, etc.), and high brightness temperature (Holden et al. 2005).

The formula is

$$\text{NBRT} = \frac{\left(\text{NIR} - \text{SWIR} \times \left(\frac{\text{Thermal}}{1000}\right)\right)}{\left(\text{NIR} + \text{SWIR} \times \left(\frac{\text{Thermal}}{1000}\right)\right)} \tag{9.6}$$

where NIR should be between 0.76 and 0.9 μm, SWIR 2.08 and 2.35 μm, and thermal 10.4 and 12.5 μm.

To display this result, remember that a lower NBRT is the result of more burning.

Bonus: Here is another way to reverse a color palette (note the min and max values):

```
Map.addLayer(nbrt, {
    min: 1,
    max: 0.9,
    palette: burnPalette
}, 'NBRT');
```

The difference in this index, before compared with after the fire, can be used as a diagnostic of burn severity (see van Wagtendonk et al. 2004).

9.4 Conclusion

Linear image transformations are a powerful tool in remote sensing analysis. By choosing your linear transformation carefully, you can highlight specific aspects of your data that make image classification easier and more accurate. For example, spectral unmixing is frequently used in change detection applications like detecting forest degradation. By using the endmembers (pure spectra) as inputs to the change detection algorithms, the model is better able to detect subtle changes due to the removal of some but not all the trees in the pixel.

References

Baig MHA, Zhang L, Shuai T, Tong Q (2014) Derivation of a tasselled cap transformation based on Landsat 8 at-satellite reflectance. Remote Sens Lett 5:423–431. https://doi.org/10.1080/2150704X.2014.915434

Bullock EL, Woodcock CE, Olofsson P (2020) Monitoring tropical forest degradation using spectral unmixing and Landsat time series analysis. Remote Sens Environ 238:110968. https://doi.org/10.1016/j.rse.2018.11.011

Crist EP (1985) A TM tasseled cap equivalent transformation for reflectance factor data. Remote Sens Environ 17:301–306. https://doi.org/10.1016/0034-4257(85)90102-6

Holden ZA, Smith AMS, Morgan P et al (2005) Evaluation of novel thermally enhanced spectral indices for mapping fire perimeters and comparisons with fire atlas data. Int J Remote Sens 26:4801–4808. https://doi.org/10.1080/01431160500239008

Huang C, Wylie B, Yang L et al (2002) Derivation of a tasselled cap transformation based on Landsat 7 at-satellite reflectance. Int J Remote Sens 23:1741–1748. https://doi.org/10.1080/01431160110106113

Huete A, Didan K, Miura T et al (2002) Overview of the radiometric and biophysical performance of the MODIS vegetation indices. Remote Sens Environ 83:195–213. https://doi.org/10.1016/S0034-4257(02)00096-2

Kauth RJ, Thomas GS (1976) The tasselled cap--a graphic description of the spectral-temporal development of agricultural crops as seen by Landsat. In LARS symposia (p. 159)

Martín MP (1998) Cartografía e inventario de incendios forestales en la Península Ibérica a partir de imágenes NOAA-AVHRR. Universidad de Alcalá

Rufin P, Frantz D, Ernst S et al (2019) Mapping cropping practices on a national scale using intra-annual Landsat time series binning. Remote Sens 11:232. https://doi.org/10.3390/rs11030232

Schultz M, Clevers JGPW, Carter S et al (2016) Performance of vegetation indices from Landsat time series in deforestation monitoring. Int J Appl Earth Obs Geoinf 52:318–327. https://doi.org/10.1016/j.jag.2016.06.020

Souza CM Jr, Roberts DA, Cochrane MA (2005) Combining spectral and spatial information to map canopy damage from selective logging and forest fires. Remote Sens Environ 98:329–343. https://doi.org/10.1016/j.rse.2005.07.013

Souza CM Jr, Siqueira JV, Sales MH et al (2013) Ten-year landsat classification of deforestation and forest degradation in the Brazilian Amazon. Remote Sens 5:5493–5513. https://doi.org/10.3390/rs5115493

Van Wagtendonk JW, Root RR, Key CH (2004) Comparison of AVIRIS and Landsat ETM + detection capabilities for burn severity. Remote Sens Environ 92:397–408. https://doi.org/10.1016/j.rse.2003.12.015

Neighborhood-Based Image Transformation

10

Karen Dyson, Andréa Puzzi Nicolau, David Saah, and Nicholas Clinton

Overview

This chapter builds on image transformations to include a spatial component. All of these transformations leverage a neighborhood of multiple pixels around the focal pixel to inform the transformation of the focal pixel.

Learning Outcomes

- Performing image morphological operations.
- Defining kernels in Earth Engine.
- Applying kernels for image convolution to smooth and enhance images.
- Viewing a variety of neighborhood-based image transformations in Earth Engine.

K. Dyson · A. P. Nicolau · D. Saah
Spatial Informatics Group, Pleasanton, CA, USA
e-mail: kdyson@sig-gis.com

A. P. Nicolau
e-mail: apnicolau@sig-gis.com

K. Dyson · A. P. Nicolau
SERVIR-Amazonia, Cali, Colombia

K. Dyson
Dendrolytics, Seattle, WA, USA

D. Saah (✉)
University of San Francisco, San Francisco, CA, USA
e-mail: dssaah@usfca.edu

N. Clinton
Google LLC, Inc, Mountain View, CA, USA
e-mail: nclinton@google.com

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_10

Assumes you know how to

- Import images and image collections, filter, and visualize (Part I).
- Perform basic image analysis: select bands, compute indices, create masks, classify images (Part II).

10.1 Introduction to Theory

Neighborhood-based image transformations enable information from the pixels surrounding a focal pixel to inform the function transforming the value in the focal pixel (Fig. 10.1). For example, if your image has some pixels with outlier values, you can use a neighborhood-based transformation to diminish the effect of the outlier values (here, we would recommend the median; see the Practicum section below). This is similar to how a moving average is performed (e.g., loess smoothing), but instead of averaging across one dimension (e.g., time), it averages across two dimensions (latitude and longitude).

Neighborhood-based image transformations are a foundational part of many remote sensing analysis workflows. For example, edge detection has been a critical part of land cover classification research efforts, including using Landsat 5 data in Minneapolis-St. Paul, Minnesota, USA (Stuckens et al. 2000), and using Landsat 7 and higher- resolution data including IKONOS in Accra, Ghana (Toure et al. 2018). Another type of neighborhood-based image transformation, the median operation, is an important part of remote sensing workflows due to its ability to dampen

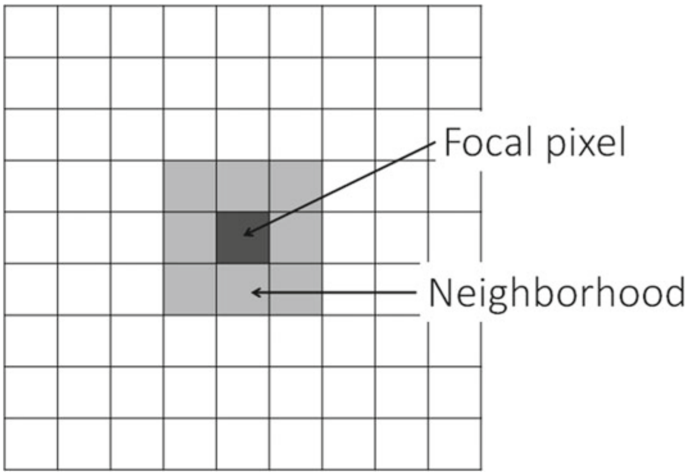


Fig. 10.1 Neighborhood surrounds the focal pixel. Different neighborhoods can vary in size and shape. The focal pixel is the pixel for which new values are calculated, based on values in the neighborhood

noise in images or classifications while maintaining edges (see ZhiYong et al. 2018; Lüttig et al. 2017). We will discuss these methods and others in this chapter.

10.2 Practicum

10.2.1 Section 1: Linear Convolution

Linear convolution refers to calculating a linear combination of pixel values in a neighborhood for the focal pixel (Fig. 10.1).

In Earth Engine, the neighborhood of a given pixel is specified by a *kernel*. The kernel defines the size and shape of the neighborhood and a weight for each position in the kernel. To implement linear convolution in Earth Engine, we will use the `convolve` with an `ee.Kernel` for the argument.

Convolving an image can be useful for extracting image information at different spatial frequencies. For this reason, smoothing kernels are called *low-pass filters* (they let low-frequency data pass through), and edge detection kernels are called *high-pass filters*. In the Earth Engine context, this use of the word "filter" is distinct from the filtering of image collections and feature collections seen throughout this book. In general in this book, filtering refers to retaining items in a set that have specified characteristics. In contrast, in this specific context of image convolution, "filter" is often used interchangeably with "kernel" when it is applied to the pixels of a single image. This chapter refers to these as "kernels" or "neighborhoods" wherever possible, but be aware of the distinction when you encounter the term "filter" in technical documentation elsewhere.

Smoothing

A square kernel with uniform weights that sum to one is an example of a smoothing kernel. Using this kernel replaces the value of each pixel with the value of a summarizing function (usually, the mean) of its neighborhood. Because averages are less extreme than the values used to compute them, this tends to diminish image noise by replacing extreme values in individual pixels with a blend of surrounding values. When using a kernel to smooth an image in practice, the statistical properties of the data should be carefully considered (Vaiphasa 2006).

Let us create and print a square kernel with uniform weights for our smoothing kernel.

```
// Create and print a uniform kernel to see its weights.
print('A uniform kernel:', ee.Kernel.square(2));
```

Expand the kernel object in the **Console** to see the weights. This kernel is defined by how many pixels it covers (i.e., `radius` is in units of pixels). Remember that your pixels may represent different real-world distances (spatial resolution is discussed in more detail in Chap. 4).

A kernel with radius defined in meters adjusts its size to an image's pixel size, so you cannot visualize its weights, but it is more flexible in terms of adapting to inputs of different spatial resolutions. In the next example, we will use kernels with radius defined in meters.

As first explored in Chap. 4, the National Agriculture Imagery Program (NAIP) is a U.S. government program to acquire imagery over the continental United States using airborne sensors. The imagery has a spatial resolution of 0.5–2 m, depending on the state and the date collected.

Define a new point named `point`. We will locate it near the small town of Odessa in eastern Washington, USA. You can also search for "Odessa, WA, USA" in the search bar and define your own point.

Now filter and display the NAIP `ImageCollection`. For Washington State, there was NAIP imagery collected in 2018. We will use the `reduce` function in order to convert the image collection to an image for the convolution.

```
// Define a point of interest in Odessa, Washington, USA.
var point = ee.Geometry.Point([-118.71845096212049,
    47.15743083101999]);
Map.centerObject(point);

// Load NAIP data.
var imageNAIP = ee.ImageCollection('USDA/NAIP/DOQQ')
    .filterBounds(point)
    .filter(ee.Filter.date('2017-01-01', '2018-12-31'))
    .first();

Map.centerObject(point, 17);

var trueColor = {
    bands: ['R', 'G', 'B'],
    min: 0,
    max: 255
};
Map.addLayer(imageNAIP, trueColor, 'true color');
```

You will notice that the NAIP imagery selected with these operations covers only a very small area. This is because the NAIP image tiles are quite small, covering only a 3.75 × 3.75 min quarter quadrangle plus a 300 m buffer on all four sides. For your own work using NAIP, you may want to use a rectangle or polygon to filter over a larger area and the `reduce` function instead of `first`.

Now define a kernel with a 2 m radius with uniform weights to use for smoothing.

```
// Begin smoothing example.
// Define a square, uniform kernel.
var uniformKernel = ee.Kernel.square({
    radius: 2,
    units: 'meters',
});
```

Apply the smoothing operation by convolving the image with the kernel we have just defined.

```
// Convolve the image by convolving with the smoothing
kernel.
var smoothed = imageNAIP.convolve(uniformKernel);
Map.addLayer(smoothed, {
    min: 0,
    max: 255
}, 'smoothed image');
```

Now, compare the input image with the smoothed image. In Fig. 10.2, notice how sharp outlines around, for example, the patch of vegetation or the road stripes are less distinct.

To make the image even more smooth, you can increase the size of the neighborhood by increasing the pixel radius.

Gaussian Smoothing

A Gaussian kernel can also be used for smoothing. A Gaussian curve is also known as a bell curve. Think of convolving with a Gaussian kernel as computing the weighted average in each pixel's neighborhood, where closer pixels are weighted more heavily than pixels that are further away. Gaussian kernels preserve lines better than the smoothing kernel we just used, allowing them to be used when

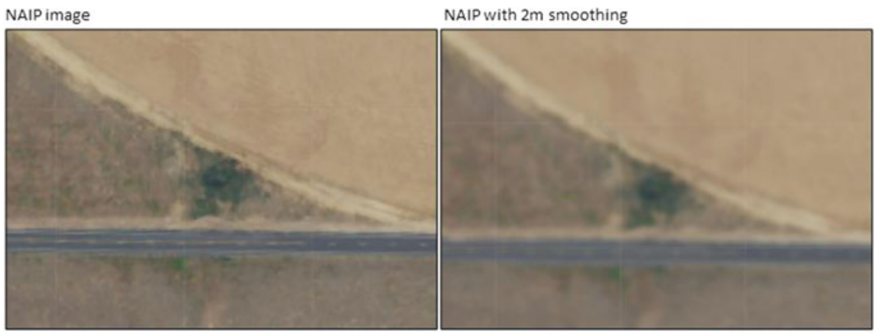


Fig. 10.2 Example of the effects of a smoothing convolution on NAIP imagery using a 2 m kernel

feature conservation is important, such as in detecting oil slicks (Wang and Hu 2015).

```
// Begin Gaussian smoothing example.
// Print a Gaussian kernel to see its weights.
print('A Gaussian kernel:', ee.Kernel.gaussian(2));
```

Now, we will apply a Gaussian kernel to the same NAIP image.

```
// Define a square Gaussian kernel:
var gaussianKernel = ee.Kernel.gaussian({
    radius: 2,
    units: 'meters',
});

// Convolve the image with the Gaussian kernel.
var gaussian = imageNAIP.convolve(gaussianKernel);
Map.addLayer(gaussian, {
    min: 0,
    max: 255
}, 'Gaussian smoothed image');
```

Pan and zoom around the NAIP image, switching between the two smoothing functions. Notice how the Gaussian smoothing preserves more of the detail of the image, such as the road lines and vegetation in Fig. 10.3.

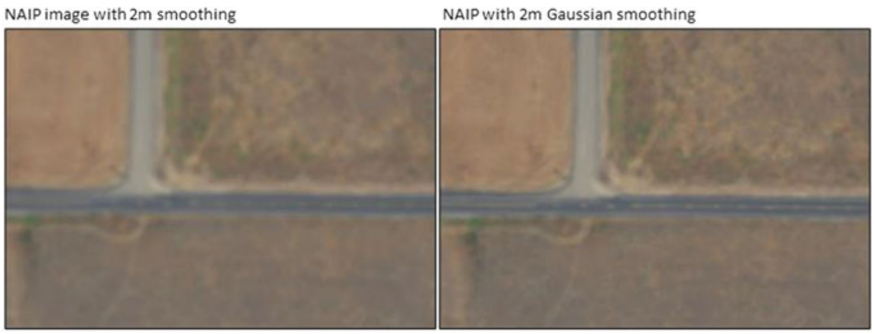


Fig. 10.3 Example of the effects of a Gaussian smoothing kernel on NAIP imagery using a 2 m kernel

Edge Detection

Edge detection kernels are used to find rapid changes in remote sensing image values. These rapid changes usually signify edges of objects represented in the image data. Finding edges is useful for many applications, including identifying transitions between land cover and land use during classification.

A common edge detection kernel is the Laplacian kernel. Other edge detection kernels include the Sobel, Prewitt, and Roberts kernels. First, look at the Laplacian kernel weights:

```
// Begin edge detection example.
// For edge detection, define a Laplacian kernel.
var laplacianKernel = ee.Kernel.laplacian8();

// Print the kernel to see its weights.
print('Edge detection Laplacian kernel:',
laplacianKernel);
```

Notice that if you sum all of the neighborhood values, the focal cell value is the negative of that sum. Now apply the kernel to our NAIP image and display the result:

```
// Convolve the image with the Laplacian kernel.
var edges = imageNAIP.convolve(laplacianKernel);
Map.addLayer(edges, {
    min: 0,
    max: 255
}, 'Laplacian convolution image');
```

Edge detection algorithms remove contrast within the image (e.g., between fields) and focus on the edge information (Fig. 10.4).

There are also algorithms in Earth Engine that perform edge detection. One of these is the Canny edge detection algorithm (Canny 1986), which identifies the diagonal, vertical, and horizontal edges by using four separate kernels.

Sharpening

Image sharpening, also known as edge enhancement, leverages the edge detection techniques we just explored to make the edges in an image sharper. This mimics the human eye's ability to enhance separation between objects via Mach bands. To achieve this from a technical perspective, you add the image to the second derivative of the image.

To implement this in Earth Engine, we use a combination of Gaussian kernels through the Difference-of-Gaussians convolution (see Schowengerdt 2006 for

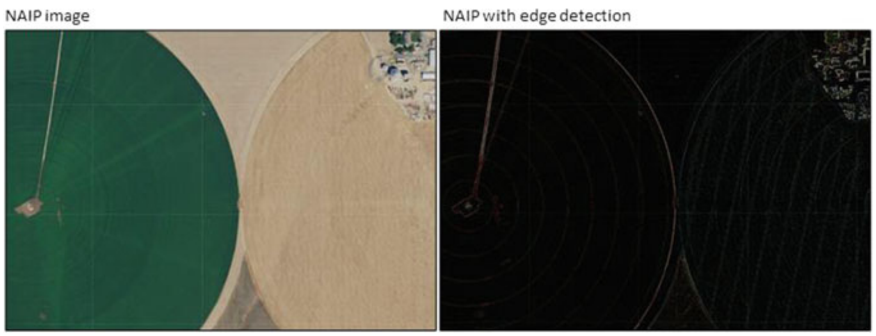


Fig. 10.4 Example of the effects of Laplacian edge detection on NAIP imagery

details) and then add this to the input image. Start by creating two Gaussian kernels:

```
// Begin image sharpening example.
// Define a "fat" Gaussian kernel.
var fat = ee.Kernel.gaussian({
    radius: 3,
    sigma: 3,
    magnitude: -1,
    units: 'meters'
});

// Define a "skinny" Gaussian kernel.
var skinny = ee.Kernel.gaussian({
    radius: 3,
    sigma: 0.5,
    units: 'meters'
});
```

Next, combine the two Gaussian kernels into a Difference-of-Gaussians kernel and print the result:

```
// Compute a difference-of-Gaussians (DOG) kernel.
var dog = fat.add(skinny);

// Print the kernel to see its weights.
print('DoG kernel for image sharpening', dog);
```

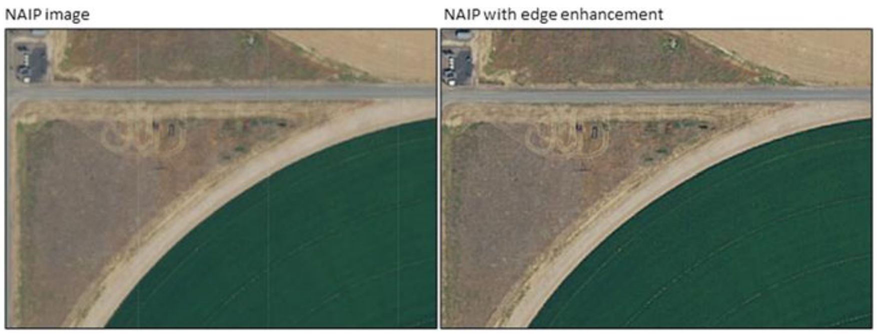


Fig. 10.5 Example of the effects of difference-of-Gaussians edge sharpening on NAIP imagery

Finally, apply the new kernel to the NAIP imagery with the `convolve` command and then add the `DoG` convolved image to the original image with the `add` command (Fig. 10.5).

```
// Add the DoG convolved image to the original image.
var sharpened = imageNAIP.add(imageNAIP.convolve(dog));
Map.addLayer(sharpened, {
    min: 0,
    max: 255
}, 'DoG edge enhancement');
```

Code Checkpoint F32a. The book's repository contains a script that shows what your code should look like at this point.

10.2.2 Section 2: Nonlinear Convolution

Where linear convolution functions involve calculating a linear combination of neighborhood pixel values for the focal pixel, nonlinear convolution functions use nonlinear combinations. Both linear and nonlinear convolution use the same concepts of the neighborhood, focal pixel, and kernel. The main difference from a practical standpoint is that nonlinear convolution approaches are implemented in Earth Engine using the `reduceNeighborhood` method on images.

Median
Median neighborhood filters are used for denoising images. For example, some individual pixels in your image may have abnormally high or low values resulting from measurement error, sensor noise, or another cause. Using the mean operation described earlier to average values within a kernel would result in these extreme

values polluting other pixels. That is, when a noisy pixel is present in the neighborhood of a focal pixel, the calculated mean will be pulled up or down due to that abnormally high- or low-value pixel. The median neighborhood filter can be used to minimize this issue. This approach is also useful because it preserves edges better than other smoothing approaches, an important feature for many types of classification.

Let us reuse the uniform 5 × 5 kernel from above (`uniformKernel`) to implement a median neighborhood filter. As seen below, nonlinear convolution functions are implemented using `reduceNeighborhood`.

```
// Begin median example.
// Pass a median neighborhood filter using our
uniformKernel.
var median = imageNAIP.reduceNeighborhood({
    reducer: ee.Reducer.median(),
    kernel: uniformKernel
});

Map.addLayer(median, {
    min: 0,
    max: 255
}, 'Median Neighborhood Filter');
```

Inspect the median neighborhood filter map layer you have just added (Fig. 10.6). Notice how the edges are preserved instead of a uniform smoothing seen with the mean neighborhood filter. Look closely at features such as road intersections, field corners, and buildings.

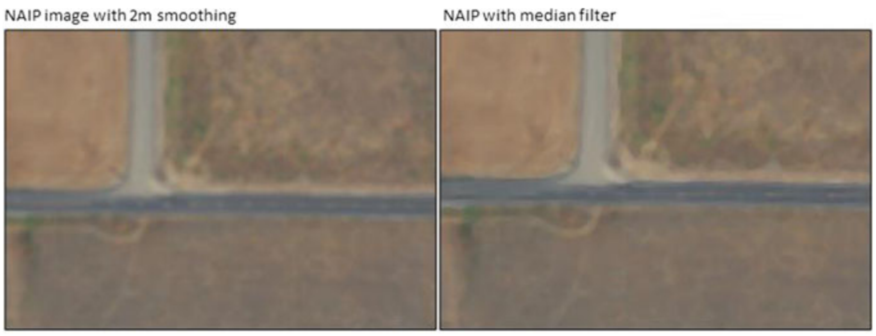


Fig. 10.6 Example of the effects of a median neighborhood filter on NAIP imagery

Mode

The mode operation, which identifies the most commonly used number in a set, is particularly useful for categorical maps. Methods such as median and mean, which blend values found in a set, do not make sense for aggregating nominal data. Instead, we use the mode operation to get the value that occurs most frequently within each focal pixel's neighborhood. The mode operation can be useful when you want to eliminate individual, rare pixel occurrences, or small groups of pixels that are classified differently than their surroundings.

For this example, we will make a categorical map by thresholding the NIR band. First we will select the NIR band and then threshold it at 200 using the `gt` function (see also Chap. 5). Values higher than 200 will map to 1, while values equal to or below 200 will map to 0. We will then display the two classes as black and green. Thresholding the NIR band in this way is a very rough approximation of where vegetation occurs on the landscape, so we will call our layer `veg`.

```
// Mode example
// Create and display a simple two-class image.
var veg = imageNAIP.select('N').gt(200);

// Display the two-class (binary) result.
var binaryVis = {
    min: 0,
    max: 1,
    palette: ['black', 'green']
};
Map.addLayer(veg, binaryVis, 'Vegetation categorical
image');
```

Now use our uniform kernel to compute the mode in each 5 × 5 neighborhood.

```
// Compute the mode in each 5x5 neighborhood and display
the result.
var mode = veg.reduceNeighborhood({
    reducer: ee.Reducer.mode(),
    kernel: uniformKernel
});

Map.addLayer(mode, binaryVis, 'Mode Neighborhood Filter on
Vegetation categorical image');
```

The resulting image following the mode neighborhood filter has less individual pixel noise and more cohesive areas of vegetation (Fig. 10.7).

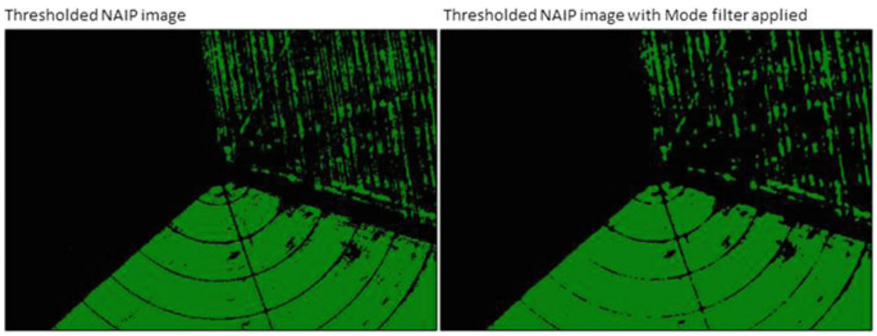


Fig. 10.7 Example of the effects of the mode neighborhood filter on thresholded NAIP imagery using a uniform kernel

Code Checkpoint F32b. The book's repository contains a script that shows what your code should look like at this point.

10.2.3 Section 3: Morphological Processing

The idea of morphology is tied to the concept of objects in images. For example, suppose the patches of 1 s in the `veg` image from the previous section represent patches of vegetation. Morphological processing helps define these objects so that the processed images can better inform classification processes, such as object-based classification (Chap. 11), and as a post-processing approach to reduce noise caused by the classification process. Below are four of the most important morphological processing approaches.

Dilation

If the classification underestimates the actual distribution of vegetation and contains "holes," a `max` operation can be applied across the neighborhood to expand the areas of vegetation. This process is known as a *dilation* (Fig. 10.8).

```
// Begin Dilation example.
// Dilate by taking the max in each 5x5 neighborhood.
var max = veg.reduceNeighborhood({
    reducer: ee.Reducer.max(),
    kernel: uniformKernel
});

Map.addLayer(max, binaryVis, 'Dilation using max');
```

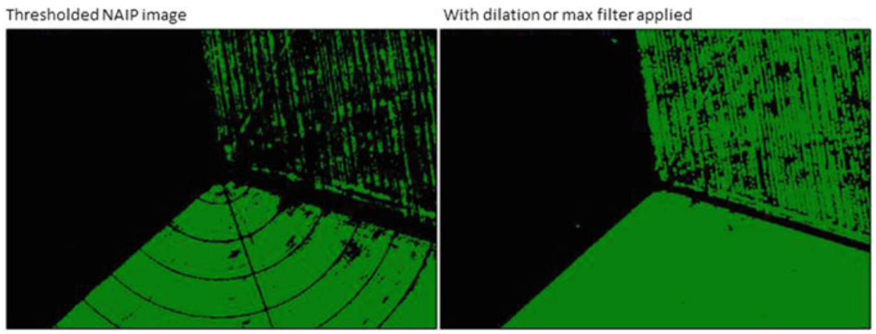


Fig. 10.8 Example of the effects of the dilation on thresholded NAIP imagery using a uniform kernel

To explore the effects of dilation, you might try to increase the size of the kernel (i.e., increase the `radius`), or to apply `reduceNeighborhood` repeatedly. There are shortcuts in the API for some common `reduceNeighborhood` actions, including focalMax and focalMin, for example.

Erosion

The opposite of dilation is *erosion*, for decreasing the size of the patches. To effect an erosion, a `min` operation can be applied to the values inside the kernel as each pixel is evaluated.

```
// Begin Erosion example.
// Erode by taking the min in each 5x5 neighborhood.
var min = veg.reduceNeighborhood({
    reducer: ee.Reducer.min(),
    kernel: uniformKernel
});

Map.addLayer(min, binaryVis, 'Erosion using min');
```

Carefully inspect the result compared to the input (Fig. 10.9). Note that the shape of the kernel affects the shape of the eroded patches (the same effect occurs in the dilation). Because we used a square kernel, the eroded patches and dilated areas are square. You can explore this effect by testing kernels of different shapes.

As with the dilation, note that you can get more erosion by increasing the size of the kernel or applying the operation more than once.

Opening

To remove small patches of green that may be unwanted, we will perform an erosion followed by a dilation. This process is called *opening*, works to delete

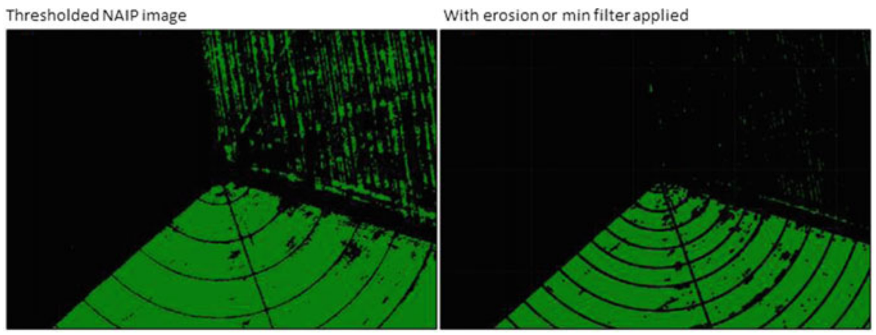


Fig. 10.9 Example of the effects of the erosion on thresholded NAIP imagery using a uniform kernel

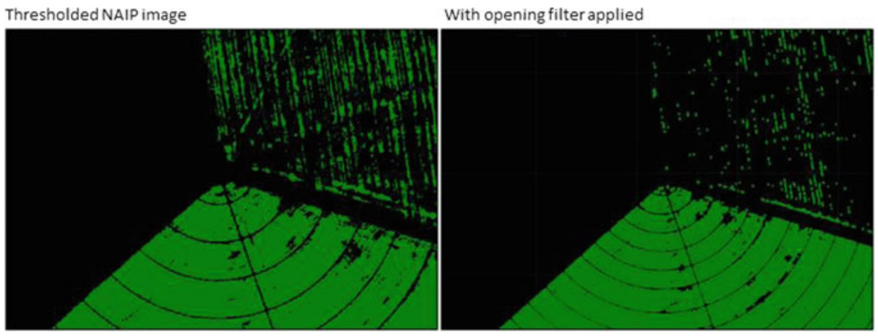


Fig. 10.10 Example of the effects of the opening operation on thresholded NAIP imagery using a uniform kernel

small details, and is useful for removing noise. We can use our eroded image and perform a dilation on it (Fig. 10.10).

```
// Begin Opening example.
// Perform an opening by dilating the eroded image.
var openedVeg = min.reduceNeighborhood({
    reducer: ee.Reducer.max(),
    kernel: uniformKernel
});

Map.addLayer(openedVeg, binaryVis, 'Opened image');
```

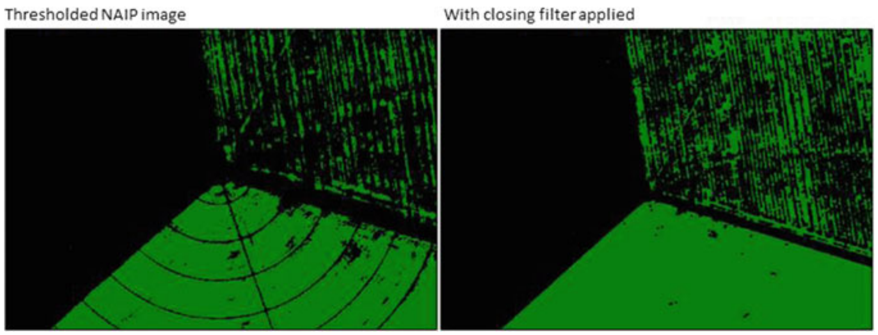


Fig. 10.11 Example of the effects of the closing operation on thresholded NAIP imagery using a uniform kernel

Closing

Finally, the opposite of opening is *closing*, which is a dilation operation followed by an erosion. This series of transformations is used to remove small holes in the input patches (Fig. 10.11).

```
// Begin Closing example.
// Perform a closing by eroding the dilated image.
var closedVeg = max.reduceNeighborhood({
    reducer: ee.Reducer.min(),
    kernel: uniformKernel
});

Map.addLayer(closedVeg, binaryVis, 'Closed image');
```

Closely examine the difference between each morphological operation and the `veg` input. You can adjust the effect of these morphological operators by adjusting the size and shape of the kernel (also called a “structuring element” in this context, because of its effect on the spatial structure of the result), or applying the operations repeatedly. When used for post-processing of, for example, a classification output, this process will usually require multiple iterations to balance accuracy with class cohesion.

Code Checkpoint F32c. The book’s repository contains a script that shows what your code should look like at this point.

10.2.4 Section 4: Texture

The final group of neighborhood-based operations we will discuss is meant to detect or enhance the “texture” of the image. Texture measures use a potentially complex, usually nonlinear calculation using the pixel values within a neighborhood. From a practical perspective, texture is one of the cues we use (often unconsciously) when looking at a remote sensing image in order to identify features. Some examples include distinguishing tree cover, examining the type of canopy cover, and distinguishing crops. Measures of texture may be used on their own or may be useful as inputs to regression, classification, and other analyzes when they help distinguish between different types of land cover/land use or other features on the landscape.

There are many ways to assess texture in an image, and a variety of functions have been implemented to compute texture in Earth Engine.

Standard Deviation

The standard deviation (SD) measures the spread of the distribution of image values in the neighborhood. A textureless neighborhood, in which there is only one value within the neighborhood, has a standard deviation of 0. A neighborhood with significant texture will have a high standard deviation, the value of which will be influenced by the magnitude of the values within the neighborhood.

Compute neighborhood SD for the NAIP image by first defining a 7 m radius kernel and then using the `stdDev` reducer with the kernel.

```
// Begin Standard Deviation example.
// Define a big neighborhood with a 7-meter radius kernel.
var bigKernel = ee.Kernel.square({
    radius: 7,
    units: 'meters'
});

// Compute SD in a neighborhood.
var sd = imageNAIP.reduceNeighborhood({
    reducer: ee.Reducer.stdDev(),
    kernel: bigKernel
});

Map.addLayer(sd, {
    min: 0,
    max: 70
}, 'SD');
```

The resulting image for our fields somewhat resembles the image for edge detection (Fig. 10.12).

You can pan around the example area to find buildings or pasture land and examine these features. Notice how local variation and features appear, such as the washes (texture variation caused by water) in Fig. 10.13.

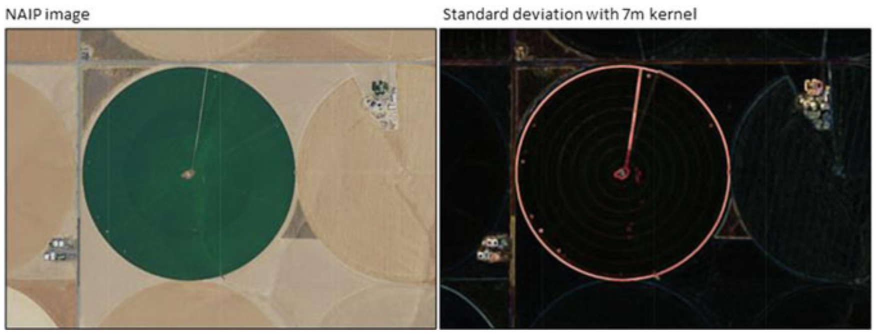


Fig. 10.12 Example of the effects of a standard deviation convolution on an irrigated field in NAIP imagery using a 7 m kernel

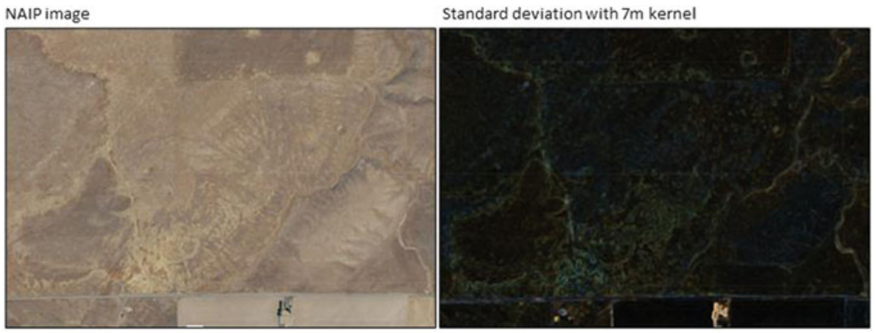


Fig. 10.13 Example of the effects of a standard deviation convolution on a natural landscape in NAIP imagery using a 7 m kernel

Entropy

For discrete valued inputs, you can compute entropy in a neighborhood. Broadly, entropy is a concept of disorder or randomness. In this case, entropy is an index of the numerical diversity in the neighborhood.

We will compute entropy using the `entropy` function in Earth Engine and our `bigKernel` structuring element. Notice that if you try to run the entropy on the entire NAIP image (`imageNAIP`), you will get an error that only 32-bit or smaller integer types are currently supported. So, let us cast the image to contain an integer in every pixel using the `int` function. We will operate on the near-infrared band since it is important for vegetation.

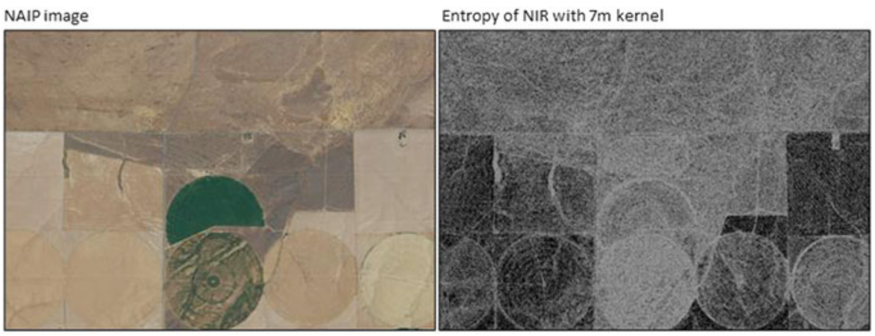


Fig. 10.14 Example of an entropy convolution of the near-infrared integer band in NAIP imagery using a 7 m kernel

```
// Begin entropy example.
// Create an integer version of the NAIP image.
var intNAIP = imageNAIP.int();

// Compute entropy in a neighborhood.
var entropy = intNAIP.select('N').entropy(bigKernel);

Map.addLayer(entropy, {
    min: 1,
    max: 3
}, 'entropy');
```

The resulting entropy image has low values where the 7 m neighborhood around a pixel is homogeneous and high values where the neighborhood is heterogeneous (Fig. 10.14).

Gray-Level Co-occurrence Matrices

The gray-level co-occurrence matrix (GLCM) is based on gray-scale images. It evaluates the co-occurrence of similar values occurring horizontally, vertically, or diagonally. More formally, the GLCM is computed by forming an $M \times M$ matrix for an image with M possible DN values, then computing entry i,j as the frequency at which DN $= i$ is adjacent to DN $= j$. In other words, the matrix represents the relationship between two adjacent pixels.

Once the GLCM has been calculated, a variety of texture metrics can be computed based on that matrix. One of these is contrast. To do this, we first use the `glcmTexture` function. This function computes 14 original GLCM metrics (Haralick et al. 1973) and four later GLCM metrics (Conners et al.1984). The glcmTexture function creates an image where each band is a different metric. Note that the input needs to be an integer, so we will use the same integer NAIP layer as above, and we need to provide a size for the neighborhood (here, it is 7).

```
// Begin GLCM example.
// Use the GLCM to compute a large number of texture
measures.
var glcmTexture = intNAIP.glcmTexture(7);
print('view the glcmTexture output', glcmTexture);
```

Now let us display the contrast results for the red, green, and blue bands. Contrast is the second band and measures the local contrast of an image.

```
// Display the 'contrast' results for the red, green and
blue bands.
var contrastVis = {
    bands: ['R_contrast', 'G_contrast', 'B_contrast'],
    min: 40,
    max: 1000
};

Map.addLayer(glcmTexture, contrastVis, 'contrast');
```

The resulting image highlights where there are differences in the contrast. For example, in Fig. 10.15, we can see that the red band has high contrast within this patchy field.

Spatial Statistics
Spatial statistics describe the distribution of different events across space and are extremely useful for remote sensing (Stein et al. 1998). Uses include anomaly detection, topographical analysis including terrain segmentation, and texture analysis using spatial association, which is how we will use it here. Two interesting

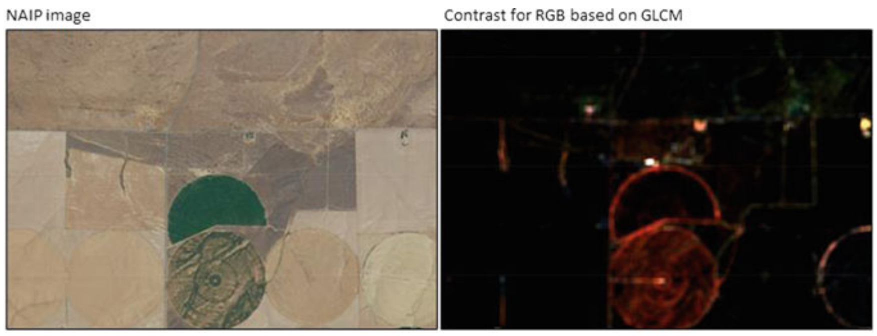


Fig. 10.15 Example of the contrast metric of GLCM for the NIR integer band in NAIP imagery using a 7 m kernel

texture measures from the field of spatial statistics include local Moran's I and local Geary's C (Anselin 1995).

To compute a local Geary's C with the NAIP image as input, first create a 9 × 9 kernel and then calculate local Geary's C.

```
// Begin spatial statistics example using Geary's C.

// Create a list of weights for a 9x9 kernel.
var list = [1, 1, 1, 1, 1, 1, 1, 1, 1];
// The center of the kernel is zero.
var centerList = [1, 1, 1, 1, 0, 1, 1, 1, 1];
// Assemble a list of lists: the 9x9 kernel weights as a
2-D matrix.
var lists = [list, list, list, list, centerList, list,
list, list, list
];
// Create the kernel from the weights.
// Non-zero weights represent the spatial neighborhood.
var kernel = ee.Kernel.fixed(9, 9, lists, -4, -4, false);
```

Now that we have a kernel, we can calculate the maximum of the four NAIP bands and use this with the kernel to calculate local Geary's C. There is no built-in function for Geary's C in Earth Engine, so we create our own using the `subtract`, `pow` (power), `sum`, and `divide` functions (Chap. 9).

```
// Use the max among bands as the input.
var maxBands = imageNAIP.reduce(ee.Reducer.max());

// Convert the neighborhood into multiple bands.
var neighBands = maxBands.neighborhoodToBands(kernel);

// Compute local Geary's C, a measure of spatial
association.
var gearys =
maxBands.subtract(neighBands).pow(2).reduce(ee.Reducer
        .sum())
    .divide(Math.pow(9, 2));

Map.addLayer(gearys, {
    min: 20,
    max: 2500
}, "Geary's C");
```

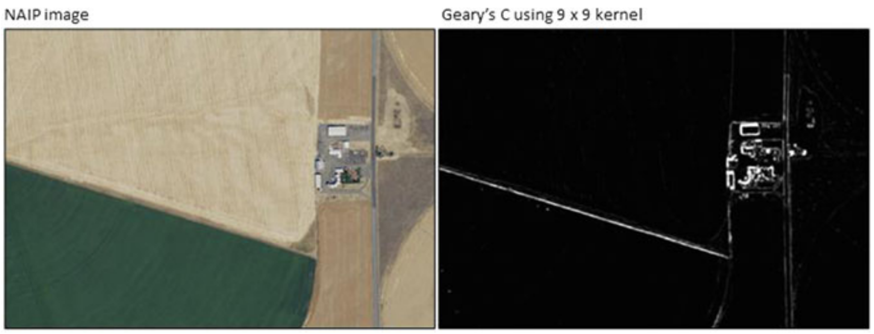


Fig. 10.16 Example of Geary's C for the NAIP imagery using a 9 m kernel

Inspecting the resulting layer shows that boundaries between fields, building outlines, and roads have high values of Geary's C. This makes sense because across bands, there will be high spatial autocorrelation within fields that are homogenous, whereas between fields (at the field boundary) the area will be highly heterogeneous (Fig. 10.16).

Code Checkpoint F32d. The book's repository contains a script that shows what your code should look like at this point.

10.3 Synthesis

In this chapter, we have explored many different neighborhood-based image transformations. These transformations have practical applications for remote sensing image analysis. Using transformation, you can use what you have learned in this chapter and in 6 to:

- Use raw imagery (red, green, blue, and near-infrared bands) to create an image classification.
- Use neighborhood transformations to create input imagery for an image classification and run the image classification.
- Use one or more of the morphological transformations to clean up your image classification.

Assignment 1. Compare and contrast your image classifications when using raw imagery compared with using neighborhood transformations.

Assignment 2. Compare and contrast your unaltered image classification and your image classification following morphological transformations.

10.4 Conclusion

Neighborhood-based image transformations enable you to extract information about each pixel's neighborhood to perform multiple important operations. Among the most important are smoothing, edge detection and definition, morphological processing, texture analysis, and spatial analysis. These transformations are a foundational part of many larger remote sensing analysis workflows. They may be used as imagery pre-processing steps and as individual layers in regression and classifications, to inform change detection, and for other purposes.

References

Anselin L (1995) Local indicators of spatial association—LISA. Geogr Anal 27:93–115. https://doi.org/10.1111/j.1538-4632.1995.tb00338.x

Canny J (1986) A computational approach to edge detection. IEEE Trans Pattern Anal Mach Intell PAMI-8:679–698. https://doi.org/10.1109/TPAMI.1986.4767851

Castleman KR (1996) Digital image processing. Prentice Hall Press

Conners RW, Trivedi MM, Harlow CA (1984) Segmentation of a high-resolution urban scene using texture operators. Comput Vision, Graph Image Process 25:273–310. https://doi.org/10.1016/0734-189X(84)90197-X

Haralick RM, Dinstein I, Shanmugam K (1973) Textural features for image classification. IEEE Trans Syst Man Cybern SMC-3:610–621. https://doi.org/10.1109/TSMC.1973.4309314

Lüttig C, Neckel N, Humbert A (2017) A combined approach for filtering ice surface velocity fields derived from remote sensing methods. Remote Sens 9:1062. https://doi.org/10.3390/rs9101062

Schowengerdt RA (2006) Remote sensing: models and methods for image processing. Elsevier

Stein A, Bastiaanssen WGM, De Bruin S et al (1998) Integrating spatial statistics and remote sensing. Int J Remote Sens 19:1793–1814. https://doi.org/10.1080/014311698215252

Stuckens J, Coppin PR, Bauer ME (2000) Integrating contextual information with per-pixel classification for improved land cover classification. Remote Sens Environ 71:282–296. https://doi.org/10.1016/S0034-4257(99)00083-8

Toure SI, Stow DA, Shih H-C et al (2018) Land cover and land use change analysis using multi-spatial resolution data and object-based image analysis. Remote Sens Environ 210:259–268. https://doi.org/10.1016/j.rse.2018.03.023

Vaiphasa C (2006) Consideration of smoothing techniques for hyperspectral remote sensing. ISPRS J Photogramm Remote Sens 60:91–99. https://doi.org/10.1016/j.isprsjprs.2005.11.002

Wang M, Hu C (2015) Extracting oil slick features from VIIRS nighttime imagery using a Gaussian filter and morphological constraints. IEEE Geosci Remote Sens Lett 12:2051–2055. https://doi.org/10.1109/LGRS.2015.2444871

ZhiYong L, Shi W, Benediktsson JA, Gao L (2018) A modified mean filter for improving the classification performance of very high-resolution remote-sensing imagery. Int J Remote Sens 39:770–785. https://doi.org/10.1080/01431161.2017.1390275

Object-Based Image Analysis

11

Morgan A. Crowley, Jeffrey A. Cardille, and Noel Gorelick

Overview

Pixel-based classification can include unwanted noise. Techniques for object-based image analysis are designed to detect objects within images, making classifications that can address this issue of classification noise. In this chapter, you will learn how region growing can be used to identify objects in satellite imagery within Earth Engine. By understanding how objects can be delineated and treated in an image, students can apply this technique to their own images to produce landscape assessments with less extraneous noise. Here, we treat images with an object delineator and view the results of simple classifications to view similarities and differences.

Learning Outcomes

- Learning about object-based image classification in Earth Engine.
- Controlling noise in images by adjusting different aspects of object segmentation.

M. A. Crowley (✉)
Natural Resources Canada, Canadian Forest Service—Great Lakes Forestry Centre, 1219 Queen Street E, Sault Ste Marie, ON, Canada
e-mail: morgan.crowley@nrcan-rncan.gc.ca

M. A. Crowley · J. A. Cardille
Department of Natural Resource Sciences, McGill University, Macdonald Campus, 21111 Lakeshore, Sainte-Anne-de-Bellevue, QC, Canada
e-mail: jeffrey.cardille@mcgill.ca

J. A. Cardille
Bieler School of Environment, McGill University, 3534 Rue University, Montreal, QC, Canada

N. Gorelick
Google Switzerland, Brandschenkestrasse 110, 8002 Zurich, Switzerland
e-mail: gorelick@google.com

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_11

- Understanding differences through time of noise in images.
- Creating and viewing objects from different sensors.

Assumes you know how to

- Import images and image collections, filter, and visualize (Part I).
- Perform basic image analysis: select bands, compute indices, create masks, classify images (Part II).
- Create a function for code reuse (Chap. 1).
- Perform pixel-based supervised and unsupervised classifications (Chap. 6).

11.1 Introduction to Theory

Building upon traditional pixel-based classification techniques, object-based image analysis classifies imagery into objects using perception-based, meaningful knowledge (Blaschke et al. 2000; Blaschke 2010; Weih and Riggan 2010). Detecting and classifying objects in a satellite image are a two-step approach. First, the image is segmented using a segmentation algorithm. Second, the landscape objects are classified using either supervised or unsupervised approaches. Segmentation algorithms create pixel clusters using imagery information such as texture, color or pixel values, shape, and size. Object-based image analysis is especially useful for mapping forest disturbances (Blaschke 2010; Wulder et al. 2004) because additional information and context are integrated into the classification through the segmentation process. One object-based image analysis approach available in Earth Engine is the Simple Non-Iterative Clustering (SNIC) segmentation algorithm (Achanta and Süsstrunk 2017). SNIC is a bottom-up, seed-based segmentation algorithm that assembles clusters from neighboring pixels based on parameters of compactness, connectivity, and neighborhood size. SNIC has been used in previous Earth Engine-based research for mapping land use and land cover (Shafizadeh-Moghadam et al. 2021; Tassi and Vizzari 2020), wetlands (Mahdianpari et al. 2018 and 2020, Amani et al. 2019), burned areas (Crowley et al. 2019), sustainable development goal indicators (Mariathasan et al. 2019), and ecosystem services (Verde et al. 2020).

11.2 Practicum

11.2.1 Section 1: Unsupervised Classification

In earlier chapters (see Chap. 6), you saw how to perform a supervised and unsupervised classification. In this lab, we will focus on object-based segmentation and unsupervised classifications—a clean and simple way to look at the spectral and spatial variability that is seen by a classification algorithm.

We will now build a script in several numbered sections, giving you a chance to see how it is constructed as well as to observe intermediate and contrasting results as you proceed. We will start by defining a function for taking an image and breaking it into a set of unsupervised classes. When called, this function will divide the image into a specified number of classes, without directly using any spatial characteristics of the image.

Paste the following block into a new script.

```
// 1.1 Unsupervised k-Means classification

// This function does unsupervised clustering classification
// input = any image. All bands will be used for clustering.
// numberOfUnsupervisedClusters = tunable parameter for how
//        many clusters to create.
var afn_Kmeans = function(input,
numberOfUnsupervisedClusters,
    defaultStudyArea, nativeScaleOfImage) {

    // Make a new sample set on the input. Here the sample
set is
    // randomly selected spatially.
    var training = input.sample({
        region: defaultStudyArea,
        scale: nativeScaleOfImage,
        numPixels: 1000
    });

    var cluster = ee.Clusterer.wekaKMeans(
            numberOfUnsupervisedClusters)
        .train(training);

    // Now apply that clusterer to the raw image that was
also passed in.
    var toexport = input.cluster(cluster);

    // The first item is the unsupervised classification.
Name the band.
    var clusterUnsup = toexport.select(0).rename(
        'unsupervisedClass');
    return (clusterUnsup);
};
```

We will also need a function to normalize the band values to a common scale from 0 to 1. This will be most useful when we are creating objects. Additionally, we will need a function to add the mean to the band name. Paste the following functions into your code. Note that, the code numbering skips intentionally from 1.2 to 1.4; we will add Sect. 1.3 later.

```
// 1.2 Simple normalization by maxes function.
var afn_normalize_by_maxes = function(img, bandMaxes) {
    return img.divide(bandMaxes);
};

// 1.4 Simple add mean to Band Name function
var afn_addMeanToBandName = (function(i) {
    return i + '_mean';
});
```

We will create a section that defines variables that you will be able to adjust. One important adjustable parameter is the number of clusters for the clusterer to use. Add the following code beneath the function definitions.

```
//////////////////////////////////////////////////////////////
// 2. Parameters to function calls
//////////////////////////////////////////////////////////////

// 2.1. Unsupervised KMeans Classification Parameters
var numberOfUnsupervisedClusters = 4;
```

The script will allow you to zoom to a specified area for better viewing and exists already in the code repository check points. Add this code below.

```
/////////////////////////////////////////////////////////////
// 2.2. Visualization and Saving parameters
// For different images, you might want to change the min
and max
// values to stretch. Useful for images 2 and 3, the
normalized images.
var centerObjectYN = true;
```

Now, with these functions, parameters, and flags in place, let us define a new image and set image-specific values that will help analyze it. We will put this in a new section of the code that contains “if” statements for images from multiple sensors. We set up the code like this because we will use several images from different sensors in the following sections; therefore, they are preloaded, so all that you have to do are to change the parameter “whichImage”. In this particular Sentinel-2 image, focus on differentiating forest and non-forest regions in the

Puget Sound, Washington, USA. The script will automatically zoom to the region of interest.

```
//////////////////////////////////////////////////////////////
// 3. Statements
//////////////////////////////////////////////////////////////

// 3.1  Selecting Image to Classify
var whichImage = 1; // will be used to select among images
if (whichImage == 1) {
    // Image 1.
    // Puget Sound, WA: Forest Harvest
    // (April 21, 2016)
    // Harvested Parcels
    // Clear Parcel Boundaries
    // Sentinel 2, 10m
    var whichCollection = 'COPERNICUS/S2';
    var ImageToUseID =
'20160421T191704_20160421T212107_T10TDT';
    var originalImage = ee.Image(whichCollection + '/' +
    ImageToUseID);
    print(ImageToUseID, originalImage);
    var nativeScaleOfImage = 10;
    var threeBandsToDraw = ['B4', 'B3', 'B2'];
    var bandsToUse = ['B4', 'B3', 'B2'];
    var bandMaxes = [1e4, 1e4, 1e4];
    var drawMin = 0;
    var drawMax = 0.3;
    var defaultStudyArea = ee.Geometry.Polygon(
        [
            [
                [-123.13105468749993, 47.612974066532004],
                [-123.13105468749993, 47.56214700543596],
                [-123.00179367065422, 47.56214700543596],
                [-123.00179367065422, 47.612974066532004]
            ]
        ]);
    var zoomArea = ee.Geometry.Polygon(
        [
            [
                [-123.13105468749993, 47.612974066532004],
                [-123.13105468749993, 47.56214700543596],
                [-123.00179367065422, 47.56214700543596],
                [-123.00179367065422, 47.612974066532004]
```

```
              ]
         ], null, false);
  }
  Map.addLayer(originalImage.select(threeBandsToDraw), {

      min:0,
      max:2000
  }, '3.1 '+ ImageToUseID, true, 1);
```

Now, let us clip the image to the study area we are interested in, then extract the bands to use for the classification process.

```
////////////////////////////////////////////////////////////
// 4. Image Preprocessing
////////////////////////////////////////////////////////////
var clippedImageSelectedBands =
originalImage.clip(defaultStudyArea)
    .select(bandsToUse);
var ImageToUse =
afn_normalize_by_maxes(clippedImageSelectedBands,
    bandMaxes);

Map.addLayer(ImageToUse.select(threeBandsToDraw), {
        min: 0.028,
        max: 0.12
    },
    '4.3 Pre-normalized image', true, 0);
```

Now, let us view the per-pixel unsupervised classification, produced using the *k*-means classifier. Note that, as we did earlier, we skip a section of the code numbering (moving from Sects. 4 to 6), which we will fill in later as the script is developed further.

```
////////////////////////////////////////////////////////////
// 6. Execute Classifications
////////////////////////////////////////////////////////////

// 6.1 Per Pixel Unsupervised Classification for Comparison
var PerPixelUnsupervised = afn_Kmeans(ImageToUse,
    numberOfUnsupervisedClusters, defaultStudyArea,
    nativeScaleOfImage);
Map.addLayer(PerPixelUnsupervised.select('unsupervisedClass'
)
    .randomVisualizer(), {}, '6.1 Per-Pixel Unsupervised',
true, 0
);
print('6.1b Per-Pixel Unsupervised Results:',
PerPixelUnsupervised);
```

Then, insert this code, so that you can zoom if requested.

```
////////////////////////////////////////////////////////////
// 7. Zoom if requested
////////////////////////////////////////////////////////////
if (centerObjectYN === true) {
    Map.centerObject(zoomArea, 14);
}
```

Code Checkpoint F33a. The book's repository contains a script that shows what your code should look like at this point.

Run the script. It will draw the study area in a true-color view (Fig. 11.1), where you can inspect the complexity of the landscape as it would have appeared to your eye in 2016, when the image was captured.

Note harvested forests of different ages, the spots in the northwest part of the study area that might be naturally treeless, and the straight easements for transmission lines in the eastern part of the study area. You can switch Earth Engine to satellite view and change the transparency of the drawn layer to inspect what has changed in the years since the image was captured.

As it drew your true-color image, Earth Engine also executed the *k*-means classification and added it to your set of layers. Turn up the visibility of layer 6.1 Per-Pixel Unsupervised, which shows the four-class per-pixel classification result using randomly selected colors. The result should look something like Fig. 11.2.

Take a look at the image that was produced, using the transparency slider to inspect how well you think the classification captured the variability in the landscape and classified similar classes together, then answer the following questions.

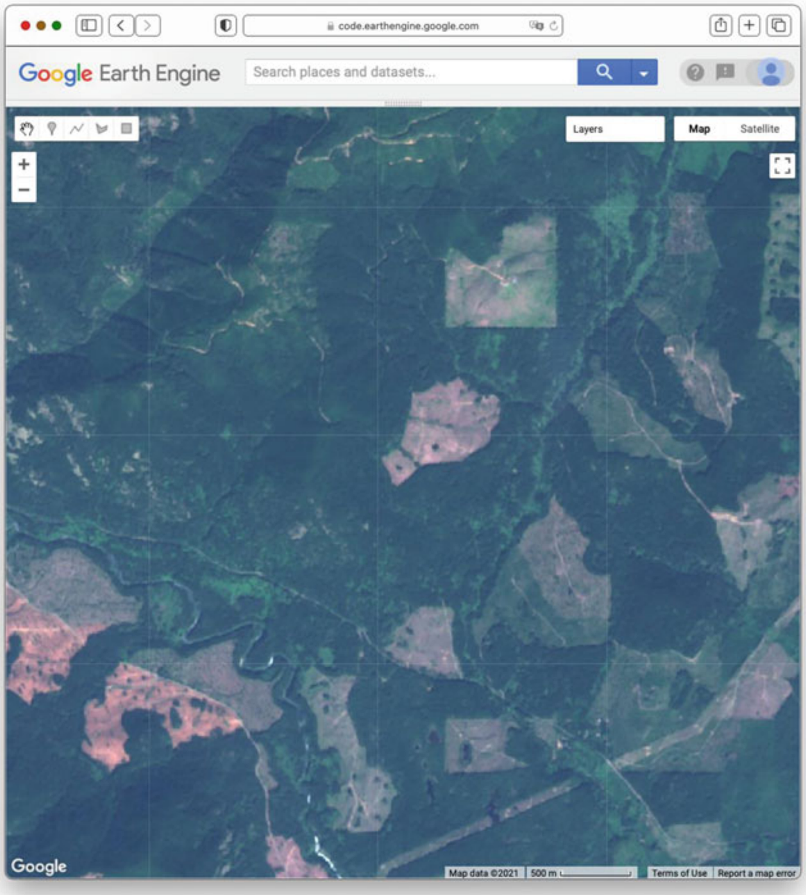


Fig. 11.1 True-color Sentinel-2 image from 2016 for the study area

Question 1. In your opinion, what are some of the strengths and weaknesses of the map that resulted from your settings?

Question 2. Part of the image that appears to our eye to represent a single land use might be classified by the *k*-means classification as containing different clusters. Is that a problem? Why or why not?

Question 3. A given unsupervised class might represent more than one land use/land cover type in the image. Use the **Inspector** to find classes for which there were these types of overlaps. Is that a problem? Why or why not?

Question 4. You can change the `numberOfUnsupervisedClusters` variable to be more or less than the default value of 4. Which, if any, of the resulting maps

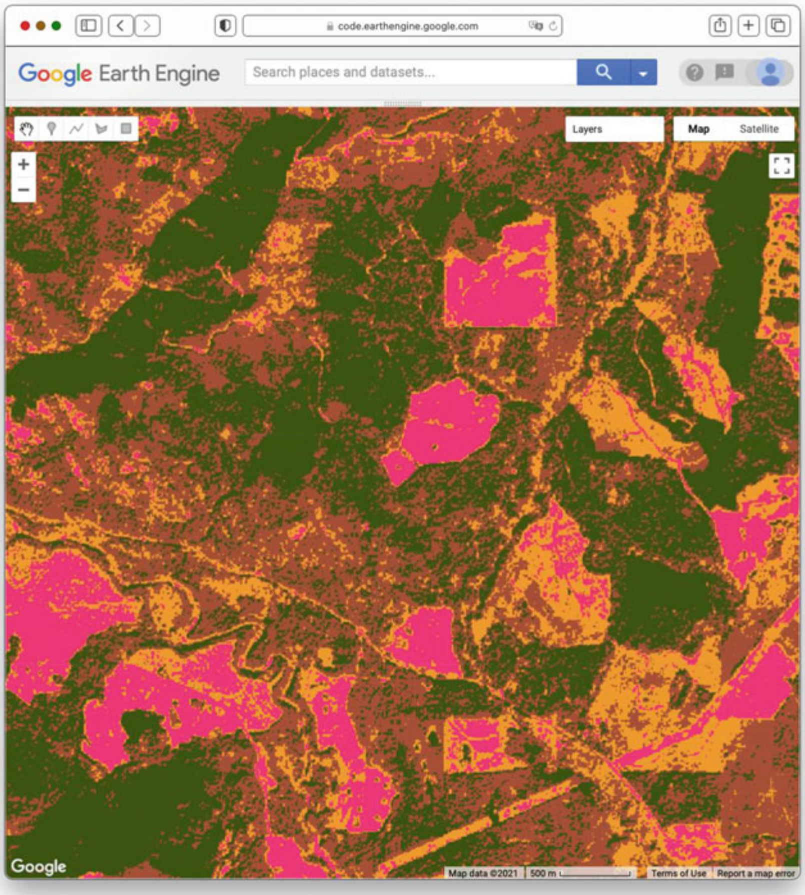


Fig. 11.2 Pixel-based unsupervised classification using four-class *k*-means unsupervised classification using bands from the visible spectrum

produce a more satisfying image? Is there an upper limit at which it is hard for you to tell whether the classification was successful or not?

As discussed in earlier chapters, the visible part of the electromagnetic spectrum contains only part of the information that might be used for a classification. The short-wave infrared bands have been seen in many applications to be more informative than the true-color bands.

Return to your script and find the place where `threeBandsToDraw` is set. That variable is currently set to B4, B3, and B2. Comment out that line and use the one below, which will set the variable to B8, B11, and B12. Make the same change for the variable `bandsToUse`. Now, run this modified script, which will use three new bands for the classification and also draw them to the screen for you to see. You will notice that this band combination provides different contrasts among cover types. For example, you might now notice that there are small bodies of water and a river in the scene, details that are easy to overlook in a true-color image. With `numberOfUnsupervisedClusters` still set at 4, your resulting classification should look like Fig. 11.3.

Question 5. Did using the bands from outside the visible part of the spectrum change any classes so that they are more cleanly separated by land use or land cover? Keep in mind that the colors which are randomly chosen in each of the

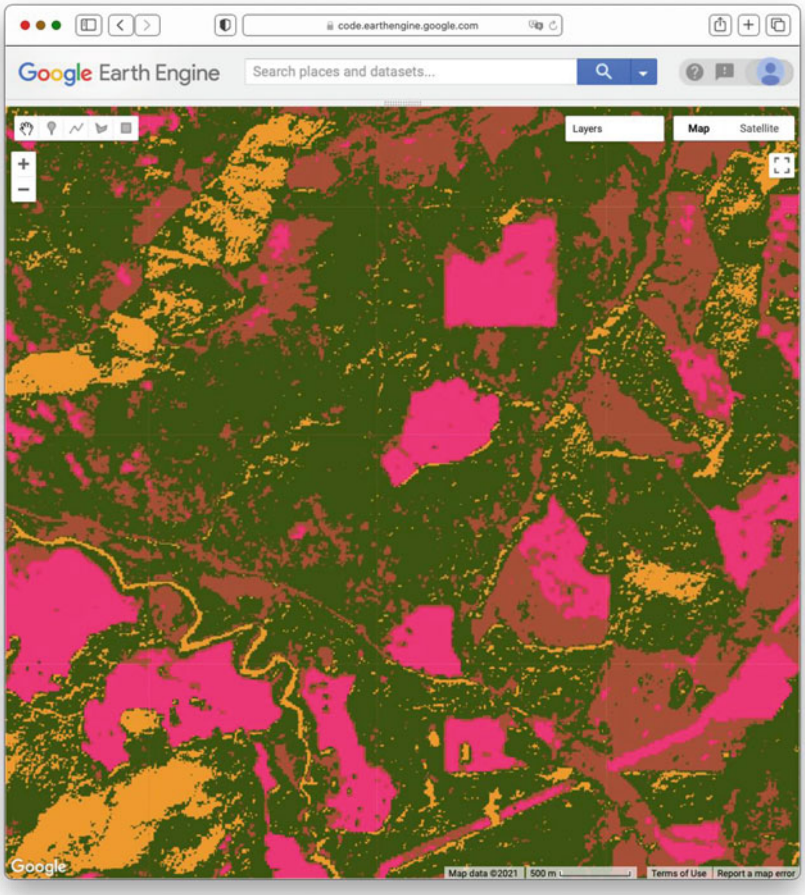


Fig. 11.3 Pixel-based unsupervised classification using four-class *k*-means unsupervised classification using bands from outside of the visible spectrum

images are unrelated—a class colored brown in Fig. 11.2 might well be pink in Fig. 11.3.

Question 6. Experiment with adjusting the `numberOfUnsupervised Clusters` with this new dataset. Is one combination preferable to another, in your opinion? Keep in mind that there is no single answer about the usefulness of an unsupervised classification beyond asking whether it separates classes of importance to the user.

Code Checkpoint F33b. The book's repository contains a script that shows what your code should look like at this point. In that code, the `numberOfUnsupervisedClusters` is set to 4, and the infrared bands are used as part of the classification process.

11.2.2 Section 2: Detecting Objects in Imagery with the SNIC Algorithm

The noise you noticed in the pixel-based classification will now be improved using a two-step approach for object-based image analysis. First, you will segment the image using the SNIC algorithm, and then, you will classify it using a *k*-means unsupervised classifier. Return to your script, where we will add a new function. Noting that the code's sections are numbered, find code Sect. 1.2 and add the function below beneath it.

```
// 1.3 Seed Creation and SNIC segmentation Function
var afn_SNIC = function(imageOriginal, SuperPixelSize,
Compactness,
    Connectivity, NeighborhoodSize, SeedShape) {
    var theSeeds =
ee.Algorithms.Image.Segmentation.seedGrid(
        SuperPixelSize, SeedShape);
    var snic2 = ee.Algorithms.Image.Segmentation.SNIC({
        image: imageOriginal,
        size: SuperPixelSize,
        compactness: Compactness,
        connectivity: Connectivity,
        neighborhoodSize: NeighborhoodSize,
        seeds: theSeeds
    });
    var theStack = snic2.addBands(theSeeds);
    return (theStack);
};
```

As you see, the function assembles parameters needed for running SNIC (Achanta and Süsstrunk 2017; Crowley et al. 2019), the function that delineates objects in an image. A call to SNIC takes several parameters that we will explore. Add the following code below code Sect. 2.2.

```
// 2.3 Object-growing parameters to change
// Adjustable Superpixel Seed and SNIC segmentation Parameters:
// The superpixel seed location spacing, in pixels.
var SNIC_SuperPixelSize = 16;
// Larger values cause clusters to be more compact
(square/hexagonal).
// Setting this to 0 disables spatial distance weighting.
var SNIC_Compactness = 0;
// Connectivity. Either 4 or 8.
var SNIC_Connectivity = 4;
// Either 'square' or 'hex'.
var SNIC_SeedShape = 'square';

// 2.4 Parameters that can stay unchanged
// Tile neighborhood size (to avoid tile boundary artifacts).
Defaults to 2 * size.
var SNIC_NeighborhoodSize = 2 * SNIC_SuperPixelSize;
```

Now, add a call to the SNIC function. You will notice that it takes the parameters specified in code Sect. 11.2.2 and sends them to the SNIC algorithm. Place the code below into the script as the code's Sect. 11.2.5, between Sects. 4 and 6.

```
////////////////////////////////////////////////////////////
// 5. SNIC Clustering
////////////////////////////////////////////////////////////

// This function returns a multi-banded image that has had
SNIC
// applied to it. It automatically determine the new names
// of the bands that will be returned from the segmentation.
print('5.1 Execute SNIC');
var SNIC_MultiBandedResults = afn_SNIC(
    ImageToUse,
    SNIC_SuperPixelSize,
    SNIC_Compactness,
    SNIC_Connectivity,
    SNIC_NeighborhoodSize,
    SNIC_SeedShape
);

var SNIC_MultiBandedResults = SNIC_MultiBandedResults
    .reproject('EPSG:3857', null, nativeScaleOfImage);
print('5.2 SNIC Multi-Banded Results',
SNIC_MultiBandedResults);

Map.addLayer(SNIC_MultiBandedResults.select('clusters')
    .randomVisualizer(), {}, '5.3 SNIC Segment Clusters',
true, 1);

var theSeeds = SNIC_MultiBandedResults.select('seeds');
Map.addLayer(theSeeds, {
    palette: 'red'
}, '5.4 Seed points of clusters', true, 1);

var bandMeansToDraw =
threeBandsToDraw.map(afn_addMeanToBandName);
print('5.5 band means to draw', bandMeansToDraw);
var clusterMeans =
SNIC_MultiBandedResults.select(bandMeansToDraw);
print('5.6 Cluster Means by Band', clusterMeans);
Map.addLayer(clusterMeans, {
    min: drawMin,
    max: drawMax
}, '5.7 Image repainted by segments', true, 0);
```

Now, run the script. It will draw several layers, with the one shown in Fig. 11.4 on top.

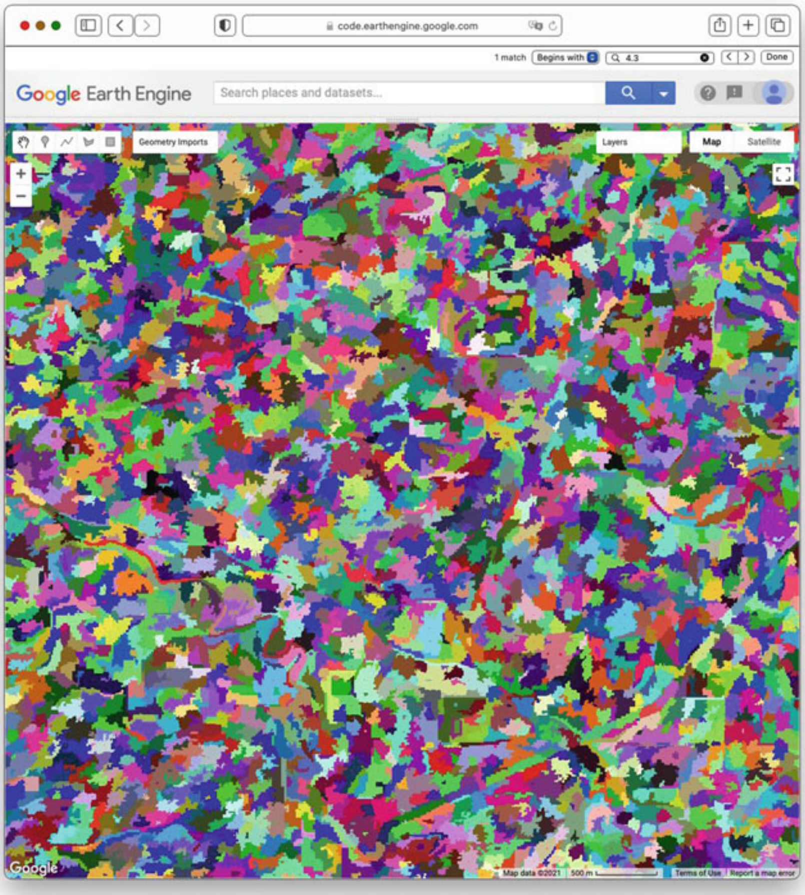


Fig. 11.4 SNIC clusters, with randomly chosen colors for each cluster

This shows the work of SNIC on the image sent to it—in this case, on the composite of bands 8, 11, and 12. If you look closely at the multicolored layer, you can see small red "seed" pixels. To initiate the process, these seeds are created and used to form square or hexagonal "superpixels" at the spacing given by the parameters passed to the function. The edges of these blocks are then pushed and pulled and directed to stop at edges in the input image. As part of the algorithm, some superpixels are then merged to form larger blocks, which is why you will find that some of the shapes contain two or more seed pixels.

Explore the structure by changing the transparency of layer 5 to judge how the image segmentation performs for the given set of parameter values. You can also compare layer 5.7–layer 3.1. Layer 5.7 is a reinterpretation of layer 3.1 in which every pixel in a given shape of layer 5.3 is assigned the mean value of the pixels inside the shape. When parameterized in a way that is useful for a given project goal, parts of the image that are homogeneous will get the same color, while areas of high heterogeneity will get multiple colors.

Now, spend some time exploring the effect of the parameters that control the code's behavior. Use your tests to answer the questions below.

Question 7. What is the effect on the SNIC clusters of changing the parameter `SNIC_SuperPixelSize`?

Question 8. What is the effect of changing the parameter `SNIC_Compactness`?

Question 9. What are the effects of changing the parameters `SNIC_Connectivity` and `SNIC_SeedShape`?

Code Checkpoint F33c. The book's repository contains a script that shows what your code should look like at this point.

11.2.3 Section 3: Object-Based Unsupervised Classification

The *k*-means classifier used in this tutorial is not aware that we would often prefer to have adjacent pixels to be grouped into the same class—it has no sense of physical space. This is why you see the noise in the unsupervised classification. However, because we have re-colored the pixels in a SNIC cluster to all share the exact same band values, *k*-means will group all pixels of each cluster to have the same class. In the best-case scenario, this allows us to enhance our classification from being pixel-based to reveal clean and unambiguous objects in the landscape. In this section, we will classify these objects, exploring the strengths and limitations of finding objects in this image.

Return the SNIC settings to their first values, namely:

```
// The superpixel seed location spacing, in pixels.
var SNIC_SuperPixelSize = 16;
// Larger values cause clusters to be more compact
(square/hexagonal).
// Setting this to 0 disables spatial distance weighting.
var SNIC_Compactness = 0;
// Connectivity. Either 4 or 8.
var SNIC_Connectivity = 4;
// Either 'square' or 'hex'.
var SNIC_SeedShape = 'square';
```

As code Sect. 6.2, add this code, which will call the SNIC function and draw the results.

```
// 6.2 SNIC Unsupervised Classification for Comparison
var bandMeansNames = bandsToUse.map(afn_addMeanToBandName);
print('6.2 band mean names returned by segmentation',
bandMeansNames);
var meanSegments =
SNIC_MultiBandedResults.select(bandMeansNames);
var SegmentUnsupervised = afn_Kmeans(meanSegments,
    numberOfUnsupervisedClusters, defaultStudyArea,
    nativeScaleOfImage);
Map.addLayer(SegmentUnsupervised.randomVisualizer(), {},
    '6.3 SNIC Clusters Unsupervised', true, 0);
print('6.3b Per-Segment Unsupervised Results:',
SegmentUnsupervised);
/////////////////////////////////////////////////////////////
```

When you run the script, that new function will classify the image in layer 5.7, which is the recoloring of the original image according to the segments shown in layer 5. Compare the classification of the superpixels (6.3) with the unsupervised classification of the pixel-by-pixel values (6.1). You should be able to change the transparency of those two layers to compare them directly.

Question 10. What are the differences between the unsupervised classifications of the per-pixel and SNIC-interpreted images? Describe the tradeoff between removing noise and erasing important details.

Code Checkpoint F33d. The book's repository contains a script that shows what your code should look like at this point.

11.2.4 Section 4: Classifications with More or Less Categorical Detail

Recall the variable `numberOfUnsupervisedClusters`, which directs the *k*-means algorithm to partition the dataset into that number of classes. Because the colors are chosen randomly for layer 6.3, any change to this number typically results in an entirely different color scheme. Changes in the color scheme can also occur if you were to use a slightly different study area size between two runs. Although this can make it hard to compare the results of two unsupervised algorithms, it is a useful reminder that the unsupervised classification labels do not necessarily correspond to a single land use/land cover type.

Question 11. Find the `numberOfUnsupervisedClusters` variable in the code and set it to different values. You might test it across powers of two: 2, 4, 8, 16, 32, and 64 clusters will all look visually distinct. In your opinion, does one of them best discriminate between the classes in the image? Is there a particular number of colors that is too complicated for you to understand?

Question 12. What concrete criteria could you use to determine whether a particular unsupervised classification is good or bad for a given goal?

11.2.5 Section 5: Effects of SNIC Parameters

The number of classes controls the partition of the landscape for a given set of SNIC clusters. The four parameters of SNIC, in turn, influence the spatial characteristics of the clusters produced for the image. Adjust the four parameters of SNIC: `SNIC_SuperPixelSize`, `SNIC_Compactness`, `SNIC_Connectivity`, and `SNIC_SeedShape`. Although their workings can be complex, you should be able to learn what characteristics of the SNIC clustering they control by changing each one individually. At that point, you can explore the effects of changing multiple values for a single run. Recall that the ultimate goal of this workflow is to produce an unsupervised classification of landscape objects, which may relate to the SNIC parameters in very complex ways. You may want to start by focusing on the effect of the SNIC parameters on the cluster characteristics (layer 5.3) and then look at the associated unsupervised classification layer 6.

Question 13. What is the effect on the unsupervised classification of SNIC clusters of changing the parameter `SNIC_SuperPixelSize`?

Question 14. What is the effect of changing the parameter `SNIC_Compactness`?

Question 15. What are the effects of changing the parameters `SNIC_Connectivity` and `SNIC_SeedShape`?

Question 16. For this image, what is the combination of parameters that, in your subjective judgment, best captures the variability in the scene while minimizing unwanted noise?

11.3 Synthesis

Assignment 1. Additional images from other remote sensing platforms can be found in script **F33s1** in the book's repository. Run the classification procedure on these images and compare the results from multiple parameter combinations.

Assignment 2. Although this exercise was designed to remove or minimize spatial noise, it does not treat temporal noise. With a careful choice of imagery, you can explore the stability of these methods and settings for images from different dates. Because the MODIS sensor, for example, can produce images on consecutive days, you would expect that the objects identified in a landscape would be nearly identical from one day to the next. Is this the case? To go deeper, you might contrast temporal stability as measured by different sensors. Are some sensors more stable in their object creation than others? To go even deeper, you might consider how you would quantify this stability using concrete measures that could be compared across different sensors, places, and times. What would these measures be?

11.4 Conclusion

Object-based image analysis is a method for classifying satellite imagery by segmenting neighboring pixels into objects using pre-segmented objects. The identification of candidate image objects is readily available in Earth Engine using the SNIC segmentation algorithm. In this chapter, you applied the SNIC segmentation and the unsupervised *k*-means algorithm to satellite imagery. You illustrated how the segmentation and classification parameters can be customized to meet your classification objective and to reduce classification noise. Now that you understand the basics of detecting and classifying image objects in Earth Engine, you can explore further by applying these methods on additional data sources.

References

Achanta R, Süsstrunk S (2017) Superpixels and polygons using simple non-iterative clustering. In: Proceedings—30th IEEE conference on computer vision and pattern recognition, CVPR 2017, pp 4895–4904

Amani M, Mahdavi S, Afshar M et al (2019) Canadian wetland inventory using Google Earth Engine: the first map and preliminary results. Remote Sens 11:842. https://doi.org/10.3390/RS11070842

Blaschke T (2010) Object based image analysis for remote sensing. ISPRS J Photogram Remote Sens 65:2–16. https://doi.org/10.1016/j.isprsjprs.2009.06.004

Blaschke T, Lang S, Lorup E et al (2000) Object-oriented image processing in an integrated GIS/remote sensing environment and perspectives for environmental applications. Environ Inf Plann Polit Public 2:555–570

Crowley MA, Cardille JA, White JC, Wulder MA (2019) Generating intra-year metrics of wildfire progression using multiple open-access satellite data streams. Remote Sens Environ 232:111295. https://doi.org/10.1016/j.rse.2019.111295

Mahdianpari M, Salehi B, Mohammadimanesh F et al (2018) The first wetland inventory map of Newfoundland at a spatial resolution of 10 m using Sentinel-1 and Sentinel-2 data on the Google Earth Engine cloud computing platform. Remote Sens 11:43. https://doi.org/10.3390/rs11010043

Mahdianpari M, Salehi B, Mohammadimanesh F et al (2020) Big data for a big country: the first generation of Canadian wetland inventory map at a spatial resolution of 10-m using Sentinel-1 and Sentinel-2 data on the Google Earth Engine cloud computing platform. Can J Remote Sens 46:15–33. https://doi.org/10.1080/07038992.2019.1711366

Mariathasan V, Bezuidenhoudt E, Olympio KR (2019) Evaluation of Earth observation solutions for Namibia's SDG monitoring system. Remote Sens 11:1612. https://doi.org/10.3390/rs11131612

Shafizadeh-Moghadam H, Khazaei M, Alavipanah SK, Weng Q (2021) Google Earth Engine for large-scale land use and land cover mapping: an object-based classification approach using spectral, textural and topographical factors. Giscience Remote Sens 58:914–928. https://doi.org/10.1080/15481603.2021.1947623

Tassi A, Vizzari M (2020) Object-oriented LULC classification in Google Earth Engine combining SNIC, GLCM, and machine learning algorithms. Remote Sens 12:1–17. https://doi.org/10.3390/rs12223776

Verde N, Kokkoris IP, Georgiadis C et al (2020) National scale land cover classification for ecosystem services mapping and assessment, using multitemporal Copernicus EO data and Google Earth Engine. Remote Sens 12:1–24. https://doi.org/10.3390/rs12203303

Weih RC, Riggan ND (2010) Object-based classification vs. pixel-based classification: comparative importance of multi-resolution imagery. Int Arch Photogram Remote Sens Spat Inf Sci 38:C7

Wulder MA, Skakun RS, Kurz WA, White JC (2004) Estimating time since forest harvest using segmented Landsat ETM+ imagery. Remote Sens Environ 93:179–187. https://doi.org/10.1016/j.rse.2004.07.009

Part IV
Interpreting Image Series

One of the paradigm-changing features of Earth Engine is the ability to access decades of imagery without the previous limitation of needing to download all the data to a local disk for processing. Because remote-sensing data files can be enormous, this used to limit many projects to viewing two or three images from different periods. With Earth Engine, users can access tens or hundreds of thousands of images to understand the status of places across decades.

Filter, Map, Reduce

12

Jeffrey A. Cardille

Overview

The purpose of this chapter is to teach you important programming concepts as they are applied in Earth Engine. We first illustrate how the order and type of these operations can matter with a real-world, non-programming example. We then demonstrate these concepts with an `ImageCollection`, a key data type that distinguishes Earth Engine from desktop image processing implementations.

Learning Outcomes

- Visualizing the concepts of filtering, mapping, and reducing with a hypothetical, non-programming example.
- Gaining context and experience with filtering an `ImageCollection`.
- Learning how to efficiently map a user-written function over the images of a filtered `ImageCollection`.
- Learning how to summarize a set of assembled values using Earth Engine reducers.

Assumes you know how to

- Import images and image collections, filter, and visualize (Part I).
- Perform basic image analysis: select bands, compute indices, create masks (Part II).

J. A. Cardille (✉)
McGill University, Quebec, Canada
e-mail: jeffrey.cardille@mcgill.ca

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_12

12.1 Introduction to Theory

Prior chapters focused on exploring individual images—for example, viewing the characteristics of single satellite images by displaying different combinations of bands (Chap. 2), viewing single images from different datasets (Chap 3, and 4), and exploring image processing principles (Parts II, III) as they are implemented for cloud-based remote sensing in Earth Engine. Each image encountered in those chapters was pulled from a larger assemblage of images taken from the same sensor. The chapters used a few ways to narrow down the number of images in order to view just one for inspection (Part I) or manipulation (Part II, Part III).

In this chapter and most of the chapters that follow, we will move from the domain of single images to the more complex and distinctive world of working with image collections, one of the fundamental data types within Earth Engine. The ability to conceptualize and manipulate entire image collections distinguishes Earth Engine and gives it considerable power for interpreting change and stability across space and time.

When looking for change or seeking to understand differences in an area through time, we often proceed through three ordered stages, which we will color code in this first explanatory part of the lab:

1. *Filter*:: selecting subsets of images based on criteria of interest.
2. *Map*:: manipulating each image in a set in some way to suit our goals.
3. *Reduce*:: estimating characteristics of the time series.

For users of other programming languages—R, MATLAB, C, Karel, and many others—this approach might seem awkward at first. We explain it below with a non-programming example: going to the store to buy milk.

Suppose you need to go shopping for milk, and you have two criteria for determining where you will buy your milk: location and price. The store needs to be close to your home, and as a first step in deciding whether to buy milk today, you want to identify the lowest price among those stores. You do not know the cost of milk at any store ahead of time, so you need to efficiently contact each one and determine the minimum price to know whether it fits in your budget. If we were discussing this with a friend, we might say, “I need to find out how much milk costs at all the stores around here”. To solve that problem in a programming language, these words imply precise operations on sets of information. We can write the following “pseudocode”, which uses words that indicate logical thinking but that cannot be pasted directly into a program:

```
AllStoresOnEarth.filterNearbyStores.filterStoresWithMilk.getMilkPricesFromEach
Store.determineTheMinimumValue
```

Imagine doing these actions not on a computer but in a more old-fashioned way: calling on the telephone for milk prices, writing the milk prices on paper, and inspecting the list to find the lowest value. In this approach, we begin with AllStoresOnEarth, since there is at least some possibility that we could decide to visit any store on Earth, a set that could include millions of stores, with prices for millions or billions of items. A wise first action would be to limit ourselves to nearby stores. Asking to filterNearbyStores would reduce the number of potential stores to hundreds, depending on how far we are willing to travel for milk. Then, working with that smaller set, we further filterStoresWithMilk, limiting ourselves to stores that sell our target item. At that point in the filtering, imagine that just ten possibilities remain. Then, by telephone, we getMilkPricesFromEachStore, making a short paper list of prices. We then scan the list to determineTheMinimumValue to decide which store to visit.

In that example, each color plays a different role in the workflow. The AllStoresOnEarth set, any one of which might contain inexpensive milk, is an enormous collection. The filtering actions filterNearbyStores and filterStoresWithMilk are operations that can happen on any set of stores. These actions take a set of stores, do some operation to limit that set, and return that smaller set of stores as an answer. The action to getMilkPricesFromEachStore takes a simple idea—calling a store for a milk price—and “maps” it over a given set of stores. Finally, with the list of nearby milk prices assembled, the action to determineTheMinimumValue, a general idea that could be applied to any list of numbers, identifies the cheapest one.

The list of steps above might seem almost too obvious, but the choice and order of operations can have a big impact on the feasibility of the problem. Imagine if we had decided to do the same operations in a slightly different order:

```
AllStoresOnEarth.filterStoresWithMilk.getMilkPricesFromEachStore.filterNearbyStor
es.determineMinimumValue
```

In this approach, we first identify all the stores on Earth that have milk, then contact them one by one to get their current milk price. If the contact is done by phone, this could be a painfully slow process involving millions of phone calls. It would take considerable “processing” time to make each call and careful work to record each price onto a giant list. Processing the operations in this order would demand that only after entirely finishing the process of contacting every milk proprietor on Earth, we then identify the ones on our list that are not nearby enough to visit, then scan the prices on the list of nearby stores to find the cheapest one. This should ultimately give the same answer as the more efficient first example, but only after requiring so much effort that we might want to give up.

In addition to the greater order of magnitude of the list size, you can see that there are also possible slow points in the process. Could you make a million phone calls yourself? Maybe, but it might be pretty appealing to hire, say, 1000 people to help. While being able to make a large number of calls in parallel would speed up the calling stage, it is important to note that you would need to wait for all 1000 callers to return their sublists of prices. Why wait? Nearby stores could be on any caller's sublist, so any caller might be the one to find the lowest nearby price. The identification of the lowest nearby price would need to wait for the slowest caller, even if it turned out that all of that last caller's prices came from stores on the other side of the world.

This counterexample would also have other complications—such as the need to track store locations on the list of milk prices—that could present serious problems if you did those operations in that unwise order. For now, the point is to *filter*, then *map*, then *reduce*. Below, we will apply these concepts to image collections.

12.2 Practicum

12.2.1 Section 1: Filtering Image Collections in Earth Engine

The first part of the *filter, map, reduce* paradigm is "filtering" to get a smaller `ImageCollection` from a larger one. As in the milk example, filters take a large set of items, limit it by some criterion, and return a smaller set for consideration. Here, filters take an `ImageCollection`, limit it by some criterion of date, location, or image characteristics, and return a smaller `ImageCollection` (Fig. 12.1).

As described first in Chap. 3, the Earth Engine API provides a set of filters for the `ImageCollection` type. The filters can limit an `ImageCollection` based on spatial, temporal, or attribute characteristics. Filters were used in Parts I, II, and III without much context or explanation, to isolate an image from an `ImageCollection` for inspection or manipulation. The information below

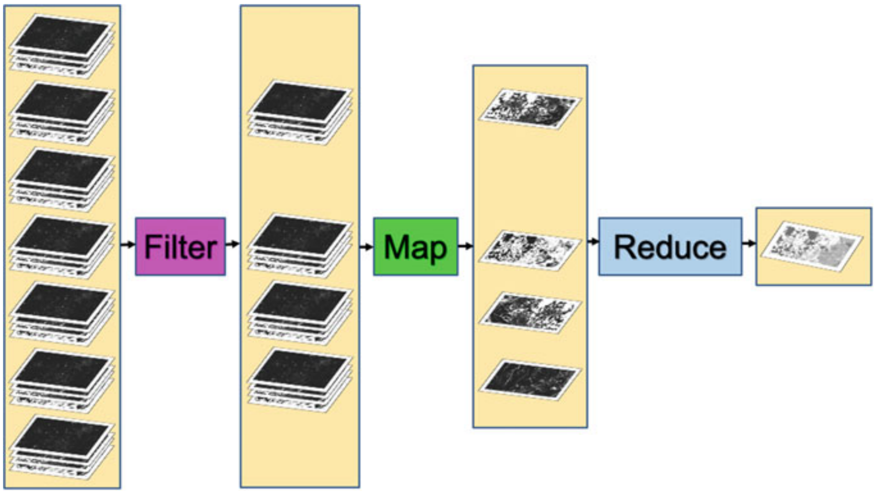


Fig. 12.1 *Filter*, *map*, *reduce* as applied to image collections in Earth Engine

should give perspective on that work while introducing some new tools for filtering image collections.

Below are three examples of limiting a Landsat 5 `ImageCollection` by characteristics and assessing the size of the resulting set.

FilterDate This takes an `ImageCollection` as input and returns an `ImageCollection` whose members satisfy the specified date criteria. We will adapt the earlier filtering logic seen in Chap. 3:

```
var imgCol = ee.ImageCollection('LANDSAT/LT05/C02/T1_L2');
// How many Tier 1 Landsat 5 images have ever been collected?
print("All images ever: ", imgCol.size()); // A very large
number

// How many images were collected in the 2000s?
var startDate = '2000-01-01';
var endDate = '2010-01-01';

var imgColfilteredByDate = imgCol.filterDate(startDate,
endDate);
print("All images 2000-2010: ", imgColfilteredByDate.size());
// A smaller (but still large) number
```

After running the code, you should get a very large number for the full set of images. You also will likely get a very large number for the subset of images over the decade-scale interval.

FilterBounds It may be that—similar to the milk example—only images near to a place of interest are useful for you. As first presented in Part I, `filterBounds` takes an `ImageCollection` as input and returns an `ImageCollection` whose images surround a specified location. If we take the `ImageCollection` that was filtered by date and then filter it by bounds, we will have filtered the collection to those images near a specified point within the specified date interval. With the code below, we will count the number of images in the Shanghai vicinity, first visited in Chap. 2, from the early 2000s:

```
var ShanghaiImage = ee.Image(
    'LANDSAT/LT05/C02/T1_L2/LT05_118038_20000606');
Map.centerObject(ShanghaiImage, 9);

var imgColfilteredByDateHere =
imgColfilteredByDate.filterBounds(Map
    .getCenter());
print("All images here, 2000-2010: ", imgColfilteredByDateHere
.size()); // A smaller number
```

If you could like, you could take a few minutes to explore the behavior of the script in different parts of the world. To do that, you would need to comment out the `Map.centerObject` command to keep the map from moving to that location each time you run the script.

Filter by Other Image Metadata As first explained in Chap. 4, the date and location of an image are characteristics stored with each image. Another important factor in image processing is the cloud cover, an image-level value computed for each image in many collections, including the Landsat and Sentinel-2 collections. The overall cloudiness score might be stored under different metadata tag names in different datasets. For example, for Sentinel-2, this overall cloudiness score is stored in the `CLOUDY_PIXEL_PERCENTAGE` metadata field. For Landsat 5, the `ImageCollection` we are using in this example, the image-level cloudiness score is stored using the tag `CLOUD_COVER`. If you are unfamiliar with how to find this information, these skills are first presented in Part I.

Here, we will access the `ImageCollection` that we just built using `filterBounds` and `filterDate` and then further filter the images by the image-level cloud cover score, using the `filterMetadata` function.

Next, let us remove any images with 50% or more cloudiness. As will be described in subsequent chapters working with per-pixel cloudiness information, you might want to retain those images in a real-life study, if you feel some values within cloudy images might be useful. For now, to illustrate the filtering concept, let us keep only images whose image-level cloudiness values indicate that the cloud coverage is lower than 50%. Here, we will take the set already filtered by bounds and date and further filter it using the cloud percentage into a new `ImageCollection`. Add this line to the script to filter by cloudiness and print the size to the **Console**.

```
var L5FilteredLowCloudImages = imgColfilteredByDateHere
    .filterMetadata('CLOUD_COVER', 'less_than', 50);
print("Less than 50% clouds in this area, 2000-2010",
    L5FilteredLowCloudImages.size()); // A smaller number
```

Filtering in an Efficient Order As you saw earlier in the hypothetical milk example, we typically filter, then map, and then reduce, in that order. In the same way that we would not want to call every store on Earth, preferring instead to narrow down the list of potential stores first, we filter images first in our workflow in Earth Engine. In addition, you may have noticed that the ordering of the filters within the filtering stage also mattered in the milk example. This is also true in Earth Engine. For problems with a non-global spatial component in which `filterBounds` is to be used, it is most efficient to do that spatial filtering first.

In the code below, you will see that you can "chain" the filter commands, which are then executed from left to right. Below, we chain the filters in the same order as you specified above. Note that, it gives an `ImageCollection` of the same size as when you applied the filters one at a time.

```
var chainedFilteredSet = imgCol.filterDate(startDate, endDate)
    .filterBounds(Map.getCenter())
    .filterMetadata('CLOUD_COVER', 'less_than', 50);
print('Chained: Less than 50% clouds in this area, 2000-2010',
    chainedFilteredSet.size());
```

In the code below, we chain the filters in a more efficient order, implementing `filterBounds` first. This, too, gives an `ImageCollection` of the same size as when you applied the filters in the less efficient order, whether the filters were chained or not.

```
var efficientFilteredSet = imgCol.filterBounds(Map.getCenter())
    .filterDate(startDate, endDate)
    .filterMetadata('CLOUD_COVER', 'less_than', 50);
print('Efficient filtering: Less than 50% clouds in this area,
2000-2010',
    efficientFilteredSet.size());
```

Each of the two chained sets of operations will give the same result as before for the number of images. While the second order is more efficient, both approaches are likely to return the answer to the Code Editor at roughly the same time for this very small example. The order of operations is most important in larger problems in which you might be challenged to manage memory carefully. As in the milk example in which you narrowed geographically first, it is good practice in Earth Engine to order the filters with the `filterBounds` first, followed by metadata filters in order of decreasing specificity.

Code Checkpoint F40a. The book's repository contains a script that shows what your code should look like at this point.

Now, with an efficiently filtered collection that satisfies our chosen criteria, we will next explore the second stage: executing a function for all of the images in the set.

12.2.2 Section 2: Mapping over Image Collections in Earth Engine

In Chap. 9, we calculated the Enhanced Vegetation Index (EVI) in very small steps to illustrate band arithmetic on satellite images. In that chapter, code was called once, on a single image. What if we wanted to compute the EVI in the same way for every image of an entire `ImageCollection`? Here, we use the key tool for the second part of the workflow in Earth Engine, a `.map` command (Fig. 12.1). This is roughly analogous to the step of making phone calls in the milk example that began this chapter, in which you took a list of store names and transformed it through effort into a list of milk prices.

Before beginning to code the EVI functionality, it is worth noting that the word "map" is encountered in multiple settings during cloud-based remote sensing, and it is important to be able to distinguish the uses. A good way to think of it is that "map" can act as a verb or as a noun in Earth Engine. There are two uses of "map" as a noun. We might refer casually to "the map" or more precisely to "the **Map** panel"; these terms refer to the place where the images are shown in the code interface. A second way "map" is used as a noun which is to refer to an Earth Engine object, which has functions that can be called on it. Examples of this are the familiar `Map.addLayer` and `Map.setCenter`. Where that use of the word is intended, it will be shown in purple text and capitalized in the Code Editor. What we are discussing here is the use of `.map` as a verb, representing the idea of performing a set of actions repeatedly on a set. This is typically referred to as "mapping over the set".

To map a given set of operations efficiently over an entire `ImageCollection`, the processing needs to be set up in a particular way. Users familiar with other programming languages might expect to see "loop" code to do this, but the processing is not done exactly that way in Earth Engine. Instead, we will create a function and then map it over the `ImageCollection`. To begin, envision creating a function that takes exactly one parameter, an `ee.Image`. The function is then designed to perform a specified set of operations on the input `ee.Image` and then, importantly, returns an `ee.Image` as the last step of the function. When we map that function over an `ImageCollection`, as we will illustrate below, the effect is that we begin with an `ImageCollection`, do operations to each image, and receive a processed `ImageCollection` as the output.

What kinds of functions could we create? For example, you could imagine a function taking an image and returning an image whose pixels have the value 1 where the value of a given band was lower than a certain threshold and 0 otherwise. The effect of mapping this function would be an entire `ImageCollection` of images with zeroes and ones representing the results of that test on each image. Or, you could imagine a function computing a complex self-defined index and sending back an image of that index calculated in each pixel. Here, we will create a function to compute the EVI for any input Landsat 5 image and return the one-band image for which the index is computed for each pixel. Copy and paste the function definition below into the Code Editor, adding it to the end of the script from the previous section.

```
var makeLandsat5EVI = function(oneL5Image) {
    // compute the EVI for any Landsat 5 image. Note it's
specific to
    // Landsat 5 images due to the band numbers. Don't run this
exact
    // function for images from sensors other than Landsat 5.

    // Extract the bands and divide by 1e4 to account for
scaling done.
    var nirScaled = oneL5Image.select('SR_B4').divide(10000);
    var redScaled = oneL5Image.select('SR_B3').divide(10000);
    var blueScaled = oneL5Image.select('SR_B1').divide(10000);

    // Calculate the numerator, note that order goes from left
to right.
    var numeratorEVI = (nirScaled.subtract(redScaled)).multiply(
        2.5);

    // Calculate the denominator
    var denomClause1 = redScaled.multiply(6);
    var denomClause2 = blueScaled.multiply(7.5);
    var denominatorEVI = nirScaled.add(denomClause1).subtract(
        denomClause2).add(1);

    // Calculate EVI and name it.
    var landsat5EVI =
numeratorEVI.divide(denominatorEVI).rename(
        'EVI');
    return (landsat5EVI);
};
```

It is worth emphasizing that, in general, band names are specific to each `ImageCollection`. As a result, if that function was run on an image without the band '`SR_B4`', for example, the function call would fail. Here, we have emphasized in the function's name that it is specifically for creating EVI for Landsat 5.

The function `makeLandsat5EVI` is built to receive a single image, select the proper bands for calculating EVI, make the calculation, and return a one-banded image. If we had the name of each image comprising our `ImageCollection`, we could enter the names into the Code Editor and call the function one at a time for each, assembling the images into variables and then combining them into an `ImageCollection`. This would be very tedious and highly prone to mistakes: lists of items might get mistyped, an image might be missed, etc. Instead, as mentioned above, we will use `.map`. With the code below, let us print the information about the cloud-filtered collection and display it, execute the `.map` command, and explore the resulting `ImageCollection`.

```
var L5EVIimages = efficientFilteredSet.map(makeLandsat5EVI);
print('Verifying that the .map gives back the same number of
images: ',
    L5EVIimages.size());
print(L5EVIimages);

Map.addLayer(L5EVIimages, {}, 'L5EVIimages', 1, 1);
```

After entering and executing this code, you will see a grayscale image. If you look closely at the edges of the image, you might spot other images drawn behind it in a way that looks somewhat like a stack of papers on a table. This is the drawing of the `ImageCollection` made from the `makeLandsat5EVI` function. You can select the **Inspector** panel and click on one of the grayscale pixels to view the values of the entire `ImageCollection`. After clicking on a pixel, look for the `Series` tag by opening and closing the list of items. When you open that tag, you will see a chart of the EVI values at that pixel, created by mapping the `makeLandsat5EVI` function over the filtered `ImageCollection`.

Code Checkpoint F40b. The book's repository contains a script that shows what your code should look like at this point.

12.2.3 Section 3: Reducing an Image Collection

The third part of the *filter, map, reduce* paradigm is "reducing" values in an `ImageCollection` to extract meaningful values (Fig. 12.1). In the milk example, we reduced a large list of milk prices to find the minimum value. The Earth Engine API provides a large set of reducers for reducing a set of values to a summary statistic.

Here, you can think of each location, after the calculation of EVI has been executed through the `.map` command, as having a list of EVI values on it. Each pixel contains a potentially very large set of EVI values; the stack might be 15 items high in one location and perhaps 200, 2000, or 200,000 items high in another location, especially if a looser set of filters had been used.

The code below computes the mean value, at every pixel, of the `ImageCollection` `L5EVIimages` created above. Add it at the bottom of your code.

```
var L5EVImean = L5EVIimages.reduce(ee.Reducer.mean());
print(L5EVImean);
Map.addLayer(L5EVImean, {
    min: -1,
    max: 2,
    palette: ['red', 'white', 'green']
}, 'Mean EVI');
```

Using the same principle, the code below computes and draws the median value of the ImageCollection in every pixel.

```
var L5EVImedian = L5EVIimages.reduce(ee.Reducer.median());
print(L5EVImedian);
Map.addLayer(L5EVImedian, {
    min: -1,
    max: 2,
    palette: ['red', 'white', 'green']
}, 'Median EVI');
```

There are many more reducers that work with an ImageCollection to produce a wide range of summary statistics. Reducers are not limited to returning only one item from the reduction. The minMax reducer, for example, returns a two-band image for each band it is given, one for the minimum and one for the maximum.

The reducers described here treat each pixel independently. In subsequent chapters in Part IV, you will see other kinds of reducers—for example, ones that summarize the characteristics in the neighborhood surrounding each pixel.

Code Checkpoint F40c. The book's repository contains a script that shows what your code should look like at this point.

12.3 Synthesis

Assignment 1. Compare the mean and median images produced in Sect. 3 (Fig. 12.2). In what ways do they look different, and in what ways do they look alike? To understand how they work, pick a pixel and inspect the EVI values computed. In your opinion, which is a better representative of the dataset?

Assignment 2. Adjust the filters to filter a different proportion of clouds or a different date range. What effects do these changes have on the number of images and the look of the reductions made from them?

Assignment 3. Explore the ee.Filter options in the API documentation, and select a different filter that might be of interest. Filter images using it, and comment on the number of images and the reductions made from them.

Assignment 4. Change the EVI function so that it returns the original image with the EVI band appended by replacing the **return** statement with this: **return** (oneL5Image.addBands(landsat5EVI)).

What does the median reducer return in that case? Some EVI values are 0. What are the conditions in which this occurs?

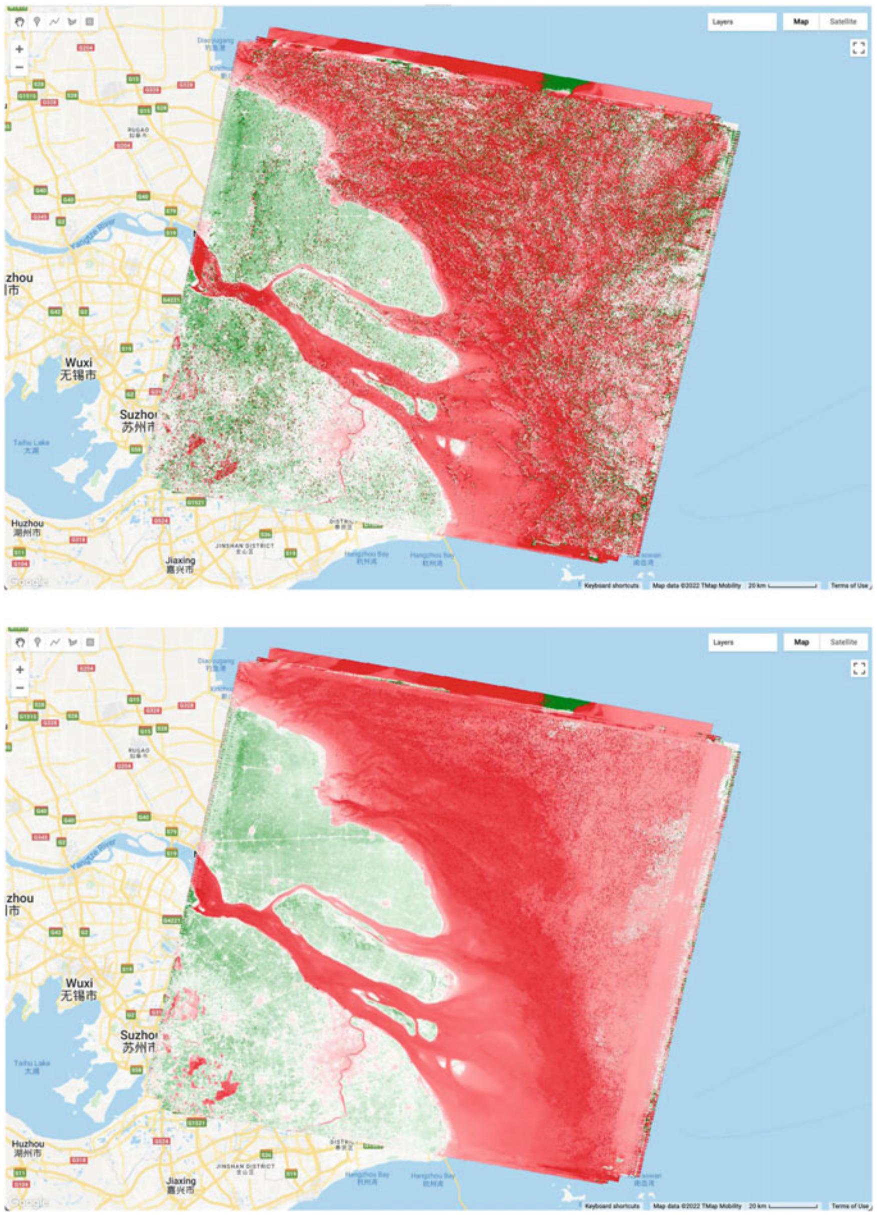


Fig. 12.2 Effects of two reducers on mapped EVI values in a filtered ImageCollection: mean image (above) and median image (below)

Assignment 5. Choose a date and location that is important to you (e.g., your birthday and your place of birth). Filter Landsat imagery to get all the low-cloud imageries at your location within 6 months of the date. Then, reduce the `ImageCollection` to find the median EVI. Describe the image and how representative of the full range of values it is, in your opinion.

12.4 Conclusion

In this chapter, you learned about the paradigm of *filter, map*, *reduce.* You learned how to use these tools to sift through, operate on, and summarize a large set of images to suit your purposes. Using the *Filter* functionality, you learned how to take a large `ImageCollection` and filter away images that do not meet your criteria, retaining only those images that match a given set of characteristics. Using the *Map* functionality, you learned how to apply a function to each image in an `ImageCollection`, treating each image one at a time and executing a requested set of operations on each. Using the *Reduce* functionality, you learned how to summarize the elements of an `ImageCollection`, extracting summary values of interest. In the subsequent chapters of Part IV, you will encounter these concepts repeatedly, manipulating image collections according to your project needs using the building blocks seen here. By building on what you have done in this chapter, you will grow in your ability to do sophisticated projects in Earth Engine.

Exploring Image Collections

13

Gennadii Donchyts

Overview
This chapter teaches how to explore image collections, including their spatiotemporal extent, resolution, and values stored in images and image properties. You will learn how to map and inspect image collections using maps, charts, and interactive tools and how to compute different statistics of values stored in image collections using reducers.

Learning Outcomes

- Inspecting the spatiotemporal extent and resolution of image collections by mapping image geometry and plotting image time properties.
- Exploring properties of images stored in an `ImageCollection` by plotting charts and deriving statistics.
- Filtering image collections by using stored or computed image properties.
- Exploring the distribution of values stored in image pixels of an `ImageCollection` through percentile reducers.

Assumes you know how to:

- Import images and image collections, filter, and visualize (Part 1).
- Perform basic image analysis: select bands, compute indices, create masks (Part 2).
- Summarize an `ImageCollection` with reducers (Chap. 12).

G. Donchyts (✉)
Google, Mountain View, CA, USA
e-mail: dgena@google.com

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_13

13.1 Practicum

In the previous chapter (Chap. 12), the *filter, map, reduce* paradigm was introduced. The main goal of this chapter is to demonstrate some of the ways that those concepts can be used within Earth Engine to better understand the variability of values stored in image collections. Section 13.1.1 demonstrates how time-dependent values stored in the images of an ImageCollection can be inspected using the Code Editor user interface after filtering them to a limited spatiotemporal range (i.e., geometry and time ranges). Section 13.1.2 shows how the extent of images, as well as basic statistics, such as the number of observations, can be visualized to better understand the spatiotemporal extent of image collections. Then, Sects. 13.1.3 and 13.1.4 demonstrate how simple reducers such as mean and median and more advanced reducers such as percentiles can be used to better understand how the values of a filtered ImageCollection are distributed.

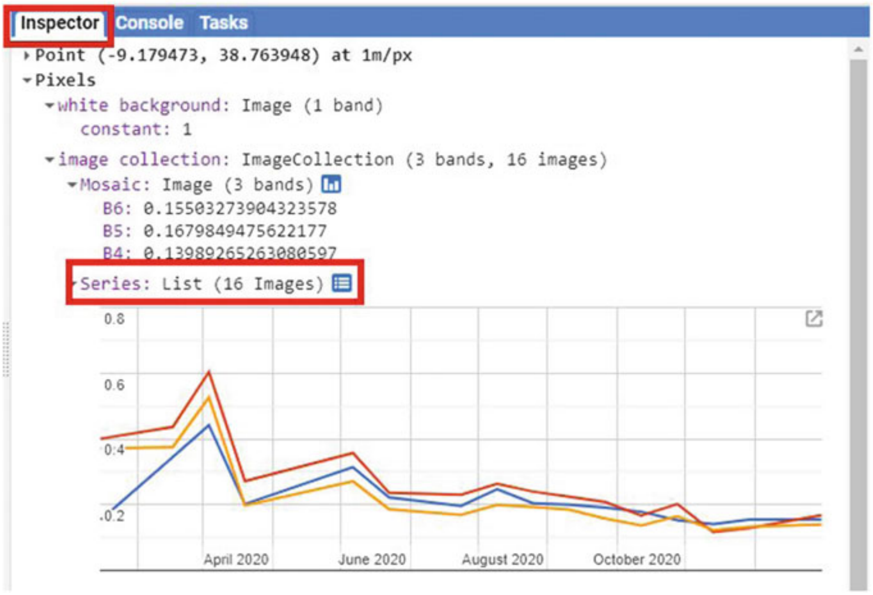
13.1.1 Section 1: Filtering and Inspecting an Image Collection

We will focus on the area in and surrounding Lisbon, Portugal. Below, we will define a point, lisbonPoint, located in the city; access the very large Landsat ImageCollection and limit it to the year 2020 and to the images that contain Lisbon; and select bands 6, 5, and 4 from each of the images in the resulting filtered ImageCollection.

```
// Define a region of interest as a point in Lisbon, Portugal.
var lisbonPoint = ee.Geometry.Point(-9.179473, 38.763948);

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

// filter the large ImageCollection to be just images from 2020
// around Lisbon. From each image, select true-color bands to
draw
var filteredIC = ee.ImageCollection('LANDSAT/LC08/C02/T1_TOA')
    .filterDate('2020-01-01', '2021-01-01')
    .filterBounds(lisbonPoint)
    .select(['B6', 'B5', 'B4']);

// Add the filtered ImageCollection so that we can inspect
values
// via the Inspector tool
Map.addLayer(filteredIC, {}, 'TOA image collection');
```

The three selected bands (which correspond to SWIR1, NIR, and Red) display a false-color image that accentuates differences between different land covers (e.g., concrete, vegetation) in Lisbon. With the **Inspector** tab highlighted (Fig. 13.1), clicking on a point will bring up the values of bands 6, 5, and 4 from each of the images. If you open the **Series** option, you will see the values through time. For the specified point and for all other points in Lisbon (since they are all enclosed in the same Landsat scene), there are 16 images gathered in 2020. By following one of the graphed lines (in blue, yellow, or red) with your finger, you should be able to count that many distinct values. Moving the mouse along the lines will show the specific values and the image dates.

We can also show this kind of chart automatically by making use of the `ui.Chart` function of the Earth Engine API. The following code snippet should result in the same chart as we could observe in the **Inspector** tab, assuming that the same pixel is clicked.



```
▸Point (-9.179473, 38.763948) at 1m/px
▾Pixels
  ▾white background: Image (1 band)
     constant: 1
  ▾image collection: ImageCollection (3 bands, 16 images)
    ▾Mosaic: Image (3 bands)
       B6: 0.15503273904323578
       B5: 0.1679849475622177
       B4: 0.13989265263080597
    ▾Series: List (16 Images)
```

Fig. 13.1 Inspect values in an `ImageCollection` at a selected point by making use of the **Inspector** tool in the Code Editor

```
// Construct a chart using values queried from image
collection.
var chart = ui.Chart.image.series({
    imageCollection: filteredIC,
    region: lisbonPoint,
    reducer: ee.Reducer.first(),
    scale: 10
});

// Show the chart in the Console.
print(chart);
```

Code Checkpoint F41a. The book's repository contains a script that shows what your code should look like at this point.

13.1.2 Section 2: How Many Images Are There, Everywhere on Earth?

Suppose we are interested to find out how many valid observations we have at every map pixel on Earth for a given `ImageCollection`. This enormously computationally demanding task is surprisingly easy to do in Earth Engine. The API provides a set of *reducer* functions to summarize values to a single number in each pixel, as described in Chap. 12. We can apply this reducer, `count`, to our filtered `ImageCollection` with the code below. We will return to the same dataset and filter for 2020, but without the geographic limitation. This will assemble images from all over the world and then count the number of images in each pixel. The following code does that count and adds the resulting image to the map with a predefined red/yellow/green color palette stretched between values 0 and 50. Continue pasting the code below into the same script.

```
// compute and show the number of observations in an image
collection
var count = ee.ImageCollection('LANDSAT/LC08/C02/T1_TOA')
    .filterDate('2020-01-01', '2021-01-01')
    .select(['B6'])
    .count();

// add white background and switch to HYBRID basemap
Map.addLayer(ee.Image(1), {
    palette: ['white']
}, 'white', true, 0.5);
Map.setOptions('HYBRID');

// show image count
Map.addLayer(count, {
    min: 0,
    max: 50,
    palette: ['d7191c', 'fdae61', 'ffffbf', 'a6d96a',
        '1a9641']
}, 'landsat 8 image count (2020)');

// Center the map at that point.
Map.centerObject(lisbonPoint, 5);
```

Run the command and zoom out. If the count of images over the entire Earth is viewed, the resulting map should look like Fig. 13.2. The created map data may take a few minutes to fully load in.

Note the checkered pattern, somewhat reminiscent of a Mondrian painting. To understand why the image looks this way, it is useful to consider the overlapping image footprints. As Landsat passes over, each image is wide enough to produce substantial “sidelap” with the images from the adjacent paths, which are collected at different dates according to the satellite’s orbit schedule. In the north–south direction, there is also some overlap to ensure that there are no gaps in the data. Because these are served as distinct images and stored distinctly in Earth Engine, you will find that there can be two images from the same day with the same value for points in these overlap areas. Depending on the purposes of a study, you might find a way to ignore the duplicate pixel values during the analysis process.

You might have noticed that we summarized a single band from the original `ImageCollection` to ensure that the resulting image would give a single count in each pixel. The `count` reducer operates on every band passed to it. Since every image has the same number of bands, passing an `ImageCollection` of all seven Landsat bands to the `count` reducer would have returned seven identical values of 16 for every point. To limit any confusion from seeing the same number seven times, we selected one of the bands from each image in the collection. In your own work, you might want to use a different reducer, such as a `median`

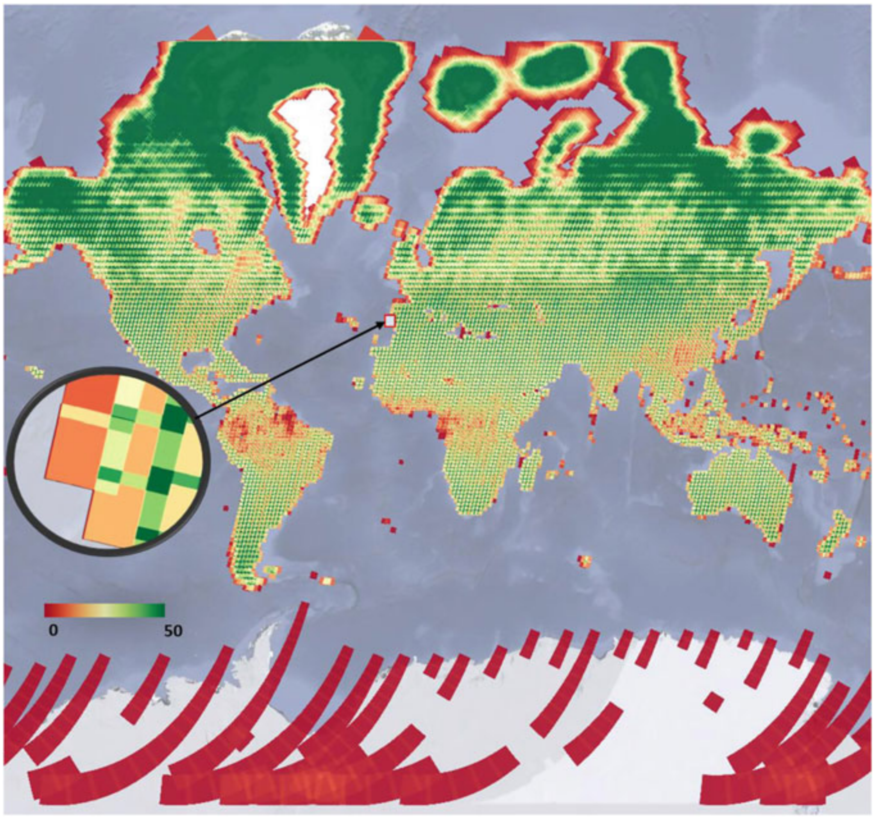


Fig. 13.2 Number of Landsat 8 images acquired during 2020

operation, that would give different, useful answers for each band. A few of these reducers are described below.

Code Checkpoint F41b. The book's repository contains a script that shows what your code should look like at this point.

13.1.3 Section 3: Reducing Image Collections to Understand Band Values

As we have seen, you could click at any point on Earth's surface and see both the number of Landsat images recorded there in 2020 and the values of any image in any band through time. This is impressive and perhaps mind-bending, given the enormous amount of data in play. In this section and the next, we will explore two ways to summarize the numerical values of the bands—one straightforward way and one more complex but highly powerful way to see what information is contained in image collections.

First, we will make a new layer that represents the mean value of each band in every pixel across every image from 2020 for the filtered set, add this layer to the layer set, and explore again with the **Inspector**. The previous section's `count` reducer was called directly using a sort of simple shorthand, that could be done similarly here by calling `mean` on the assembled bands. In this example, we will use the reducer to get the mean using the more general `reduce` call. Continue pasting the code below into the same script.

```
// Zoom to an informative scale for the code that follows.
Map.centerObject(lisbonPoint, 10);

// Add a mean composite image.
var meanFilteredIC = filteredIC.reduce(ee.Reducer.mean());
Map.addLayer(meanFilteredIC, {},
    'Mean values within image collection');
```

Now, let us look at the median value for each band among all the values gathered in 2020. Using the code below, calculate the median and explore the image with the **Inspector**. Compare this image briefly to the mean image by eye and by clicking in a few pixels in the **Inspector**. They should have different values, but in most places they will look very similar.

```
// Add a median composite image.
var medianFilteredIC = filteredIC.reduce(ee.Reducer.median());
Map.addLayer(medianFilteredIC, {},
    'Median values within image collection');
```

There is a wide range of reducers available in Earth Engine. If you are curious about which reducers can be used to summarize band values across a collection of images, use the **Docs** tab in the Code Editor to list all reducers and look for those beginning with `ee.Reducer`.

Code Checkpoint F41c. The book's repository contains a script that shows what your code should look like at this point.

13.1.4 Section 4: Compute Multiple Percentile Images for an Image Collection

One particularly useful reducer that can help you better understand the variability of values in image collections is `ee.Reducer.percentile`. The *n*th percentile gives the value that is the *n*th largest in a set. In this context, you can imagine accessing all of the values for a given band in a given `ImageCollection` for a given pixel and sorting them. The 30th percentile, for example, is the value 30% of the way along the list from smallest to largest. This provides an easy way to explore the variability of the values in image collections by computing a cumulative density function of values on a per-pixel basis. The following code shows how we can calculate a single 30th percentile on a per-pixel and per-band basis for our Landsat 8 `ImageCollection`. Continue pasting the code below into the same script.

```
// compute a single 30% percentile
var p30 = filteredIC.reduce(ee.Reducer.percentile([30]));

Map.addLayer(p30, {
    min: 0.05,
    max: 0.35
}, '30%');
```

We can see that the resulting composite image (Fig. 4.1.3) has almost no cloudy pixels present for this area. This happens because cloudy pixels usually have higher reflectance values. At the lowest end of the values, other unwanted effects like cloud or hill shadows typically have very low-reflectance values. This is why this 30th percentile composite image looks so much cleaner than the mean composite image (`meanFilteredIC`) calculated earlier. Note that, the reducers operate per-pixel: adjacent pixels are drawn from different images. This means that one pixel's value could be taken from an image from one date, and the adjacent pixel's value drawn from an entirely different period. Although, like the mean and median images, percentile images such as that seen in Fig. 13.3 never existed on a single day, composite images allow us to view Earth's surface without the noise that can make analysis difficult.

We can explore the range of values in an entire `ImageCollection` by viewing a series of increasingly bright percentile images, as shown in Fig. 13.4. Paste and run the following code.

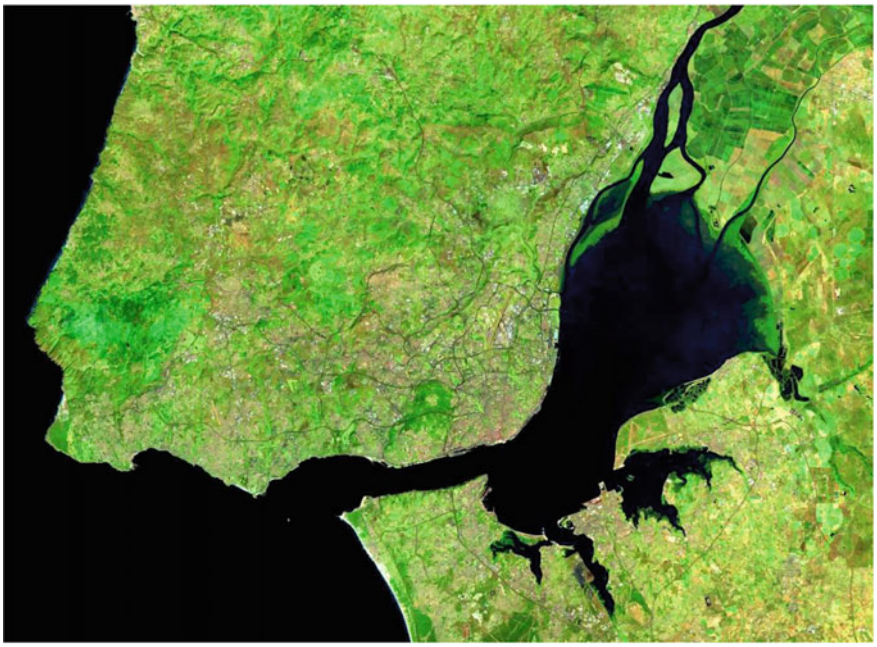


Fig. 13.3 Landsat 8 TOA reflectance 30th percentile image computed for `ImageCollection` with images acquired during 2020

```
var percentiles = [0, 10, 20, 30, 40, 50, 60, 70, 80];

// let's compute percentile images and add them as separate
layers
percentiles.map(function(p) {
    var image =
filteredIC.reduce(ee.Reducer.percentile([p]));
    Map.addLayer(image, {
        min: 0.05,
        max: 0.35
    }, p + '%');
});
```

Note that, the code adds every percentile image as a separate map layer, so you need to go to the **Layers** control and show/hide different layers to explore differences. Here, we can see that low-percentile composite images depict darker, low-reflectance land features, such as water and cloud or hill shadows, while higher-percentile composite images (> 70% in our example) depict clouds and any other atmospheric or land effects corresponding to bright reflectance values.

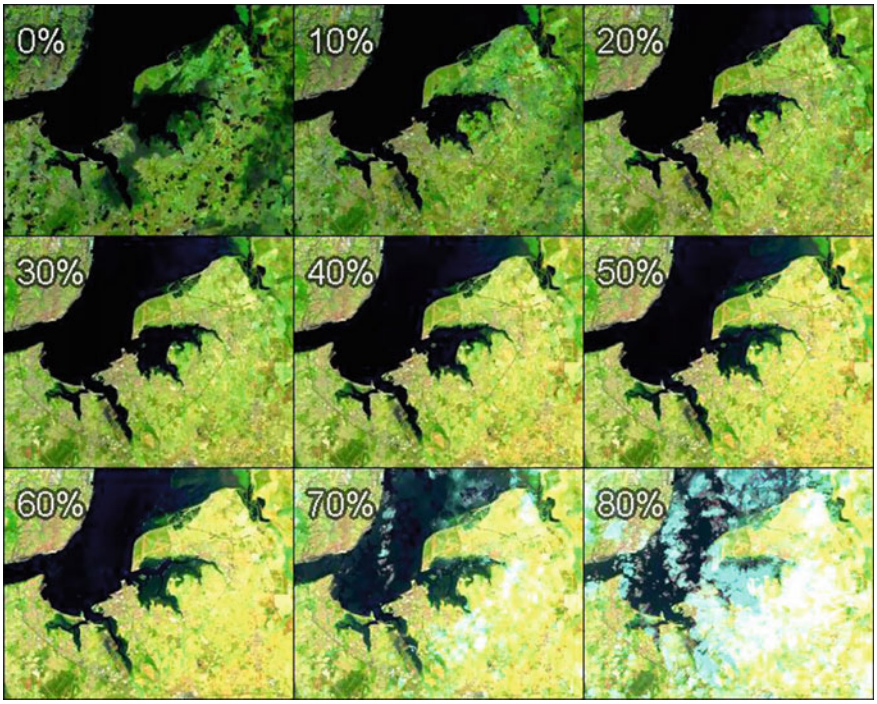


Fig. 13.4 Landsat 8 TOA reflectance percentile composite images

Earth Engine provides a very rich API, allowing users to explore image collections to better understand the extent and variability of data in space, time, and across bands, as well as tools to analyze values stored in image collections in a frequency domain. Exploring these values in different forms should be the first step of any study before developing data analysis algorithms.

Code Checkpoint F41d. The book's repository contains a script that shows what your code should look like at this point.

13.2 Synthesis

In the example above, the 30th percentile composite image would be useful for typical studies that need cloud-free data for analysis. The "best" composite to use, however, will depend on the goal of a study, the characteristics of the given dataset, and the location being viewed. You can imagine choosing different percentile composite values if exploring image collections over the Sahara Desert or over Congo, where cloud frequency would vary substantially (Wilson and Jetz 2016).

Assignment 1. Noting that your own interpretation of what constitutes a good composite is subjective, create a series of composites of a different location, or perhaps a pair of locations, for a given set of dates.

Assignment 2. Filter to create a relevant dataset—for example, for Landsat 8 or Sentinel-2 over an agricultural growing season. Create percentile composites for a given location. Which image composite is the most satisfying, and what type of project do you have in mind when giving that response?

Assignment 3. Do you think it is possible to generalize about the relationship between the time window of an `ImageCollection` and the percentile value that will be the most useful for a given project, or will every region need to be inspected separately?

13.3 Conclusion

In this chapter, you have learned different ways to explore image collections using Earth Engine in addition to looking at individual images. You have learned that image collections in Earth Engine may have global footprints as well as images with a smaller, local footprint, and how to visualize the number of images in a given filtered `ImageCollection`. You have learned how to explore the temporal and spatial extent of images stored in image collections and how to quickly examine the variability of values in these image collections by computing simple statistics like mean or median, as well as how to use a percentile reducer to better understand this variability.

Reference

Wilson AM, Jetz W (2016) Remotely sensed high-resolution global cloud dynamics for predicting ecosystem and biodiversity distributions. PLoS Biol 14:e1002415. https://doi.org/10.1371/journal.pbio.1002415

Aggregating Images for Time Series 14

Ujaval Gandhi

Overview

Many remote sensing datasets consist of repeated observations over time. The interval between observations can vary widely. The Global Precipitation Measurement dataset, for example, produces observations of rain and snow worldwide every three hours. The Climate Hazards Group InfraRed Precipitation with Station (CHIRPS) project produces a gridded global dataset at the daily level and also for each five-day period (Funk et al. 2015). The Landsat 8 mission produces a new scene of each location on Earth every 16 days. With its constellation of two satellites, the Sentinel-2 mission images every location every five days.

Many applications, however, require computing aggregations of data at time intervals different from those at which the datasets were produced. For example, for determining rainfall anomalies, it is useful to compare monthly rainfall against a long-period monthly average.

While individual scenes are informative, many days are cloudy, and it is useful to build a robust cloud-free time series for many applications. Producing less cloudy or even cloud-free composites can be done by aggregating data to form monthly, seasonal, or yearly composites built from individual scenes. For example, if you are interested in detecting long-term changes in an urban landscape, creating yearly median composites can enable you to detect change patterns across long time intervals with less worry about day-to-day noise.

This chapter will cover the techniques for aggregating individual images from a time series at a chosen interval. We will take the CHIRPS time series of rainfall for one year and aggregate it to create a monthly rainfall time series.

U. Gandhi (✉)
Spatial Thoughts LLP, FF105 Aaradhya, Gala Gymkhana Road, Bopal, Ahmedabad, 380058, India
e-mail: ujaval@spatialthoughts.com

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_14

Learning Outcomes

- Using the Earth Engine API to work with dates.
- Aggregating values from an `ImageCollection` to calculate monthly, seasonal, or yearly images.
- Plotting the aggregated time series at a given location.

Assumes you know how to:

- Import images and image collections, filter, and visualize (Part 1).
- Create a graph using `ui.Chart` (Chap. 4).
- Write a function and `map` it over an `ImageCollection` (Chap. 12).
- Summarize an `ImageCollection` with reducers (Chaps. 12 and 13).
- Inspect an Image and an `ImageCollection`, as well as their properties (Chap. 13).

14.1 Introduction to Theory

CHIRPS is a high-resolution global gridded rainfall dataset that combines satellite-measured precipitation with ground station data in a consistent, long time-series dataset. The data are provided by the University of California, Santa Barbara, and are available from 1981 to the present. This dataset is extremely useful in drought monitoring and assessing global environmental change over land. The satellite data are calibrated with ground station observations to create the final product.

In this exercise, we will work with the CHIRPS dataset using the *pentad*. A pentad represents the grouping of five days. There are six pentads in a calendar month, with five pentads of exactly five days each and one pentad with the remaining three to six days of the month. Pentads reset at the beginning of each month, and the first day of every month is the start of a new pentad. Values at a given pixel in the CHIRPS dataset represent the total precipitation in millimeters over the pentad.

14.2 Practicum

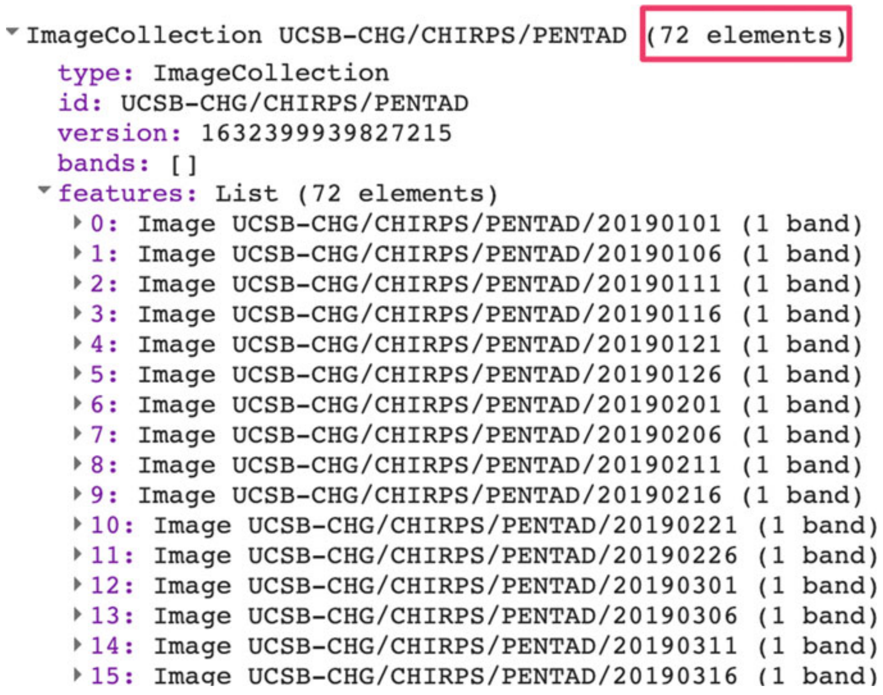
14.2.1 Section 1: Filtering an Image Collection

We will start by accessing the CHIRPS pentad collection and filtering it to create a time series for a single year.

```
var chirps = ee.ImageCollection('UCSB-CHG/CHIRPS/PENTAD');
var startDate = '2019-01-01';
var endDate = '2020-01-01';
var yearFiltered = chirps.filter(ee.Filter.date(startDate,
endDate));

print(yearFiltered, 'Date-filtered CHIRPS images');
```

The CHIRPS collection contains one image for every pentad. The filtered collection above is filtered to contain one year, which equates to 72 global images. If you expand the printed collection in the **Console**, you will be able to see the metadata for individual images; note that, their date stamps indicate that they are spaced evenly every five days (Fig. 14.1).

Each image's pixel values store the total precipitation during the pentad. Without aggregation to a period that matches other datasets, these layers are not very useful. For hydrological analysis, we typically need the total precipitation for each month or for a season. Let us aggregate this collection so that we have 12 images—one image per month, with pixel values that represent the total precipitation for that month.



```
▾ImageCollection UCSB-CHG/CHIRPS/PENTAD (72 elements)
   type: ImageCollection
   id: UCSB-CHG/CHIRPS/PENTAD
   version: 1632399939827215
   bands: []
  ▾features: List (72 elements)
    ▸0: Image UCSB-CHG/CHIRPS/PENTAD/20190101 (1 band)
    ▸1: Image UCSB-CHG/CHIRPS/PENTAD/20190106 (1 band)
    ▸2: Image UCSB-CHG/CHIRPS/PENTAD/20190111 (1 band)
    ▸3: Image UCSB-CHG/CHIRPS/PENTAD/20190116 (1 band)
    ▸4: Image UCSB-CHG/CHIRPS/PENTAD/20190121 (1 band)
    ▸5: Image UCSB-CHG/CHIRPS/PENTAD/20190126 (1 band)
    ▸6: Image UCSB-CHG/CHIRPS/PENTAD/20190201 (1 band)
    ▸7: Image UCSB-CHG/CHIRPS/PENTAD/20190206 (1 band)
    ▸8: Image UCSB-CHG/CHIRPS/PENTAD/20190211 (1 band)
    ▸9: Image UCSB-CHG/CHIRPS/PENTAD/20190216 (1 band)
    ▸10: Image UCSB-CHG/CHIRPS/PENTAD/20190221 (1 band)
    ▸11: Image UCSB-CHG/CHIRPS/PENTAD/20190226 (1 band)
    ▸12: Image UCSB-CHG/CHIRPS/PENTAD/20190301 (1 band)
    ▸13: Image UCSB-CHG/CHIRPS/PENTAD/20190306 (1 band)
    ▸14: Image UCSB-CHG/CHIRPS/PENTAD/20190311 (1 band)
    ▸15: Image UCSB-CHG/CHIRPS/PENTAD/20190316 (1 band)
```

Fig. 14.1 CHIRPS time series for one year

Code Checkpoint F42a. The book's repository contains a script that shows what your code should look like at this point.

14.2.2 Section 2: Working with Dates

To aggregate the time series, we need to learn how to create and manipulate dates programmatically. This section covers some functions from the `ee.Date` module that will be useful.

The Earth Engine API has a function called `ee.Date.fromYMD` that is designed to create a date object from `year`, `month`, and `day` values. The following code snippet shows how to define a variable containing the year value and create a date object from it. Paste the following code in a new script:

```
var chirps = ee.ImageCollection('UCSB-CHG/CHIRPS/PENTAD');
var year = 2019;
var startDate = ee.Date.fromYMD(year, 1, 1);
```

Now, let us determine how to create an end date in order to be able to specify a desired time interval. The preferred way to create a date relative to another date is using the `advance` function. It takes two parameters—a `delta` value and the `unit` of time—and returns a new date. The code below shows how to create a date one year in the future from a given date. Paste it into your script.

```
var endDate = startDate.advance(1, 'year');
```

Next, paste the code below to perform filtering of the CHIRPS data using these calculated dates. After running it, check that you had accurately set the dates by looking for the dates of the images inside the printed result.

```
var yearFiltered = chirps
    .filter(ee.Filter.date(startDate, endDate));
print(yearFiltered, 'Date-filtered CHIRPS images');
```

Another date function that is very commonly used across Earth Engine is `millis`. This function takes a date object and returns the number of milliseconds since the arbitrary reference date of the start of the year 1970: 1970-01-01T00:00:00Z. This is known as the "Unix Timestamp"; it is a standard way to convert dates to numbers and allows for easy comparison between dates with high precision. Earth Engine objects store the timestamps for images and features in special properties called system:time_start and system:time_end. Both of these properties need to be supplied with a number instead of dates, and the millis function can help you to do that. You can print the result of calling this function and check for yourself.

```
print(startDate, 'Start date');
print(endDate, 'End date');

print('Start date as timestamp', startDate.millis());
print('End date as timestamp', endDate.millis());
```

We will use the millis function in the next section when we need to set the system:time_start and system:time_end properties of the aggregated images.

Code Checkpoint F42b. The book's repository contains a script that shows what your code should look like at this point.

14.2.3 Section 3: Aggregating Images

Now, we can start aggregating the pentads into monthly sums. The process of aggregation has two fundamental steps. The first is to determine the beginning and ending dates of one time interval (in this case, one month), and the second is to sum up all of the values (in this case, the pentads) that fall within each interval. To begin, we can envision that the resulting series will contain 12 images. To prepare to create an image for each month, we create an ee.List of values from 1 to 12. We can use the ee.List.sequence function, as first presented in Chap. 1, to create the list of items of type ee.Number. Continuing with the script of the previous section, paste the following code:

```
// Aggregate this time series to compute monthly images.
// Create a list of months
var months = ee.List.sequence(1, 12);
```

Next, we write a function that takes a single month as the input and returns an aggregated image for that month. Given beginningMonth as an input parameter, we first create a start and end date for that month based on the year and month variables. Then, we filter the collection to find all images for that month. To create a monthly precipitation image, we apply ee.Reducer.sum to reduce the six pentad images for a month to a single image holding the summed value across the pentads. We also expressly set the timestamp properties system:time_start and system:time_end of the resulting summed image. We can also set year and month, which will help us to filter the resulting collection later.

```
// Write a function that takes a month number
// and returns a monthly image.
var createMonthlyImage = function(beginningMonth) {
    var startDate = ee.Date.fromYMD(year, beginningMonth, 1);
    var endDate = startDate.advance(1, 'month');
    var monthFiltered = yearFiltered
        .filter(ee.Filter.date(startDate, endDate));

    // Calculate total precipitation.
    var total = monthFiltered.reduce(ee.Reducer.sum());
    return total.set({
        'system:time_start': startDate.millis(),
        'system:time_end': endDate.millis(),
        'year': year,
        'month': beginningMonth
    });
};
```

We now have an ee.List containing items of type ee.Number from 1 to 12, with a function that can compute a monthly aggregated image for each month number. All that is left to do are to map the function over the list. As described in Chaps. 12 and 13, the map function passes over each image in the list and runs createMonthlyImage. The function first receives the number "1" and executes, returning an image to Earth Engine. Then, it runs on the number "2" and so on for all 12 numbers. The result is a list of monthly images for each month of the year.

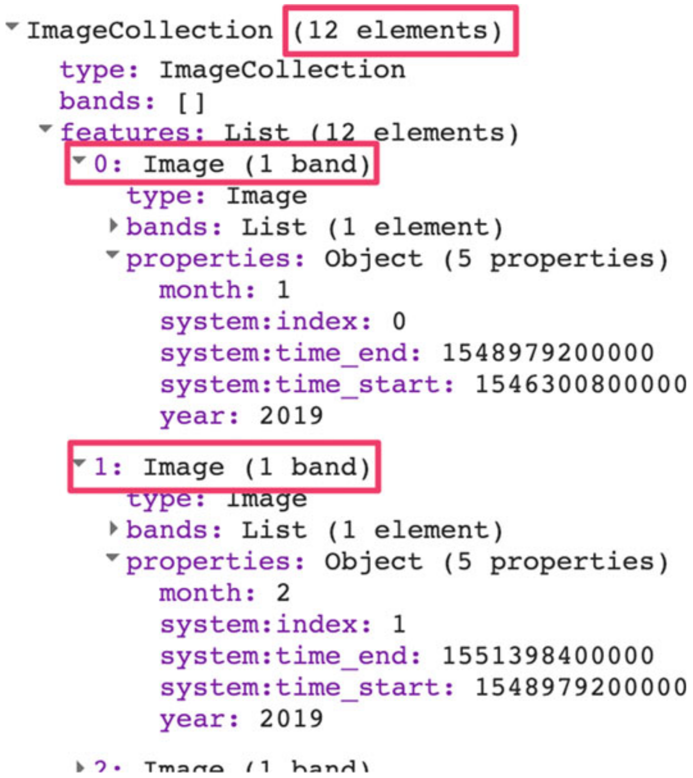
```
// map() the function on the list of months
// This creates a list with images for each month in the list
var monthlyImages = months.map(createMonthlyImage);
```

We can create an ImageCollection from this ee.List of images using the ee.ImageCollection.fromImages function.

```
// Create an ee.ImageCollection.
var monthlyCollection =
ee.ImageCollection.fromImages(monthlyImages);
print(monthlyCollection);
```

We have now successfully computed an aggregated collection from the source `ImageCollection` by filtering, mapping, and reducing, as described in Chaps. 12 and 13. Expand the printed collection in the **Console**, and you can verify that we now have 12 images in the newly created `ImageCollection` (Fig. 14.2).

Code Checkpoint F42c. The book's repository contains a script that shows what your code should look like at this point.



```
▼ImageCollection (12 elements)
   type: ImageCollection
   bands: []
  ▼features: List (12 elements)
    ▼0: Image (1 band)
       type: Image
      ▶bands: List (1 element)
      ▼properties: Object (5 properties)
         month: 1
         system:index: 0
         system:time_end: 1548979200000
         system:time_start: 1546300800000
         year: 2019

    ▼1: Image (1 band)
       type: Image
      ▶bands: List (1 element)
      ▼properties: Object (5 properties)
         month: 2
         system:index: 1
         system:time_end: 1551398400000
         system:time_start: 1548979200000
         year: 2019

    ▶2: Image (1 band)
```

Fig. 14.2 Aggregated time series

14.2.4 Section 4: Plotting Time Series

One useful application of gridded precipitation datasets is to analyze rainfall patterns. We can plot a time-series chart for a location using the newly computed time series. We can plot the pixel value at any given point or polygon. Here, we create a point geometry for a given coordinate. Continuing with the script of the previous section, paste the following code:

```
// Create a point with coordinates for the city of
Bengaluru, India.
var point = ee.Geometry.Point(77.5946, 12.9716);
```

Earth Engine comes with a built-in `ui.Chart.image.series` function that can plot time series. In addition to the `imageCollection` and `region` parameters, we need to supply a `scale` value. The CHIRPS data catalog page indicates that the resolution of the data is 5566 m, so we can use that as the scale. The resulting chart is printed in the **Console**.

```
var chart = ui.Chart.image.series({
    imageCollection: monthlyCollection,
    region: point,
    reducer: ee.Reducer.mean(),
    scale: 5566,
});
print(chart);
```

We can make the chart more informative by adding axis labels and a title. The `setOptions` function allows us to customize the chart using parameters from Google Charts. To customize the chart, paste the code below at the bottom of your script. The effect will be to see two charts in the editor: one with the old view of the data and one with the customized chart.

```
var chart = ui.Chart.image.series({
    imageCollection: monthlyCollection,
    region: point,
    reducer: ee.Reducer.mean(),
    scale: 5566
}).setOptions({
    lineWidth: 1,
    pointSize: 3,
    title: 'Monthly Rainfall at Bengaluru',
    vAxis: {
        title: 'Rainfall (mm)'
    },
    hAxis: {
        title: 'Month',
        gridlines: {
            count: 12
        }
    }
});
print(chart);
```

The customized chart (Fig. 14.3) shows the typical rainfall pattern in the city of Bengaluru, India. Bengaluru has a temperate climate, with pre-monsoon rains in April and May cooling down the city and a moderate monsoon season lasting from June to September.

Code Checkpoint F42d. The book's repository contains a script that shows what your code should look like at this point.

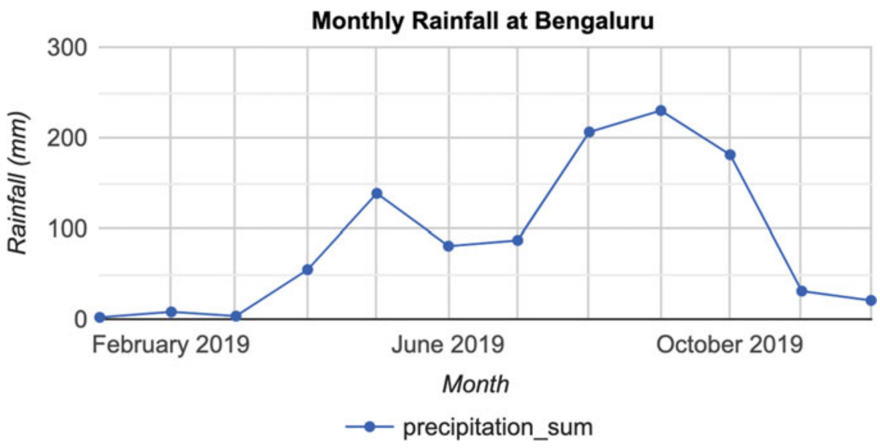


Fig. 14.3 Monthly rainfall chart

14.3 Synthesis

Assignment 1. The CHIRPS collection contains data for 40 years. Aggregate the same collection to yearly images and create a chart for annual precipitation from 1981 to 2021 at your chosen location.

Instead of creating a list of months and writing a function to create monthly images, we will create a list of years and write a function to create yearly images. The code snippet below will help you get started.

```
var chirps = ee.ImageCollection('UCSB-CHG/CHIRPS/PENTAD');

// Create a list of years
var years = ee.List.sequence(1981, 2021);

// Write a function that takes a year number
// and returns a yearly image
var createYearlyImage = function(beginningYear) {
    // Add your code
};

var yearlyImages = years.map(createYearlyImage);
var yearlyCollection =
ee.ImageCollection.fromImages(yearlyImages);
print(yearlyCollection);
```

14.4 Conclusion

In this chapter, you learned how to aggregate a collection to months and plot the resulting time series for a location. This chapter also introduced useful functions for working with the dates that will be used across many different applications. You also learned how to iterate over a list using the `map` function. The technique of mapping a function over a list or collection is essential for processing data. Mastering this technique will allow you to scale your analysis using the parallel computing capabilities of Earth Engine.

References

Funk C, Peterson P, Landsfeld M et al (2015) The climate hazards infrared precipitation with stations—a new environmental record for monitoring extremes. Sci Data 2:1–21. https://doi.org/10.1038/sdata.2015.66

Clouds and Image Compositing

15

Txomin Hermosilla, Saverio Francini, Andréa P. Nicolau, Michael A. Wulder, Joanne C. White, Nicholas C. Coops, and Gherardo Chirici

Overview
The purpose of this chapter is to provide necessary context and demonstrate different approaches for image composite generation when using data quality flags, using an initial example of removing cloud cover. We will examine different filtering options, demonstrate an approach for cloud masking, and provide additional opportunities for image composite development. Pixel selection for composite development can exclude unwanted pixels—such as those impacted by cloud, shadow, and smoke or haze—and can also preferentially select pixels based upon proximity to a target date or a preferred sensor type.

T. Hermosilla (✉) · M. A. Wulder · J. C. White
Canadian Forest Service, Victoria, Canada
e-mail: txomin.hermosillagomez@nrcan-rncan.gc.ca

M. A. Wulder
e-mail: mike.wulder@nrcan-rncan.gc.ca

J. C. White
e-mail: joanne.White@nrcan-rncan.gc.ca

S. Francini · G. Chirici
University of Florence, Florence, Italy
e-mail: saverio.francini@unifi.it

G. Chirici
e-mail: gherardo.chirici@unifi.it

A. P. Nicolau
Spatial Informatics Group, SERVIR-Amazonia, Pleasanton, USA
e-mail: apnicolau@sig-gis.com

N. C. Coops
University of British Columbia, Vancouver, Canada
e-mail: nicholas.coops@ubc.ca

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_15

Learning Outcomes

- Understanding and applying satellite-specific cloud mask functions.
- Incorporating images from different sensors.
- Using focal functions to fill in data gaps.

Assumes you know how to:

- Import images and image collections, filter, and visualize (Part 1).
- Perform basic image analysis: select bands, compute indices, create masks (Part 2).
- Use band scaling factors (Chap. 9).
- Perform pixel-based transformations (Chap. 9).
- Use neighborhood-based image transformations (Chap. 10).
- Write a function and `map` it over an `ImageCollection` (Chap. 12).
- Summarize an `ImageCollection` with reducers (Chaps. 12 and 13).

15.1 Introduction to Theory

Satellite remote sensing is an ideal source of data for monitoring large or remote regions. However, cloud cover is one of the most common limitations of optical sensors in providing continuous time series of data for surface mapping and monitoring. This is particularly relevant in tropical, polar, mountainous, and high-latitude areas, where clouds are often present. Many studies have addressed the extent to which cloudiness can restrict the monitoring of various regions (Zhu and Woodcock 2012, 2014; Eberhardt et al. 2016; Martins et al. 2018).

Clouds and cloud shadows reduce the view of optical sensors and completely block or obscure the spectral response from Earth's surface (Cao et al. 2020). Working with pixels that are cloud-contaminated can significantly influence the accuracy and information content of products derived from a variety of remote sensing activities, including land cover classification, vegetation modeling, and especially change detection, where unscreened clouds might be mapped as false changes (Braaten et al. 2015; Zhu et al. 2015). Thus, the information provided by cloud detection algorithms is critical to exclude clouds and cloud shadows from subsequent processing steps.

Historically, cloud detection algorithms derived the cloud information by considering a single date image and sun illumination geometry (Irish et al. 2006; Huang et al. 2010). In contrast, current, more accurate cloud detection algorithms are based on the analysis of Landsat time series (Zhu and Woodcock 2014; Zhu and Helmer 2018). Cloud detection algorithms inform on the presence of clouds, cloud shadows, and other atmospheric conditions (e.g., presence of snow). The presence and extent of cloud contamination within a pixel are currently provided with Landsat and Sentinel-2 images as ancillary data via quality flags at the pixel level. Additionally, quality flags inform on acquisition-related conditions, including radiometric saturation and terrain occlusion, which enables us to assess the usefulness and convenience of inclusion of each pixel in subsequent analyses. The quality flags are ideally suited to reduce users' manual supervision and maximize the automatic processing approaches.

Most automated algorithms (for classification or change detection, for example) work best on images free of clouds and cloud shadows, that cover the full area without spatial or spectral inconsistencies. Thus, the image representation over the study area should be seamless, containing as few data gaps as possible. Image compositing techniques are primarily used to reduce the impact of clouds and cloud shadows, as well as aerosol contamination, view angle effects, and data volumes (White et al. 2014). Compositing approaches typically rely on the outputs of cloud detection algorithms and quality flags to include or exclude pixels from the resulting composite products (Roy et al. 2010). Epochal image composites help overcome the limited availability of cloud-free imagery in some areas and are constructed by considering the pixels from all images acquired in a given period (e.g., season, year).

The information provided by the cloud masks and pixel flags guides the establishment of rules to rank the quality of the pixels based on the presence of and distance of clouds, cloud shadows, or atmospheric haze (Griffiths et al. 2013). Higher scores are assigned to pixels with more desirable conditions, based on the presence of clouds and other acquisition circumstances, such as acquisition date or sensor. Pixels with the highest scores are included in subsequent composite development. Image compositing approaches enable users to define the rules that are most appropriate for their particular information needs and study area to generate imagery covering large areas instead of being limited to the analysis of single scenes (Hermosilla et al. 2015; Loveland and Dwyer 2012). Moreover, generating image composites at regular intervals (e.g., annually) allows for the analysis of long temporal series over large areas, fulfilling a critical information need for monitoring programs.

15.2 Practicum

The general workflow to generate a cloud-free composite involves:

1. Defining your area of interest (AOI).
2. Filtering (`ee.Filter`) the satellite `ImageCollection` to desired parameters.
3. Applying a cloud mask.
4. Reducing (`ee.Reducer`) the collection to generate a composite.
5. Using the GEE-BAP application to generate annual best-available-pixel image composites by globally combining multiple Landsat sensors and images.

Additional steps may be necessary to improve the composite generated. These steps will be explained in the following sections.

15.2.1 Section 1: Cloud Filter and Cloud Mask

The first step is to define your AOI and center the map. The goal is to create a nationwide composite for the country of Colombia. We will use the Large-Scale International Boundary (2017) simplified dataset from the US Department of State (USDOS), which contains polygons for all countries of the world.

```
// ---------- Section 1 -----------------

// Define the AOI.
var country = ee.FeatureCollection('USDOS/LSIB_SIMPLE/2017')
    .filter(ee.Filter.equals('country_na', 'Colombia'));

// Center the Map. The second parameter is zoom level.
Map.centerObject(country, 5);
```

We will start creating a composite from the Landsat 8 collection. First, we define two time variables: `startDate` and `endDate`. Here, we will create a composite for the year 2019. Then, we will define a collection for the Landsat 8 Level 2, Collection 2, Tier 1 variable and filter it to our AOI and time period. We define and use a function to apply scaling factors to the Landsat 8 Collection 2 data.

```
// Define time variables.
var startDate = '2019-01-01';
var endDate = '2019-12-31';

// Load and filter the Landsat 8 collection.
var landsat8 = ee.ImageCollection('LANDSAT/LC08/C02/T1_L2')
    .filterBounds(country)
    .filterDate(startDate, endDate);

// Apply scaling factors.
function applyScaleFactors(image) {
    var opticalBands =
image.select('SR_B.').multiply(0.0000275).add(-0.2);
    var thermalBands =
image.select('ST_B.*').multiply(0.00341802)
        .add(149.0);
    return image.addBands(opticalBands, null, true)
        .addBands(thermalBands, null, true);
}

landsat8 = landsat8.map(applyScaleFactors);
```

To create a composite we will use the `median` function, which has the same effect as writing `reduce(ee.Reducer.median())` as seen in Chap. 12, to reduce our `ImageCollection` to a median composite. Add the resulting composite to the map using visualization parameters.

```
// Create composite.
var composite = landsat8.median().clip(country);

var visParams = {
    bands: ['SR_B4', 'SR_B3', 'SR_B2'],
    min: 0,
    max: 0.2
};
Map.addLayer(composite, visParams, 'L8 Composite');
```

The resulting composite (Fig. 15.1) has lots of clouds, especially in the western, mountainous regions of Colombia. In tropical regions, it is very challenging to generate a high-quality, cloud-free composite without first filtering images for cloud cover, even if our collection is constrained to only include images acquired during the dry season. Therefore, we will filter our collection by the CLOUD_COVER parameter to avoid cloudy images, starting with images that have less than 50% cloud cover.

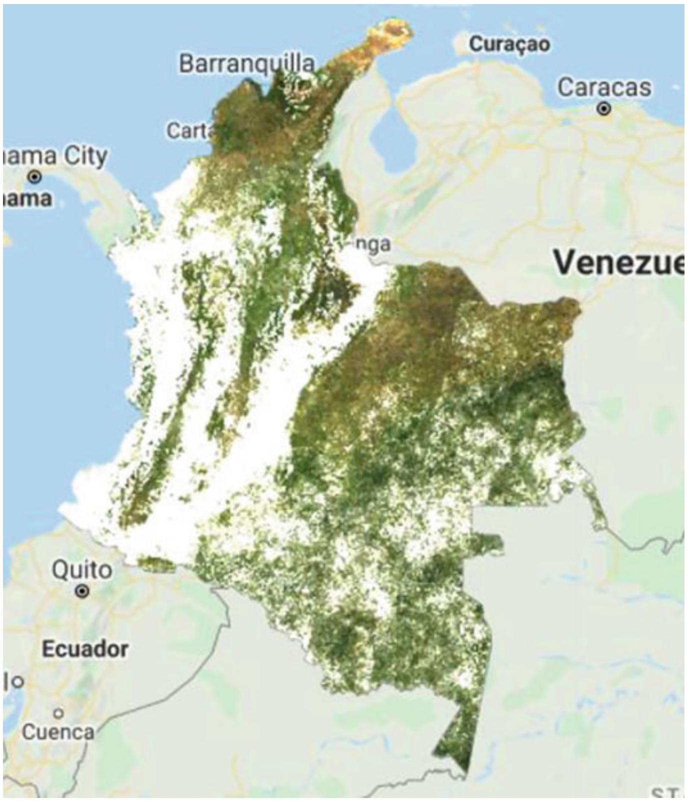


Fig. 15.1 Landsat 8 surface reflectance 2019 median composite of Colombia

```
// Filter by the CLOUD_COVER property.
var landsat8FiltClouds = landsat8
    .filterBounds(country)
    .filterDate(startDate, endDate)
    .filter(ee.Filter.lessThan('CLOUD_COVER', 50));

// Create a composite from the filtered imagery.
var compositeFiltClouds =
landsat8FiltClouds.median().clip(country);

Map.addLayer(compositeFiltClouds, visParams,
    'L8 Composite cloud filter');

// Print size of collections, for comparison.
print('Size landsat8 collection', landsat8.size());
print('Size landsat8FiltClouds collection',
landsat8FiltClouds.size());
```

This new composite (Fig. 15.2) looks slightly better than the previous one, but it is still very cloudy. Remember to turn off the first layer or adjust the transparency to only visualize this new composite. The code prints the size of these collections, using the `size` function, to see how many images were left out after we applied the cloud cover threshold. (There are 1201 images in the `landsat8` collection, compared to 493 in the `landsat8FiltClouds` collection—indicating that many images have cloud cover greater than or equal to 50%.)

Try adjusting the `CLOUD_COVER` threshold in the `landsat8FiltClouds` variable to different percentages and checking the results. For example, with the threshold set to 20% (Fig. 15.3), you can see that many parts of the country have image gaps. (Remember to turn off the first layer or adjust its transparency; you can also set the shown parameter in the `Map.addLayer` function to **false**, so the layer does not automatically load.) So, there is a trade-off between a stricter cloud cover threshold and data availability. Additionally, even with a cloud filter, some areas still present cloud cover.

This is due to persistent cloud cover in some regions of Colombia. However, a cloud mask can be applied to improve the results. The Landsat 8 Collection 2 contains a quality assessment (QA) band called `QA_PIXEL` that provides useful information on certain conditions within the data and allows users to apply per-pixel filters. Each pixel in the QA band contains unsigned integers that represent bit-packed combinations of surface, atmospheric, and sensor conditions.

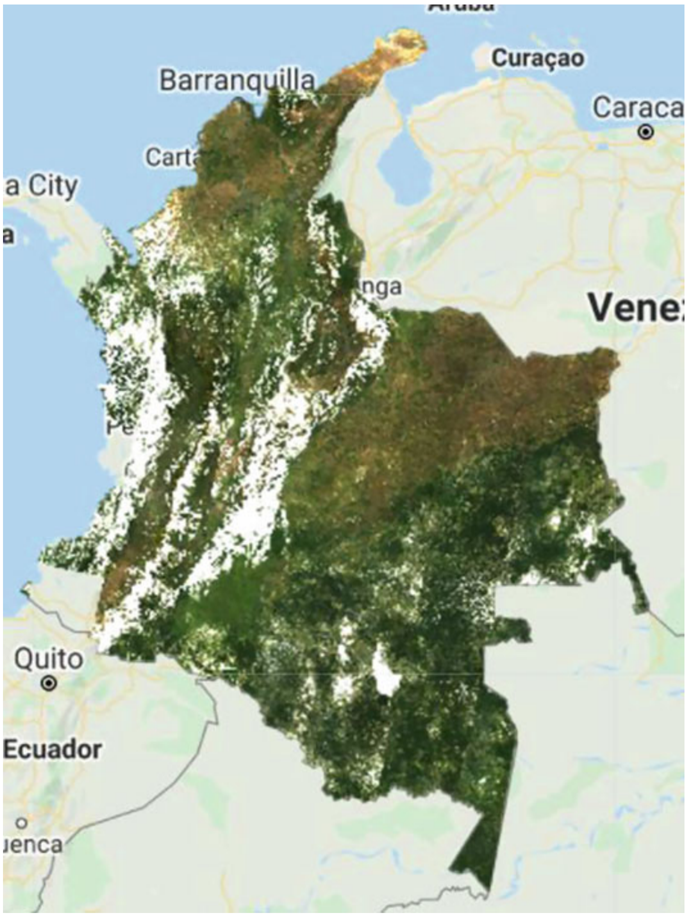


Fig. 15.2 Landsat 8 surface reflectance 2019 median composite of Colombia filtered by cloud cover less than 50%

We will also use the `QA_RADSAT` band, which indicates which bands are radiometrically saturated. A pixel value of 1 means saturated, so we will be masking these pixels.

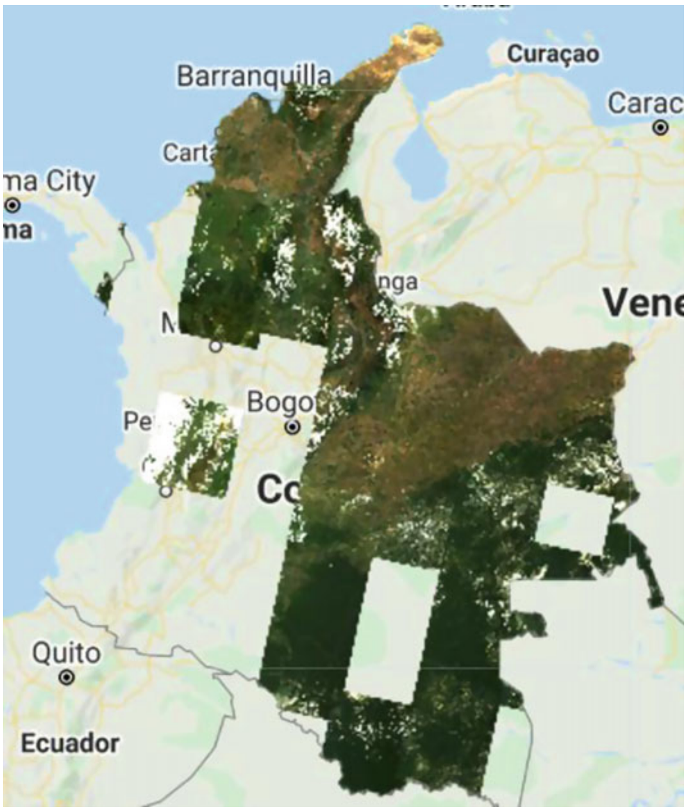


Fig. 15.3 Landsat 8 surface reflectance 2019 median composite of Colombia filtered by cloud cover less than 20%

As described in Chap. 12, we will create a function to apply a cloud mask to an image and then map this function over our collection. The mask is applied by using the `updateMask` function. This function excludes undesired pixels from the analysis, i.e., makes them transparent, by taking the mask as the input. You will see that this cloud mask function (or similar versions) is used in other chapters of the book. Note: Remember to set the cloud cover threshold back to 50 in the `landsat8FiltClouds` variable.

```
// Define the cloud mask function.
function maskSrClouds(image) {
    // Bit 0 - Fill
    // Bit 1 - Dilated Cloud
    // Bit 2 - Cirrus
    // Bit 3 - Cloud
    // Bit 4 - Cloud Shadow
    var qaMask =
image.select('QA_PIXEL').bitwiseAnd(parseInt('11111',
        2)).eq(0);
    var saturationMask = image.select('QA_RADSAT').eq(0);

    return image.updateMask(qaMask)
        .updateMask(saturationMask);
}

// Apply the cloud mask to the collection.
var landsat8FiltMasked = landsat8FiltClouds.map(maskSrClouds);

// Create a composite.
var landsat8compositeMasked =
landsat8FiltMasked.median().clip(country);

Map.addLayer(landsat8compositeMasked, visParams, 'L8 composite
masked');
```

Since we are dealing with bits, in the `maskSrClouds` function, we utilized the `bitwiseAnd` and `parseInt` functions. These functions unpack the bit information. A bitwise AND is a binary operation that takes two equal length binary representations and performs the logical AND operation on each pair of corresponding bits. Thus, if both bits in the compared positions have the value 1, the bit in the resulting binary representation is 1 ($1 \times 1 = 1$); otherwise, the result is 0 ($1 \times 0 = 0$ and $0 \times 0 = 0$). The `parseInt` function parses a string argument (in our case, five-character string `'11111'`) and returns an integer of the specified numbering system, base `2`.

The resulting composite (Fig. 15.4) shows masked clouds and is more spatially exhaustive in coverage compared to previous composites (recall to uncheck the previous layers). This is because, when compositing all the images into one, we are not considering cloudy pixels anymore, and so the resulting pixels are an actual representation of the landscape. However, data gaps due to cloud cover still persist. If an annual composite is not specifically required, a first approach is to create a two-year composite in order to mitigate the presence of data gaps or to have a series of rules that allows for selecting pixels for that particular year. Change the `startDate` variable to `2018-01-01` to include all images from 2018 and 2019 in the collection. How does the cloud-masked composite (Fig. 15.5) compare to the 2019 one?

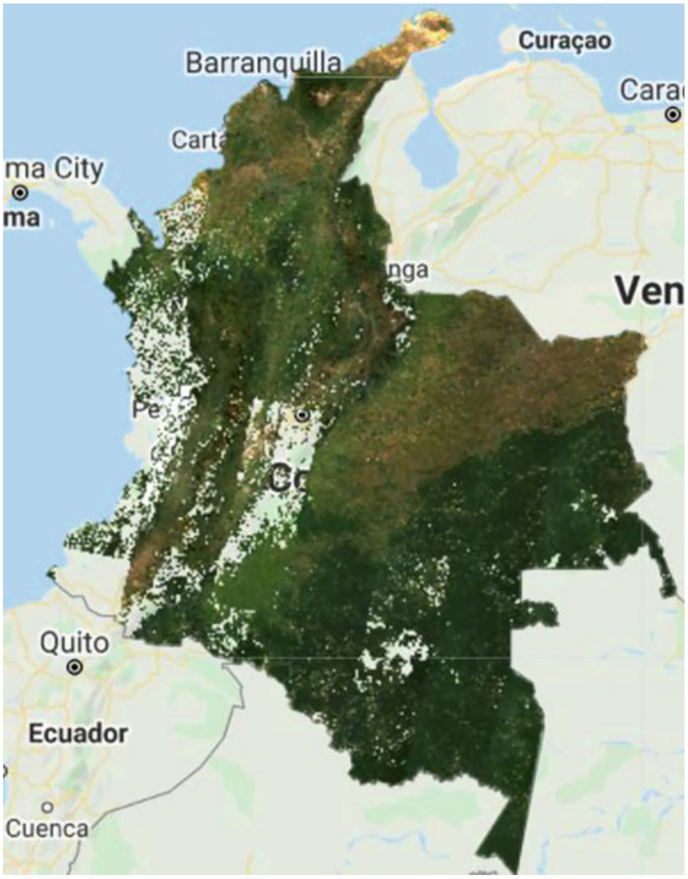


Fig. 15.4 Landsat 8 surface reflectance 2019 median composite of Colombia filtered by cloud cover less than 50% and with cloud mask applied

The resulting image has substantially fewer data gaps (you can zoom in to better see them). Again, if the time period is not a constraint for the creation of your composite, you can incorporate more images from a third year, and so on.

Code Checkpoint F43a. The book's repository contains a script that shows what your code should look like at this point.

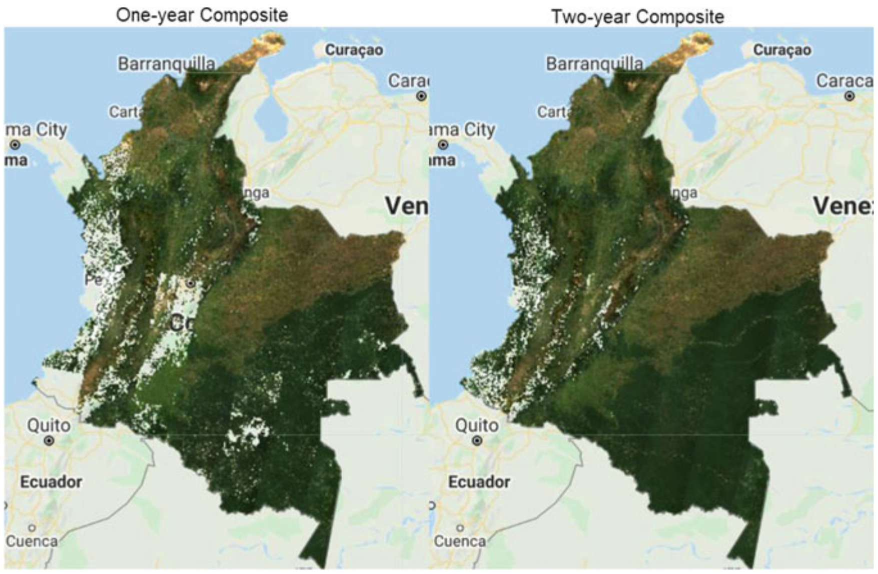


Fig. 15.5 One-year, `startDate` variable set to `2019-01-01`, (left) and two-year, `startDate` variable set to `2018-01-01`, (right) median composites with 50% cloud cover threshold and cloud mask applied

15.2.2 Section 2: Incorporating Data from Other Satellites

Another option to reduce the presence of data gaps in cloudy situations is to bring in imagery from other sensors acquired during the time period of interest. The Landsat collection spans multiple missions, which have continuously acquired uninterrupted data since 1972. Next, we will incorporate Landsat 7 Level 2, Collection 2, Tier 1 images from 2019 to fill the gaps in the 2019 Landsat 8 composite.

To generate a Landsat 7 composite, we apply similar steps to the ones we did for Landsat 8, so keep adding code to the same script from Sect. 15.2.1. First, define your Landsat 7 collection variable and the scaling function. Then, filter the collection, apply the cloud mask (since we know that Colombia has persistent cloud cover), and apply the scaling function. Note that we will use the same cloud mask function defined above, since the bits' information for Landsat 7 is the same as for Landsat 8. Finally, create the median composite. Paste in the code below and change the `startDate` variable back to `2019-01-01` before executing it to create a one-year composite of 2019.

```
// ---------- Section 2 -----------------

// Define Landsat 7 Level 2, Collection 2, Tier 1
collection.
var landsat7 =
ee.ImageCollection('LANDSAT/LE07/C02/T1_L2');

// Scaling factors for L7.
function applyScaleFactorsL7(image) {
    var opticalBands =
image.select('SR_B.').multiply(0.0000275).add(-0.2);
    var thermalBand =
image.select('ST_B6').multiply(0.00341802).add(149.0);
    return image.addBands(opticalBands, null, true)
        .addBands(thermalBand, null, true);
}

// Filter collection, apply cloud mask, and scaling
factors.
var landsat7FiltMasked = landsat7
    .filterBounds(country)
    .filterDate(startDate, endDate)
    .filter(ee.Filter.lessThan('CLOUD_COVER', 50))
    .map(maskSrClouds)
    .map(applyScaleFactorsL7);

// Create composite.
var landsat7compositeMasked = landsat7FiltMasked
    .median()
    .clip(country);

Map.addLayer(landsat7compositeMasked,
    {
        bands: ['SR_B3', 'SR_B2', 'SR_B1'],
        min: 0,
        max: 0.2
    },
    'L7 composite masked');
```

Note that we used `bands: ['SR_B3', 'SR_B2', 'SR_B1']` to visualize the composite because Landsat 7 has different band designations. The sensors aboard each of the Landsat satellites were designed to acquire data in different ranges along the electromagnetic spectrum. For Landsat 8, the red, green, and blue bands are B4, B3, and B2, whereas, for Landsat 7, these same bands are B3, B2, and B1, respectively.

The result is an image with systematic gaps like the one shown in Fig. 15.6 (remember to turn off the other layers, and zoom in to better see the data gaps). Landsat 7 was launched in 1999, but since 2003, the sensor has acquired and delivered data with data gaps caused by a scan line corrector (SLC) failure: SLC-off. Without an operating SLC, the sensor's line of sight traces a zig-zag pattern along the satellite ground track, and as a result, the imaged area is duplicated and some areas are missed. When the Level 1 data are processed, the duplicated areas are removed, which results in data gaps (Fig. 15.7). For more information about Landsat 7 and SLC-off malfunction, please refer to the USGS Landsat 7 website (https://www.usgs.gov/landsat-missions/landsat-7). However, even with the SLC-off malfunction, we can use the Landsat 7 data in our composite. Next, we will combine the Landsat 7 and 8 collections.

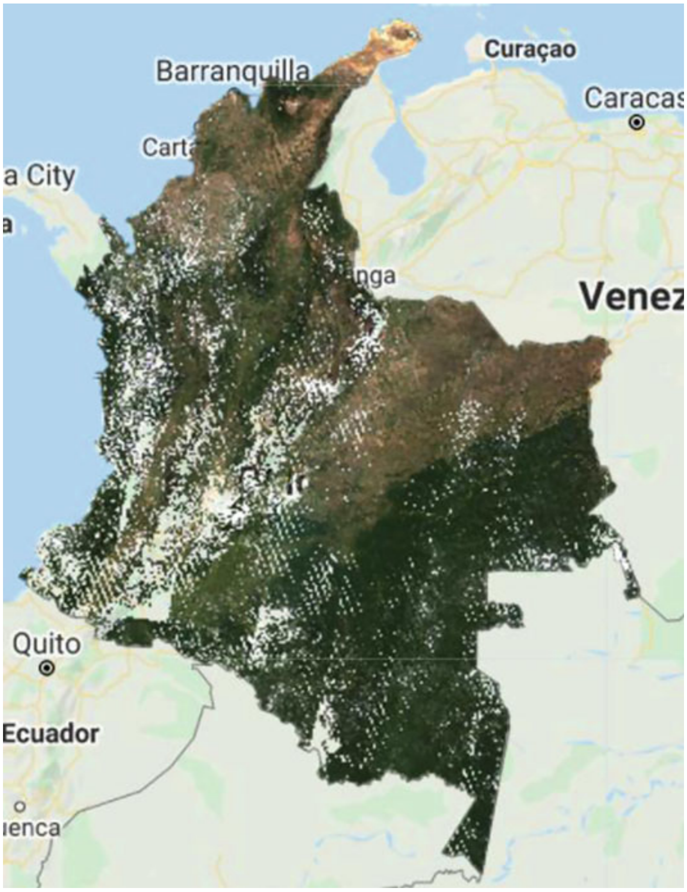


Fig. 15.6 One-year Landsat 7 median composite with 50% cloud cover threshold and cloud mask applied

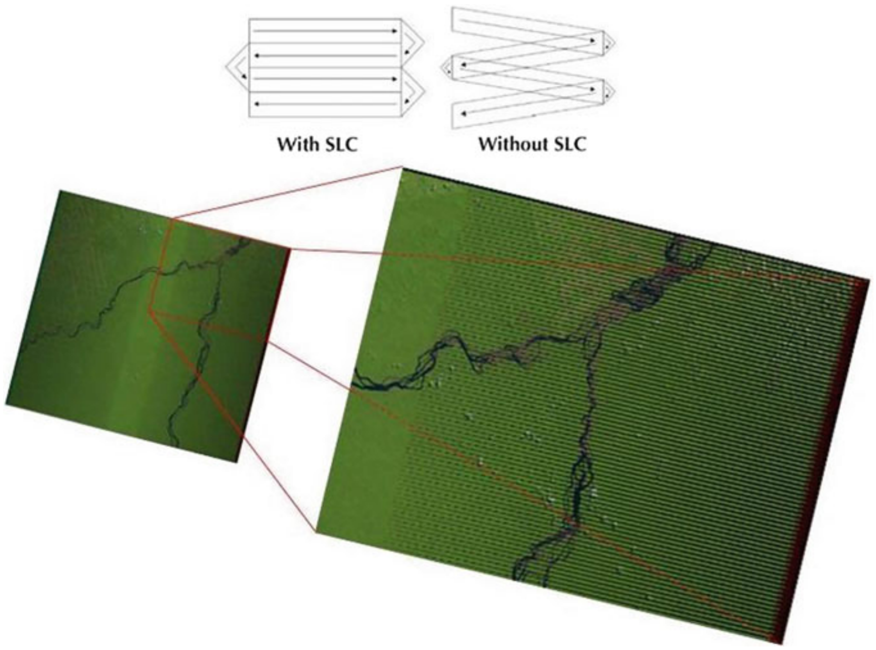
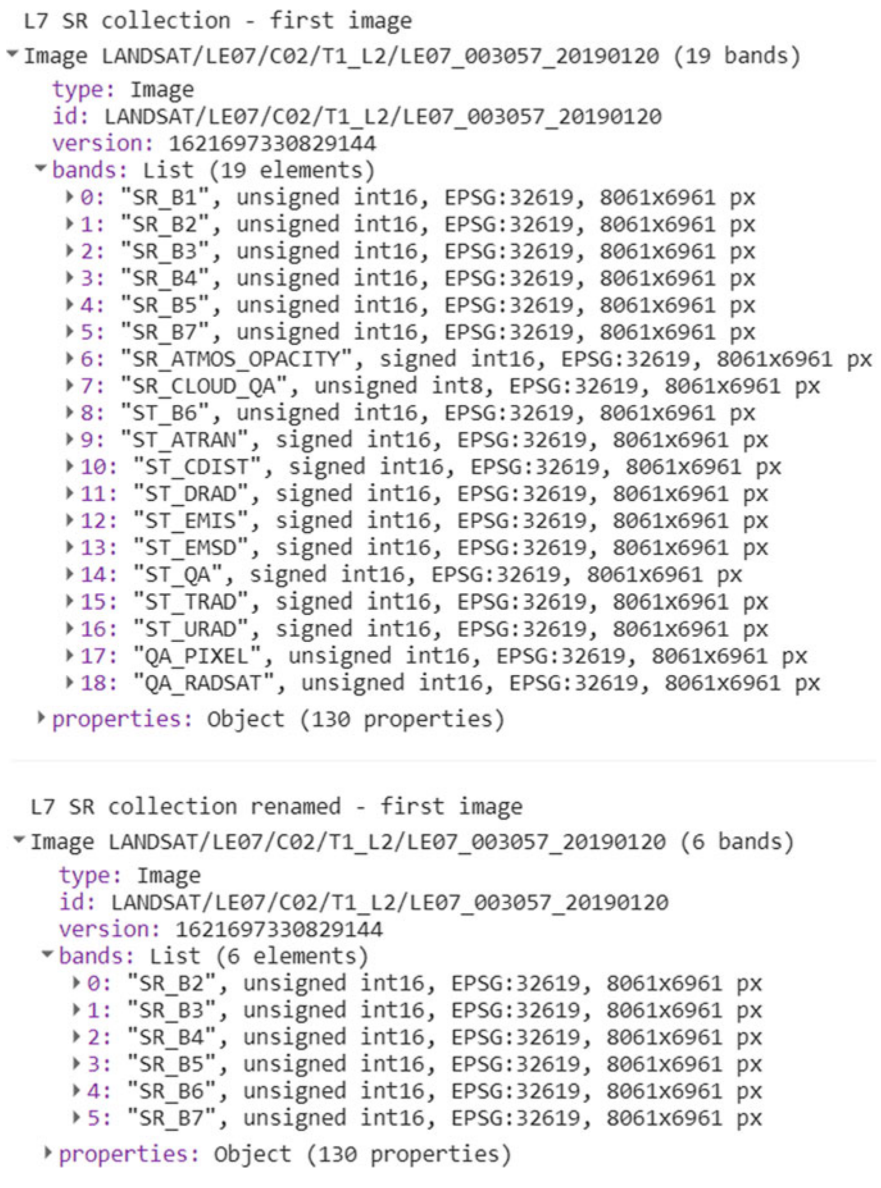


Fig. 15.7 Landsat 7's SLC-off condition. *Source* USGS

Since Landsat's 7 and 8 have different band designations, first we create a function to rename the bands from Landsat 7 to match the names used for Landsat 8 and map that function over our Landsat 7 collection.

```
// Since Landsat 7 and 8 have different band designations,
// let's create a function to rename L7 bands to match to L8.
function rename(image) {
    return image.select(
        ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5',
'SR_B7'],
        ['SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6',
'SR_B7']);
}

// Apply the rename function.
var landsat7FiltMaskedRenamed =
landsat7FiltMasked.map(rename);
```

If you print the first images of both the `landsat7FiltMasked` and `landsat7FiltMaskedRenamed` collections (Fig. 15.8), you will see that the bands were renamed, but not all bands were copied over (`SR_ATMOS_OPACITY`, `SR_CLOUD_QA`, `SR_B6`, etc.). To copy these additional bands, simply add them to the `rename` function. You will need to rename `SR_B6`, so it does not have the same name as the new band 5.



```
 L7 SR collection - first image
▾Image LANDSAT/LE07/C02/T1_L2/LE07_003057_20190120 (19 bands)
   type: Image
   id: LANDSAT/LE07/C02/T1_L2/LE07_003057_20190120
   version: 1621697330829144
  ▾bands: List (19 elements)
    ▸0: "SR_B1", unsigned int16, EPSG:32619, 8061x6961 px
    ▸1: "SR_B2", unsigned int16, EPSG:32619, 8061x6961 px
    ▸2: "SR_B3", unsigned int16, EPSG:32619, 8061x6961 px
    ▸3: "SR_B4", unsigned int16, EPSG:32619, 8061x6961 px
    ▸4: "SR_B5", unsigned int16, EPSG:32619, 8061x6961 px
    ▸5: "SR_B7", unsigned int16, EPSG:32619, 8061x6961 px
    ▸6: "SR_ATMOS_OPACITY", signed int16, EPSG:32619, 8061x6961 px
    ▸7: "SR_CLOUD_QA", unsigned int8, EPSG:32619, 8061x6961 px
    ▸8: "ST_B6", unsigned int16, EPSG:32619, 8061x6961 px
    ▸9: "ST_ATRAN", signed int16, EPSG:32619, 8061x6961 px
    ▸10: "ST_CDIST", signed int16, EPSG:32619, 8061x6961 px
    ▸11: "ST_DRAD", signed int16, EPSG:32619, 8061x6961 px
    ▸12: "ST_EMIS", signed int16, EPSG:32619, 8061x6961 px
    ▸13: "ST_EMSD", signed int16, EPSG:32619, 8061x6961 px
    ▸14: "ST_QA", signed int16, EPSG:32619, 8061x6961 px
    ▸15: "ST_TRAD", signed int16, EPSG:32619, 8061x6961 px
    ▸16: "ST_URAD", signed int16, EPSG:32619, 8061x6961 px
    ▸17: "QA_PIXEL", unsigned int16, EPSG:32619, 8061x6961 px
    ▸18: "QA_RADSAT", unsigned int16, EPSG:32619, 8061x6961 px
  ▸properties: Object (130 properties)

 L7 SR collection renamed - first image
▾Image LANDSAT/LE07/C02/T1_L2/LE07_003057_20190120 (6 bands)
   type: Image
   id: LANDSAT/LE07/C02/T1_L2/LE07_003057_20190120
   version: 1621697330829144
  ▾bands: List (6 elements)
    ▸0: "SR_B2", unsigned int16, EPSG:32619, 8061x6961 px
    ▸1: "SR_B3", unsigned int16, EPSG:32619, 8061x6961 px
    ▸2: "SR_B4", unsigned int16, EPSG:32619, 8061x6961 px
    ▸3: "SR_B5", unsigned int16, EPSG:32619, 8061x6961 px
    ▸4: "SR_B6", unsigned int16, EPSG:32619, 8061x6961 px
    ▸5: "SR_B7", unsigned int16, EPSG:32619, 8061x6961 px
  ▸properties: Object (130 properties)
```

Fig. 15.8 First images of `landsat7FiltMasked` and `landsat7FiltMaskedRenamed`, respectively

We merge the two collections using the merge function for ImageCollection and mapping over a function to cast the Landsat 7 input values to 32-bit float using the toFloat function for consistency. To merge collections, the number and names of the bands must be the same in each collection. We use the select function (Chap. 2) to select the Landsat 8 bands that match Landsat 7's. When creating the new Landsat 7 and 8 composite, if we did not select these six bands, we would get an error message for attempting to composite a collection that has six bands (Landsat 7) with a collection that has 19 bands (Landsat 8).

```
// Merge Landsat collections.
var landsat78 = landsat7FiltMaskedRenamed
    .merge(landsat8FiltMasked.select(
        ['SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6',
'SR_B7']))
    .map(function(img) {
        return img.toFloat();
    });
print('Merged collections', landsat78);
```

The result is a collection with about 1000 images. Next, we will take the median of the values across the ImageCollection.

```
// Create Landsat 7 and 8 image composite and add to the Map.
var landsat78composite = landsat78.median().clip(country);
Map.addLayer(landsat78composite, visParams, 'L7 and L8
composite');
```

Comparing the composite generated considering both Landsat 7 and 8 to the Landsat 8-only composite, a reduction in the amount of data gaps in the final result is evident (Fig. 15.9). The resulting Landsat 7 and 8 image composite still has data gaps due to the presence of clouds and Landsat 7's SLC-off data. Set the center of the map to the point with latitude 3.6023 and longitude − 75.0741 to see the inset example of Fig. 15.9.

Code Checkpoint F43b. The book's repository contains a script that shows what your code should look like at this point.

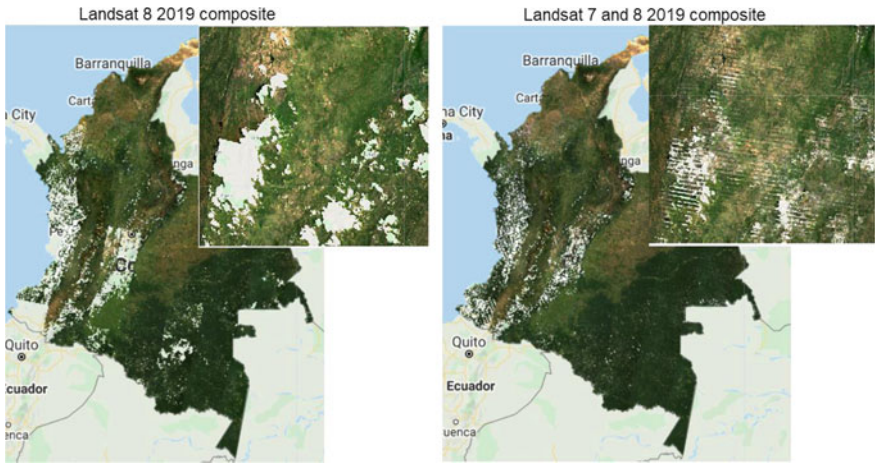


Fig. 15.9 Landsat 8-only composite (left) and Landsat 7 and 8 composite (right) for 2019. Inset centered at latitude 3.6023, longitude − 75.0741

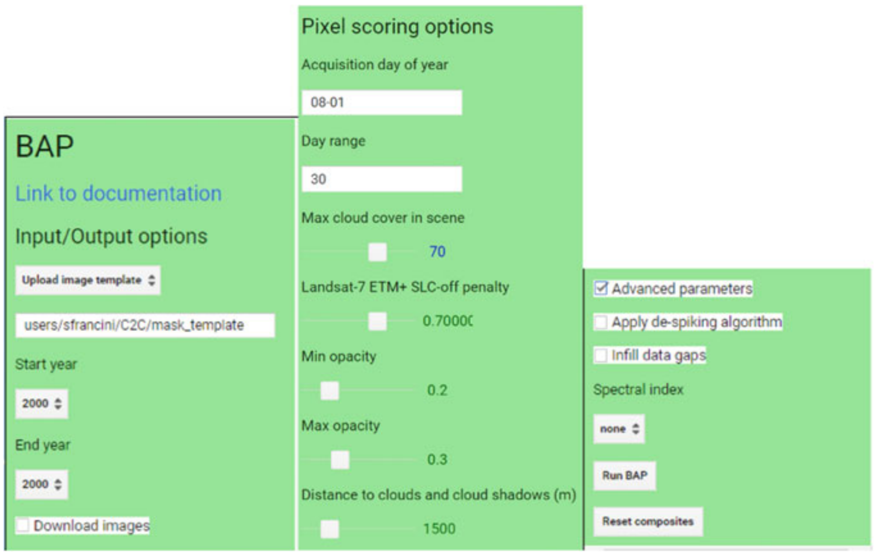
15.2.3 Section 3: Best-Available-Pixel Compositing Earth Engine Application

This section presents an Earth Engine application that enables the generation of annual best-available-pixel (BAP) image composites by globally combining multiple Landsat sensors and images: GEE-BAP. Annual BAP image composites are generated by choosing optimal observations for each pixel from all available Landsat 5 TM, Landsat 7 ETM+, and Landsat 8 OLI images within a given year and within a given day range from a specified acquisition day of year, in addition to other constraints defined by the user. The data accessible via Earth Engine are from the USGS free and open archive of Landsat data. The Landsat images used are atmospherically corrected to surface reflectance values. Following White et al. (2014), a series of scoring functions ranks each pixel observation for (1) acquisition day of year, (2) cloud cover in the scene, (3) distance to clouds and cloud shadows, (4) presence of haze, and (5) acquisition sensor. Further information on the BAP image compositing approach can be found in Griffiths et al. (2013), and detailed information on tuning parameters can be found in White et al. (2014).

Code Checkpoint F43c. The book's repository contains information about accessing the GEE-BAP interface and its related functions.

Once you have loaded the GEE-BAP interface (Fig. 15.10) using the instructions in the Code Checkpoint, you will notice that it is divided into three sections: (1) **Input/Output options**, (2) **Pixel scoring options**, and (3) **Advanced parameters**. Users indicate the study area, the time period for generating annual BAP composites (i.e., start and end years), and the path to store the results in the **Input/Output options**. Users have three options to define the study area. The **Draw study area** option uses the **Draw a shape** and **Draw a rectangle** tools to define the area of interest. The **Upload image template** option utilizes an image template uploaded by the user in TIFF format. This option is well suited to generating BAP composites that match the projection, pixel size, and extent to existing raster datasets. The **Work globally** option generates BAP composites for the entire globe; note that when this option is selected, complete data download is not available due to the Earth's size. With **Start year** and **End year**, users can indicate the beginning and end of the annual time series of BAP image composites to be generated. Multiple image composites are then generated—one composite for each year—resulting in a time series of annual composites. For each year, composites are uniquely generated utilizing images acquired on the days within the specified **Date range**. Produced BAP composites can be saved in the indicated (**Path**) Google Drive folder using the **Tasks** tab. Results are generated in a tiled, TIFF format, accompanied by a CSV file that indicates the parameters used to construct the composite.

As noted, GEE-BAP implements five pixel scoring functions: (1) target acquisition day of year and day range, (2) maximum cloud coverage per scene, (3) distance to clouds and cloud shadows, (4) atmospheric opacity, and (5) a penalty for images acquired under the Landsat 7 ETM+ SLC-off malfunction. By defining the **Acquisition day of year** and **Day range**, those candidate pixels acquired closer to a defined acquisition day of year are ranked higher. Note that pixels acquired outside the day range window are excluded from subsequent composite development. For example, if the target day of year is defined as "08-01" and the day range as "31," only those pixels acquired between July 1 and August 31 are considered, and the ones acquired closer to August 1 will receive a higher score.


BAP

Link to documentation

Input/Output options

Upload image template

users/sfrancini/C2C/mask_template

Start year: 2000

End year: 2000

☐ Download images

Pixel scoring options

Acquisition day of year: 08-01

Day range: 30

Max cloud cover in scene: 70

Landsat-7 ETM+ SLC-off penalty: 0.7000(

Min opacity: 0.2

Max opacity: 0.3

Distance to clouds and cloud shadows (m): 1500

☑ Advanced parameters

☐ Apply de-spiking algorithm

☐ Infill data gaps

Spectral index: none

Fig. 15.10 GEE-BAP user interface controls

The scoring function **Max cloud cover in scene** indicates the maximum percentage of cloud cover in an image that will be accepted by the user in the BAP image compositing process. Defining a value of 70% implies that only those scenes with less than or equal to 70% cloud cover will be considered as a candidate for compositing.

The **Distance to clouds and cloud shadows** scoring function enables the user to exclude those pixels identified to contain clouds and shadows by the QA mask from the generated BAP, as well as decreasing a pixel's score if the pixel is within a specified proximity of a cloud or cloud shadow.

The **Atmospheric opacity** scoring function ranks pixels based on their atmospheric opacity values, which are indicative of hazy imagery. Pixels with opacity values that exceed a defined haze expectation (**Max opacity**) are excluded. Pixels with opacity values lower than a defined value (**Min opacity**) get the maximum score. Pixels with values in between these limits are scored following the functions defined by Griffiths et al. (2013). This scoring function is available only for Landsat 5 TM and Landsat 7 ETM+ images, which provides the opacity attribute in the image metadata file.

Finally, there is a **Landsat 7 ETM+ SLC-off penalty** scoring function that de-emphasizes images acquired following the ETM+ SLC-off malfunction in 2003. The aim of this scoring element is to ensure that TM or OLI data, which do not have stripes, take precedence over ETM+ when using dates after the SLC failure. This allows users to avoid the inclusion of multiple discontinuous small portions of images being used to produce the BAP image composites, thus reducing the spatial variability of the spectral data. The penalty applied to SLC-off imagery is defined directly proportional to the overall score. A large score reduces the chance that SLC-off imagery will be used in the composite. A value of 1 prevents SLC-off imagery from being used.

By default, the GEE-BAP application produces image composites using all the visible bands. The **Spectral index** option enables the user to produce selected spectral indices from the resulting BAP image composites. Available spectral indices include: Normalized Difference Vegetation Index (NDVI, Fig. 15.11), Enhanced Vegetation Index (EVI), and Normalized Burn Ratio (NBR), as well as several indices derived from the Tasseled Cap transformation: Wetness (TCW), Greenness (TCG), Brightness (TCB), and Angle (TCA). Composited indices are able to be downloaded as well as viewed on the map.

GEE-BAP functions can be accessed programmatically, including pixel scoring parameters, as well as BAP image compositing (`BAP`), de-spiking (`despikeCollection`), data-gap infilling (`infill`), and displaying (`ShowCollection`) functions. The following code sets the scoring parameter values, then generates and displays the compositing results (Fig. 15.12) for a BAP composite that is de-spiked, with data gaps infilled using temporal interpolation. Copy and paste the code below into a new script.

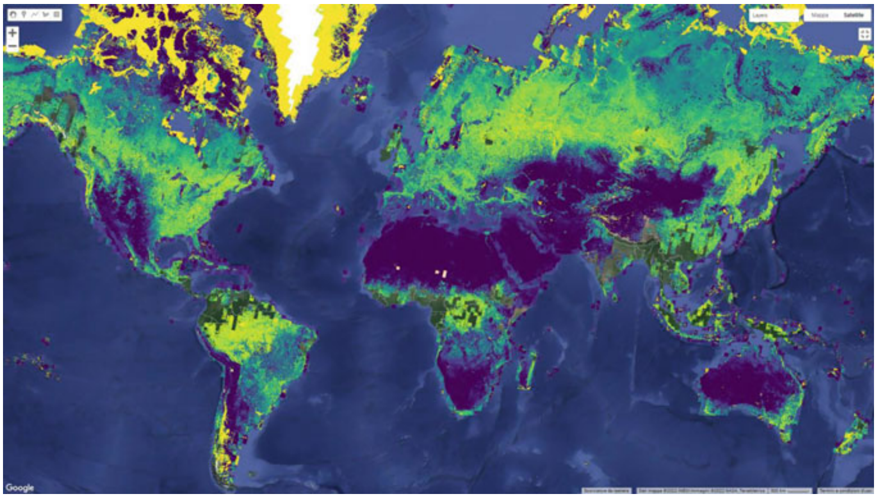


Fig. 15.11 Example of a global BAP image composite showing NDVI values generated using the GEE-BAP user interface

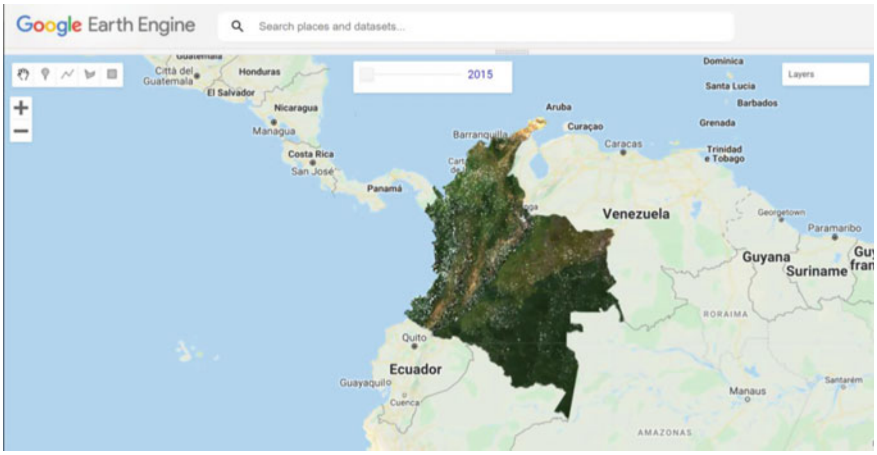


Fig. 15.12 Outcome of the compositing code

```
// Define required parameters.
var targetDay = '06-01';
var daysRange = 75;
var cloudsTh = 70;
var SLCoffPenalty = 0.7;
var opacityScoreMin = 0.2;
var opacityScoreMax = 0.3;
var cloudDistMax = 1500;
var despikeTh = 0.65;
var despikeNbands = 3;
var startYear = 2015;
var endYear = 2017;

// Define study area.
var worldCountries =
ee.FeatureCollection('USDOS/LSIB_SIMPLE/2017');
var colombia = worldCountries.filter(ee.Filter.eq('country_na',
    'Colombia'));

// Load the bap library.
var library = require('users/sfrancini/bap:library');

// Calculate BAP.
var BAPCS = library.BAP(null, targetDay, daysRange, cloudsTh,
    SLCoffPenalty, opacityScoreMin, opacityScoreMax,
cloudDistMax);

// Despike the collection.
BAPCS = library.despikeCollection(despikeTh, despikeNbands,
BAPCS,
    1984, 2021, true);

// Infill datagaps.
BAPCS = library.infill(BAPCS, 1984, 2021, false, true);

// Visualize the image.
Map.centerObject(colombia, 5);
library.ShowCollection(BAPCS, startYear, endYear, colombia,
false,null);
library.AddSLider(startYear, endYear);
```

Code Checkpoint F43d. The book's repository contains a script that shows what your code should look like at this point.

15.3 Synthesis

Assignment 1. Create composites for other cloudy regions or less cloudy regions. For example, change the `country` variable to `'Cambodia'` or `'Mozambique'`. Are more gaps present in the resulting composite? Can you change the compositing rules to improve this (using **Acquisition day of year** and **Day range**)? Different regions of the Earth have different cloud seasonal patterns, so the most appropriate date windows to acquire cloud-free composites will change depending on location. Also be aware that the larger the country, the longer it will take to generate the composite.

Assignment 2. Similarly, try creating composites for the wet and dry seasons of a region separately. Compare the two composites. Are some features brighter or darker? Is there evidence of drying of vegetation, such as leaf loss or reduction in herbaceous ground vegetation?

Assignment 3. Test different cloud threshold values and see if you can find an optimal threshold that balances data gaps against area coverage for your particular target date.

15.4 Conclusion

We cannot monitor what we cannot see. Image compositing algorithms provide robust and transparent tools to address issues with clouds, cloud shadows, haze, and smoke in remotely sensed images derived from optical satellite data and expand data availability for remote sensing applications. The tools and approaches described here should provide you with some useful strategies to aid in mitigating the presence of cloud cover in your data. Note that the quality of image outcomes is a function of the quality of cloud masking routines applied to the source data to generate the various flags that are used in the scoring functions described herein. Different compositing parameters can be used to represent a given location as a function of conditions that are present at a given point in time and the information needs of the end user. Tuning or optimization of compositing parameters is possible (and recommended) to ensure best capture of the physical conditions of interest.

References

Braaten JD, Cohen WB, Yang Z (2015) Automated cloud and cloud shadow identification in Landsat MSS imagery for temperate ecosystems. Remote Sens Environ 169:128–138. https://doi.org/10.1016/j.rse.2015.08.006

Cao R, Chen Y, Chen J et al (2020) Thick cloud removal in Landsat images based on autoregression of Landsat time-series data. Remote Sens Environ 249:112001. https://doi.org/10.1016/j.rse.2020.112001

Eberhardt IDR, Schultz B, Rizzi R et al (2016) Cloud cover assessment for operational crop monitoring systems in tropical areas. Remote Sens 8:219. https://doi.org/10.3390/rs8030219
Griffiths P, van der Linden S, Kuemmerle T, Hostert P (2013) A pixel-based Landsat compositing algorithm for large area land cover mapping. IEEE J Sel Top Appl Earth Obs Remote Sens 6:2088–2101. https://doi.org/10.1109/JSTARS.2012.2228167
Hermosilla T, Wulder MA, White JC et al (2015) An integrated Landsat time series protocol for change detection and generation of annual gap-free surface reflectance composites. Remote Sens Environ 158:220–234. https://doi.org/10.1016/j.rse.2014.11.005
Huang C, Thomas N, Goward SN et al (2010) Automated masking of cloud and cloud shadow for forest change analysis using Landsat images. Int J Remote Sens 31:5449–5464. https://doi.org/10.1080/01431160903369642
Irish RR, Barker JL, Goward SN, Arvidson T (2006) Characterization of the Landsat-7 ETM+ automated cloud-cover assessment (ACCA) algorithm. Photogramm Eng Remote Sens 72:1179–1188. https://doi.org/10.14358/PERS.72.10.1179
Loveland TR, Dwyer JL (2012) Landsat: building a strong future. Remote Sens Environ 122:22–29. https://doi.org/10.1016/j.rse.2011.09.022
Martins VS, Novo EMLM, Lyapustin A et al (2018) Seasonal and interannual assessment of cloud cover and atmospheric constituents across the Amazon (2000–2015): insights for remote sensing and climate analysis. ISPRS J Photogramm Remote Sens 145:309–327. https://doi.org/10.1016/j.isprsjprs.2018.05.013
Roy DP, Ju J, Kline K et al (2010) Web-enabled Landsat data (WELD): Landsat ETM+ composited mosaics of the conterminous United States. Remote Sens Environ 114:35–49. https://doi.org/10.1016/j.rse.2009.08.011
White JC, Wulder MA, Hobart GW et al (2014) Pixel-based image compositing for large-area dense time series applications and science. Can J Remote Sens 40:192–212. https://doi.org/10.1080/07038992.2014.945827
Zhu X, Helmer EH (2018) An automatic method for screening clouds and cloud shadows in optical satellite image time series in cloudy regions. Remote Sens Environ 214:135–153. https://doi.org/10.1016/j.rse.2018.05.024
Zhu Z, Woodcock CE (2012) Object-based cloud and cloud shadow detection in Landsat imagery. Remote Sens Environ 118:83–94. https://doi.org/10.1016/j.rse.2011.10.028
Zhu Z, Woodcock CE (2014) Automated cloud, cloud shadow, and snow detection in multitemporal Landsat data: an algorithm designed specifically for monitoring land cover change. Remote Sens Environ 152:217–234. https://doi.org/10.1016/j.rse.2014.06.012
Zhu Z, Wang S, Woodcock CE (2015) Improvement and expansion of the Fmask algorithm: cloud, cloud shadow, and snow detection for Landsats 4–7, 8, and Sentinel 2 images. Remote Sens Environ 159:269–277. https://doi.org/10.1016/j.rse.2014.12.014

Change Detection

16

Karis Tenneson, John Dilger, Crystal Wespestad, Brian Zutta, Andréa Puzzi Nicolau, Karen Dyson, and Paula Paz

Overview
This chapter introduces change detection mapping. It will teach you how to make a two-date land cover change map using image differencing and threshold-based classification. You will use what you have learned so far in this book to produce a map highlighting changes in the land cover between two time steps. You will

K. Tenneson · J. Dilger (✉) · C. Wespestad · B. Zutta · A. P. Nicolau · K. Dyson
Spatial Informatics Group, Pleasanton, CA, USA
e-mail: jdilger@sig-gis.com

K. Tenneson
e-mail: ktenneson@sig-gis.com

C. Wespestad
e-mail: cwespestad@sig-gis.com

B. Zutta
e-mail: bzutta@sig-gis.com

A. P. Nicolau
e-mail: apnicolau@sig-gis.com

K. Dyson
e-mail: kdyson@sig-gis.com

SERVIR-Amazonia, Cali, Colombia

K. Dyson
Dendrolytics, Seattle, WA, USA

P. Paz
Alliance Bioversity-CIAT, Cali, Colombia
e-mail: p.a.paz@cgiar.org

J. Dilger
Astraea, Charlottesville, Virginia, USA

first explore differences between the two images extracted from these time steps by creating a difference layer. You will then learn how to directly classify change based on the information in both of your images.

Learning Outcomes

- Creating and exploring how to read a false-color cloud-free Landsat composite image
- Calculating the normalized burn ratio (NBR) index.
- Creating a two-image difference to help locate areas of change.
- Producing a change map and classifying changes using thresholding.

Assumes you know how to:

- Import images and image collections, filter, and visualize (Part 1).
- Perform basic image analysis: select bands, compute indices, create masks (Part 2).

16.1 Introduction to Theory

Change detection is the process of assessing how landscape conditions are changing by looking at differences in images acquired at different times. This can be used to quantify changes in forest cover—such as those following a volcanic eruption, logging activity, or wildfire—or when crops are harvested (Fig. 16.1). For example, using time-series change detection methods, Hansen et al. (2013) quantified annual changes in forest loss and regrowth. Change detection mapping is important for observing, monitoring, and quantifying changes in landscapes over time. Key questions that can be answered using these techniques include identifying whether a change has occurred, measuring the area or the spatial extent of the region undergoing change, characterizing the nature of the change, and measuring the pattern (configuration or composition) of the change (MacLeod and Congalton 1998).

Many change detection techniques use the same basic premise: that most changes on the landscape result in spectral values that differ between pre-event and post-event images. The challenge can be to separate the real changes of interest—those due to activities on the landscape—from noise in the spectral signal, which can be caused by seasonal variation and phenology, image misregistration, clouds and shadows, radiometric inconsistencies, variability in illumination (e.g., sun angle, sensor position), and atmospheric effects.

Activities that result in pronounced changes in radiance values for a sufficiently long time period are easier to detect using remote sensing change detection techniques than are subtle or short-lived changes in landscape conditions. Mapping challenges can arise if the change event is short-lived, as these are difficult to capture using satellite instruments that only observe a location every several days. Other types of changes occur so slowly or are so vast that they are not easily detected until they are observed using satellite images gathered over a sufficiently long interval of time. Subtle changes that occur slowly on the landscape may be better suited to more computationally demanding methods, such as time-series analysis. Kennedy et al. (2009) provide a nice overview of the concepts and tradeoffs involved when designing landscape monitoring approaches. Additional summaries of change detection methods and recent advances include Singh (1989), Coppin et al. (2004), Lu et al. (2004) and Woodcock et al. (2020).

For land cover changes that occur abruptly over large areas on the landscape and are long-lived, a simple two-date image differencing approach is suitable.

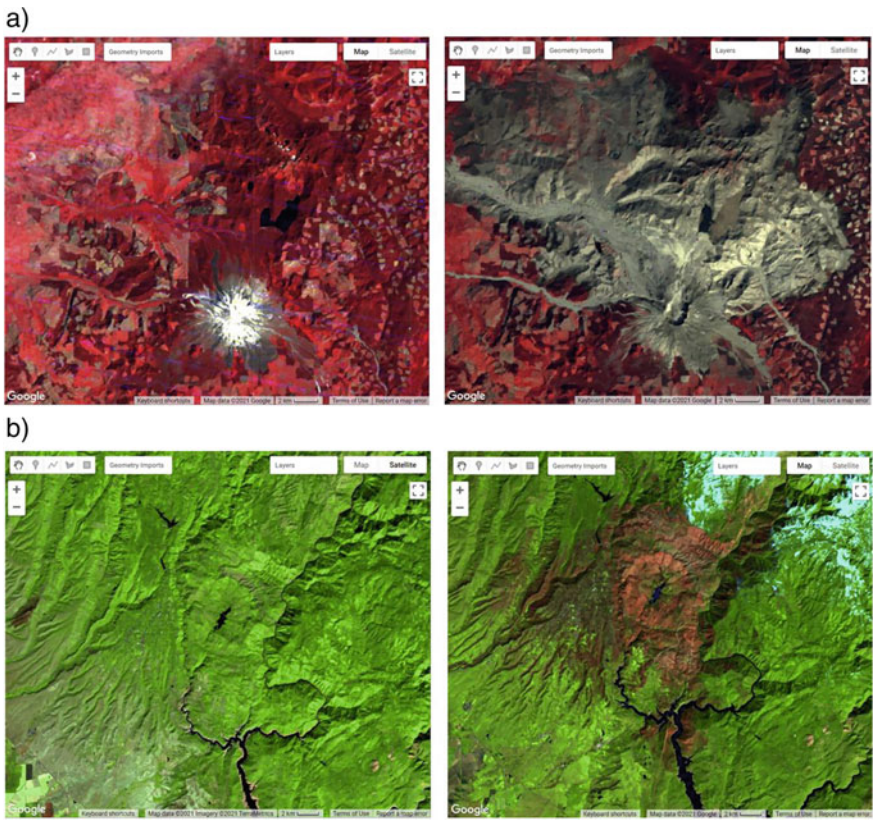


Fig. 16.1 Before and after images of **a** the eruption of Mount St. Helens in Washington State, USA, in 1980 (before, July 10, 1979; after, September 5, 1980); **b** the Camp Fire in California, USA, in 2018 (before, October 7, 2018; after, March 16, 2019); **c** illegal gold mining in the Madre de Dios region of Peru (before, March 31, 2001; after, August 22, 2020); and **d** shoreline changes in Incheon, South Korea (before, May 29, 1981; after, March 11, 2020)

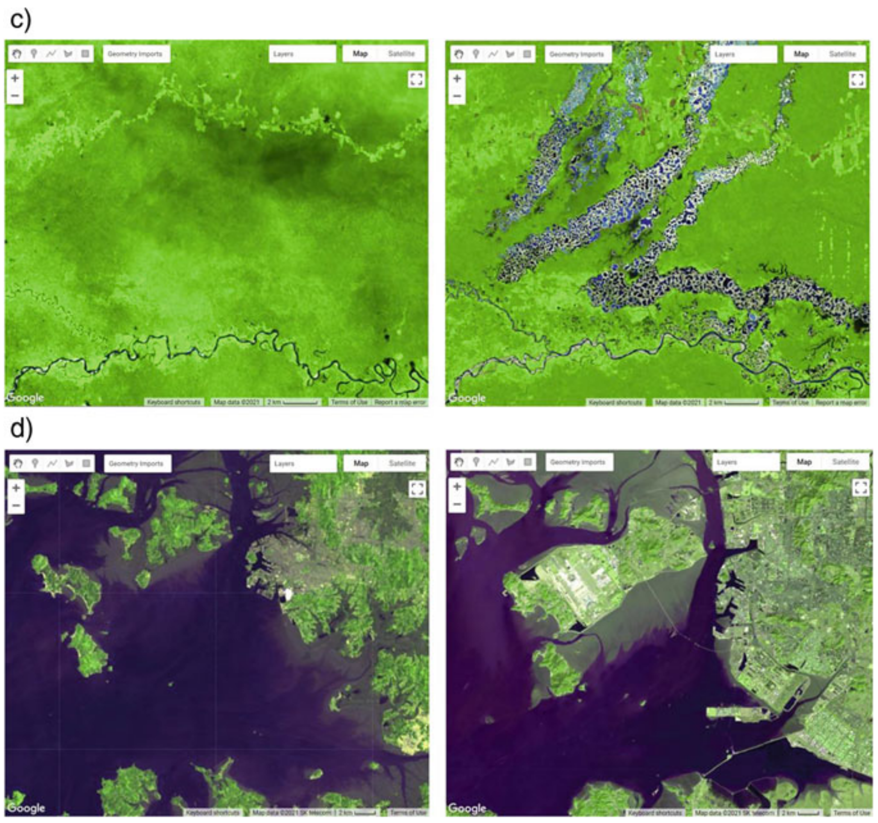


Fig. 16.1 (continued)

Two-date image differencing techniques are long-established methods for identifying changes that produce easily interpretable results (Singh 1989). The process typically involves four steps: (1) image selection and preprocessing; (2) data transformation, such as calculating the difference between indices of interest [e.g., the normalized difference vegetation index (NDVI)] in the pre-event and post-event images; (3) classifying the differenced image(s) using thresholding or supervised classification techniques; and (4) evaluation.

16.2 Practicum

For the practicum, you will select pre-event and post-event image scenes and investigate the conditions in these images in a false-color composite display. Next, you will calculate the NBR index for each scene and create a difference image using the two NBR maps. Finally, you will apply a threshold to the difference image to establish categories of changed versus stable areas (Fig. 16.2).

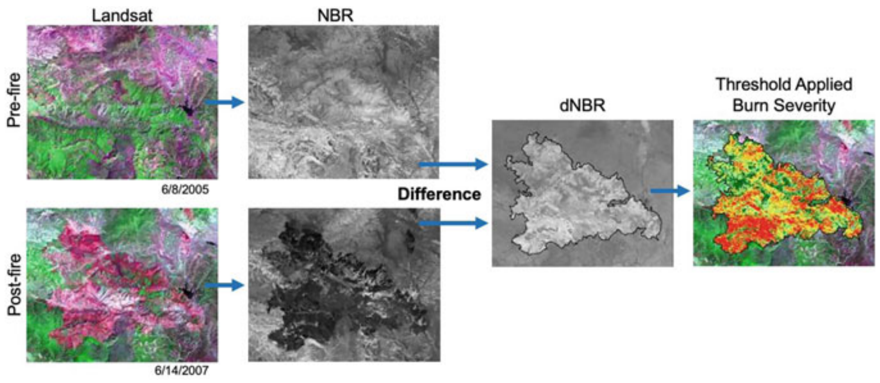


Fig. 16.2 Change detection workflow for this practicum

16.2.1 Section 1: Preparing Imagery

Before beginning a change detection workflow, image preprocessing is essential. The goal is to ensure that each pixel records the same type of measurement at the same location over time. These steps include multitemporal image registration and radiometric and atmospheric corrections, which are especially important. A lot of this work has been automated and already applied to the images that are available in Earth Engine. Image selection is also important. Selection considerations include finding images with low cloud cover and representing the same phenology (e.g., leaf-on or leaf-off).

The code in the block below accesses the USGS Landsat 8 Level 2, Collection 2, Tier 1 dataset and assigns it to the variable `landsat8`. To improve readability when working with the Landsat 8 `ImageCollection`, the code selects bands 2–7 and renames them to band names instead of band numbers.

```
var landsat8 = ee.ImageCollection('LANDSAT/LC08/C02/T1_L2')
    .select(
        ['SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7'],
        ['blue', 'green', 'red', 'nir', 'swir1', 'swir2']
    );
```

Next, you will split the Landsat 8 `ImageCollection` into two collections, one for each time period, and apply some filtering and sorting to get an image for each of two time periods. In this example, we know that there are few clouds for the months of the analysis; if you are working in a different area, you may need to apply some cloud masking or mosaicing techniques (see Chap. 15).

The code below does several things. First, it creates a new geometry variable to filter the geographic bounds of the image collections. Next, it creates a new variable for the pre-event image by (1) filtering the collection by the date range of interest (e.g., June 2013), (2) filtering the collection by the geometry, (3) sorting by cloud cover so the first image will have the least cloud cover, and (4) getting the first image from the collection.

Now repeat the previous step, but assign it to a post-event image variable and change the filter date to a period after the pre-event image's date range (e.g., June 2020).

```
var point = ee.Geometry.Point([-123.64, 42.96]);
Map.centerObject(point, 11);

var preImage = landsat8
    .filterBounds(point)
    .filterDate('2013-06-01', '2013-06-30')
    .sort('CLOUD_COVER', true)
    .first();

var postImage = landsat8
    .filterBounds(point)
    .filterDate('2020-06-01', '2020-06-30')
    .sort('CLOUD_COVER', true)
    .first();
```

16.2.2 Section 2: Creating False-Color Composites

Before running any sort of change detection analysis, it is useful to first visualize your input images to get a sense of the landscape, visually inspect where changes might occur, and identify any problems in the inputs before moving further. As described in Chap. 2, false-color composites draw bands from multispectral sensors in the red, green, and blue channels in ways that are designed to illustrate contrast in imagery. Below, you will produce a false-color composite using SWIR2 in the red channel, NIR in the green channel, and red in the blue channel (Fig. 16.3).

Following the format in the code block below, first create a variable `visParam` to hold the display parameters, selecting the SWIR2, NIR, and red bands, with values drawn that are between 7750 and 22,200. Next, add the pre-event and post-event images to the map and click **Run**. Click and drag the opacity slider on the post-event image layer back and forth to view the changes between your two images.

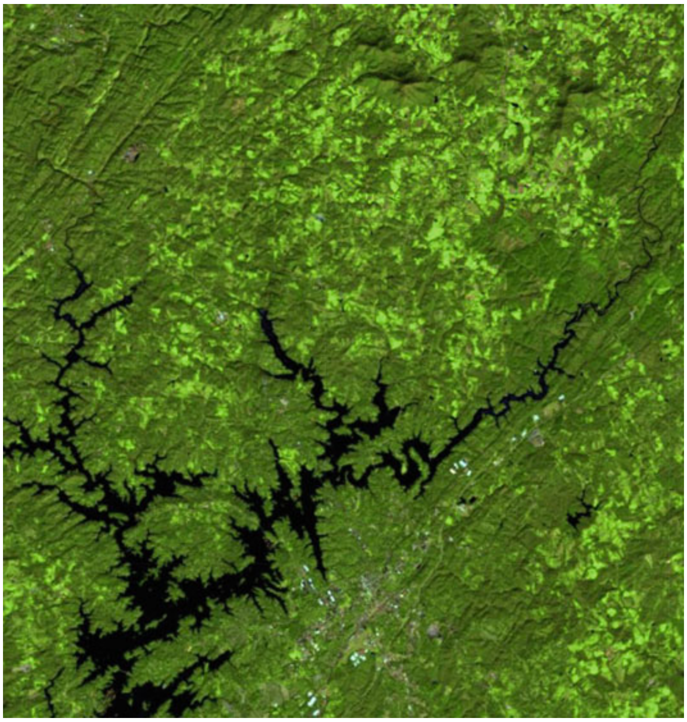


Fig. 16.3 False-color composite using SWIR2, NIR, and red. Vegetation shows up vividly in the green channel due to vegetation being highly reflective in the NIR band. Shades of green can be indicative of vegetation density; water typically shows up as black to dark blue; and burned or barren areas show up as brown

```
var visParam = {
    'bands': ['swir2', 'nir', 'red'],
    'min': 7750,
    'max': 22200
};
Map.addLayer(preImage, visParam, 'pre');
Map.addLayer(postImage, visParam, 'post');
```

16.2.3 Section 3: Calculating NBR

The next step is data transformation, such as calculating NBR. The advantage of using these techniques is that the data, along with the noise inherent in the data, have been reduced in order to simplify a comparison between two images.

Image differencing is done by subtracting the spectral value of the first-date image from that of the second-date image, pixel by pixel (Fig. 16.2). Two-date image differencing can be used with a single band or with spectral indices, depending on the application. Identifying the correct band or index to identify change and finding the correct thresholds to classify it are critical to producing meaningful results. Working with indices known to highlight the land cover conditions before and after a change event of interest is a good starting point. For example, the normalized difference water index would be good for mapping water level changes during flooding events; the NBR is good at detecting soil brightness; and the NDVI can be used for tracking changes in vegetation (although this index does saturate quickly). In some cases, using derived band combinations that have been customized to represent the phenomenon of interest is suggested, such as using the normalized difference fraction index to monitor forest degradation (see Chap. 49).

Examine changes to the landscape caused by fires using NBR, which measures the severity of fires using the equation (NIR − SWIR)/(NIR + SWIR). These bands were chosen because they respond most strongly to the specific changes in forests caused by fire. This type of equation, a difference of variables divided by their sum, is referred to as a normalized difference equation (see Chap. 5). The resulting value will always fall between − 1 and 1. NBR is useful for determining whether a fire recently occurred and caused damage to the vegetation, but it is not designed to detect other types of land cover changes especially well.

First, calculate the NBR for each time period using the built-in normalized difference function. For Landsat 8, be sure to utilize the NIR and SWIR2 bands to calculate NBR. Then, rename each image band with the built-in `rename` function.

```
// Calculate NBR.
var nbrPre = preImage.normalizedDifference(['nir', 'swir2'])
    .rename('nbr_pre');
var nbrPost = postImage.normalizedDifference(['nir', 'swir2'])
    .rename('nbr_post');
```

Code Checkpoint F44a. The book’s repository contains a script that shows what your code should look like at this point.

16.2.4 Section 4: Single Date Transformation

Next, we will examine the changes that have occurred, as seen when comparing two specific dates in time.

Subtract the pre-event image from the post-event image using the `subtract` function. Add the two-date change image to the map with the specialized Fabio Crameri *batlow* color ramp (Crameri et al. 2020). This color ramp is an example of a color combination specifically designed to be readable by colorblind and color-deficient viewers. Being cognizant of your cartographic choices is an important part of making a good change map.

```
// 2-date change.
var diff = nbrPost.subtract(nbrPre).rename('change');

var palette = [
    '011959', '0E365E', '1D5561', '3E6C55', '687B3E',
    '9B882E', 'D59448', 'F9A380', 'FDB7BD', 'FACCFA'
];
var visParams = {
    palette: palette,
    min: -0.2,
    max: 0.2
};
Map.addLayer(diff, visParams, 'change');
```

Question 1. Try to interpret the resulting image before reading on. What patterns of change can you identify? Can you find areas that look like vegetation loss or gain?

The color ramp has dark blues for the lowest values, greens, and oranges in the midrange and pink for the highest values. We used `nbrPre` subtracted from `nbrPost` to identify changes in each pixel. Since NBR values are higher when vegetation is present, areas that are negative in the change image will represent pixels that were higher in the `nbrPre` image than in the `nbrPost` image. Conversely, positive differences mean that an area gained vegetation (Fig. 16.4).

16.2.5 Section 5: Classifying Change

Once the images have been transformed and differenced to highlight areas undergoing change, the next step is image classification into a thematic map consisting of stable and change classes. This can be done rather simply by thresholding the change layer, or by using classification techniques such as machine learning algorithms. One challenge of working with simple thresholding of the difference layer is knowing how to select a suitable threshold to partition changed areas from stable classes. On the other hand, classification techniques using machine learning algorithms partition the landscape using examples of reference data that you provide to train the classifier. This may or may not yield better results but does require additional work to collect reference data and train the classifier. In the end, resources, timing, and the patterns of the phenomenon you are trying to map will determine which approach is suitable—or perhaps the activity you are trying to track requires

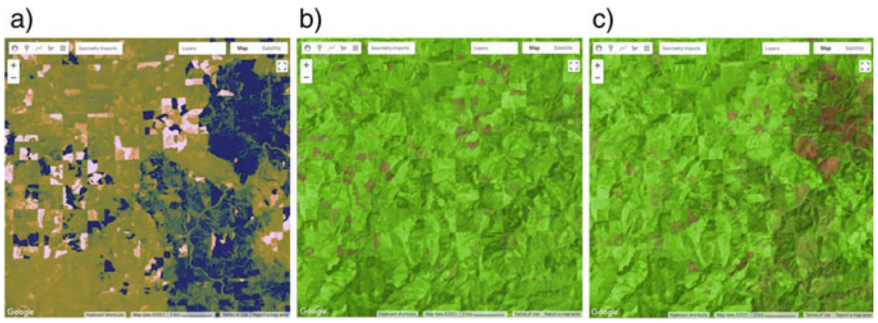


Fig. 16.4 **a** Two-date NBR difference; **b** pre-event image (June 2013) false-color composite; **c** post-event image (June 2020) false-color composite. In the change map (**a**), areas on the lower range of values (blue) depict areas where vegetation has been negatively affected, and areas on the higher range of values (pink) depict areas where there has been vegetation gain; the green/orange areas have experienced little change. In the pre-event and post-event images (**b** and **c**), the green areas indicate vegetation, while the brown regions are barren ground

something more advanced, such as a time-series approach that uses more than two dates of imagery.

For this chapter, we will classify our image into categories using a simple, manual thresholding method, meaning we will decide the optimal values for when a pixel will be considered change or no-change in the image. Finding the ideal value is a considerable task and will be unique to each use case and set of inputs (e.g., the threshold values for a SWIR2 single-band change would be different from the thresholds for NDVI). For a look at a more-advanced method of thresholding, check out the thresholding methods in Chap. 42.

First, you will define two variables for the threshold values for gain and loss. Next, create a new image with a constant value of 0. This will be the basis of our classification. Reclassify the new image using the `where` function. Classify loss areas as 2 where the difference image is less than or equal to the loss threshold value. Reclassify gain areas to 1 where the difference image is greater than or equal to the gain threshold value. Finally, mask the image by itself and add the classified image to the map (Fig. 16.5). Note: It is not necessary to self-mask the image, and in many cases, you might be just as interested in areas that did not change as you are in areas that did.

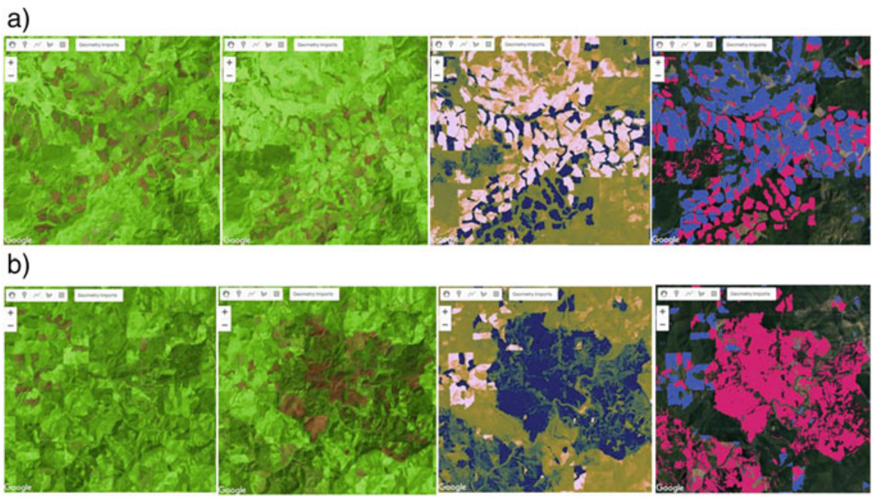


Fig. 16.5 **a** Change detection in timber forests of southern Oregon, including maps of the (left to right) pre-event false-color composite, post-event false-color composite, difference image, and classified change using NBR; **b** the same map types for an example of change caused by fire in southern Oregon. The false-color maps highlight vegetation in green and barren ground in brown. The difference images show NBR gain in pink to NBR loss in blue. The classified change images show NBR gain in blue and NBR loss in red

```
// Classify change
var thresholdGain = 0.10;
var thresholdLoss = -0.10;

var diffClassified = ee.Image(0);

diffClassified =
diffClassified.where(diff.lte(thresholdLoss), 2);
diffClassified =
diffClassified.where(diff.gte(thresholdGain), 1);

var changeVis = {
    palette: 'fcffc8,2659eb,fa1373',
    min: 0,
    max: 2
};

Map.addLayer(diffClassified.selfMask(),
    changeVis,
    'change classified by threshold');
```

Chapters 17 through 21 present more-advanced change detection algorithms that go beyond differencing and thresholding between two images, instead allowing you to analyze changes indicated across several images as a time series.

Code Checkpoint F44b. The book's repository contains a script that shows what your code should look like at this point.

16.3 Synthesis

Evaluating any maps you create, including change detection maps, is essential to determining whether the method you have selected is appropriate for informing land management and decision-making (Stehman and Czaplewski 1998), or whether you need to iterate on the mapping process to improve the final results. Maps generally, and change maps specifically, will always have errors. This is due to a suite of factors, such as the geometric registration between images, the calibration between images, the data resolution (e.g., temporal, spectral, radiometric) compared to the characteristics of the activity of interest, the complexity of the landscape of the study region (topography, atmospheric conditions, etc.), and the classification techniques employed (Lu et al. 2004). This means that similar studies can present different, sometimes controversial, conclusions about landscape dynamics (e.g., Cohen et al. 2017). In order to be useful for decision-making, a change detection mapping effort should provide the user with an understanding of the strengths and weaknesses of the product, such as by presenting omission and commission error rates. The quantification of classification quality is presented in Chap. 7.

Assignment 1. Try using a different index, such as NDVI or a Tasseled Cap Transformation, to run the change detection steps, and compare the results with those obtained from using NBR.

Assignment 2. Experiment with adjusting the `thresholdLoss` and `thresholdGain` values.

Assignment 3. Use what you have learned in the classification chapter (Chap. 6) to run a supervised classification on the difference layer (or layers, if you have created additional ones). Hint: To complete a supervised classification, you would need reference examples of both the stable and change classes of interest to train the classifier.

Assignment 4. Think about how things like clouds and cloud shadows could affect the results of change detection. What do you think the two-date differencing method would pick up for images in the same year in different seasons?

16.4 Conclusion

In this chapter, you learned how to make a change detection map using two-image differencing. The importance of visualizing changes in this way instead of using a post-classification comparison, where two classified maps are compared instead of two satellite images, is that it avoids multiplicative errors from the classifications and is better at observing more subtle changes in the landscape. You also learned that how you visualize your images and change maps—such as what band combinations and color ramps you select, and what threshold values you use for a classification map—has an impact on how easily and what types of changes can be seen.

References

Cohen WB, Healey SP, Yang Z et al (2017) How similar are forest disturbance maps derived from different Landsat time series algorithms? Forests 8:98. https://doi.org/10.3390/f8040098

Coppin P, Jonckheere I, Nackaerts K et al (2004) Digital change detection methods in ecosystem monitoring: a review. Int J Remote Sens 25:1565–1596. https://doi.org/10.1080/0143116031000101675

Crameri F, Shephard GE, Heron PJ (2020) The misuse of colour in science communication. Nat Commun 11:1–10. https://doi.org/10.1038/s41467-020-19160-7

Hansen MC, Potapov PV, Moore R et al (2013) High-resolution global maps of 21st-century forest cover change. Science 342:850–853. https://doi.org/10.1126/science.1244693

Kennedy RE, Townsend PA, Gross JE et al (2009) Remote sensing change detection tools for natural resource managers: understanding concepts and tradeoffs in the design of landscape monitoring projects. Remote Sens Environ 113:1382–1396. https://doi.org/10.1016/j.rse.2008.07.018

Lu D, Mausel P, Brondízio E, Moran E (2004) Change detection techniques. Int J Remote Sens 25:2365–2401. https://doi.org/10.1080/0143116031000139863

Macleod RD, Congalton RG (1998) A quantitative comparison of change-detection algorithms for monitoring eelgrass from remotely sensed data. Photogramm Eng Remote Sens 64:207–216

Singh A (1989) Digital change detection techniques using remotely-sensed data. Int J Remote Sens 10:989–1003. https://doi.org/10.1080/01431168908903939

Stehman SV, Czaplewski RL (1998) Design and analysis for thematic map accuracy assessment: fundamental principles. Remote Sens Environ 64:331–344. https://doi.org/10.1016/S0034-4257(98)00010-8

Woodcock CE, Loveland TR, Herold M, Bauer ME (2020) Transitioning from change detection to monitoring with remote sensing: a paradigm shift. Remote Sens Environ 238:111558. https://doi.org/10.1016/j.rse.2019.111558

Interpreting Annual Time Series with LandTrendr

17

Robert Kennedy, Justin Braaten, and Peter Clary

Overview
Time-series analysis of change can be achieved by fitting the entire spectral trajectory using simple statistical models. These allow us to both simplify the time series and to extract useful information about the changes occurring. In this chapter, you will get an introduction to the use of LandTrendr, one of these time-series approaches used to characterize time series of spectral values.

Learning Outcomes

- Evaluating yearly time-series spectral values to distinguish between true change and artifacts.
- Recognizing disturbance and growth signals in the time series of annual spectral values for individual pixels.
- Interpreting change segments and translating them to maps.
- Applying parameters in a graphical user interface to create disturbance maps in forests.

R. Kennedy (✉) · P. Clary
Geography Program, College of Earth Ocean and Atmospheric Sciences, Oregon State University, Corvallis, OR 97331, USA
e-mail: robert.kennedy@oregonstate.edu

P. Clary
e-mail: clarype@oregonstate.edu

J. Braaten
Google, Mountain View, CA 97331, USA
e-mail: braaten@google.com

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_17

Assumes you know how to

- Calculate and interpret vegetation indices (Chap. 5)
- Interpret bands and indices in terms of land surface characteristics (Chap. 5).

17.1 Introduction to Theory

Land surface change happens all the time, and satellite sensors witness it. If a spectral index is chosen to match the type of change being sought, surface change can be inferred from changes in spectral index values. Over time, the progression of spectral values witnessed in each pixel tells a story of the processes of change, such as growth and disturbance. Time-series algorithms are designed to leverage many observations of spectral values over time to isolate and describe changes of interest, while ignoring uninteresting change or noise.

In this lab, we use the LandTrendr time-series algorithms to map change. The LandTrendr algorithms apply "temporal segmentation" strategies to distill a multiyear time series into sequential straight-line segments that describe the change processes occurring in each pixel. We then isolate the segment of interest in each pixel and make maps of when, how long, and how intensely each process occurred. Similar strategies can be applied to more complicated descriptions of the time series, as is seen in some of the chapters that follow this one.

17.2 Practicum

For this lab, we will use a graphical user interface (GUI) to teach the concepts of LandTrendr.

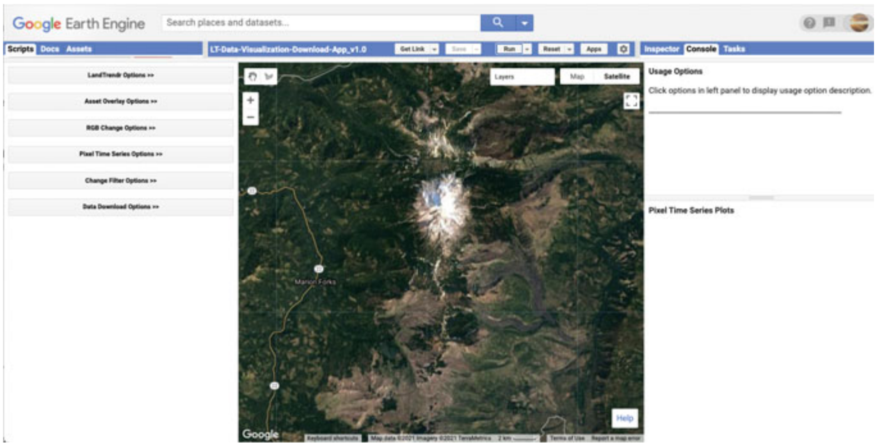
Code Checkpoint F45a. The book's repository contains information about accessing the LandTrendr interface.

17.2.1 Section 1: Pixel Time Series

When working with LandTrendr for the first time in your area, there are two questions you must address.

First, is the change of interest detectable in the spectral reflectance record? If the change you are interested in does not leave a pattern in the spectral reflectance record, then an algorithm will not be able to find it.

Second, can you identify fitting parameters that allow the algorithm to capture that record? Time-series algorithms apply rules to a temporal sequence of spectral values in a pixel and simplify the many observations into more digestible forms, such as the linear segments we will work with using LandTrendr. The algorithms



Usage Options

Click options in left panel to display usage option description.

Fig. 17.1 LandTrendr GUI interface, with the control panel on the left, the **Map** panel in the center, and the reporting panel on the right

that do the simplification are often guided by parameters that control the way the algorithm does its job.

The best way to begin assessing these questions is to look at the time series of individual pixels. In Earth Engine, open and run the script that generates the GUI we have developed to easily deploy the LandTrendr algorithms (Kennedy et al. 2018). Run the script, and you should see an interface that looks like the one shown in Fig. 17.1.

The LandTrendr GUI consists of three panels: a control panel on the left, a reporting panel on the right, and a **Map** panel in the center. The control panel is where all of the functionality of the interface resides. There are several modules, and each is accessed by clicking on the double arrow to the right of the title. The **Map** panel defaults to a location in Oregon but can be manually moved anywhere in the world. The reporting panel shows messages about how to use functions, as well as providing graphical outputs.

Next, expand the "Pixel Time-Series Options" function. For now, simply use your mouse to click somewhere on the map. *Wait a few seconds even though it looks like nothing is happening—be patient!!* The GUI has sent information to Earth Engine to run the LandTrendr algorithms at the location you have clicked and is waiting for the results. Eventually, you should see a chart appear in the reporting panel on the right. Figure 17.2 shows what one pixel looks like in an area where the forest burned and began regrowth. Your chart will probably look different.

The key to success with the LandTrendr algorithm is interpreting these time series. First, let us examine the components of the chart. The *x*-axis shows the year of observation. With LandTrendr, only one observation per year is used to describe the history of a pixel; later, we will cover how you control that value. The *y*-axis shows the spectral value of the index that is chosen. In the default

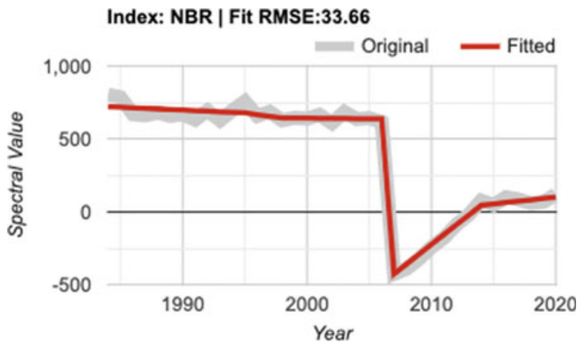


Fig. 17.2 Typical trajectory for a single pixel. The *x*-axis shows the year, the *y*-axis the spectral index value, and the title the index chosen. The gray line represents the original spectral values observed by Landsat, and the red line the result of the LandTrendr temporal segmentation algorithms

mode, the normalized burn ratio (as described in Chap. 16). Note that you also have the ability to pick more indices using the checkboxes on the control panel on the left. Note that we scale floating point (decimal) indices by 1000. Thus, an NBR value of 1.0 would be displayed as 1000.

In the chart area, the thick gray line represents the spectral values observed by the satellite for the period of the year selected for a single 30 m Landsat pixel at the location you have chosen. The red line is the output from the temporal segmentation that is the heart of the LandTrendr algorithms. The title of the chart shows the spectral index, as well as the root-mean-square error of the fit.

To interpret the time series, first know which way is "up" and "down" for the spectral index you are interested in. For the NBR, the index goes up in value when there is more vegetation and less soil in a pixel. It goes down when there is less vegetation. For vegetation disturbance monitoring, this is useful.

Next, translate that change into the changes of interest for the change processes you are interested in. For conifer forest systems, the NBR is useful because it drops precipitously when a disturbance occurs, and it rises as vegetation grows.

In the case of Fig. 17.2, we interpret the abrupt drop as a disturbance, and the subsequent rise of the index as regrowth or recovery (though not necessarily to the same type of vegetation) (Fig. 17.3).

Tip: LandTrendr is able to accept any index, and advanced users are welcome to use indices of their own design. An important consideration is knowing which direction indicates "recovery" and "disturbance" for the topic you are interested in. The algorithms favor detection of disturbance and can be controlled to constrain how quickly recovery is assumed to occur (see parameters below).

For LandTrendr to have any hope of finding the change of interest, that change must be manifested in the gray line showing the original spectral values. If you know that some process is occurring and it is not evident in the gray line, what can you do?

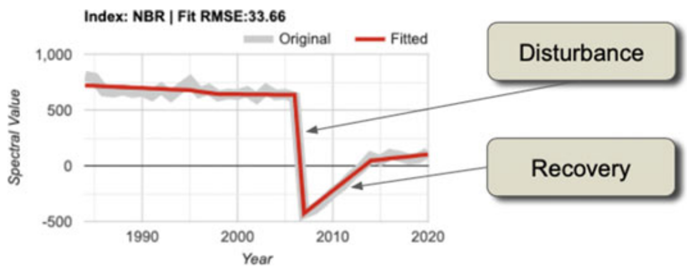


Fig. 17.3 For the trajectory in Fig. 17.2, we can identify a segment capturing disturbance based on its abrupt drop in the NBR index, and the subsequent vegetative recovery

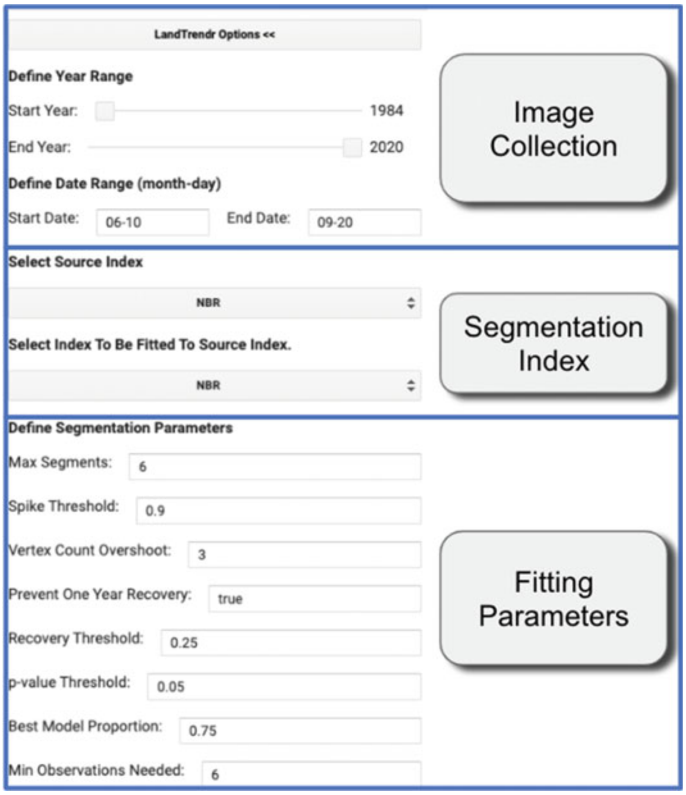
One option is to change the index. Any single index is simply one view of the larger spectral space of the Landsat Thematic Mapper sensors. The change you are interested in may cause spectral change in a different direction than that captured with some indices. Try choosing different indices from the list. If you click on different checkboxes and re-submit the pixel, the fits for all of the different indices will appear.

Another option is to change the date range. LandTrendr uses one value per year, but the value that is chosen can be controlled by the user. It is possible that the change of interest is better identified in some seasons than others. We use a *medoid* image compositing approach, which picks the best single observation each year from a date range of images in an `ImageCollection`. In the GUI, you can change the date range of imagery used for compositing in the **Image Collection** portion of the **LandTrendr Options** menu (Fig. 17.4).

Change the **Start Date** and **End Date** to find a time of year when the distinction between cover conditions before and during the change process of interest is greatest.

There are other considerations to keep in mind. First, seasonality of vegetation, water, or snow often can affect the signal of the change of interest. And because we use an `ImageCollection` that spans a range of dates, it is best to choose a date range where there is not likely to be a substantial change in vegetative state from the beginning to the end of the date range. Clouds can be a factor too. Some seasons will have more cloudiness, which can make it difficult to find good images. Often with optical sensors, we are constrained to working with periods where clouds are less prevalent, or using wide date ranges to provide many opportunities for a pixel to be cloud-free.

It is possible that no combination of index or data range is sensitive to the change of interest. If that is the case, there are two options: try using a different sensor and change detection technique, or accept that the change is not discernible. This can often occur if the change of interest occupies a small portion of a given 30 m by 30 m Landsat pixel, or if the spectral manifestation of the change is so subtle that it is not spectrally separable from non-changed pixels.



Define Year Range

Start Year: 1984

End Year: 2020

Define Date Range (month-day)

Start Date: 06-10 End Date: 09-20

Select Source Index

NBR

Select Index To Be Fitted To Source Index.

NBR

Define Segmentation Parameters

Max Segments:	6
Spike Threshold:	0.9
Vertex Count Overshoot:	3
Prevent One Year Recovery:	true
Recovery Threshold:	0.25
p-value Threshold:	0.05
Best Model Proportion:	0.75
Min Observations Needed:	6

Fig. 17.4 LandTrendr options menu. Users control the year and date range in the **Image Collection** section, the index used for temporal segmentation in the middle section, and the parameters controlling the temporal segmentation in the bottom section

Even if you as a human can identify the change of interest in the spectral trajectory of the gray line, an algorithm may not be able to similarly track it. To give the algorithm a fighting chance, you need to explore whether different fitting parameters could be used to match the red fitted line with the gray source image line.

The overall fitting process includes steps to reduce noise and best identify the underlying signal. The temporal segmentation algorithms are controlled by fitting parameters that are described in detail in Kennedy et al. (2010). You adjust these parameters using the **Fitting Parameters** block of the LandTrendr **Options** menu. Below is a brief overview of what values are often useful, but these will likely change as you use different spectral indices.

First, the *minimum observations needed* criterion is used to evaluate whether a given trajectory has enough unfiltered (i.e., clear observation) years to run the fitting. We suggest leaving this at the default of 6.

The segmentation begins with a noise-dampening step to remove spikes that could be caused by unfiltered clouds or shadows. The *spike threshold* parameter controls the degree of filtering. A value of 1.0 corresponds to no filtering, and lower values corresponding to more severe filtering. We suggest leaving this at 0.9; if changed, a range from 0.7 to 1.0 is appropriate.

The next step is finding vertices. This begins with the start and end year as vertex years, progressively adding candidate vertex years based on deviation from linear fits. To avoid getting an overabundance of vertex years initially found using this method, we suggest leaving the *vertex count overshoot* at a value of 3. A second set of algorithms uses deflection angle to cull back this overabundance to a set number of maximum candidate vertex years.

That number of vertex years is controlled by the *max_segments* parameter. As a general rule, your number of segments should be no more than one-third of the total number of likely yearly observations. The years of these vertices (*X*-values) are then passed to the model-building step. Assuming you are using at least 30 years of the archive, and your area has reasonable availability of images, a value of 8 is a good starting point.

In the model-building step, straight-line segments are built by fitting *Y*-values (spectral values) for the periods defined by the vertex years (*X*-values). The process moves from left to right—early years to late years. Regressions of each subsequent segment are connected to the end of the prior segment. Regressions are also constrained to prevent unrealistic recovery after disturbance, as controlled by the *recovery threshold* parameter. A lower value indicates greater constraint: a value of 1.0 means the constraint is turned off; a value of 0.25 means that segments that fully recover in faster than four years ($4 = 1/0.25$) are not permitted. Note: This parameter has strong control on the fitting and is one of the first to explore when testing parameters. Additionally, the *preventOneYearRecovery* will disallow fits that have one-year-duration recovery segments. This may be useful to prevent overfitting of noisy data in environments where such quick vegetative recovery is not ecologically realistic.

Once a model of the maximum number of segments is found, successively simpler models are made by iteratively removing the least informative vertex. Each model is scored using a pseudo-*f* statistic, which penalizes models with more segments, to create a pseudo *p*-value for each model. The *p-value threshold* parameter is used to identify all fits that are deemed good enough. Start with a value of 0.05, but check to see if the fitted line appears to capture the salient shape and features of the gray source trajectory. If you see temporal patterns in the gray line that are likely not noise (based on your understanding of the system under study), consider switching the *p-value threshold* to 0.10 or even 0.15.

Note: Because of temporal autocorrelation, these cannot be interpreted as true *f*- and *p*-values, but rather as relative scalars to distinguish goodness of fit among models. If no good models can be found using these criteria based on the *p*-value parameter set by the user, a second approach is used to solve for the *Y*-value of all vertex years simultaneously. If no good model is found, then a straight-line mean value model is used.

From the models that pass the *p*-value threshold, one is chosen as the final fit. It may be the one with the lowest *p*-value. However, an adjustment is made to allow more complicated models (those with more segments) to be picked even if their *p*-value is within a defined proportion of the best-scoring model. That proportion is set by the *best model proportion* parameter. As an example, a *best model proportion* value of 0.75 would allow a more complicated model to be chosen if its score were greater than 75% that of the best model.

17.2.2 Section 2: Translating Pixels to Maps

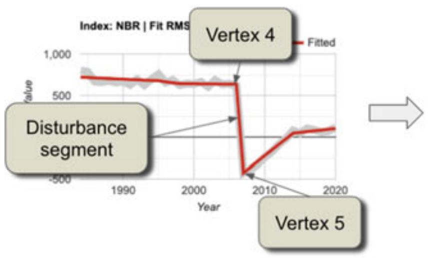
Although the full time series is the best description of each pixel's "life history," we typically are interested in the behavior of *all* of the pixels in our study area. It would be both inefficient to manually visualize all of them and ineffective to try to summarize areas and locations. Thus, we seek to make maps.

There are three post-processing steps to convert a segmented trajectory to a map. First, we identify segments of interest; if we are interested in disturbance, we find segments whose spectral change indicates loss. Second, we filter out segments of that type that do not meet criteria of interest. For example, very low magnitude disturbances can occur when the algorithm mistakenly finds a pattern in the random noise of the signal, and thus, we do not want to include it. Third, we extract from the segment of interest something about its character to map on a pixel-by-pixel basis: its start year, duration, spectral value, or the value of the spectral change.

Theory: We will start with a single pixel to learn how to interpret a disturbance pixel time series in terms of the dominant disturbance segment. For the disturbance time series, we have used in figures above, and we can identify the key parameters of the segment associated with the disturbance. For the example above, we have extracted the actual NBR values of the fitted time series and noted them in a table (Fig. 17.5). This is not part of the GUI—it is simply used here to work through the concepts.

From the table shown in Fig. 17.5, we can infer several key things about this pixel:

- It was likely disturbed between 2006 and 2007. This is because the NBR value drops precipitously in the segment bounded by vertices (breakpoints) in 2006 and 2007.
- The magnitude of spectral change was large: 1175 scaled NBR units out of a possible range of 2000 scaled units.
- There were small drops in NBR earlier, which may indicate some subtle loss of vegetation over a long period in the pixel. These drops, however, would need to be explored in a separate analysis because of their subtle nature.



	Year	NBR	Delta NBR
Vertex 1	1986	725	NA
Vertex 2	1996	700	-25
Vertex 3	1998	690	-10
Vertex 4	2006	685	-5
Vertex 5	2007	-490	-1175
Vertex 6	2013	50	540
Vertex 7	2020	100	50

Fig. 17.5 Tracking actual values of fitted trajectories to learn how we focus on quantification of disturbance. Because we know that the NBR index drops when vegetation is lost and soil exposure is increased, we know that a precipitous drop suggests an abrupt loss of vegetation. Although some early segments show very subtle change, only the segment between vertex 4 and 5 shows large-magnitude vegetation loss

- The main disturbance had a disturbance duration of just one year. This abruptness combined with the high magnitude suggests a major vegetative disturbance such as a harvest or a fire.
- The disturbance was then followed by recovery of vegetation, but not to the level before the disturbance. Note: Ecologists will recognize the growth signal as one of succession, or active revegetation by human intervention.

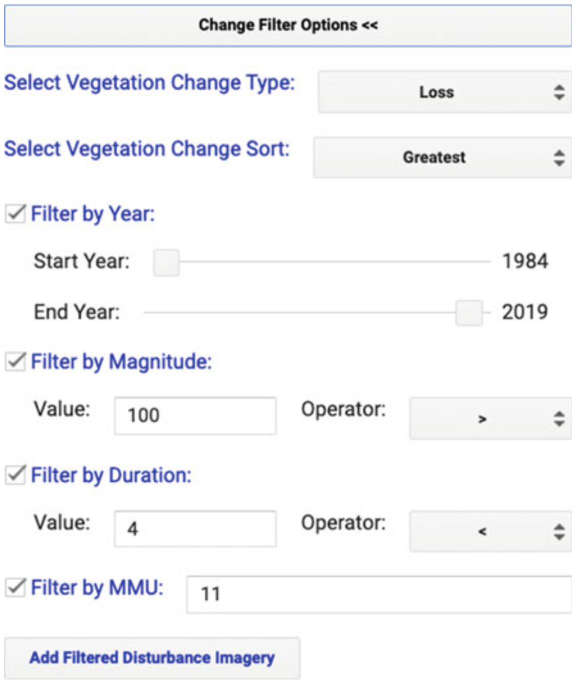
Following the three post-processing steps noted in the introduction to this section, to map the year of disturbance for this pixel, we would first identify the potential disturbance segments as those with negative NBR. Then, we would hone in on the disturbance of interest by filtering out potential disturbance segments that are not abrupt and/or of small magnitude. This would leave only the high-magnitude and short-duration segment. For that segment, the first year that we have evidence of disturbance is the first year after the start of the segment. The segment starts in 2006, which means that 2007 is the first year we have such evidence. Thus, we would assign 2007 to this pixel.

If we wanted to map the magnitude of the disturbance, we would follow the same first two steps but then report for the pixel value the magnitude difference between the starting and ending segment.

The LandTrendr GUI provides a set of tools to easily apply the same logic rules to all pixels of interest and create maps. Click on the **Change Filter Options** menu. The interface shown in Fig. 17.6 appears.

The first two sections are used to identify the segments of interest.

Select Vegetation Change Type offers the options of gain or loss, which refer to gain or loss of vegetation, with disturbance assumed to be related to loss of vegetation. Note: Advanced users can look in the landtrendr.js library in the "calcindex" function to add new indices with gain and loss defined as they choose.


Change Filter Options <<

Select Vegetation Change Type: Loss

Select Vegetation Change Sort: Greatest

✓ Filter by Year:

Start Year: 1984

End Year: 2019

✓ Filter by Magnitude:

Value: 100 Operator: >

✓ Filter by Duration:

Value: 4 Operator: <

✓ Filter by MMU: 11

Add Filtered Disturbance Imagery

Fig. 17.6 Menu used to post-process disturbance trajectories into maps. Select vegetation change type and sort to hone in on the segment type of interest, then check boxes to apply selective filters to eliminate uninteresting changes

The underlying algorithm is built to find disturbance in indices that increase when disturbance occurs, so indices such as NBR or NDVI need to be multiplied by (−1) before being fed to the LandTrendr algorithm. This is handled in the calcIndex function.

Select Vegetation Change Sort offers various options that allow you to choose the segment of interest based on timing or duration. By default, the greatest magnitude disturbance is chosen.

Each filter (magnitude, duration, etc.) is used to further winnow the possible segments of interest. All other filters are applied at the pixel scale, but **Filter by MMU** is applied to groups of pixels based on a given minimum mapping unit (MMU). Once all other filters have been defined, some pixels are flagged as being of interest, and others are not. The MMU filter looks to see how many connected pixels have been flagged as occurring in the same year, and omits groups smaller in pixel count than the number indicated here (which defaults to 11 pixels, or approximately 1 ha).

If you are following along and making changes, or if you are just using the default location and parameters, click the **Add Filtered Disturbance Imagery** to add this to the map. You should see something like Fig. 17.7.

There are multiple layers of disturbance added to the map. Use the map layers checkboxes to change which is shown. Magnitude of disturbance, for example,

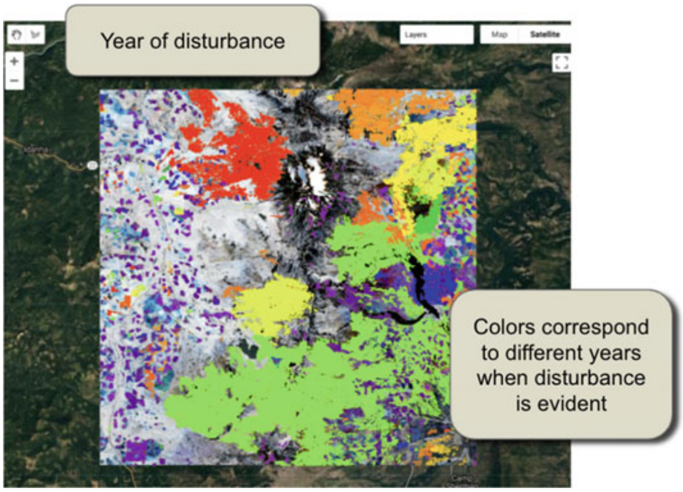

Year of disturbance
Colors correspond to different years when disturbance is evident
Layers
Map
Satellite

Fig. 17.7 Basic output from a disturbance mapping exercise

is a map of the delta change between beginning and endpoints of the segments (Fig. 17.8).

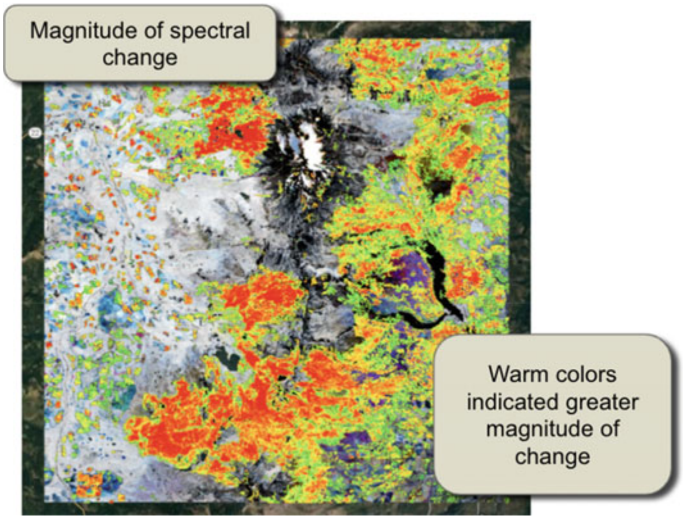

Magnitude of spectral change
Warm colors indicated greater magnitude of change

Fig. 17.8 Magnitude of change for the same area

17.3 Synthesis

In this chapter, you have learned how to work with annual time series to interpret regions of interest. Looking at annual snapshots of the landscape provides three key benefits: (1) the ability to view your area of interest without the clouds and noise that typically obscure single-day views; (2) gage the amount by which the noise-dampened signal still varies from year to year in response to large-scale forcing mechanisms; and (3) the ability to view the response of landscapes as they recover, sometimes slowly, from disturbance.

To learn more about LandTrendr, see the assignments below.

Assignment 1. Find your own change processes of interest. First, navigate the map (zooming and dragging) to an area of the world where you are interested in a change process, and the spectral index that would capture it. Make sure that the UI control panel is open to the **Pixel Time-Series Options** section. Next, click on the map in areas where you know change has occurred, and observe the spectral trajectories in the charts. Then, describe whether the change of interest is detectable in the spectral reflectance record and what are its characteristics in different parts of the study area.

Assignment 2. Find a pixel in your area of interest that shows a distinctive disturbance process, as you define it for your topic of interest. Adjust date ranges, parameters, etc., using the steps outlined in Sect. 17.2.1 above and then answer these questions:

- **Question 1**. Which index and date range did you use?
- **Question 2**. Did you need to change fitting parameters to make the algorithm find the disturbance? If so, which ones, and why?
- **Question 3**. How do you know this is a disturbance?

Assignment 3. Switch the control panel in the GUI to **Change Filter Options**, and use the guidance in Sect. 17.2.2 to set parameters and make disturbance maps.

- **Question 4**. Do the disturbance year and magnitude as mapped in the image match with what you would expect from the trajectory itself?
- **Question 5**. Can you change some of the filters to create a map where your disturbance process is not mapped? If so, what did you change?
- **Question 6**. Can you change filters to create a map that includes a different disturbance process, perhaps subtler, longer duration, etc.? Find a pixel and use the "pixel time-series" plotter to look at the time series of those processes.

Assignment 4. Return to the **Pixel Time-Series Options** section of the control panel, and navigate to a pixel in your area of interest that you believe would show a distinctive recovery or growth process, as you define it for your topic of interest. You may want to modify the index, parameters, etc., as covered in Sect. 17.2.1 to adequately capture the growth process with the fitted trajectories.

- **Question 7**. Did you use the same spectral index? If not, why?
- **Question 8**. Were the fitting parameters the same as those for disturbance? If not, what did you change, and why?
- **Question 9**. What evidence do you have that this is a vegetative growth signal?

Assignment 5. For vegetation gain mapping, switch the control panel back to **Change Filter Options** and use the guidance in Sect. 17.2.2 to set parameters, etc., to make maps of growth.

- **Question 10**. For the pixel or pixels you found for Assignment 3, does the year and magnitude as mapped in the "gain" image match with what you would expect from the trajectory itself?
- **Question 11**. Compare what the map looks like when you run it with and without the MMU filter. What differences do you see?
- **Question 12**. Try changing the recovery duration filter to a very high number (perhaps the full length of your archive) and to a very low number (say, one or two years). What differences do you see?

17.4 Conclusion

This exercise provides a baseline sense of how the LandTrendr algorithm works. The key points are learning how to interpret change in spectral values in terms of the processes occurring on the ground and then translating those into maps.

You can export the images you have made here using **Download Options**. Links to materials are available in the chapter checkpoints and LandTrendr documentation about both the GUI and the script-based versions of the algorithm. In particular, there are scripts that handle different components of the fitting and mapping process and that allow you to keep track of the fitting and image selection criteria.

References

Kennedy RE, Yang Z, Cohen WB (2010) Detecting trends in forest disturbance and recovery using yearly Landsat time series: 1. LandTrendr—temporal segmentation algorithms. Remote Sens Environ 114:2897–2910. https://doi.org/10.1016/j.rse.2010.07.008

Kennedy RE, Yang Z, Gorelick N et al (2018) Implementation of the LandTrendr algorithm on Google Earth Engine. Remote Sens 10:691. https://doi.org/10.3390/rs10050691

Fitting Functions to Time Series 18

Andréa Puzzi Nicolau, Karen Dyson, Biplov Bhandari, David Saah, and Nicholas Clinton

Overview
The purpose of this chapter is to establish a foundation for time-series analysis of remotely sensed data, which is typically arranged as an ordered stack of images. You will be introduced to the concepts of graphing time series, using linear modeling to detrend time series, and fitting harmonic models to time-series data. At the completion of this chapter, you will be able to perform analysis of multi-temporal data for determining trend and seasonality on a per-pixel basis.

A. P. Nicolau · K. Dyson · D. Saah
Spatial Informatics Group, Pleasanton, CA, USA
e-mail: apnicolau@sig-gis.com

K. Dyson
e-mail: kdyson@sig-gis.com

A. P. Nicolau · K. Dyson
SERVIR-Amazonia, Cali, Colombia

K. Dyson
Dendrolytics, Seattle, WA, USA

B. Bhandari
NASA SERVIR Science Coordination Office, Huntsville, AL, USA
e-mail: bb0134@uah.edu

University of Alabama in Huntsville, Huntsville, AL, USA

D. Saah (✉)
University of San Francisco, San Francisco, CA, USA
e-mail: dssaah@usfca.edu

N. Clinton
Google LLC, Mountain View, CA, USA
e-mail: nclinton@google.com

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_18

Learning Outcomes

- Graphing satellite imagery values across a time series.
- Quantifying and potentially removing linear trends in time series.
- Fitting linear and harmonic models to individual pixels in time-series data.

Assumes you know how to

- Import images and image collections, filter, and visualize (Part I).
- Perform basic image analysis: select bands, compute indices, create masks (Part II).
- Create a graph using `ui.Chart` (Chap. 4).
- Use `normalizedDifference` to calculate vegetation indices (Chap. 5).
- Write a function and `map` it over an `ImageCollection` (Chap. 12).
- Mask cloud, cloud shadow, snow/ice, and other undesired pixels (Chap. 15).

18.1 Introduction to Theory

Many natural and man-made phenomena exhibit important annual, interannual, or longer-term trends that recur—that is, they occur at roughly regular intervals. Examples include seasonality in leaf patterns in deciduous forests and seasonal crop growth patterns. Over time, indices such as the Normalized Difference Vegetation Index (NDVI) will show regular increase (e.g., leaf on, crop growth) and decrease (e.g., leaf-off, crop senescence), and typically have a long term, if noisy, trend such as a gradual increase in NDVI value as an area recovers from a disturbance.

Earth engine supports the ability to do complex linear and nonlinear regressions of values in each pixel of a study area. Simple linear regressions of indices can reveal linear trends that can span multiple years. Meanwhile, harmonic terms can be used to fit a sine wave-like curve. Once you have the ability to fit these functions to time series, you can answer many important questions. For example, you can define vegetation dynamics over multiple time scales, identify phenology and track changes year to year, and identify deviations from the expected patterns (Bradley et al. 2007; Bullock et al. 2020). There are multiple applications for these analyses. For example, algorithms to detect deviations from the expected pattern can be used to identify disturbance events, including deforestation and forest degradation (Bullock et al. 2020).

18.2 Practicum

18.2.1 Section 1: Multi-temporal Data in Earth Engine

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.

As explained in Chaps. 12 and 13, a time series in Earth Engine is typically represented as an `ImageCollection`. Because of image overlaps, cloud treatments, and filtering choices, an `ImageCollection` can have any of the following complex characteristics:

- At each pixel, there might be a distinct number of observations taken from a unique set of dates.
- The size (length) of the time series can vary across pixels.
- Data may be missing in any pixel at any point in the sequence (e.g., due to cloud masking).

The use of multi-temporal data in Earth Engine introduces two mind-bending concepts, which we will describe below.

Per-pixel curve fitting. As you have likely encountered in many settings, a function can be fit through a series of values. In the most familiar example, a function of the form $y = mx + b$ can represent a linear trend in data of all kinds. Fitting a straight "curve" with linear regression techniques involves estimating m and b for a set of x and y values. In a time series, x typically represents time, while y values represent observations at specific times. This chapter introduces how to estimate m and b for computed indices through time to model a potential linear trend in a time series. We then demonstrate how to fit a sinusoidal wave, which is useful for modeling rising and falling values, such as NDVI over a growing season. What can be particularly mind bending in this setting is the fact that when Earth Engine is asked to estimate values across a large area, it will fit a function *in every pixel* of the study area. Each pixel, then, has its own m and b values, determined by the number of observations in that pixel, the observed values, and the dates for which they were observed.

Higher-dimension band values: array images. That more complex conception of the potential information contained in a single pixel can be represented in a higher-order Earth Engine structure: the *array image*. As you will encounter in this lab, it is possible for a single pixel in a single band of a single image to contain *more than one value*. If you choose to implement an array image, a single pixel might contain a one-dimensional vector of numbers, perhaps holding the slope and intercept values resulting from a linear regression, for example. Other examples,

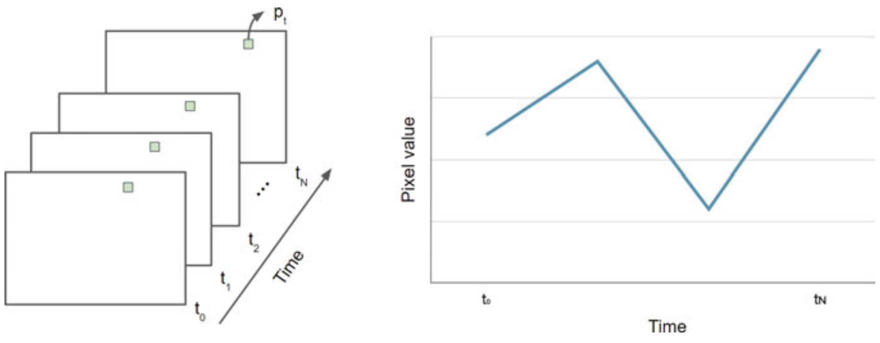


Fig. 18.1 Time series representation of pixel *p*

outside the scope of this chapter but used in the next chapter, might employ a two-dimensional matrix of values for each pixel within a single band of an image. Higher-order dimensions are available, as well as array image manipulations borrowed from the world of matrix algebra. Additionally, there are functions to move between the multidimensional array image structure and the more familiar, more easily displayed, simple `Image` type. Some of these array image functions were encountered in Chap. 9, but with less explanatory context.

First, we will give some very basic notation (Fig. 18.1). A scalar pixel at time t is given by p_t, and a pixel vector by $\mathbf{p}_t$. A variable with a "hat" represents an estimated value: in this context, $\hat{p}_t$ is the estimated pixel value at time t. A time series is a collection of pixel values, usually sorted chronologically: $\{\mathbf{p}_t; t = t_0 \ldots t_N\}$, where t might be in any units, t_0 is the smallest, and t_N is the largest such t in the series.

18.2.2 Section 2: Data Preparation and Preprocessing

The first step in analysis of time-series data is to import data of interest and plot it at an interesting location. We will work with the USGS Landsat 8 Level 2, Collection 2, Tier 1 `ImageCollection` and a cloud-masking function (Chap. 15), scale the image values, and add variables of interest to the collection as bands. Copy and paste the code below to filter the Landsat 8 collection to a point of interest over California (variable `roi`) and specific dates, and to apply the defined function. The variables of interest added by the function are: (1) NDVI (Chap. 5), (2) a time variable that is the difference between the image's current year and the year 1970 (a start point), and (3) a constant variable with value 1.

```
//////////////////// Sections 1 & 2 ////////////////////////////

// Define function to mask clouds, scale, and add variables
// (NDVI, time and a constant) to Landsat 8 imagery.
function maskScaleAndAddVariable(image) {
    // Bit 0 - Fill
    // Bit 1 - Dilated Cloud
    // Bit 2 - Cirrus
    // Bit 3 - Cloud
    // Bit 4 - Cloud Shadow
    var qaMask =
image.select('QA_PIXEL').bitwiseAnd(parseInt('11111',
        2)).eq(0);
    var saturationMask = image.select('QA_RADSAT').eq(0);

    // Apply the scaling factors to the appropriate bands.
    var opticalBands =
image.select('SR_B.').multiply(0.0000275).add(-
        0.2);
    var thermalBands = image.select('ST_B.*').multiply(0.00341802)
        .add(149.0);

    // Replace the original bands with the scaled ones and apply the
masks.
    var img = image.addBands(opticalBands, null, true)
        .addBands(thermalBands, null, true)
        .updateMask(qaMask)
        .updateMask(saturationMask);
    var imgScaled = image.addBands(img, null, true);

    // Now we start to add variables of interest.
    // Compute time in fractional years since the epoch.
    var date = ee.Date(image.get('system:time_start'));
    var years = date.difference(ee.Date('1970-01-01'), 'year');
    // Return the image with the added bands.
    return imgScaled
        // Add an NDVI band.
        .addBands(imgScaled.normalizedDifference(['SR_B5', 'SR_B4'])
            .rename('NDVI'))
        // Add a time band.
        .addBands(ee.Image(years).rename('t'))
        .float()
        // Add a constant band.
        .addBands(ee.Image.constant(1));
}

// Import point of interest over California, USA.
var roi = ee.Geometry.Point([-121.059, 37.9242]);

// Import the USGS Landsat 8 Level 2, Collection 2, Tier 1 image
collection),
// filter, mask clouds, scale, and add variables.
var landsat8sr = ee.ImageCollection('LANDSAT/LC08/C02/T1_L2')
    .filterBounds(roi)
    .filterDate('2013-01-01', '2018-01-01')
    .map(maskScaleAndAddVariable);

// Set map center over the ROI.
Map.centerObject(roi, 6);
```

Next, to visualize the NDVI at the point of interest over time, copy and paste the code below to print a chart of the time series (Chap. 4) at the location of interest (Fig. 18.2).

```
// Plot a time series of NDVI at a single location.
var landsat8Chart =
ui.Chart.image.series(landsat8sr.select('NDVI'), roi)
    .setChartType('ScatterChart')
    .setOptions({
        title: 'Landsat 8 NDVI time series at ROI',
        lineWidth: 1,
        pointSize: 3,
    });
print(landsat8Chart);
```

We can add a linear trend line to our chart using the `trendlines` parameters in the `setOptions` function for image series charts. Copy and paste the code below to print the same chart but with a linear trend line plotted (Fig. 18.3). In the next section, you will learn how to estimate linear trends over time.

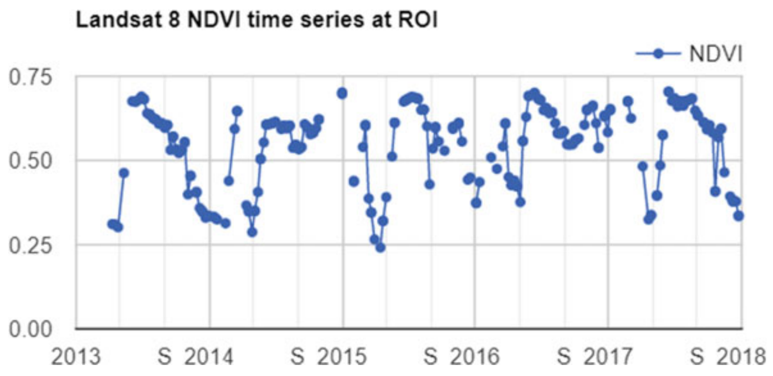


Fig. 18.2 Time-series representation of pixel *p*

```
// Plot a time series of NDVI with a linear trend line
// at a single location.
var landsat8ChartTL =
ui.Chart.image.series(landsat8sr.select('NDVI'), roi)
    .setChartType('ScatterChart')
    .setOptions({
        title: 'Landsat 8 NDVI time series at ROI',
        trendlines: {
            0: {
                color: 'CC0000'
            }
        },
        lineWidth: 1,
        pointSize: 3,
    });
print(landsat8ChartTL);
```

Now that we have plotted and visualized the data, lots of interesting analyses can be done to the time series by harnessing Earth Engine tools for fitting curves through this data. We will see a couple of examples in the following sections.

Code Checkpoint F46a. The book’s repository contains a script that shows what your code should look like at this point.

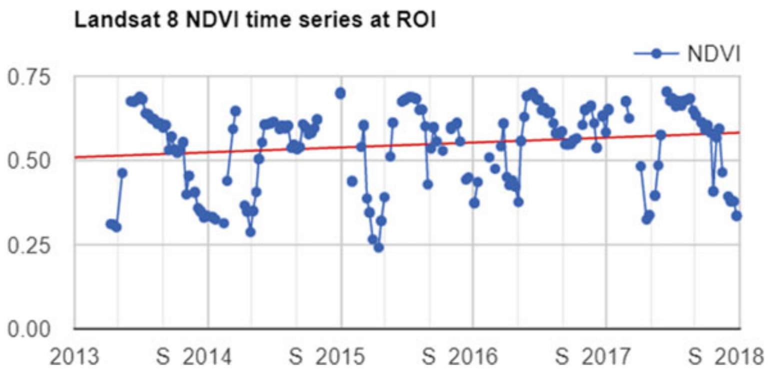


Fig. 18.3 Time series representation of pixel *p* with the trend line in red

18.2.3 Section 3: Estimating Linear Trend Over Time

Time-series datasets may contain not only trends but also seasonality, both of which may need to be removed prior to modeling. Trends and seasonality can result in a varying mean and a varying variance over time, both of which define a time series as non-stationary. Stationary datasets, on the other hand, have a stable mean and variance, and are therefore much easier to model.

Consider the following linear model, where e_t is a random error:

$$p_t = \beta_0 + \beta_1 t + e_t \tag{18.1}$$

This is the model behind the trend line added to the chart created in the previous section (Fig. 18.3). Identifying trends at different scales is a big topic, with many approaches being used (e.g., differencing, modeling).

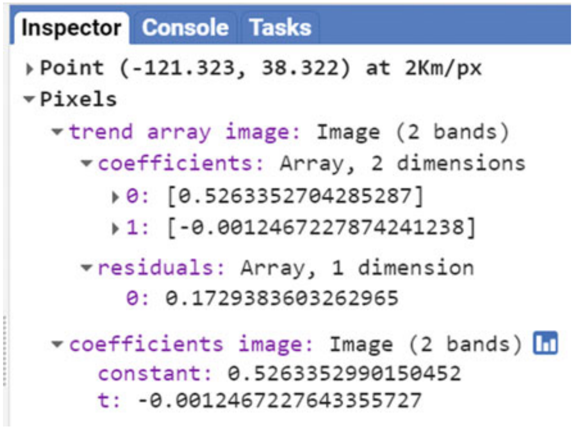
Removing unwanted to uninteresting trends for a given problem is often a first step to understand complex patterns in time series. There are several approaches to remove trends. Here, we will remove the linear trend that is evident in the data shown in Fig. 18.3 using Earth Engine's built-in tools for regression modeling. This approach is a useful, straightforward way to detrend data in time series (Shumway and Stoffer 2019). Here, the goal is to discover the values of the β's in Eq. 18.1 for each pixel.

Copy and paste code below into the Code Editor, adding it to the end of the script from the previous section. Running this code will fit this trend model to the Landsat-based NDVI series using ordinary least squares, using the `linearRegression` reducer (Chap. 8).

```
///////////////////// Section 3 /////////////////////////////

// List of the independent variable names
var independents = ee.List(['constant', 't']);

// Name of the dependent variable.
var dependent = ee.String('NDVI');

// Compute a linear trend.  This will have two bands:
'residuals' and
// a 2x1 (Array Image) band called 'coefficients'.
// (Columns are for dependent variables)
var trend = landsat8sr.select(independents.add(dependent))

.reduce(ee.Reducer.linearRegression(independents.length(),
1));
Map.addLayer(trend, {}, 'trend array image');

// Flatten the coefficients into a 2-band image.
var coefficients = trend.select('coefficients')
    // Get rid of extra dimensions and convert back to a
regular image
    .arrayProject([0])
    .arrayFlatten([independents]);
Map.addLayer(coefficients, {}, 'coefficients image');
```

If you click over a point using the **Inspector** tab, you will see the pixel values for the array image (coefficients "*t*" and "constant", and residuals) and two-band image (coefficients "*t*" and "constant") (Fig. 18.4).

Now, copy and paste the code below to use the model to detrend the original NDVI time series and plot the time series chart with the `trendlines` parameter (Fig. 18.5).


Inspector Console Tasks

```
▸Point (-121.323, 38.322) at 2Km/px
▾Pixels
  ▾trend array image: Image (2 bands)
    ▾coefficients: Array, 2 dimensions
      ▸0: [0.5263352704285287]
      ▸1: [-0.0012467227874241238]
    ▾residuals: Array, 1 dimension
       0: 0.1729383603262965
  ▾coefficients image: Image (2 bands)
     constant: 0.5263352990150452
     t: -0.0012467227643355727
```

Fig. 18.4 Pixel values of array image and coefficients image

```
// Compute a detrended series.
var detrended = landsat8sr.map(function(image) {
    return image.select(dependent).subtract(

image.select(independents).multiply(coefficients)
            .reduce('sum'))
        .rename(dependent)
        .copyProperties(image, ['system:time_start']);
});

// Plot the detrended results.
var detrendedChart = ui.Chart.image.series(detrended, roi,
null, 30)
    .setOptions({
        title: 'Detrended Landsat time series at ROI',
        lineWidth: 1,
        pointSize: 3,
        trendlines: {
            0: {
                color: 'CC0000'
            }
        },
    });
print(detrendedChart);
```

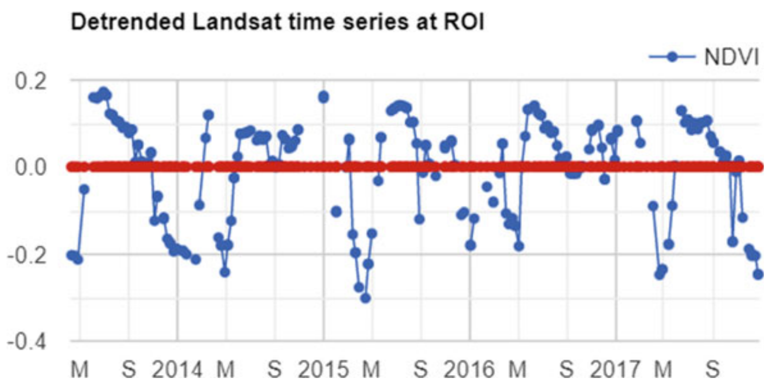


Fig. 18.5 Detrended NDVI time series

Code Checkpoint F46b. The book's repository contains a script that shows what your code should look like at this point.

18.2.4 Section 4: Estimating Seasonality with a Harmonic Model

A linear trend is one of several possible types of trends in time series. Time series can also present harmonic trends, in which a value goes up and down in a predictable wave pattern. These are of particular interest and usefulness in the natural world, where harmonic changes in greenness of deciduous vegetation can occur across the spring, summer, and autumn. Now, we will return to the initial time series (`landsat8sr`) of Fig. 18.2 and fit a harmonic pattern through the data. Consider the following harmonic model, where A is amplitude, ω is frequency, φ is phase, and e_t is a random error.

$$\begin{aligned} p_t &= \beta_0 + \beta_1 t + A\cos(2\pi\omega t - \varphi) + e_t \\ &= \beta_0 + \beta_1 t + \beta_2\cos(2\pi\omega t) + \beta_3\sin(2\pi\omega t) + e_t \end{aligned} \tag{18.2}$$

Note that $\beta_2 = A\,\cos(\varphi)$ and $\beta_3 = A\,\sin(\varphi)$, implying $A = ({\beta_2}^2 + {\beta_3}^2)^{½}$ and $\varphi = \mathrm{atan}(\beta_3/\beta_2)$ (as described in Shumway and Stoffer 2019). To fit this model to an annual time series, set $\omega = 1$ (one cycle per year) and use ordinary least squares regression.

The setup for fitting the model is to first add the harmonic variables (the third and fourth terms of Eq. 18.2) to the `ImageCollection`. Then, fit the model as with the linear trend, using the `linearRegression` reducer, which will yield a 4×1 array image.

```
///////////////////// Section 4 /////////////////////////////

// Use these independent variables in the harmonic
regression.
var harmonicIndependents = ee.List(['constant', 't', 'cos',
'sin']);

// Add harmonic terms as new image bands.
var harmonicLandsat = landsat8sr.map(function(image) {
    var timeRadians = image.select('t').multiply(2 *
Math.PI);
    return image
        .addBands(timeRadians.cos().rename('cos'))
        .addBands(timeRadians.sin().rename('sin'));
});

// Fit the model.
var harmonicTrend = harmonicLandsat
    .select(harmonicIndependents.add(dependent))
    // The output of this reducer is a 4x1 array image.

.reduce(ee.Reducer.linearRegression(harmonicIndependents.leng
th(),
        1));
```

Now, copy and paste the code below to plug the coefficients into Eq. 18.2 in order to get a time series of fitted values and plot the harmonic model time series (Fig. 18.6).

```
// Turn the array image into a multi-band image of coefficients.
var harmonicTrendCoefficients =
harmonicTrend.select('coefficients')
    .arrayProject([0])
    .arrayFlatten([harmonicIndependents]);

// Compute fitted values.
var fittedHarmonic = harmonicLandsat.map(function(image) {
    return image.addBands(
        image.select(harmonicIndependents)
        .multiply(harmonicTrendCoefficients)
        .reduce('sum')
        .rename('fitted'));
});

// Plot the fitted model and the original data at the ROI.
print(ui.Chart.image.series(
        fittedHarmonic.select(['fitted', 'NDVI']), roi,
ee.Reducer
        .mean(), 30)
    .setSeriesNames(['NDVI', 'fitted'])
    .setOptions({
        title: 'Harmonic model: original and fitted values',
        lineWidth: 1,
        pointSize: 3,
    }));
```

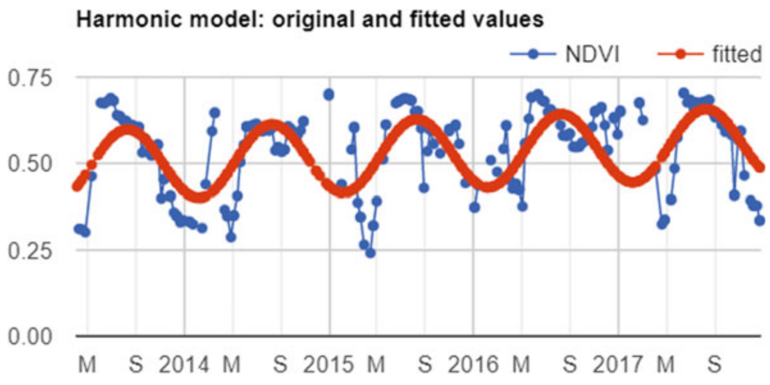


Fig. 18.6 Harmonic model of NDVI time series

Returning to the mind-bending nature of curve fitting, it is worth remembering that the harmonic waves seen in Fig. 18.6 are the fit of the data to a *single point* across the image. Next, we will map the outcomes of millions of these fits, pixel-by-pixel, across the entire study area.

We'll compute and map the phase and amplitude of the estimated harmonic model for each pixel. Phase and amplitude (Fig. 18.7) can give us additional information to facilitate remote sensing applications such as agricultural mapping and

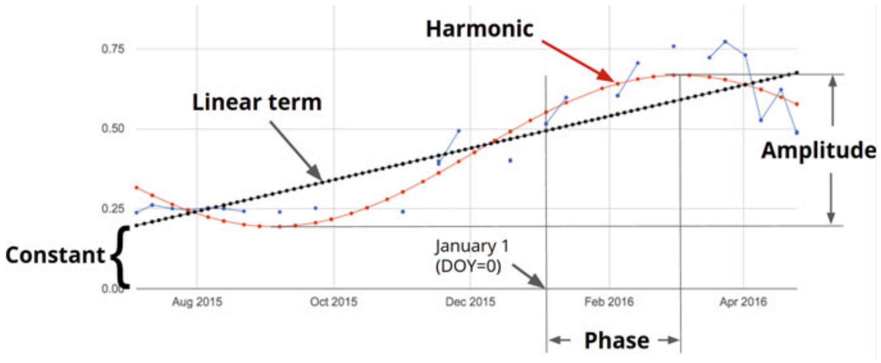


Fig. 18.7 Example of phase and amplitude in harmonic model

land use and land cover monitoring. Agricultural crops with different phenological cycles can be distinguished with phase and amplitude information, something that perhaps would not be possible with spectral information alone.

Copy and paste the code below to compute phase and amplitude from the coefficients and add this image to the map (Fig. 18.8).

```
// Compute phase and amplitude.
var phase = harmonicTrendCoefficients.select('sin')
    .atan2(harmonicTrendCoefficients.select('cos'))
    // Scale to [0, 1] from radians.
    .unitScale(-Math.PI, Math.PI);

var amplitude = harmonicTrendCoefficients.select('sin')
    .hypot(harmonicTrendCoefficients.select('cos'))
    // Add a scale factor for visualization.
    .multiply(5);

// Compute the mean NDVI.
var meanNdvi = landsat8sr.select('NDVI').mean();

// Use the HSV to RGB transformation to display phase and
amplitude.
var rgb = ee.Image.cat([
    phase, // hue
    amplitude, // saturation (difference from white)
    meanNdvi // value (difference from black)
]).hsvToRgb();

Map.addLayer(rgb, {}, 'phase (hue), amplitude (sat), ndvi
(val)');
```

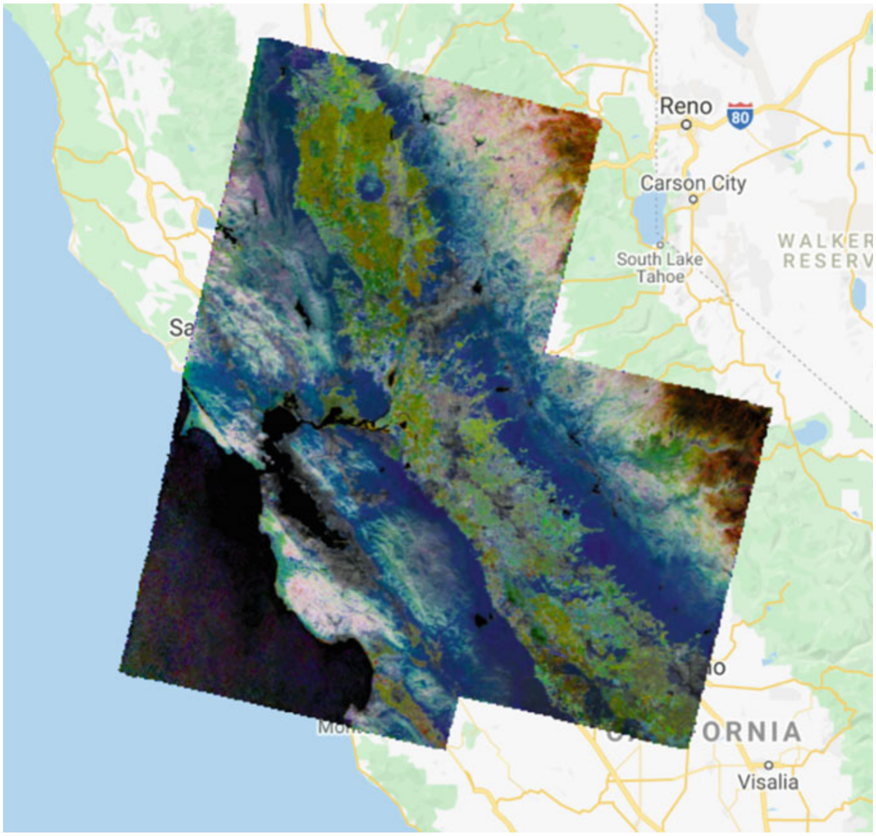


Fig. 18.8 Phase, amplitude, and NDVI concatenated image

The code uses the HSV to RGB transformation `hsvToRgb` for visualization purposes (Chap. 9). We use this transformation to separate color components from intensity for a better visualization. Without this transformation, we would visualize a very colorful image that would not look as intuitive as the image with the transformation. With this transformation, phase, amplitude, and mean NDVI are displayed in terms of hue (color), saturation (difference from white), and value (difference from black), respectively. Therefore, darker pixels are areas with low NDVI. For example, water bodies will appear as black, since NDVI values are zero or negative. The different colors are distinct phase values, and the saturation of the color refers to the amplitude: whiter colors mean amplitude closer to zero (e.g., forested areas), and the more vivid the colors, the higher the amplitude (e.g., croplands). Note that if you use the **Inspector** tool to analyze the values of a pixel, you will not get values of phase, amplitude, and NDVI, but the transformed values into values of blue, green, and red colors.

Code Checkpoint F46c. The book's repository contains a script that shows what your code should look like at this point.

18.2.5 Section 5: An Application of Curve Fitting

The rich data about the curve fits can be viewed in a multitude of different ways. Add the code below to your script to produce the view in Fig. 18.9. The image will be a close-up of the area around Modesto, California.

```
///////////////////// Section 5 /////////////////////////////

// Import point of interest over California, USA.
var roi = ee.Geometry.Point([-121.04, 37.641]);

// Set map center over the ROI.
Map.centerObject(roi, 14);

var trend0D = trend.select('coefficients').arrayProject([0])
    .arrayFlatten([independents]).select('t');

var anotherView =
ee.Image(harmonicTrendCoefficients.select('sin'))
    .addBands(trend0D)
    .addBands(harmonicTrendCoefficients.select('cos'));

Map.addLayer(anotherView,
    {
        min: -0.03,
        max: 0.03
    },
    'Another combination of fit characteristics');
```

The upper image in Fig. 18.9 is a closer view of Fig. 18.8, showing an image that transforms the sine and cosine coefficient values and incorporates information from the mean NDVI. The lower image draws the sine and cosine in the red and blue bands, and extracts the slope of the linear trend that you calculated earlier in the chapter, placing that in the green band. The two views of the fit are similarly structured in their spatial pattern—both show fields to the west and the city to the east. But the pixel-by-pixel variability emphasizes a key point of this chapter: that a fit to the NDVI data is done independently in each pixel in the image. Using different elements of the fit, these two views, like other combinations of the data you might imagine, can reveal the rich variability of the landscape around Modesto.

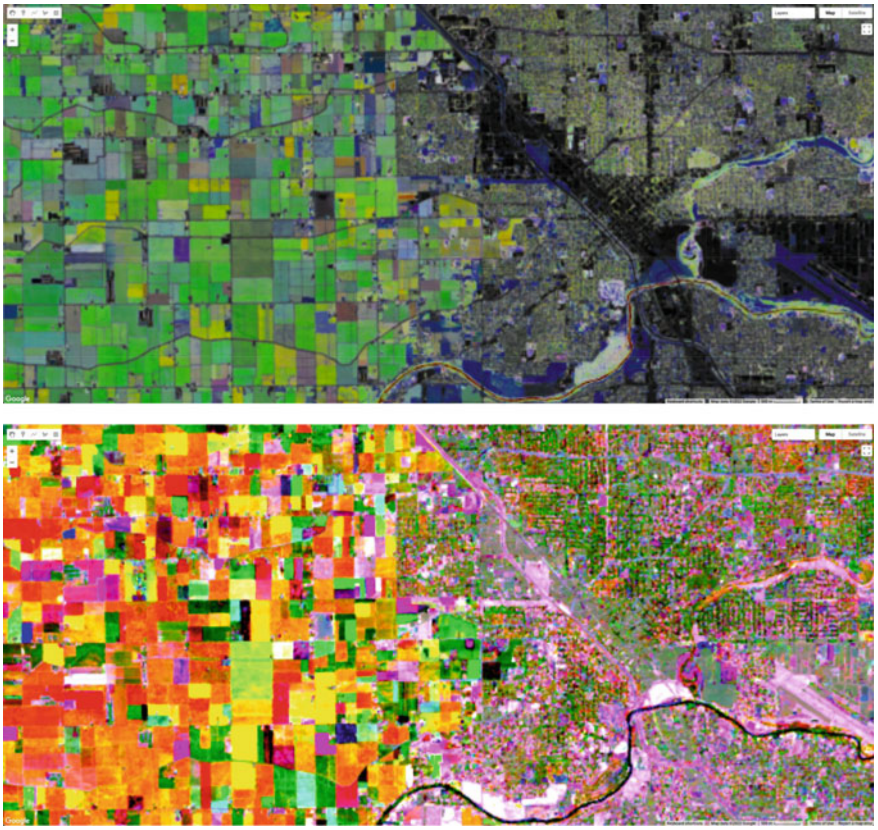


Fig. 18.9 Two views of the harmonic fits for NDVI for the Modesto, California area

Code Checkpoint F46d. The book’s repository contains a script that shows what your code should look like at this point.

18.2.6 Section 6: Higher-Order Harmonic Models

Harmonic models are not limited to fitting a single wave through a set of points. In some situations, there may be more than one cycle within a given year—for example, when an agricultural field is double-cropped. Modeling multiple waves within a given year can be done by adding more harmonic terms to Eq. 18.2. The code at the following checkpoint allows the fitting of any number of cycles through a given point.

Code Checkpoint F46e. The book’s repository contains a script to use to begin this section. You will need to start with that script and edit the code to produce the charts in this section.

Beginning with the repository script, changing the value of the `harmonics` variable will change the complexity of the harmonic curve fit by superimposing more or fewer harmonic waves on each other. While fitting higher-order functions improves the goodness-of-fit of the model to a given set of data, many of the coefficients may be close to zero at higher numbers or harmonic terms. Figure 18.10 shows the fit through the example point using one, two, and three harmonic curves.

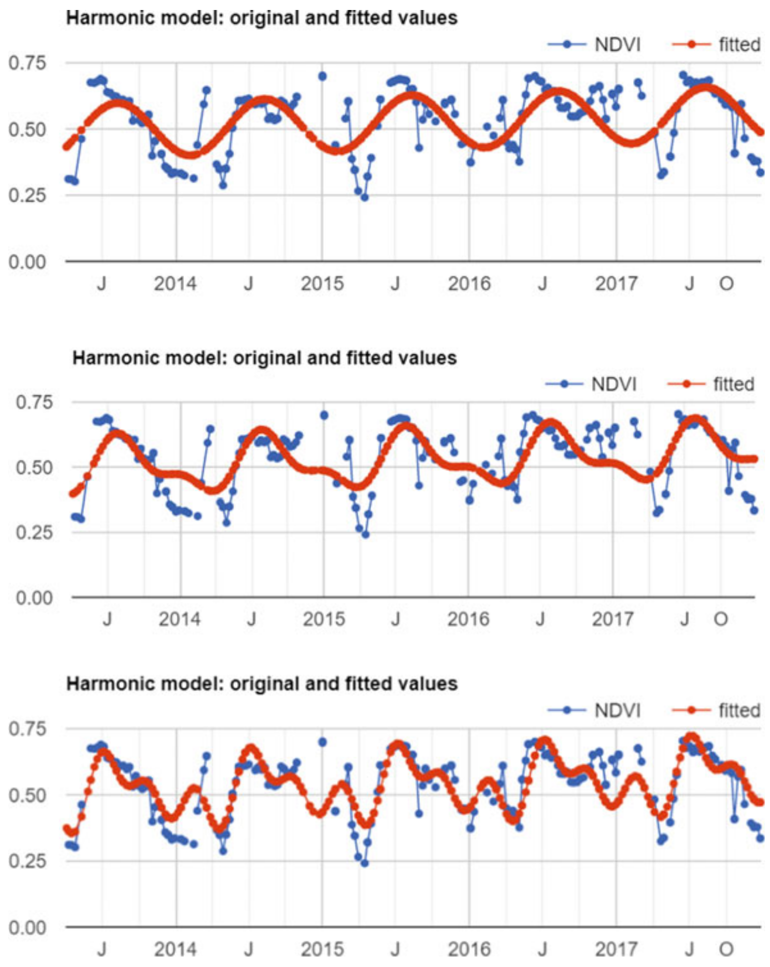


Fig. 18.10 Fit with harmonic curves of increasing complexity, fitted for data at a given point

18.3 Synthesis

Assignment 1. Fit two NDVI harmonic models for a point close to Manaus, Brazil: one prior to a disturbance event and one after the disturbance event (Fig. 18.11). You can start with the code checkpoint below, which gives you the point coordinates and defines the initial functions needed. The disturbance event happened in mid-December 2014, so set filter dates for the first `ImageCollection` to `'2013-01-01', '2014-12-12'`, and set the filter dates for the second `ImageCollection` to `‘2014-12-13', ‘2019-01-01'`. Merge both fitted collections and plot both NDVI and fitted values. The result should look like Fig. 18.12.

Code Checkpoint F46s1. The book's repository contains a script that shows what your code should look like at this point.

What do you notice? Think about how the harmonic model would look if you tried to fit the entire period. In this example, you were given the date of the breakpoint between the two conditions of the land surface within the time series. State-of-the-art land cover change algorithms work by assessing the difference between the modeled and observed pixel values. These algorithms look for breakpoints in the model, typically flagging changes after a predefined number of consecutive observations.

Code Checkpoint F46s2. The book's repository contains a script that shows what your code should look like at this point.

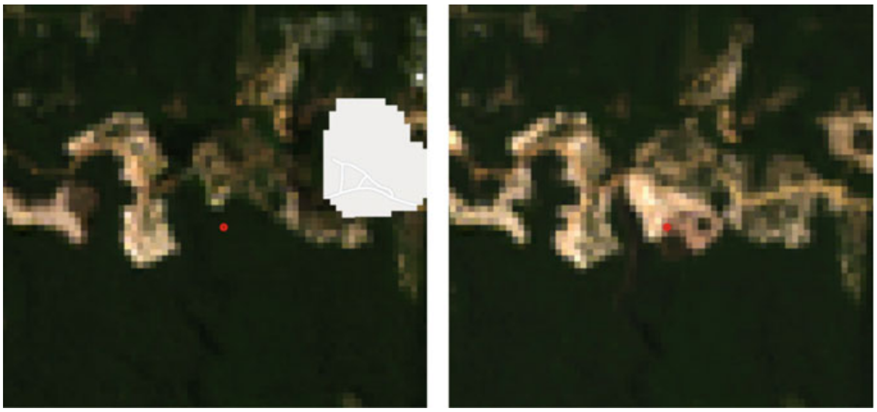


Fig. 18.11 Landsat 8 images showing the land cover change at a point in Manaus, Brazil; (left) July 6, 2014, (right) August 8, 2015

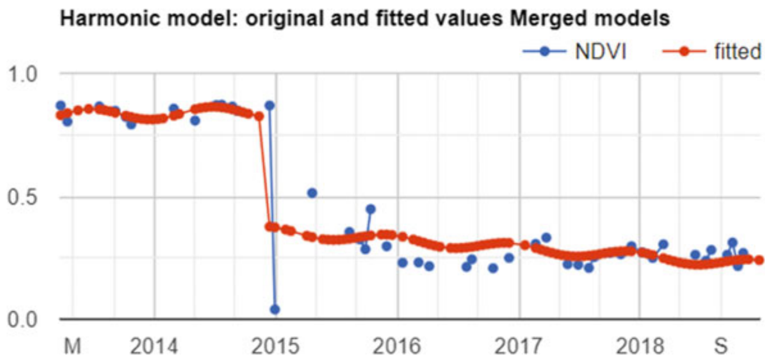


Fig. 18.12 Fitted harmonic models before and after disturbance events to a given point in the Brazilian Amazon

18.4 Conclusion

In this chapter, we learned how to graph and fit both linear and harmonic functions to time series of remotely sensed data. These skills underpin important tools such as Continuous Change Detection and Classification (CCDC, Chap. 19) and Continuous Degradation Detection (CODED, Chap. 49). These approaches are used by many organizations to detect forest degradation and deforestation (e.g., Tang et al. 2019; Bullock et al. 2020). These approaches can also be used to identify crops (Chap. 32) with high degrees of accuracy (Ghazaryan et al. 2018).

References

Bradley BA, Jacob RW, Hermance JF, Mustard JF (2007) A curve fitting procedure to derive inter-annual phenologies from time series of noisy satellite NDVI data. Remote Sens Environ 106:137–145. https://doi.org/10.1016/j.rse.2006.08.002

Bullock EL, Woodcock CE, Olofsson P (2020) Monitoring tropical forest degradation using spectral unmixing and Landsat time series analysis. Remote Sens Environ 238:110968. https://doi.org/10.1016/j.rse.2018.11.011

Ghazaryan G, Dubovyk O, Löw F et al (2018) A rule-based approach for crop identification using multi-temporal and multi-sensor phenological metrics. Eur J Remote Sens 51:511–524. https://doi.org/10.1080/22797254.2018.1455540

Shumway RH, Stoffer DS (2019) Time series: a data analysis approach using R. Chapman and Hall/CRC

Tang X, Bullock EL, Olofsson P et al (2019) Near real-time monitoring of tropical forest disturbance: new algorithms and assessment framework. Remote Sens Environ 224:202–218. https://doi.org/10.1016/j.rse.2019.02.003

Interpreting Time Series with CCDC

Paulo Arévalo and Pontus Olofsson

Overview
Continuous change detection and classification (CCDC) is a land change monitoring algorithm designed to operate on time series of satellite data, particularly Landsat data. This chapter focuses on the portion that is the change detection component (CCD); you will learn how to run the algorithm, interpret its outputs, and visualize coefficients and change information.

Learning Outcomes

- Exploring pixel-level time series of Landsat observations, as well as the temporal segments that CCDC fits to the observations.
- Visualizing the coefficients of the temporal segments in space.
- Visualizing predicted images made from detected temporal segments.
- Visualizing change information.
- Using array image functions.
- Attaching user-defined metadata to an image when exporting.

Assumes you know how to

- Import images and image collections, filter, and visualize (Part I).
- Perform basic image analysis: select bands, compute indices, create masks (Part II).

P. Arévalo (✉) · P. Olofsson
Department of Earth and Environment, Boston University, Boston, MA, USA
e-mail: parevalo@bu.edu

P. Olofsson
e-mail: olofsson@bu.edu

© The Author(s) 2024
J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_19

- Visualize images with a variety of false-color band combinations (Chap. 2).
- Interpret bands and indices in terms of land surface characteristics (Chap. 5).
- Work with array images (Chaps. 9 and 18).
- Interpret fitted harmonic models (Chap. 18).

19.1 Introduction to Theory

"A time series is a sequence of observations taken sequentially in time. … An intrinsic feature of a time series is that, typically, adjacent observations are dependent. Time-series analysis is concerned with techniques for the analysis of this dependency." This is the formal definition of time-series analysis by Box et al. (1994). In a remote sensing context, the observations of interest are measurements of radiation reflected from the surface of the Earth from the Sun or an instrument emitting energy toward Earth. Consecutive measurements made over a given area result in a time series of surface reflectance. By analyzing such time series, we can achieve a comprehensive characterization of ecosystem and land surface processes (Kennedy et al. 2014). The result is a shift away from traditional, retrospective change detection approaches based on data acquired over the same area at two or a few points in time to continuous monitoring of the landscape (Woodcock et al. 2020). Previous obstacles related to data storage, preprocessing, and computing power have been largely overcome with the emergence of powerful cloud-computing platforms that provide direct access to the data (Gorelick et al. 2017). In this chapter, we will illustrate how to study landscape dynamics in the Amazon river basin by analyzing dense time series of Landsat data using the CCDC algorithm. Unlike LandTrendr (Chap. 17), which uses anniversary images to fit straight line segments that describe the spectral trajectory over time, CCDC uses all available clear observations. This has multiple advantages, including the ability to detect changes within a year and capture seasonal patterns, although at the expense of much higher computational demands and more complexity to manipulate the outputs, compared to LandTrendr.

19.2 Practicum

19.2.1 Section 1: Understanding Temporal Segmentation with CCDC

Spectral change is detected at the pixel level by testing for structural breaks in a time series of reflectance. In Earth Engine, this process is referred to as "temporal segmentation," as pixel-level time series are segmented according to periods of unique reflectance. It does so by fitting harmonic regression models to all spectral bands in the time series. The model-fitting starts at the beginning of the time series and moves forward in time in an "online" approach to change detection. The coefficients are used to predict future observations, and if the residuals of future observations exceed a statistical threshold for numerous consecutive observations, then the algorithm flags that a change has occurred. After the change, a new regression model is fit and the process continues until the end of the time series. The details of the original algorithm are described in Zhu and Woodcock (2014). We have created an interface-based tool (Arévalo et al. 2020) that facilitates the exploration of time series of Landsat observations and the CCDC results.

Code Checkpoint F47a. The book's repository contains information about accessing the CCDC interface.

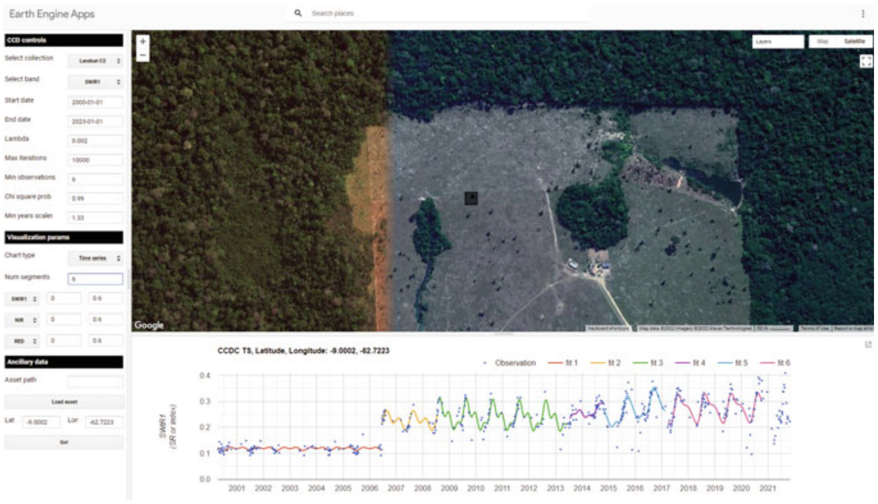
Once you have loaded the CCDC interface (Fig. 19.1), you will be able to navigate to any location, pick a Landsat spectral band or index to plot, and click on the map to see the fit by CCDC at the location you clicked. For this exercise, we will study landscape dynamics in the state of Rondônia, Brazil. We can use the panel on the left-bottom corner to enter the following coordinates (latitude, longitude): − 9.0002, − 62.7223. A point will be added in that location and the map will zoom in to it. Once there, click on the point and wait for the chart at the bottom to load. This example shows the Landsat time series for the first shortwave infrared (SWIR1) band (as blue dots) and the time segments (as colored lines) run using CCDC default parameters. The first segment represents stable forest, which was abruptly cut in mid-2006. The algorithm detects this change event and fits a new segment afterward, representing a new temporal pattern of agriculture. Other subsequent patterns are detected as new segments are fitted that may correspond to cycles of harvest and regrowth, or a different crop. To investigate the dynamics over time, you can click on the points in the chart, and the Landsat images they correspond to will be added to the map according to the visualization parameters selected for the RGB combination in the left panel. Currently, changes made in that panel are not immediate but must be set before clicking on the map.

Pay special attention to the characteristics of each segment. For example, look at the average surface reflectance value for each segment. The presence of a pronounced slope may be indicative of phenomena like vegetation regrowth or degradation. The number of harmonics used in each segment may represent seasonality in vegetation (either natural or due to agricultural practices) or landscape dynamics (e.g., seasonal flooding).

Question 1. While still using the SWIR1 band, click on a pixel that is forested. What do the time series and time segments look like?

19.2.2 Section 2: Running CCDC

The tool shown above is useful for understanding the temporal dynamics for a specific point. However, we can do a similar analysis for larger areas by first running the CCDC algorithm over a group of pixels. The CCDC function in Earth Engine can take any `ImageCollection`, ideally one with little or no noise, such as a Landsat `ImageCollection` where clouds and cloud shadows have been



CCD controls

Select collection: Landsat C2
Select band: SWIR1
Start date: 2000-01-01
End date: 2023-01-01
Lambda: 0.002
Max iterations: 10000
Min observations: 6
Chi square prob: 0.99
Min years scaler: 1.33

Visualization params

Chart type: Time series
Num segments: 6
SWIR1: 0 0.6
NIR: 0 0.6
RED: 0 0.6

Ancillary data

Asset path:
Load asset
Lat: -9.0002 Lon: -62.7223
Go!

Fig. 19.1 Landsat time series for the SWIR1 band (blue dots) and CCDC time segments (colored lines) showing a forest loss event circa 2006 for a place in Rondônia, Brazil

masked. CCDC contains an internal cloud masking algorithm and is rather robust against missed clouds, but the cleaner the data the better. To simplify the process, we have developed a function library that contains functions for generating input data and processing CCDC results. Paste this line of code in a new script:

```
var utils = require(
    'users/parevalo_bu/gee-ccdc-tools:ccdcUtilities/api');
```

For the current exercise, we will obtain an `ImageCollection` of Landsat 4, 5, 7, and 8 data (Collection 2 Tier 1) that has been filtered for clouds, cloud shadows, haze, and radiometrically saturated pixels. If we were to do this manually, we would retrieve each `ImageCollection` for each satellite, apply the corresponding filters and then merge them all into a single `ImageCollection`. Instead, to simplify that process, we will use the function `getLandsat`, included in the "Inputs" module of our utilities, and then filter the resulting `ImageCollection` to a small study region for the period between 2000 and 2020. The `getLandsat` function will retrieve all surface reflectance bands (renamed and scaled to actual surface reflectance units) as well as other vegetation indices. To simplify the exercise, we will select only the surface reflectance bands we are going to use, adding the following code to your script:

```
var studyRegion = ee.Geometry.Rectangle([
    [-63.9533, -10.1315],
    [-64.9118, -10.6813]
]);

// Define start, end dates and Landsat bands to use.
var startDate = '2000-01-01';
var endDate = '2020-01-01';
var bands = ['BLUE', 'GREEN', 'RED', 'NIR', 'SWIR1',
'SWIR2'];

// Retrieve all clear, Landsat 4, 5, 7 and 8 observations
(Collection 2, Tier 1).
var filteredLandsat = utils.Inputs.getLandsat({
        collection: 2
    })
    .filterBounds(studyRegion)
    .filterDate(startDate, endDate)
    .select(bands);

print(filteredLandsat.first());
```

With the `ImageCollection` ready, we can specify the CCDC parameters and run the algorithm. For this exercise, we will use the default parameters, which tend to work reasonably well in most circumstances. The only parameters we will modify are the breakpoint bands, date format, and lambda. We will set all the parameter values in a dictionary that we will pass to the CCDC function. For the break detection process, we use all bands except for the blue and surface temperature bands (`'BLUE'` and `'TEMP'`, respectively). The `minObservations` default value of 6 represents the number of consecutive observations required to flag a change. The `chiSquareProbability` and `minNumOfYearsScaler` default parameters of 0.99 and 1.33, respectively, control the sensitivity of the algorithm to detect change and the iterative curve fitting process required to detect change. We set the date format to 1, which corresponds to fractional years and tends to be easier to interpret. For instance, a change detected in the middle day of the year 2010 would be stored in a pixel as 2010.5. Finally, we use the default value of `lambda` of 20, but we scale it to match the scale of the inputs (surface reflectance units), and we specify a `maxIterations` value of 10,000, instead of the default of 25,000, which might take longer to complete. Those two parameters control the curve fitting process.

To complete the input parameters, we specify the ImageCollection to use, which we derived in the previous code section. Add this code below:

```
// Set CCD params to use.
var ccdParams = {
    breakpointBands: ['GREEN', 'RED', 'NIR', 'SWIR1', 'SWIR2'],
    tmaskBands: ['GREEN', 'SWIR2'],
    minObservations: 6,
    chiSquareProbability: 0.99,
    minNumOfYearsScaler: 1.33,
    dateFormat: 1,
    lambda: 0.002,
    maxIterations: 10000,
    collection: filteredLandsat
};

// Run CCD.
var ccdResults =
ee.Algorithms.TemporalSegmentation.Ccdc(ccdParams);
print(ccdResults);
```

Notice that the output ccdResults contains a large number of bands, with some of them corresponding to two-dimensional arrays. We will explore these bands more in the following section. The process of running the algorithm interactively for more than a handful of pixels can become very taxing to the system very quickly, resulting in memory errors. To avoid having such issues, we typically export the results to an Earth Engine asset first, and then inspect the asset. This approach ensures that CCDC completes its run successfully, and also allows us to access the results easily later. In the following sections of this chapter, we will use a precomputed asset, instead of asking you to export the asset yourself. For your reference, the code required to export CCDC results is shown below, with the flag set to false to help you remember to not export the results now, but instead to use the precomputed asset in the following sections.

```
var exportResults = false
if (exportResults) {
    // Create a metadata dictionary with the parameters and
arguments used.
    var metadata = ccdParams;
    metadata['breakpointBands'] =
        metadata['breakpointBands'].toString();
    metadata['tmaskBands'] = metadata['tmaskBands'].toString();
    metadata['startDate'] = startDate;
    metadata['endDate'] = endDate;
    metadata['bands'] = bands.toString();

    // Export results, assigning the metadata as image
properties.
    //
    Export.image.toAsset({
        image: ccdResults.set(metadata),
        region: studyRegion,
        pyramidingPolicy: {
            ".default": 'sample'
        },
        scale: 30
    });
}
```

Note the `metadata` variable above. This is not strictly required for exporting the per-pixel CCDC results, but it allows us to keep a record of important properties of the run by attaching this information as metadata to the image. Additionally, some of the tools we have created to interact with CCDC outputs use this user-created metadata to facilitate using the asset. Note also that setting the value of `pyramidingPolicy` to `'sample'` ensures that all the bands in the output have the proper policy.

As a general rule, try to use pre-existing CCDC results if possible, and if you want to try running it yourself outside of this lab exercise, start with very small areas. For instance, the study area in this exercise would take approximately 30 min on average to export, but larger tiles may take several hours to complete, depending on the number of images in the collection and the parameters used.

Code Checkpoint F47b. The book's repository contains a script that shows what your code should look like at this point.

19.2.3 Section 3: Extracting Break Information

We will now start exploring the pre-exported CCDC results mentioned in the previous section. We will make use of the third-party module `palettes`, described in detail in Chap. 27, that simplifies the use of palettes for visualization. Paste the following code in a new script:

```
var palettes = require('users/gena/packages:palettes');

var resultsPath =
    'projects/gee-book/assets/F4-7/Rondonia_example_small';
var ccdResults = ee.Image(resultsPath);
Map.centerObject(ccdResults, 10);
print(ccdResults);
```

The first line calls a library that will facilitate visualizing the images. The second line contains the path to the precomputed results of the CCDC run shown in the previous section. The printed asset will contain the following bands:

- `tStart`: The start date of each time segment
- `tEnd`: The end date of each time segment
- `tBreak`: The time segment break date if a change is detected
- `numObs`: The number of observations used in each time segment
- `changeProb`: A numeric value representing the change probability for each of the bands used for change detection
- `*_coefs`: The regression coefficients for each of the bands in the input image collection
- `*_rmse`: The model root-mean-square error for each time segment and input band
- `*_magnitude`: For time segments with detected changes, this represents the normalized residuals during the change period.

Notice that next to the band name and band type, there is also the number of dimensions (i.e., 1 dimension, 2 dimensions). This is an indication that we are dealing with an array image, which typically requires a specific set of functions for proper manipulation, some of which we will use in the next steps. We will start by looking at the change bands, which are one of the key outputs of the CCDC algorithm. We will select the band containing the information on the timing of break, and find the number of breaks for a given time range. In the same script, paste the code below:

```
// Select time of break and change probability array images.
var change = ccdResults.select('tBreak');
var changeProb = ccdResults.select('changeProb');

// Set the time range we want to use and get as mask of
// places that meet the condition.
var start = 2000;
var end = 2021;
var mask =
change.gt(start).and(change.lte(end)).and(changeProb.eq(
1));
Map.addLayer(changeProb, {}, 'change prob');

// Obtain the number of breaks for the time range.
var numBreaks = mask.arrayReduce(ee.Reducer.sum(), [0]);
Map.addLayer(numBreaks, {
    min: 0,
    max: 5
}, 'Number of breaks');
```

With this code, we define the time range that we want to use, and then we generate a mask that will indicate all the positions in the image array with breaks detected in that range that also meet the condition of having a change probability of 1, effectively removing some spurious breaks. For each pixel, we can count the number of times that the mask retrieved a valid result, indicating the number of breaks detected by CCDC. In the loaded layer, places that appear brighter will show a higher number of breaks, potentially indicating the conversion from forest to agriculture, followed by multiple agricultural cycles. Keep in mind that the detection of a break does not always imply a change of land cover. Natural events, small-scale disturbances, and seasonal cycles, among others can result in the detection of a break by CCDC. Similarly, changes in the *condition* of the land cover in a pixel can also be detected as breaks by CCDC, and some erroneous breaks can also happen due to noisy time series or other factors.

For places with many changes, visualizing the first or last time when a break was recorded can be helpful to understand the change dynamics happening in the landscape. Paste the code below in the same script:

```
// Obtain the first change in that time period.
var dates = change.arrayMask(mask).arrayPad([1]);
var firstChange = dates
    .arraySlice(0, 0, 1)
    .arrayFlatten([
        ['firstChange']
    ])
    .selfMask();

var timeVisParams = {
    palette: palettes.colorbrewer.YlOrRd[9],
    min: start,
    max: end
};
Map.addLayer(firstChange, timeVisParams, 'First change');

// Obtain the last change in that time period.
var lastChange = dates
    .arraySlice(0, -1)
    .arrayFlatten([
        ['lastChange']
    ])
    .selfMask();
Map.addLayer(lastChange, timeVisParams, 'Last change');
```

Here, we use `arrayMask` to keep only the change dates that meet our condition, by using the mask we created previously. We use the function `arrayPad` to fill or "pad" those pixels that did not experience any change and therefore have no value in the `tBreak` band. Then we select either the first or last values in the array, and we convert the image from a one-dimensional array to a regular image, in order to apply a visualization to it, using a custom palette. The results should look like Fig. 19.2.

Finally, we can use the magnitude bands to visualize where and when the largest changes as recorded by CCDC have occurred, during our selected time period. We are going to use the magnitude of change in the SWIR1 band, masking it and padding it in the same way we did before. Paste this code in your script:

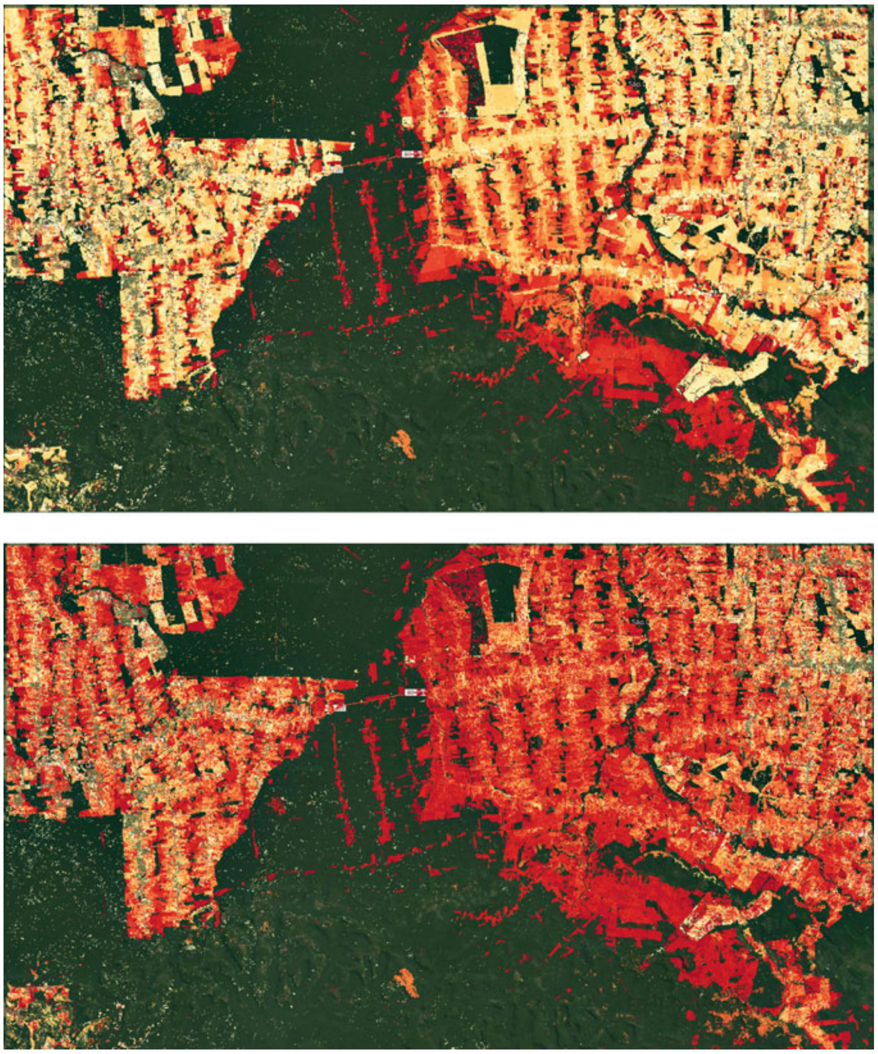


Fig. 19.2 First (top) and last (bottom) detected breaks for the study area. Darker colors represent more recent dates, while brighter colors represent older dates. The first change layer shows the clear patterns of original agricultural expansion closer to the year 2000. The last change layer shows the more recently detected and noisy breaks in the same areas. The thin areas in the center of the image have only one time of change, corresponding to a single deforestation event. Pixels with no detected breaks are masked and therefore show the basemap underneath, set to show satellite imagery

```
// Get masked magnitudes.
var magnitudes = ccdResults
    .select('SWIR1_magnitude')
    .arrayMask(mask)
    .arrayPad([1]);

// Get index of max abs magnitude of change.
var maxIndex = magnitudes
    .abs()
    .arrayArgmax()
    .arrayFlatten([
        ['index']
    ]);

// Select max magnitude and its timing
var selectedMag = magnitudes.arrayGet(maxIndex);
var selectedTbreak = dates.arrayGet(maxIndex).selfMask();

var magVisParams = {
    palette: palettes.matplotlib.viridis[7],
    min: -0.15,
    max: 0.15
};
Map.addLayer(selectedMag, magVisParams, 'Max mag');
Map.addLayer(selectedTbreak, timeVisParams, 'Time of max
mag');
```

We first take the absolute value because the magnitudes can be positive or negative, depending on the direction of the change and the band used. For example, a positive value in the SWIR1 may show a forest loss event, where surface reflectance goes from low to higher values. Brighter values in Fig. 19.3 represent events of that type. Conversely, a flooding event would have a negative value, due to the corresponding drop in reflectance. Once we find the maximum absolute value, we find its position on the array and then use that index to extract the original magnitude value, as well as the time when that break occurred.

Code Checkpoint F47c. The book's repository contains a script that shows what your code should look like at this point.

Question 2. Compare the "first change" and "last change" layers with the layer showing the timing of the maximum magnitude of change. Use the **Inspector** to check the values for specific pixels if necessary. What does the timing of the layers tell you about the change processes happening in the area?

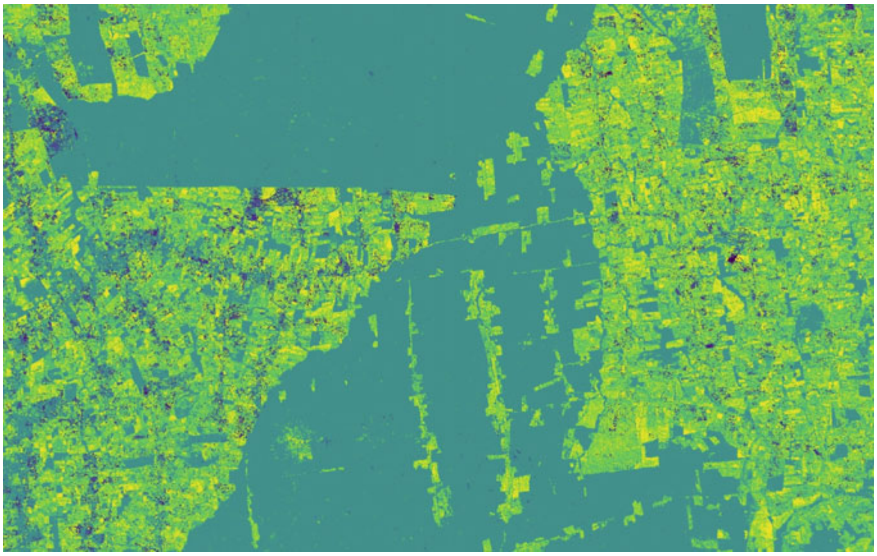


Fig. 19.3 Maximum magnitude of change for the SWIR1 band for the selected study period

Question 3. Looking at the “max magnitude of change” layer, find places showing the largest and the smallest values. What type of changes do you think are happening in each of those places?

19.2.4 Section 4: Extracting Coefficients Manually

In addition to the change information generated by the CCDC algorithm, we can use the coefficients of the time segments for multiple purposes, like land cover classification. Each time segment can be described as a harmonic function with an intercept, slope, and three pairs of sine and cosine terms that allow the time segments to represent seasonality occurring at different temporal scales. These coefficients, as well as the root-mean-square error (RMSE) obtained by comparing each predicted and actual Landsat value, are produced when the CCDC algorithm is run. The following example will show you how to retrieve the intercept coefficient for a segment intersecting a specific date. In a new script, paste the code below:

```
var palettes = require('users/gena/packages:palettes');

var resultsPath =
    'projects/gee-book/assets/F4-7/Rondonia_example_small';
var ccdResults = ee.Image(resultsPath);
Map.centerObject(ccdResults, 10);
print(ccdResults);

// Display segment start and end times.
var start = ccdResults.select('tStart');
var end = ccdResults.select('tEnd');
Map.addLayer(start, {
    min: 1999,
    max: 2001
}, 'Segment start');
Map.addLayer(end, {
    min: 2010,
    max: 2020
}, 'Segment end');
```

Check the **Console** and expand the bands section in the printed image information. We will be using the `tStart`, `tEnd`, and `SWIR1_coefs` bands, which are array images containing the date when the time segments start, date time segments end, and the coefficients for each of those segments for the SWIR1 band. Run the code above and switch the map to **Satellite** mode. Using the **Inspector**, click anywhere on the images, noticing the number of dates printed and their values for multiple clicked pixels. You will notice that for places with stable forest cover, there is usually one value for `tStart` and one for `tEnd`. This means that for those more stable places, only one time segment was fit by CCDC. On the other hand, for places with visible transformation in the basemap, the number of dates is usually two or three, meaning that the algorithm fitted two or three time segments, respectively. To simplify the processing of the data, we can select a single segment to extract its coefficients. Paste the code below and re-run the script:

```
// Find the segment that intersects a given date.
var targetDate = 2005.5;
var selectSegment =
start.lte(targetDate).and(end.gt(targetDate));
Map.addLayer(selectSegment, {}, 'Identified segment');
```

In the code above, we set a time of interest, in this case the middle of 2005, and then we find the segments that meet the condition of starting before and ending after that date. Using the **Inspector** again, click on different locations and verify the outputs. The segment that meets the condition will have a value of 1, and the other segments will have a value of 0. We can use this information to select the coefficients for that segment, using the code below:

```
// Get all coefs in the SWIR1 band.
var SWIR1Coefs = ccdResults.select('SWIR1_coefs');
Map.addLayer(SWIR1Coefs, {}, 'SWIR1 coefs');

// Select only those for the segment that we identified
previously.
var sliceStart = selectSegment.arrayArgmax().arrayFlatten([
    ['index']
]);
var sliceEnd = sliceStart.add(1);
var selectedCoefs = SWIR1Coefs.arraySlice(0, sliceStart,
sliceEnd);
Map.addLayer(selectedCoefs, {}, 'Selected SWIR1 coefs');
```

In the piece of code above, we first select the array image with the coefficients for the SWIR1 band. Then, using the layer that we created before, we find the position where the condition is true, and use that to extract the coefficients *only* for that segment. Once again, you can verify that using the **Inspector** tab.

Finally, what we have now is the full set of coefficients for the segment that intersects the midpoint of 2005. The coefficients are in the following order: intercept, slope, cosine 1, sine 1, cosine 2, sine 2, cosine 3, and sine 3. For this exercise, we will extract the intercept coefficient (Fig. 19.4), which is the first element in the array, using the code below:

```
// Retrieve only the intercept coefficient.
var intercept = selectedCoefs.arraySlice(1, 0,
1).arrayProject([1]);
var intVisParams = {
    palette: palettes.matplotlib.viridis[7],
    min: -6,
    max: 6
};
Map.addLayer(intercept.arrayFlatten([
    ['INTP']
]), intVisParams, 'INTP_SWIR1');
```

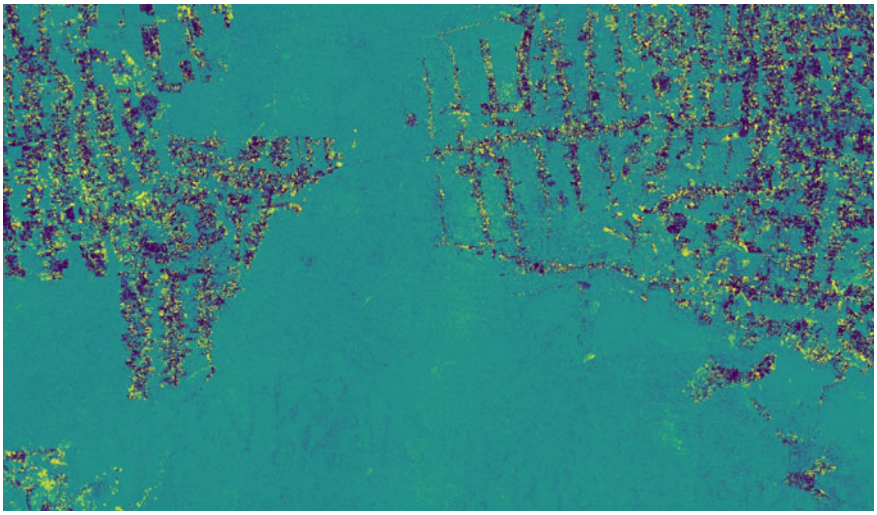


Fig. 19.4 Values for the intercept coefficient of the segments that start before and end after the midpoint of 2005

Since we run the CCDC algorithm on Landsat surface reflectance images, intercept values should represent the average reflectance of a segment. However, if you click on the image, you will see that the values are outside of the 0–1 range. This is because the intercept is calculated by the CCDC algorithm for the origin (e.g., time 0), and not for the year we requested. In order to retrieve the adjusted intercept, as well as other coefficients, we will use a different approach.

Code Checkpoint F47d. The book's repository contains a script that shows what your code should look like at this point.

19.2.5 Section 5: Extracting Coefficients Using External Functions

The code we generated in the previous section allowed us to extract a single coefficient for a single date. However, we typically want to extract a set of multiple coefficients and bands that we can use as inputs to other workflows, such as classification. To simplify that process, we will use the same function library that we saw in Sect. 19.2.2. In this section, we will extract and visualize different coefficients for a single date and produce an RGB image using the intercept coefficients for multiple spectral bands for the same date. The first step involves determining the date of interest and converting the CCDC results from array images to regular multiband images for easier manipulation and faster display. In a new script, copy the code below:

```
// Load the required libraries.
var palettes = require('users/gena/packages:palettes');
var utils = require(
    'users/parevalo_bu/gee-ccdc-tools:ccdcUtilities/api');

// Load the results.
var resultsPath =
    'projects/gee-book/assets/F4-7/Rondonia_example_small';
var ccdResults = ee.Image(resultsPath);
Map.centerObject(ccdResults, 10);

// Convert a date into fractional years.
var inputDate = '2005-09-25';
var dateParams = {
    inputFormat: 3,
    inputDate: inputDate,
    outputFormat: 1
};
var formattedDate = utils.Dates.convertDate(dateParams);

// Band names originally used as inputs to the CCD
algorithm.
var BANDS = ['BLUE', 'GREEN', 'RED', 'NIR', 'SWIR1',
'SWIR2'];

// Names for the time segments to retrieve.
var SEGS = ['S1', 'S2', 'S3', 'S4', 'S5', 'S6', 'S7', 'S8',
'S9',
    'S10'
];

// Transform CCD results into a multiband image.
var ccdImage = utils.CCDC.buildCcdImage(ccdResults,
SEGS.length,
    BANDS);
print(ccdImage);
```

In the code above, we define the date of interest (`2005-09-25`) and convert it to the date format in which we ran CCDC, which corresponds to fractional years. After that, we specify the band that we used as inputs for the CCDC algorithm. Finally, we specify the names we will assign to the time segments, with the list length indicating the maximum number of time segments to retrieve per pixel. This step is done because the results generated by CCDC are stored as variable-length arrays. For example, a pixel where there are no breaks detected will have one time segment, but another pixel where a single break was detected may have one or two segments, depending on when the break occurred. Requesting a predefined maximum number of segments ensures that the structure of the multiband image is known, and greatly facilitates its manipulation and display. Once we have set these variables, we call a function that converts the result into an image with several bands representing the combination of segments requested, input bands, and coefficients. You can see the image structure in the **Console**.

Finally, to extract a subset of coefficients for the desired bands, we can use a function in the imported library, called `getMultiCoefs`. This function expects the following ordered parameters:

- The CCDC results in the multiband format we just generated in the step above.
- The date for which we want to extract the coefficients, in the format in which the CCDC results were run (fractional years in our case).
- List of the bands to retrieve (i.e., spectral bands).
- List of coefficients to retrieve, defined as follows: `INTP` (intercept), `SLP` (slope), `COS`, `SIN`, `COS32`, `SIN2`, `COS3`, `SIN3`, and `RMSE`.
- A Boolean flag of **`true`** or **`false`**, indicating whether we want the intercepts to be calculated for the input date, instead of being calculated at the origin. If **`true`**, `SLP` must be included in the list of coefficients to retrieve.
- List of segment names, as used to create the multiband image in the prior step.
- Behavior to apply if there is no time segment for the requested date: `normal` will retrieve a value only if the date intersects a segment; `before` or `after` will use the value of the segment immediately before or after the requested date, if no segment intersects the date directly.

```
// Define bands to select.
var SELECT_BANDS = ['RED', 'GREEN', 'BLUE', 'NIR'];

// Define coefficients to select.
// This list contains all possible coefficients, and the RMSE
var SELECT_COEFS = ['INTP', 'SLP', 'RMSE'];

// Obtain coefficients.
var coefs = utils.CCDC.getMultiCoefs(
    ccdImage, formattedDate, SELECT_BANDS, SELECT_COEFS, true,
    SEGS, 'after');
print(coefs);

// Show a single coefficient.
var slpVisParams = {
    palette: palettes.matplotlib.viridis[7],
    min: -0.0005,
    max: 0.005
};
```

```
Map.addLayer(coefs.select('RED_SLP'), slpVisParams,
    'RED SLOPE 2005-09-25');

var rmseVisParams = {
    palette: palettes.matplotlib.viridis[7],
    min: 0,
    max: 0.1
};
Map.addLayer(coefs.select('NIR_RMSE'), rmseVisParams,
    'NIR RMSE 2005-09-25');

// Show an RGB with three coefficients.
var rgbVisParams = {
    bands: ['RED_INTP', 'GREEN_INTP', 'BLUE_INTP'],
    min: 0,
    max: 0.1
};
Map.addLayer(coefs, rgbVisParams, 'RGB 2005-09-25');
```

The slope and RMSE images are shown in Fig. 19.5. For the slopes, high positive values are bright, while large negative values are very dark. Most of the remaining forest is stable and has a slope close to zero, while areas that have experienced transformation and show agricultural activity tend to have positive slopes in the RED band, appearing bright in the image. Similarly, for the RMSE image, stable forests present more predictable time series of surface reflectance that are captured more faithfully by the time segments, and therefore present lower RMSE values, appearing darker in the image. Agricultural areas present noisier time series that are more challenging to model, and result in higher RMSE values, appearing brighter.

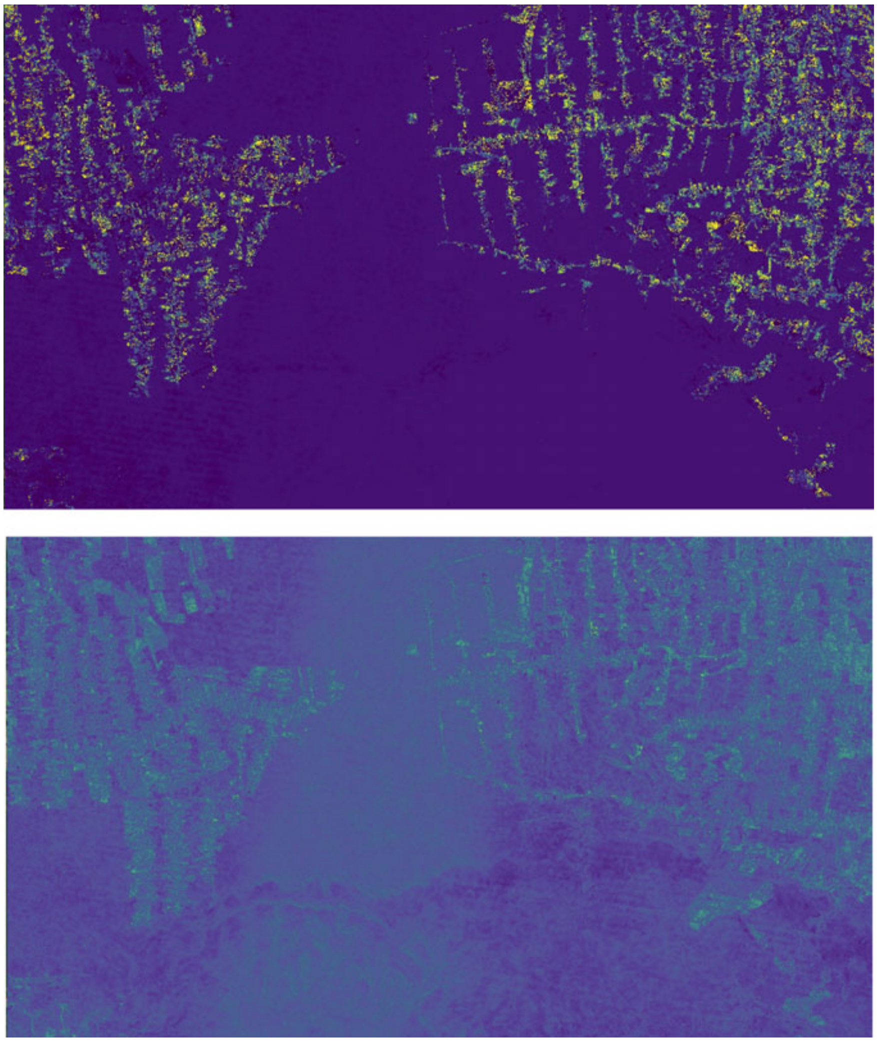


Fig. 19.5 Image showing the slopes (top) and RMSE (bottom) of the segments that intersect the requested date

Finally, the RGB image we created is shown in Fig. 19.6. The intercepts are calculated for the middle point of the time segment intercepting the date we requested, representing the average reflectance for the span of the selected segment. In that sense, when shown together as an RGB image, they are similar to a composite image for the selected date, with the advantage of always being cloud free.

Code Checkpoint F47e. The book's repository contains a script that shows what your code should look like at this point.

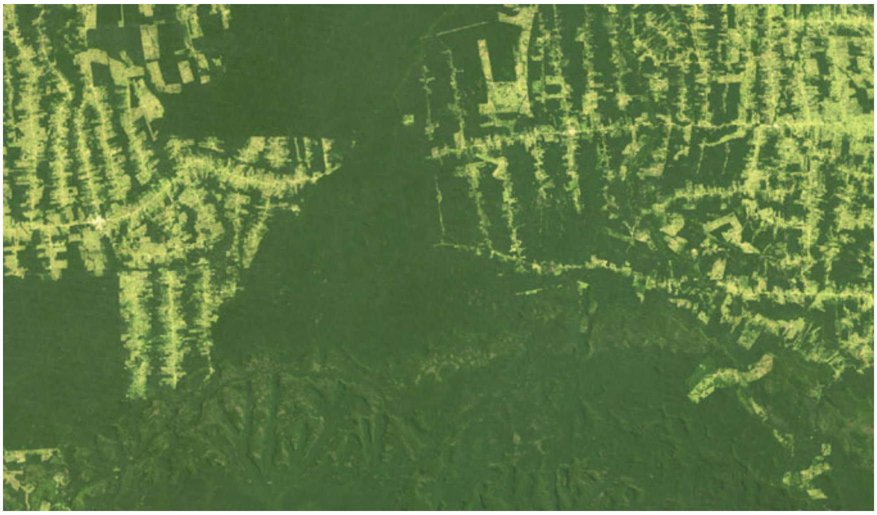


Fig. 19.6 RGB image created using the time segment intercepts for the requested date

19.3 Synthesis

Assignment 1. Use the time series from the first section of this chapter to explore the time series and time segments produced by CCDC in many locations around the world. Compare places with different land cover types, and places with more stable dynamics (e.g., lakes, primary forests) versus highly dynamic places (e.g., agricultural lands, construction sites). Pay attention to the variability in data density across continents and latitudes, and the effect that data density has on the appearance of the time segments. Use different spectral bands and indices and notice how they capture the temporal dynamics you are observing.

Assignment 2. Pick three periods within the temporal study period of the CCDC results we used earlier: one near to the start, another in the middle, and the third close to the end. For each period, visualize the maximum change magnitude. Compare the spatial patterns between periods, and reflect on the types of disturbances that might be happening at each stage.

Assignment 3. Select the intercept coefficients of the middle date of each of the periods you chose in the previous assignment. For each of those dates, load an RGB image with the band combination of your choosing (or simply use the Red, Green and Blue intercepts to obtain true-color images). Using the **Inspector** tab, compare the values across images in places with subtle and large differences between them, as well as in areas that do not change. What do the values tell you in terms of the benefits of using CCDC to study changes in a landscape?

19.4 Conclusion

This chapter provided a guide for the interpretation of the results from the CCDC algorithm for studying deforestation in the Amazon. Consider the advantages of such an analysis compared to traditional approaches to change detection, which are typically based on the comparison of two or a few images collected over the same area. For example, with time-series analysis, we can study trends and subtle processes such as vegetation recovery or degradation, determine the timing of land surface events, and move away from retrospective analyses to monitoring in near-real time. Through the use of all available clear observations, CCDC can detect intra-annual breaks and capture seasonal patterns, although at the expense of increased computational requirements and complexity, unlike faster and easier to interpret methods based on annual composites, such as LandTrendr (Chap. 17). We expect to see more applications that make use of multiple change detection approaches (also known as "Ensemble" approaches), and multisensor analyses in which data from different satellites are fused (radar and optical, for example) for higher data density.

References

Arévalo P, Bullock EL, Woodcock CE, Olofsson P (2020) A suite of tools for continuous land change monitoring in Google Earth Engine. Front Clim 2. https://doi.org/10.3389/fclim.2020.576740

Box GEP, Jenkins GM, Reinsel GC (1994) Time series analysis: forecasting and control. Prentice Hall

Gorelick N, Hancher M, Dixon M et al (2017) Google Earth Engine: planetary-scale geospatial analysis for everyone. Remote Sens Environ 202:18–27. https://doi.org/10.1016/j.rse.2017.06.031

Kennedy RE, Andréfouët S, Cohen WB et al (2014) Bringing an ecological view of change to Landsat-based remote sensing. Front Ecol Environ 12:339–346. https://doi.org/10.1890/130066

Woodcock CE, Loveland TR, Herold M, Bauer ME (2020) Transitioning from change detection to monitoring with remote sensing: a paradigm shift. Remote Sens Environ 238:111558. https://doi.org/10.1016/j.rse.2019.111558

Zhu Z, Woodcock CE (2014) Continuous change detection and classification of land cover using all available Landsat data. Remote Sens Environ 144:152–171. https://doi.org/10.1016/j.rse.2014.01.011

Data Fusion: Merging Classification Streams

20

Jeffrey A. Cardille, Rylan Boothman, Mary Villamor, Elijah Perez, Eidan Willis, and Flavie Pelletier

Overview

As the ability to rapidly produce classifications of satellite images grows, it will be increasingly important to have algorithms that can shift through them to separate the signal from inevitable classification noise. The purpose of this chapter is to explore how to update classification time series by blending information from multiple classifications made from a wide variety of data sources. In this lab, we will explore how to update the classification time series of the Roosevelt River found in Fortin et al. (2020). That time series began with the 1972 launch of Landsat 1, blending evidence from 10 sensors and more than 140 images to show the evolution of the area until 2016. How has it changed since 2016? What new tools and data streams might we tap to understand the land surface through time?

J. A. Cardille (✉) · R. Boothman · M. Villamor · E. Perez · E. Willis · F. Pelletier
Department of Natural Resource Sciences, Bieler School of Environment, McGill University, Montreal, Canada
e-mail: jeffrey.cardille@mcgill.ca

R. Boothman
e-mail: rylan.boothman@mail.mcgill.ca

E. Perez
e-mail: elijah.perez@mail.mcgill.ca

E. Willis
e-mail: eidan.willis@mail.mcgill.ca

F. Pelletier
e-mail: flavie.pelletier@mail.mcgill.ca

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_20

Learning Outcomes

- Distinguishing between merging sensor data and merging classifications made from sensors.
- Working with the Bayesian Updating of Land Cover (BULC) algorithm, in its basic form, to blend classifications made across multiple years and sensors.
- Working with the BULC-D algorithm to highlight locations that changed.

Assumes you know how to

- Import images and image collections, filter, and visualize (Part I).
- Perform basic image analysis: select bands, compute indices, create masks, classify images (Part II).
- Create a graph using `ui.Chart` (Chap. 4).
- Obtain accuracy metrics from classifications (Chap. 7).

20.1 Introduction to Theory

When working with multiple sensors, we are often presented with a challenge: What to do with classification noise? It is almost impossible to remove all noise from a classification. Given the information contained in a stream of classifications, however, you should be able to use the temporal context to distinguish noise from true changes in the landscape.

The Bayesian Updating of Land Cover (BULC) algorithm (Cardille and Fortin 2016) is designed to extract the signal from the noise in a stream of classifications made from any number of data sources. BULC's principal job is to estimate, at each time step, the likeliest state of land use and land cover (LULC) in a study area given the accumulated evidence to that point. It takes a stack of provisional classifications as input; in keeping with the terminology of Bayesian statistics, these are referred to as "Events," because they provide new evidence to the system. BULC then returns a stack of classifications as output that represents the estimated LULC time series implied by the Events.

BULC estimates, at each time step, the most likely class from a set given the evidence up to that point in time. This is done by employing an accuracy assessment matrix like that seen in Chap. 7. At each time step, the algorithm quantifies the agreement between two classifications adjacent in time within a time series.

If the Events agree strongly, they are evidence of the true condition of the landscape at that point in time. If two adjacent Events disagree, the accuracy assessment matrix limits their power to change the class of a pixel in the interpreted time series. As each new classification is processed, BULC judges the credibility of a pixel's stated class and keeps track of a set of estimates of the probability of each class for each pixel. In this way, each pixel traces its own LULC history, reflected through BULC's judgment of the confidence in each of the classifications. The specific mechanics and formulas of BULC are detailed in Cardille and Fortin (2016).

BULC's code is written in JavaScript, with modules that weigh evidence for and against change in several ways, while recording parts of the data-weighing process for you to inspect. In this lab, we will explore BULC through its graphical user interface (GUI), which allows rapid interaction with the algorithm's main functionality.

20.2 Practicum

20.2.1 Section 1: Imagery and Classifications of the Roosevelt River

How has the Roosevelt River area changed in recent decades? One way to view the area's recent history is to use Google Earth Timelapse, which shows selected annual clear images of every part of Earth's terrestrial surface since the 1980s. (You can find the site quickly with a web search.) Enter “Roosevelt River, Brazil” in the search field. For centuries, this area was very remote from agricultural development. It was so little known to Westerners that when former US President Theodore Roosevelt traversed it in the early 1900s, there was widespread doubt about whether his near-death experience there was exaggerated or even entirely fictional (Millard 2006). After World War II, the region saw increased agricultural development. Fortin et al. (2020) traced four decades of the history of this region with satellite imagery. Timelapse, meanwhile, indicates that land cover conversion continued after 2016. Can we track it using Earth Engine?

In this section, we will view the classification inputs to BULC, which were made separately from this lab exercise by identifying training points and classifying them using Earth Engine's regression tree capability. As seen in Table 20.1, the classification inputs included Sentinel-2 optical data, Landsat 7, Landsat 8, and the Advanced Spaceborne Thermal Emission and Reflection Radiometer (ASTER) aboard Terra. Though each classification was made with care, they each contain noise, with each pixel likely to have been misclassified one or more times. This could lead us to draw unrealistic conclusions if the classifications themselves were considered as a time series. For example, we would judge it highly unlikely that an area represented by a pixel would really be agriculture one day and revert to intact forest later in the month, only to be converted to agriculture again soon after, and so on. With careful (though unavoidably imperfect) classifications, we would expect that an area that had truly been converted to agriculture would consistently be classified as agriculture, while an area that remained as forest would be classified as that class most of the time. BULC's logic is to detect that persistence, extracting the true LULC change and stability from the noisy signal of the time series of classifications.

As you have seen in earlier chapters, creating classifications can be very involved and time consuming. To allow you to concentrate on BULC's efforts

Table 20.1 Images classified for updating Roosevelt River LULC with BULC

Sensor	Date	Spatial resolution (m)
Sentinel-2	2016: February 8 2017: July 7 2018: May 28 2019: June 17 2020: May 27 2021: May 27, July 11, August 15	10
Landsat 7	2017: August 16	30
Landsat 8	2021: July 18	30
ASTER	2017: July 15 2018: August 19 2019: June 19 2020: August 8	15–30

to clean noise from an existing `ImageCollection`, we have created the classifications already and stored them as an `ImageCollection` asset. You can view the Event time series using the `ui.Thumbnail` function, which creates an animation of the elements of the collection. Paste the code below into a new script to see those classifications drawn in sequence in the **Console**.

```
var events = ee.ImageCollection(
    'projects/gee-book/assets/F4-8/cleanEvents');
print(events, 'List of Events');
print('Number of events:', events.size());

print(ui.Thumbnail(events, {
    min: 0,
    max: 3,
    palette: ['black', 'green', 'blue', 'yellow'],
    framesPerSecond: 1,
    dimensions: 1000
}));
```

In the thumbnail sequence, the color palette shows Forest (class 1) as green, Water (class 2) as blue, and Active Agriculture (class 3) as yellow. Areas with no data in a particular Event are shown in black.

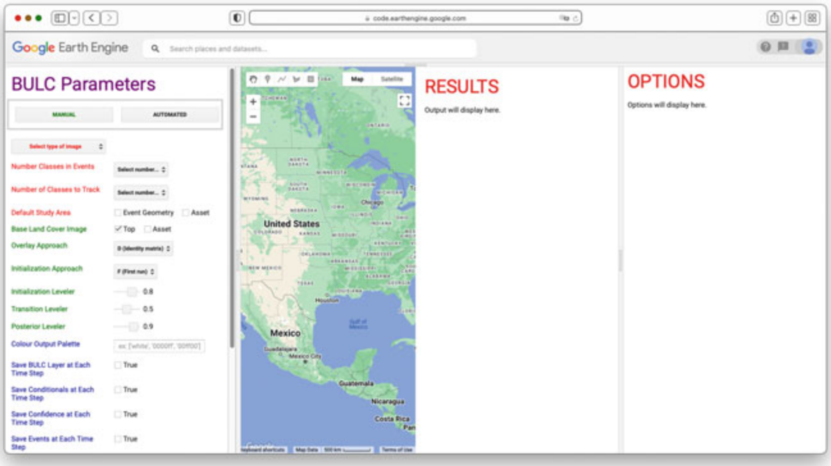
Code Checkpoint F48a. The book's repository contains a script that shows what your code should look like at this point.

20.2.2 Section 2: Basics of the BULC Interface

To see if BULC can successfully sift through these Events, we will use BULC's GUI (Fig. 20.1), which makes interacting with the functionality straightforward. **Code Checkpoint F48b** in the book's repository contains information about accessing that interface.

After you have run the script, BULC's interface requires that a few parameters be set; these are specified using the left panel. Here, we describe and populate each of the required parameters, which are shown in red. As you proceed, the default red color will change to green when a parameter receives a value.

- The interface permits new runs to be created using the **Manual** or **Automated** methods. The **Automated** setting allows information from a previous run to be used without manual entry. In this tutorial, we will enter each parameter individually using the interface, so you should set this item to **Manual** by clicking once on it.
- **Select Type of Image**: The interface can accept pre-made Event inputs in one of three forms: (1) as a stored `ImageCollection`; (2) as a single multi-banded `Image`; and (3) as a stream of Dynamic World classifications. The classifications are processed in the order they are given, either within the `ImageCollection` or sequentially through the `Image` bands. For this run, select **Image Collection** from the dropdown menu, then enter the path to this collection, without enclosing it in quotes: projects/gee-book/assets/F4-8/cleanEvents.



Number Classes in Events — Select number...
Number of Classes to Track — Select number...
Default Study Area — ☐ Event Geometry ☐ Asset
Base Land Cover Image — ✓ Top ☐ Asset
Overlay Approach — 0 (identity matrix)
Initialization Approach — F (First run)
Initialization Leveler — 0.8
Transition Leveler — 0.5
Posterior Leveler — 0.9
Colour Output Palette — ex: ['white', '0000ff', '00ff00']
Save BULC Layer at Each Time Step — ☐ True
Save Conditionals at Each Time Step — ☐ True
Save Confidence at Each Time Step — ☐ True
Save Events at Each Time Step — ☐ True

Fig. 20.1 BULC interface

- **Remap**: In some settings, you might want to remap the input value to combine classes. Leave this empty for now; an example of this is discussed later in the lab.
- **Number of Classes in Events** and **Number of Classes to Track**: The algorithm requires the number of classes in each Event and the number of meaningful classes to track to be entered. Here, there are 3 classes in each classification (Forest, Water, and Active Agriculture) and 3 classes being tracked. (In the BULC-U version of the algorithm (Lee et al. 2018, 2020), these numbers may be different when the Events are made using unsupervised classifications, which may contain many more classes than are being tracked in a given run.) Meaningful classes are assumed by BULC to begin with 1 rather than 0, while class values of 0 in Events are treated as no data. As seen in the thumbnail of Events, there are 3 classes; set both of these values to 3.
- The **Default Study Area** is used by BULC to delimit the location to analyze. This value can be pulled from a specially sized `Image` or set automatically, using the extent of the inputs. Set this parameter to **Event Geometry**, which gets the value automatically from the Event collection.
- The **Base Land Cover Image** defines the initial land cover condition to which BULC adds evidence from Events. Here, we are working to update the land cover map from the end of 2016, as estimated in Fortin et al. (2020). The ending estimated classification from that study has been loaded as an asset and placed as the first image in the input `ImageCollection`. We will direct the BULC interface to use this first image in the collection as the base land cover image by selecting **Top**.
- **Overlay Approach**: BULC can run in multiple modes, which affect the outcome of the classification updating. One option, **Overlay**, overlays each consecutive Event with the one prior in the sequence, following Cardille and Fortin (2016). Another option, **Custom**, allows a user-defined constant array to be used. For this tutorial, we will choose **D (Identity matrix)**, which uses the same transition table for every Event, regardless of how it overlays with the Event prior. That table gives a large conditional likelihood to the chance that classes agree strongly across the consecutive Event classifications that are used as inputs.
  BULC makes relatively small demands on memory since its arithmetic uses only multiplication, addition, and division, without the need for complex function fitting. The specific memory use is tied to the overlay method used. In particular, Event-by-Event comparisons (the **Overlay** setting) are considerably more computationally expensive than pre-defined transition tables (the **Identity** and **Custom** settings). The maximum working Event depth is also slightly lowered when intermediate probability values are returned for inspection. Our tests indicate that with pre-defined truth tables and no intermediate probability values returned, BULC can handle updating problems hundreds of Events deep across an arbitrarily large area.
- **Initialization Approach**: If a BULC run of the full intended size ever surpassed the memory available, you would be able to break the processing into two or

more parts using the following technique. First, create a slightly smaller run that can complete, and save the final probability image of that run. Because of the operation of Bayes' theorem, the ending probability multi-band image can be used as the prior probability to continue processing Events with the same answers as if it had been run all at once. For this small run, we will select **F (First Run)**.

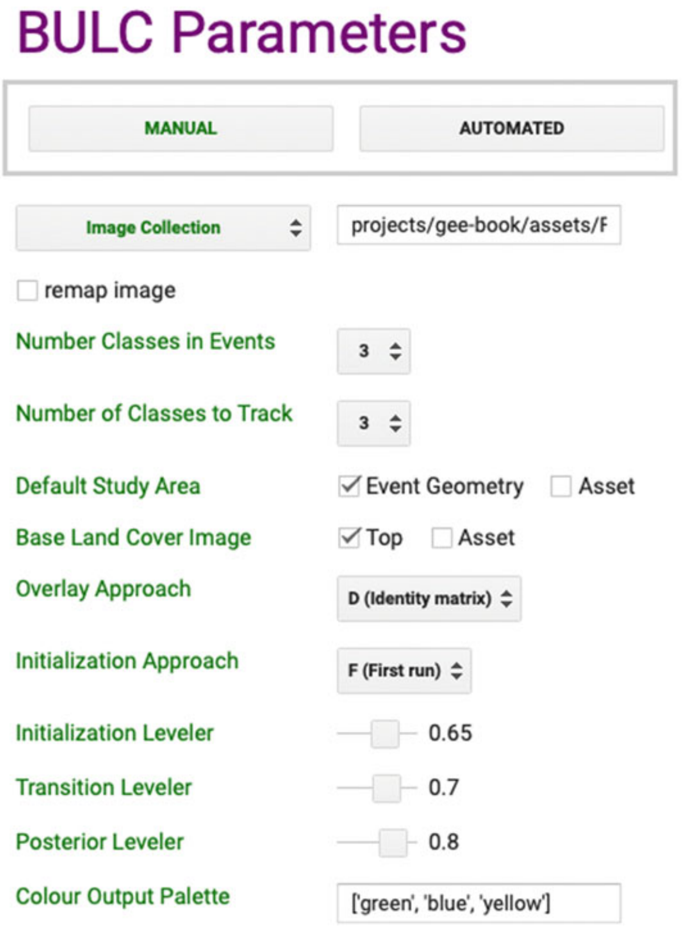
- **Levelers**: BULC uses three levelers as part of its processing, as described in Cardille and Fortin (2016). The **Initialization Leveler** creates the initial probability vector of the initial LULC image; the **Transition Leveler** dampens transitions at each time step, making BULC less reactive to new evidence; and the **Posterior Leveler** ensures that each class retains nonzero probability so that the Bayes formula can function properly throughout the run. For this run, set the parameters to 0.65, 0.3, and 0.6, respectively. This corresponds to a typical set of values that is appropriate when moderate-quality classifications are fed to BULC.
- **Color Output Palette**: We will use the same color palette as what was seen in the small script you used to draw the Events, with one exception. Because BULC will give a value for the estimated class for every pixel, there are no pixels in the study area with missing or masked data. To line up the colors with the attainable numbers, we will remove the color 'black' from the specification. For this field, enter this list: ['green', 'blue', 'yellow']. For all of the text inputs, make sure to click outside that field after entering text so that the input information is registered; the changing of the text color to green confirms that the information was received.

When you have finished setting the required parameters, the interface will look like Fig. 20.2.

Beneath the required parameters is a set of optional parameters that affect which intermediate results are stored during a run for later inspection. We are also given a choice of returning intermediate results for closer inspection. At this stage, you can leave all optional parameters out of the BULC call by leaving them blanked or unchecked.

After clicking the **Apply Parameters** button at the bottom of the left panel, the classifications and parameters are sent to the BULC modules. The **Map** will move to the study area, and after a few seconds, the **Console** will hold new thumbnails. The uppermost thumbnail is a rapidly changing view of the input classifications. Beneath that is a thumbnail of the same area as interpreted by BULC. Beneath those is a Confidence thumbnail, which is discussed in detail later in this lab.

The BULC interpretation of the landscape looks roughly like the Event inputs, but it is different in two important ways. First, depending on the leveler settings, it will usually have less noise than the Event classifications. In the settings above, we used the Transition and Posterior levelers to tell BULC to trust past accumulated evidence more than a single new image. The second key difference between the BULC result and the input classifications is that even when the inputs don't cover the whole area at each time step, BULC provides an estimate in every pixel at each



BULC Parameters

projects/gee-book/assets/F

☐ remap image

Number Classes in Events	3
Number of Classes to Track	3
Default Study Area	✓ Event Geometry ☐ Asset
Base Land Cover Image	✓ Top ☐ Asset
Overlay Approach	D (Identity matrix)
Initialization Approach	F (First run)
Initialization Leveler	0.65
Transition Leveler	0.7
Posterior Leveler	0.8
Colour Output Palette	['green', 'blue', 'yellow']

Fig. 20.2 Initial settings for the key driving parameters of BULC

time step. To create this continuous classification, if a new classification does not have data for some part of the study area (beyond the edge of a given image, for example), the last best guess from the previous iteration is carried forward. Simply put, the estimate in a given pixel is kept the same until new data arrives.

Meanwhile, below the **Console**, the rest of the interface changes when BULC is run. The **Map** panel displays BULC's classification for the final date: that is, after considering the evidence from each of the input classifications. We can use the **Satellite** background to judge whether BULC is accurately capturing the state of LULC. This can be done by unselecting the drawn layers in the map layer set and selecting **Satellite** from the choices in the upper-right part of the **Map** panel. Earth Engine's background satellite images are often updated, so you should see something like the right side of Fig. 20.3, though it may differ slightly.

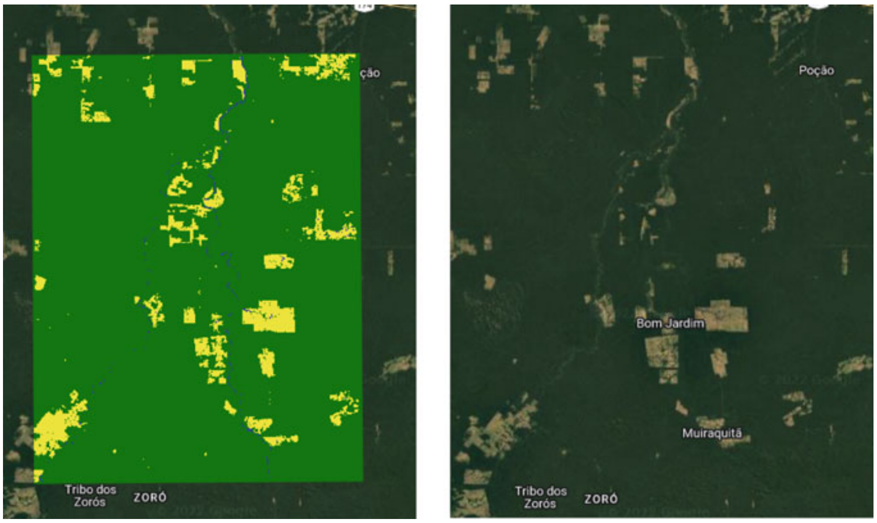


Fig. 20.3 BULC estimation of the state of LULC at the end of 2021 (left). Satellite backdrop for Earth Engine (right), which may differ from what you see due to updates

Question 1. When comparing the BULC classification for 2021 against the current Earth Engine satellite view, what are the similarities and differences? Note that in Earth Engine, the copyrighted year numbers at the bottom of the screen may not coincide with the precise date of the image shown.

In the rightmost panel below the **Console**, the interface offers you multiple options for viewing the results. These include:

1. **Movie**. This uses the `ui.Thumbnail` API function to draw the BULC results rapidly in the viewer. This option offers you a control on the frame rate (in frames per second), and a checkbox affecting the drawing resolution. The high-resolution option uses the maximum resolution permitted given the function's constraints. A lower resolution setting constructs the thumbnail more quickly, but at a loss of detail.
2. **Filmstrip**. This produces an image like the Movie option, but allows you to move on request through each image.
3. **Mosaic**. This draws every BULC result in the panel. Depending on the size of the stack of classifications, this could become quite a large set of images.
4. **Zoom**. This draws the final BULC classification at multiple scales, with the finest-scale image matching that shown in the **Map** window.

Question 2. Select the **BULC** option, then select the **Movie** tool to view the result, and choose a drawing speed and resolution. When viewing the full area, would you assess the additional LULC changes since 2016 as being minor, moderate, or major compared to the changes that occurred before 2016? Explain the reasoning for your assessment.

20.2.3 Section 3: Detailed LULC Inspection with BULC

BULC results can be viewed interactively, allowing you to view more detailed estimations of the LULC around the study area. We will zoom into a specific area where change did occur after 2016. To do that, turn on the **Satellite** view and zoom in. Watching the scale bar in the lower right of the **Map** panel, continue zooming until the scale bar says **5 km**. Then, enter "−60.7, −9.83" in the Earth Engine search tool, located above the code. The text will be interpreted as a longitude/latitude value and will offer you a nearby coordinate, indicated with a value for the degrees West and the degrees South. Click that entry and Earth Engine will move to that location, while keeping at the specified zoom level. Let us compare the BULC result in this sector against the image from Earth Engine's satellite view that is underneath it (Fig. 20.4).

BULC captured the changes between 2016 and 2021 with a classification series that suggests agricultural development (Fig. 20.4, left). Given the appearance of BULC's 2021 classification, it suggests that the satellite backdrop at the time of this writing (Fig. 20.4, right) came from an earlier time period.

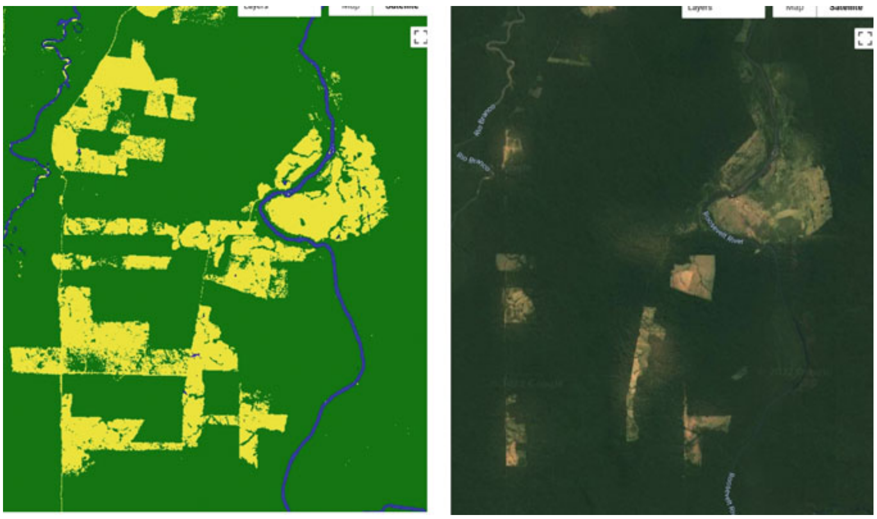


Fig. 20.4 Comparison of the final classification of the northern part of the study area to the satellite view

Now, in the **Results** panel, select **BULC**, then **Movie**. Set your desired frame speed and resolution, then select **Redraw Thumbnail**. Then, zoom the main **Map** even closer to some agriculture that appears to have been established between 2016 and 2021. Redraw the thumbnail movie as needed to find an interesting set of pixels.

With this finer-scale access to the results of BULC, you can select individual pixels to inspect. Move the horizontal divider downward to expose the **Inspector** tab and **Console** tab. Use the **Inspector** to click on several pixels to learn their history as expressed in the inputted Events and in BULC's interpretation of the noise and signal in the Event series. In a chosen pixel, you might see output that looks like Fig. 20.5. It indicates a possible conversion in the Event time series after a few classifications of the pixel as Forest. This decreases the confidence that the pixel is still Forest (Fig. 20.5, lower panel), but not enough for the Active Agriculture class (class 3) to become the dominant probability. After the subsequent Event labels the pixel as Forest, the confidence (lower panel) recovers slightly, but not to its former level. The next Event classifies the pixel as Active Agriculture, confidently, by interpreting that second Active Agriculture classification, in a setting where change was already somewhat suspected after the first non-Forest classification. BULC's label (middle panel) changes to be Active Agriculture at that point in the sequence. Subsequent Event classifications as Active Agriculture creates a growing confidence that its proper label at the end of the sequence was indeed Active Agriculture.

Question 3. Run the code again with the same data, but adjust the three levelers, then view the results presented in the **Map** window and the **Results** panel. How do each of the three parameters affect the behavior of BULC in its results? Use the thumbnail to assess your subjective satisfaction with the results, and use the **Inspector** to view the BULC behavior in individual pixels. Can you produce an optimal outcome for this given set of input classifications?

20.2.4 Section 4: Change Detection with BULC-D

What if we wanted to identify areas of likely change or stability without trying to identify the initial and final LULC class? BULC-D is an algorithm that estimates, at each time step, the probability of noteworthy change. The example below uses the Normalized Burn Ratio (NBR) as a gauge: BULC-D assesses whether the ratio has meaningfully increased, decreased, or remained the same. It is then the choice of the analyst to decide how to treat these assessed probabilities of stability and change.

BULC-D involves determining an expectation for an index across a user-specified time period and then comparing new values against that estimation. Using Bayesian logic, BULC-D then asks which of three hypotheses is most likely, given evidence from the new values to date from that index. The hypotheses are simple: Either the value has decreased meaningfully, or it has increased meaningfully, or it

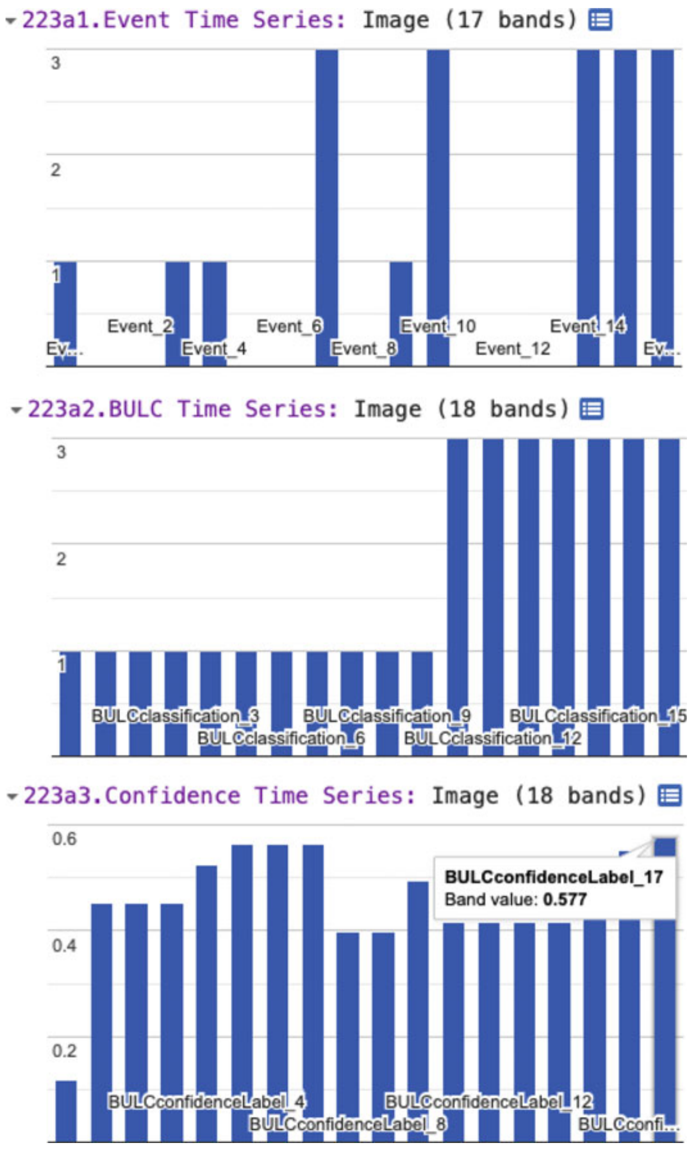


Fig. 20.5 History for 2016–2020 for a pixel that appeared to have been newly cultivated during that period. (above): the input classifications, which suggest a possible conversion from class 1 (Forest) to class 3 (Active Agriculture) midway through the time series. (middle): BULC's interpretation of the evidence, which changes its estimated classification based on the evidence, in this case after two occurrences of it being classified as Active Agriculture. (below): BULC's confidence in its estimation. This number grows in the initial part of the series as more classifications calling this pixel Forest classes are encountered, then drops as conflicting evidence is seen. Eventually, after more Active Agriculture classifications are encountered, its confidence in that new class grows

has not changed substantially compared to the previously established expectation. The details of the workings of BULC-D are beyond the scope of this exercise, but we provide it as a tool for exploration. BULC-D's basic framework is the following:

- *Establish*: Fit a harmonic curve with a user-specified number of terms to a stream of values from an index, such as the Normalized Difference Vegetation Index (NDVI), and NBR.
- *Standardize*: For each new image, quantify the deviation of the index's value from the expectation on that date.
- *Contextualize*: Assess the magnitude of that deviation in one of several ordered bins.
- *Synthesize*: Use the BULC framework to adjust the vector of change for the three possibilities: the value went down, the value stayed the same, and the value went up.

It is worth noting that BULC-D does not label the change with a LULC category; rather, it trains itself to distinguish likely LULC change from expected variability. In this way, BULC-D can be thought of as a "sieve" through which you are able to identify locations of possible change, isolated from likely background noise. In the BULC-D stage, the likeliness of change is identified across the landscape; in a separate second stage, the meaning of those changes and any changes to LULC classes are identified. We will explore the workings of BULC-D using its GUI.

Code Checkpoint F48c. The book's repository contains information about accessing that interface.

After you have run the script to initialize the interface, BULC-D's interface requires a few parameters to be set. For this run of BULC-D, we will set the parameters to the following:

- Expectation years: 2020
- Target year: 2021
- Sensors: Landsat and Sentinel
- Index: NBR
- Harmonic fit: Yes, 1 harmonic term.

Run BULC-D for this area. As a reminder, you should first zoom in enough that the scale bar reads "5 km" or finer. Then, search for the location "− 60.7624, − 9.8542". When you run BULC-D, a result like Fig. 20.6 is shown for the layer of probabilities.

The BULC-D image (Fig. 20.6) shows each pixel as a continuous three-value vector along a continuous range; the three values sum to 1. For example, a vector with values of [0.85, 0.10, 0.05] would represent an area estimated with high confidence according to BULC-D to have experienced a sustained drop in NBR in the target period compared to the values set by the expectation data. In that pixel,

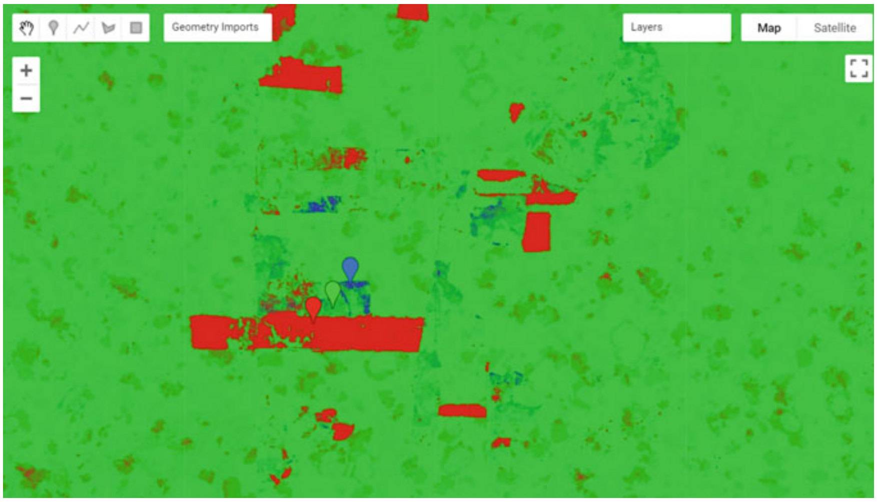


Fig. 20.6 Result for BULC-D for the Roosevelt River area, depicting estimated probability of change and stability for 2021

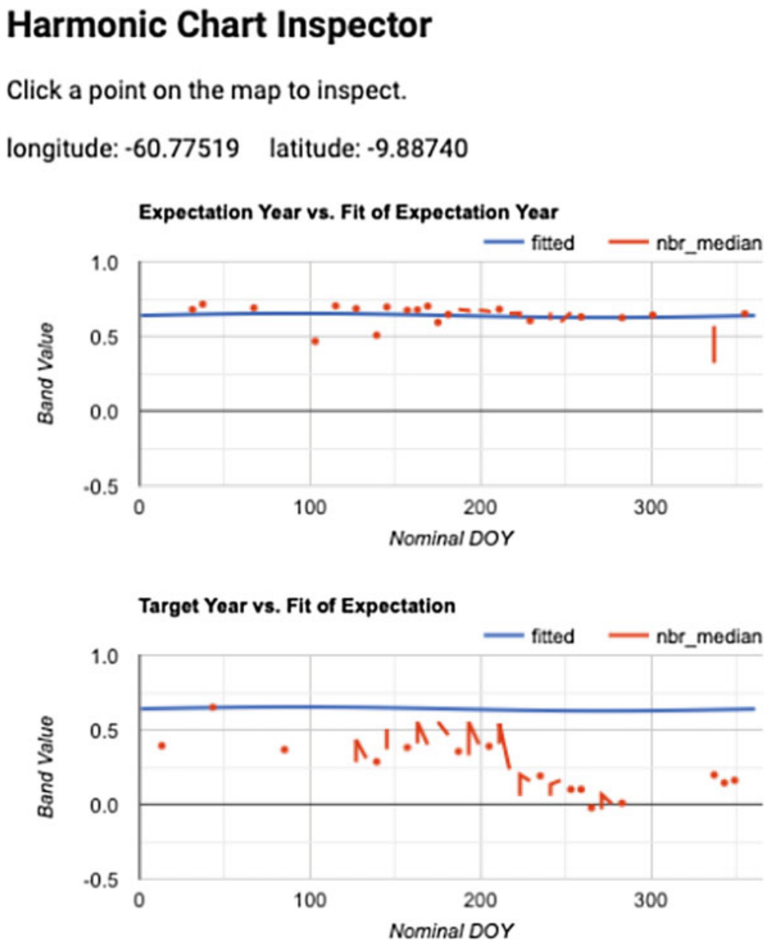
the combination of three colors would produce a value that is richly red. You can see Chap. 2 for more information on drawing bands of information to the screen using the red–green–blue additive color model in Earth Engine.

Each pixel experiences its own NBR history in both the expectation period and the target year. Next, we will highlight the history of three nearby areas: one, marked with a red balloon in your interface, that BULC assessed as having experienced a persistent drop in NBR; a second in green assessed to not have changed, and a third in blue assessed to have witnessed a persistent NBR increase.

Figure 20.7 shows the NBR history for the red balloon in the southern part of the study area in Fig. 20.4. If you click on that pixel or one like it, you can see that, whereas the values were quite stable throughout the growing season for the years used to create the pixel's expectation, they were persistently lower in the target year. This is flagged as a likely meaningful drop in the NBR by BULC-D, for consideration by the analyst.

Figure 20.8 shows the NBR history for the blue balloon in the southern part of the study area in Fig. 20.4. For that pixel, while the values were quite stable throughout the growing season for the years used to create the pixel's expectation, they were persistently higher in the target year.

Question 4. Experiment with turning off one of the satellite sensor data sources used to create the expectation collection. For example, do you get the same results if the Sentinel-2 data stream is not used, or is the outcome different. You might make screen captures of the results to compare with Fig. 20.4. How strongly does each satellite stream affect the outcome of the estimate? Do differences in the resulting estimate vary across the study area?

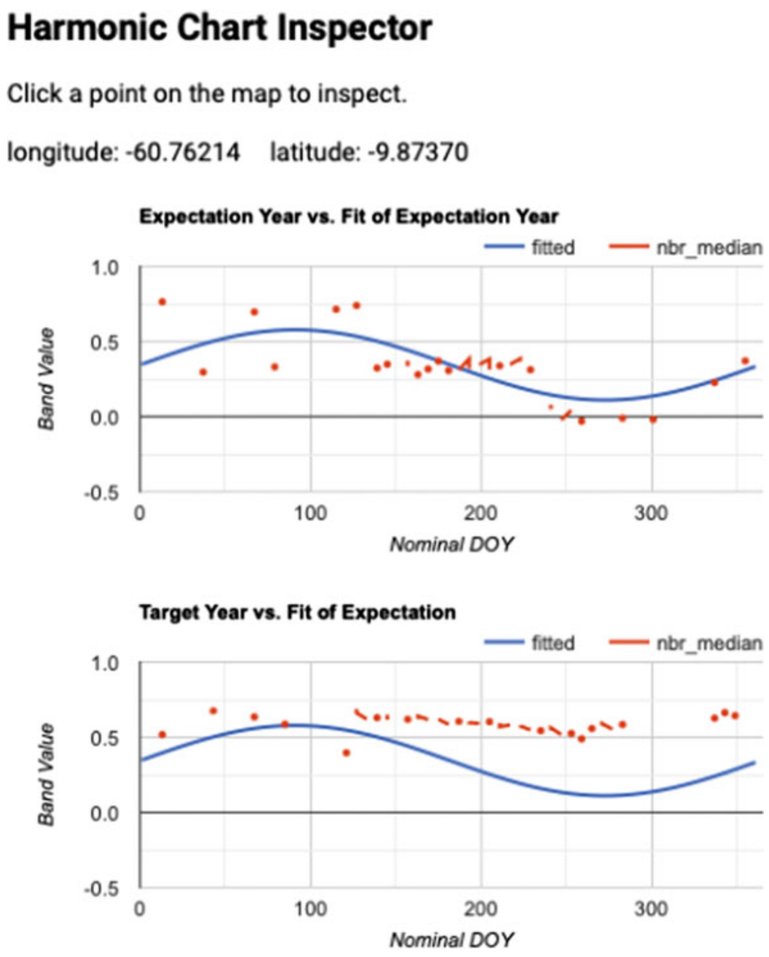

Harmonic Chart Inspector

Click a point on the map to inspect.

longitude: -60.77519   latitude: -9.88740

Fig. 20.7 NBR history for a pixel with an apparent drop in NBR in the target year (below) as compared to the expectation years (above). Pixel is colored a shade of red in Fig. 20.6

Figure 20.8 also shows that, for that pixel, the fit of values for the years used to build the expectation showed a sine wave (shown in blue), but with a fit that was not very strong. When data for the target year was assembled (Fig. 20.8, bottom), the values were persistently above expectation throughout the growing season. Note that this pixel was identified as being different in the target year as compared to earlier years, which does not rule out the possibility that the LULC of the area was changed (e.g., from Forest to Agriculture) during the years used to build the expectation collection. BULC-D is intended to be run steadily over a long period of time, with the changes marked as they occur, after which point the expectation would be recalculated.



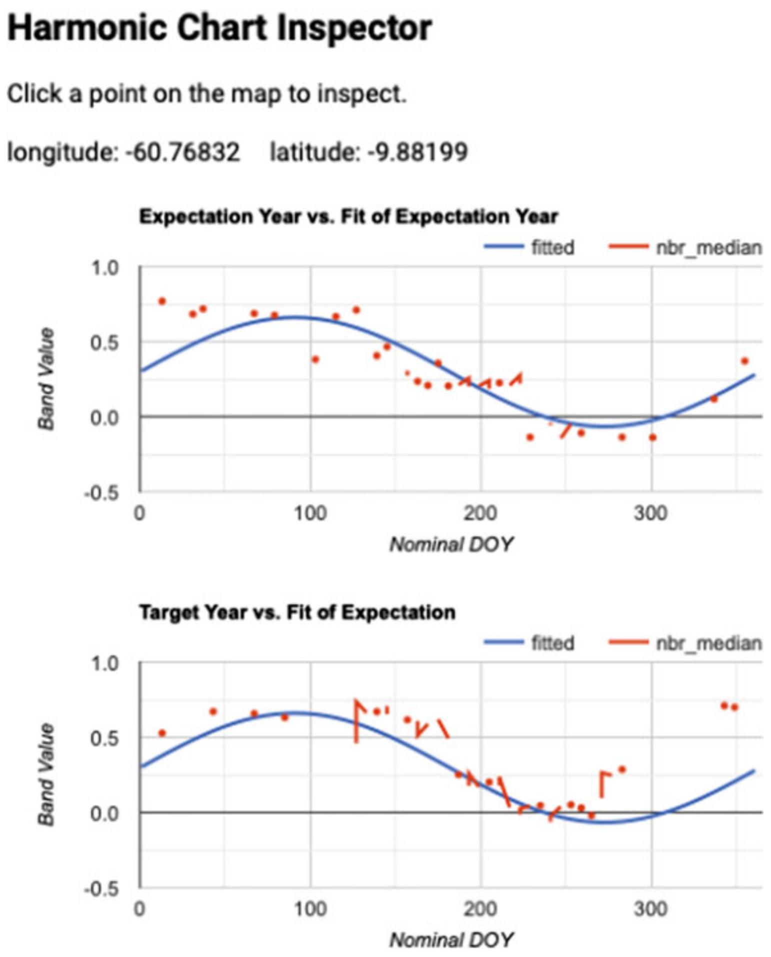
Harmonic Chart Inspector

Click a point on the map to inspect.

longitude: -60.76214 latitude: -9.87370

Fig. 20.8 NBR history for a pixel with an apparent increase in NBR in the target year (below) as compared to the expectation years (above). Pixel is colored a shade of blue in Fig. 20.6

Figure 20.9 shows the NBR history for the green balloon in the southern part of the study area in Fig. 20.4. For that pixel, the values in the expectation collection formed a sine wave, and the values in the target collection deviated only slightly from the expectation during the target year.


Harmonic Chart Inspector

Click a point on the map to inspect.

longitude: -60.76832 latitude: -9.88199

Fig. 20.9 NBR history for a pixel with no apparent increase or decrease in NBR in the target year (below) as compared to the expectation years (above). Pixel is colored a shade of green in Fig. 20.6

20.2.5 Section 5: Change Detection with BULC and Dynamic World

Recent advances in neural networks have made it easier to develop consistent models of LULC characteristics using satellite data. The Dynamic World project (Brown et al. 2022) applies a neural network, trained on a very large number of images, to each new Sentinel-2 image soon after it arrives. The result is a near-real-time classification interpreting the LULC of Earth's surface, kept continually up to date with new imagery.

What to do with the inevitable inconsistencies in a pixel's stated LULC class through time? For a given pixel on a given image, its assigned class label is chosen by the Dynamic World algorithm as the maximum class probability given the band values on that day. Individual class probabilities are given as part of the dataset and could be used to better interpret a pixel's condition and perhaps its history. Future work with BULC will involve incorporating these probabilities into BULC's probability-based structure. For this tutorial, we will explore the consistency of the assigned labels in this same Roosevelt River area as a way to illustrate BULC's potential for minimizing noise in this vast and growing dataset.

20.2.5.1 Section 5.1: Using BULC to Explore and Refine Dynamic World Classifications

Code Checkpoint A48d. The book's repository contains a script to use to begin this section. You will need to load the linked script and run it to begin.

After running the linked script, the BULC interface will initialize. Select **Dynamic World** from the dropdown menu where you earlier selected **Image Collection**. When you do, the interface opens several new fields to complete. BULC will need to know where you are interested in working with Dynamic World, since it could be anywhere on Earth. To specify the location, the interface field expects a nested list of lists of lists, which is modeled after the structure used inside the constructor `ee.Geometry.Polygon`. (When using drawing tools or specifying study areas using coordinates, you may have noticed this structure.) Enter the following nested list in the text field near the **Dynamic World** option, without enclosing it in quotes:

```
[[[-61.155, -10.559], [-60.285, -10.559], [-60.285, -9.436],
[-61.155, -9.436]]]
```

Next, BULC will need to know which years of Dynamic World you are interested in. For this exercise, select **2021**. Then, BULC will ask for the Julian days of the year that you are interested in. For this exercise, enter 150 for the start day and 300 for the end day. Because you selected Dynamic World for analysis in BULC, the interface defaults to offering the number 9 for the number of classes in Events and for the number of classes to track. This number represents the full set of classes in the Dynamic World classification scheme. You can leave other required settings shown in green with their default values. For the Color Output Palette, enter the following palette without quotes. This will render results in the Dynamic World default colors.

```
['419BDF', '397D49', '88B053'
, '7A87C6', 'E49635', 'DFC35A'
, 'C4281B', 'A59B8F', 'B39FE1']
```

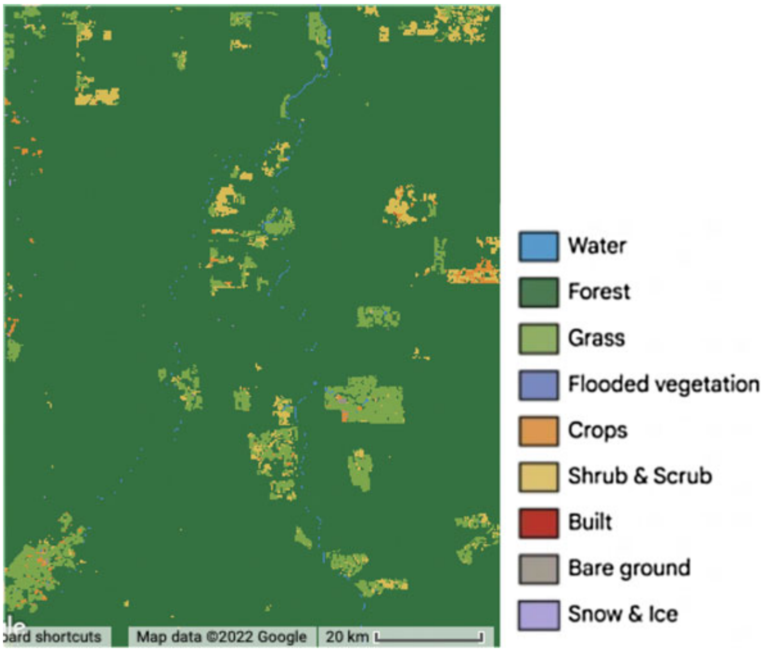
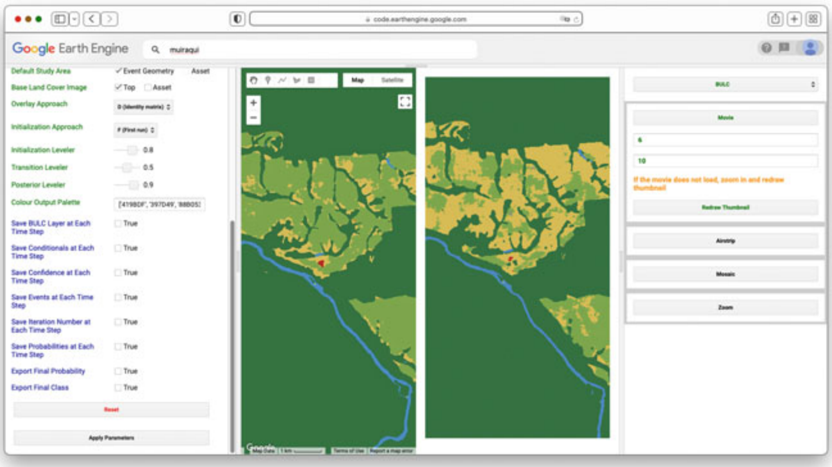


Fig. 20.10 BULC classification using default settings for Roosevelt River area for late 2021

When you have finished, select **Apply Parameters** at the bottom of the input panel. After BULC subsets the Dynamic World dataset to clip out according to the dates and location, identifying images from more than 40 distinct dates. The area covers two of the tiles in which Dynamic World classifications are partitioned to be served, so BULC receives more than 90 classifications. When BULC finishes its run, the **Map** panel will look like Fig. 20.10, BULC's estimate of the final state of the landscape at the end of the classification sequence.

Let us explore the suite of information returned by BULC about this time period in Dynamic World. Enter "Muiraquitã" in the search bar and view the results around that area to be able to see the changing LULC classifications within farm fields. Then, begin to inspect the results by viewing a **Movie** of the Events, with a data frame rate of 6 frames per second. Because the study area spans multiple Dynamic World tiles, you will find that many Event frames are black, meaning that there was no data in your sector on that particular image. Because of this, and also perhaps because of the very aggressive cloud masking built into Dynamic World, viewing Events (which, as a reminder, are the individual classified images directly from Dynamic World) can be a very challenging way to look for change and stability. BULC's goal is to sift through those classifications to produce a time series that reflects, according to its estimation, the most likely LULC value at each time step. View the **Movie** of the BULC results and ask yourself whether each



Default Study Area ✓ Event Geometry Asset
Base Land Cover Image ✓ Top ☐ Asset
Overlay Approach 0 (identity matrix)
Initialization Approach # (First run)
Initialization Leveler 0.8
Transition Leveler 0.5
Posterior Leveler 0.9
Colour Output Palette ['419BDF', '397D49', '88B053
Save BULC Layer at Each Time Step ☐ True
Save Conditionals at Each Time Step ☐ True
Save Confidence at Each Time Step ☐ True
Save Events at Each Time Step ☐ True
Save Iteration Number at Each Time Step ☐ True
Save Probabilities at Each Time Step ☐ True
Export Final Probability ☐ True
Export Final Class ☐ True

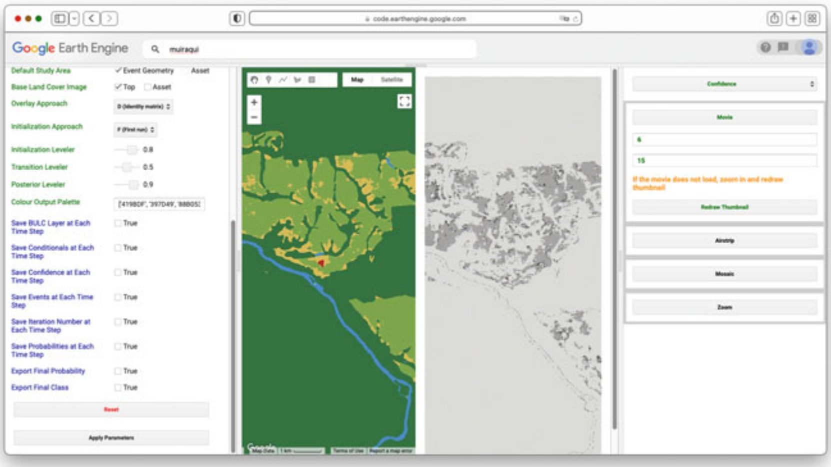
6
10
If the movie does not load, zoom in and redraw thumbnail

Fig. 20.11 Still frame (right image) from the animation of BULC's adjusted estimate of LULC through time near Muiraquitã

class is equally well replicated across the set of classifications. A still from midway through the **Movie** sequence of the BULC results can be seen in Fig. 20.11.

As BULC uses the classification inputs to estimate the state of the LULC at each time step, it also tracks its confidence in those estimates. This is shown in several ways in the interface.

- You can view a **Movie** of BULC's confidence through time as it reacts to the consistency or variability of the class identified in each pixel by Dynamic World. View that movie now over this area to see the evolution of BULC's confidence through time of the class of each pixel. A still frame from this movie can be seen in Fig. 20.12. The frame and animation indicate that BULC's confidence is lowest in pixels where the estimate flips between similar categories, such as Grass and Shrub and Scrub. It also is low at the edges of land covers, even where the covers (such as Forest and Water) are easy to discern from each other.
- You can inspect the final confidence estimate from BULC, which is shown as a grayscale image in the set of **Map** layers in the left lower panel. That single layer synthesizes how, across many Dynamic World classifications, the confidence in certain LULC classes and locations is ultimately more stable than in others. For example, generally speaking, the Forest class is classified consistently across this assemblage of Dynamic World images. Agriculture fields are less consistently classified as a single class, as evidenced by their relatively low confidence.
- Another way of viewing BULC's confidence is through the **Inspector** tab. You can click on individual pixels to view their values in the Event time



Default Study Area	✓ Event Geometry Asset
Base Land Cover Image	✓ Top ☐ Asset
Overlay Approach	0 (identity matrix)
Initialization Approach	F (First run)
Initialization Leveler	0.8
Transition Leveler	0.5
Posterior Leveler	0.9
Colour Output Palette	['419BDF', '397D49', '88B05]
Save BULC Layer at Each Time Step	☐ True
Save Conditionals at Each Time Step	☐ True
Save Confidence at Each Time Step	☐ True
Save Events at Each Time Step	☐ True
Save Iteration Number at Each Time Step	☐ True
Save Probabilities at Each Time Step	☐ True
Export Final Probability	☐ True
Export Final Class	☐ True

6

15

If the movie does not load, zoom in and redraw thumbnail

Fig. 20.12 Still frame from the animation of changing confidence through time, near Muiraquitã

series and in the BULC time series, and see BULC's corresponding confidence value changing through time in response to the relative stability of each pixel's classification.

- Another way to view BULC's confidence estimation is as a hillshade enhancement of the final BULC classification. If you select the Probability Hillshade in the set of **Map** layers, it shows the final BULC classification as a textured surface, in which you can see where lower-confidence pixels are classified.

20.2.5.2 Section 5.2: Using BULC to Visualize Uncertainty of Dynamic World in Simplified Categories

In the previous section, you may have noticed that there are two main types of uncertainty in BULC's assessment of long-term classification confidence. One type is due to spatial uncertainty at the edge of two relatively distinct phenomena, like the River/Forest boundary visible in Fig. 20.12. These are shown in dark tones in the confidence images, and emphasized in the Probability Hillshade. The other type of uncertainty is due to some cause of labeling uncertainty, due either (1) to the similarity of the classes, or (2) to persistent difficulty in distinguishing two distinct classes that are meaningfully different but spectrally similar. An example of uncertainty due to similar labels is distinguishing flooded and non-flooded wetlands in classifications that contain both those categories. An example of difficulty distinguishing distinct but spectrally similar classes might be distinguishing a parking lot from a body of water.

BULC allows you to remap the classifications it is given as input, compressing categories as a way to minimize uncertainty due to similarity among classes. In the setting of Dynamic World in this study area, we notice that several classes are functionally similar for the purposes of detecting new deforestation: Farm fields and pastures are variously labeled on any given Dynamic World classification as Grass, Flooded Vegetation, Crops, Shrub and Scrub, Built, or Bare Ground. What if we wanted to combine these categories to be similar to the distinctions of the classified Events from this lab's Sect. 20.2.1? The classes in that section were Forest, Water, and Active Agriculture. To remap the Dynamic World classification, continue with the same run as in Sect. 20.2.5.1. Near where you specified the location for clipping Dynamic World, there are two fields for remapping. Select the **Remap** checkbox and in the "from" field, enter (without quotes):

```
0, 1, 2, 3, 4, 5, 6, 7, 8
```

In the "to" field, enter (without quotes):

```
1, 0, 2, 2, 2, 2, 2, 2, 0
```

This directs BULC to create a three-class remap of each Dynamic World image. Next, in the area of the interface where you specify the palette, enter the same palette used earlier:

```
['green', 'blue', 'yellow']
```

Before continuing, think for a moment about how many classes you have now. From BULC's perspective, the Dynamic World events will have 3 classes and you will be tracking 3 classes. Set both the **Number of Classes in Events** and **Number of Classes to Track** to 3. Then click **Apply Parameters** to send this new run to BULC.

The confidence image shown in the main **Map** panel is instructive (Fig. 20.13). Using data from 2020, 2021, and 2022, it indicates that much of the uncertainty among the original Dynamic World classifications was in distinguishing labels within agricultural fields. When that uncertainty is removed by combining classes, the BULC result indicates that a substantial part of the remaining uncertainty is at the edges of distinct covers. For example, in the south-central and southern part of the frame, much of the uncertainty among classifications in the original Dynamic World classifications was due to distinction among the highly similar, easily confused classes. Much of what remained (right) after remapping (right) formed outlines of the river and the edges between farmland and forest: a graphic depiction of the "spatial uncertainty" discussed earlier. Yet not all of the uncertainty was spatial; the thicker, darker areas of uncertainty even after remapping (right, at the extreme eastern edge for example) indicates a more fundamental disagreement in the classifications. In those pixels, even when the Agriculture-like classes were compressed, there was still considerable uncertainty (likely between

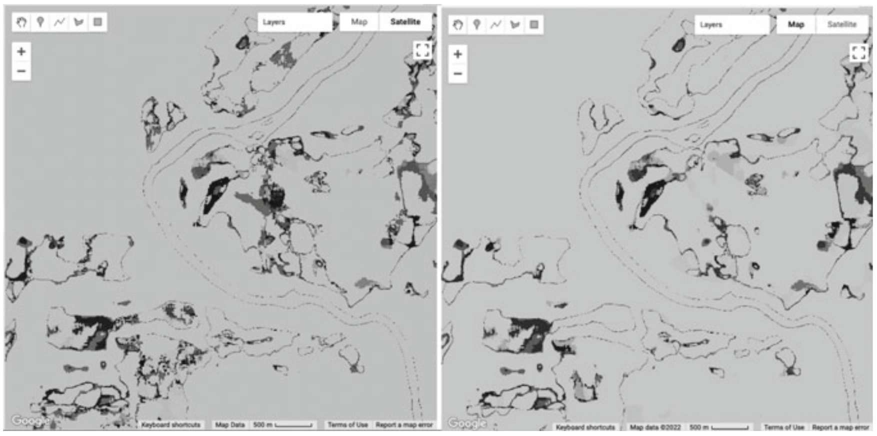


Fig. 20.13 Final confidence layer from the run with (left) and without (right) remapping to combine similar LULC classes to distinguish Forest, Water, and Active Agriculture near − 60.696W, − 9.826S

Forest and Active Agriculture) in the true state of these areas. These might be of further interest: were they places newly deforested in 2020–2022? Were they abandoned fields regrowing? Were they degraded at some point? The mapping of uncertainty may hold promise for a better understanding of uncertainty as it is encountered in real classifications, thanks to Dynamic World.

Given the tools and approaches presented in this lab, you should now be able to import your own classifications for BULC (Sects. 20.2.1, 20.2.2 and 20.2.3), detect changes in sets of raw imagery (Sect. 20.2.4), or use Dynamic World's pre-created classifications (Sect. 20.2.5). The following exercises explore this potential.

20.3 Synthesis

Assignment 1. For a given set of classifications as inputs, BULC uses three parameters that specify how strongly to trust the initial classification, how heavily to weigh the evidence of each classification, and how to adjust the confidence at the end of each time step. For this exercise, adjust the values of these three parameters to explore the strength of the effect they can have on the BULC results.

Assignment 2. The BULC-D framework produces a continuous three-value vector of the probability of change at each pixel. This variability accounts for the mottled look of the figures when those probabilities are viewed across space. Use the **Inspector** tool or the interface to explore the final estimated probabilities, both numerically and as represented by different colors of pixels in the given example. Compare and contrast the mean NBR values from the earlier and later years, which are drawn in the **Layer** list. Then answer the following questions:

(1) In general, how well does BULC-D appear to be identifying locations of likely change?
(2) Does one type of change (decrease, increase, no change) appear to be mapped better than the others? If so, why do you think this is?

Assignment 3. The BULC-D example used here was for 2021. Run it for 2022 or later at this location. How well do results from adjacent years complement each other?

Assignment 4. Run BULC-D in a different area for a year of interest of your choosing. How do you like the results?

Assignment 5. Describe how you might use BULC-D as a filter for distinguishing meaningful change from noise. In your answer, you can consider using BULC-D before or after BULC or some other time-series algorithm, like CCDC or LandTrendr.

Assignment 6. Analyze stability and change with Dynamic World for other parts of the world and for other years. For example, you might consider:

(a) Quebec, Canada, days 150–300 for 2019 and 2020:

[[[− 71.578, 49.755], [− 71.578, 49.445], [− 70.483, 49.445], [− 70.483, 49.755]]]

Location of a summer 2020 fire
(b) Addis Ababa, Ethiopia: [[[38.79, 9.00], [38.79, 8.99], [38.81, 8.99], [38.81, 9.00]]]
(c) Calacalí, Ecuador: [[[− 78.537, 0.017], [− 78.537, − 0.047], [− 78.463, − 0.047], [− 78.463, 0.017]]]
(d) Irpin, Ukraine: [[[30.22, 50.58], [30.22, 50.525], [30.346, 50.525], [30.346, 50.58]]]
(e) A different location of your own choosing. To do this, use the Earth Engine drawing tools to draw a rectangle somewhere on Earth. Then, at the top of the **Import** section, you will see an icon that looks like a sheet of paper. Click that icon and look for the polygon specification for the rectangle you drew. Paste that into the location field for the Dynamic World interface.

20.4 Conclusion

In this lab, you have viewed several related but distinct ways to use Bayesian statistics to identify locations of LULC change in complex landscapes. While they are standalone algorithms, they are each intended to provide a perspective either on the likelihood of change (BULC-D) or of extracting signal from noisy classifications (BULC). You can consider using them especially when you have pixels that, despite your best efforts, periodically flip back and forth between similar but different classes. BULC can help ignore noise, and BULC-D can help reveal whether this year's signal has precedent in past years.

To learn more about the BULC algorithm, you can view this interactive probability illustration tool by a link found in script **F48s1—Supplemental** in the book's repository. In the future, after you have learned how to use the logic of BULC, you might prefer to work with the JavaScript code version. To do that, you can find a tutorial at the website of the authors.

References

Brown CF, Brumby SP, Guzder-Williams B et al (2022) Dynamic world, near real-time global 10 m land use land cover mapping. Sci Data 9:1–17. https://doi.org/10.1038/s41597-022-01307-4

Cardille JA, Fortin JA (2016) Bayesian updating of land-cover estimates in a data-rich environment. Remote Sens Environ 186:234–249. https://doi.org/10.1016/j.rse.2016.08.021

Fortin JA, Cardille JA, Perez E (2020) Multi-sensor detection of forest-cover change across 45 years in Mato Grosso, Brazil. Remote Sens Environ 238. https://doi.org/10.1016/j.rse.2019.111266

Lee J, Cardille JA, Coe MT (2018) BULC-U: sharpening resolution and improving accuracy of land-use/land-cover classifications in Google Earth Engine. Remote Sens 10. https://doi.org/10.3390/rs10091455

Lee J, Cardille JA, Coe MT (2020) Agricultural expansion in Mato Grosso from 1986–2000: a Bayesian time series approach to tracking past land cover change. Remote Sens 12. https://doi.org/10.3390/rs12040688

Millard C (2006) The river of doubt: Theodore Roosevelt's darkest journey. Anchor

Exploring Lagged Effects in Time Series

21

Andréa Puzzi Nicolau, Karen Dyson, David Saah, and Nicholas Clinton

Overview
In this chapter, we will introduce lagged effects to build on the previous work in modeling time series data. Time-lagged effects occur when an event at one point in time impacts dependent variables at a later point in time. You will be introduced to concepts of autocovariance and autocorrelation, cross-covariance and cross-correlation, and auto-regressive models. At the end of this chapter, you will be able to examine how variables relate to one another across time and to fit time series models that take into account lagged events.

A. P. Nicolau · K. Dyson · D. Saah
Spatial Informatics Group, Pleasanton, CA, USA
e-mail: apnicolau@sig-gis.com

K. Dyson
e-mail: kdyson@sig-gis.com

A. P. Nicolau · K. Dyson
SERVIR-Amazonia, Cali, Colombia

D. Saah (✉)
University of San Francisco, San Francisco, CA, USA
e-mail: dssaah@usfca.edu

N. Clinton
Google LLC, Mountain View, CA, USA
e-mail: nclinton@google.com

K. Dyson
Dendrolytics, Seattle, WA, USA

Learning Outcomes

- Using the `ee.Join` function to create time-lagged collections.
- Calculating autocovariance and autocorrelation.
- Calculating cross-covariance and cross-correlation.
- Fitting auto-regressive models.

Assumes you know how to:

- Import images and image collections, filter, and visualize (Part I).
- Perform basic image analysis: select bands, compute indices, create masks, classify images (Part II).
- Create a graph using `ui.Chart` (Chap. 4).
- Write a function and `map` it over an `ImageCollection` (Chap. 12).
- Mask cloud, cloud shadow, snow/ice, and other undesired pixels (Chap. 15).
- Fit linear and nonlinear functions with regression in an `ImageCollection` time series (Chap. 18).

21.1 Introduction to Theory

While fitting functions to time series allows you to account for seasonality in your models, sometimes, the impact of a seasonal event does not impact your dependent variable until the next month, the next year, or even multiple years later. For example, coconuts take 18–24 months to develop from flower to harvestable size. Heavy rains during the flower development stage can severely reduce the number of coconuts that can be harvested months later, with significant negative economic repercussions. These patterns—where events in one time period impact our variable of interest in later time periods—are important to be able to include in our models.

In this chapter, we introduce lagged effects into our previous discussions on interpreting time series data (Chaps. 18 and 19). Being able to integrate lagged effects into our time series models allows us to address many important questions. For example, streamflow can be accurately modeled by taking into account previous streamflow, rainfall, and soil moisture; this improved understanding helps predict and mitigate the impacts of drought and flood events made more likely by climate change (Sazib et al. 2020). As another example, time series lag analysis was able to determine that decreased rainfall was associated with increases in livestock disease outbreaks one year later in India (Karthikeyan et al. 2021).

21.2 Practicum

21.2.1 Section 1: Autocovariance and Autocorrelation

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.

Before we dive into autocovariance and autocorrelation, let us set up an area of interest and dataset that we can use to illustrate these concepts. We will work with a detrended time series (as seen in Chap. 18) based on the USGS Landsat 8 Level 2, Collection 2, Tier 1 image collection. Copy and paste the code below to filter the Landsat 8 collection to a point of interest over California and specific dates, and apply the pre-processing function—to mask clouds (as seen in Chap. 15) and to scale and add variables of interest (as seen in Chap. 18).

```
// Define function to mask clouds, scale, and add variables
// (NDVI, time and a constant) to Landsat 8 imagery.
function maskScaleAndAddVariable(image) {
    // Bit 0 - Fill
    // Bit 1 - Dilated Cloud
    // Bit 2 - Cirrus
    // Bit 3 - Cloud
    // Bit 4 - Cloud Shadow
    var qaMask =
image.select('QA_PIXEL').bitwiseAnd(parseInt('11111',
        2)).eq(0);
    var saturationMask = image.select('QA_RADSAT').eq(0);

    // Apply the scaling factors to the appropriate bands.
    var opticalBands =
image.select('SR_B.').multiply(0.0000275).add(-
        0.2);
    var thermalBands =
image.select('ST_B.*').multiply(0.00341802)
        .add(149.0);

    // Replace the original bands with the scaled ones and apply
the masks.
    var img = image.addBands(opticalBands, null, true)
        .addBands(thermalBands, null, true)
        .updateMask(qaMask)
        .updateMask(saturationMask);
    var imgScaled = image.addBands(img, null, true);
    // Now we start to add variables of interest.
    // Compute time in fractional years since the epoch.
    var date = ee.Date(image.get('system:time_start'));
    var years = date.difference(ee.Date('1970-01-01'), 'year');
    var timeRadians = ee.Image(years.multiply(2 * Math.PI));
    // Return the image with the added bands.
    return imgScaled
        // Add an NDVI band.
        .addBands(imgScaled.normalizedDifference(['SR_B5',
'SR_B4'])
            .rename('NDVI'))
        // Add a time band.
        .addBands(timeRadians.rename('t'))
        .float()
        // Add a constant band.
        .addBands(ee.Image.constant(1));
}
```

```
// Import region of interest. Area over California.
var roi = ee.Geometry.Polygon([
    [-119.44617458417066,35.92639730653253],
    [-119.07675930096754,35.92639730653253],
    [-119.07675930096754,36.201704711823844],
    [-119.44617458417066,36.201704711823844],
    [-119.44617458417066,35.92639730653253]
]);


// Import the USGS Landsat 8 Level 2, Collection 2, Tier 1
collection,
// filter, mask clouds, scale, and add variables.
var landsat8sr = ee.ImageCollection('LANDSAT/LC08/C02/T1_L2')
    .filterBounds(roi)
    .filterDate('2013-01-01', '2018-01-01')
    .map(maskScaleAndAddVariable);

// Set map center.
Map.centerObject(roi, 10);
```

Next, copy and paste the code below to estimate the linear trend using the `linearRegression` reducer, and remove that linear trend from the time series.

```
// List of the independent variable names.
var independents = ee.List(['constant', 't']);

// Name of the dependent variable.
var dependent = ee.String('NDVI');

// Compute a linear trend.  This will have two bands:
'residuals' and
// a 2x1 band called coefficients (columns are for
dependent variables).
var trend = landsat8sr.select(independents.add(dependent))

.reduce(ee.Reducer.linearRegression(independents.length(),
1));

// Flatten the coefficients into a 2-band image
var coefficients = trend.select('coefficients')
    // Get rid of extra dimensions and convert back to a
regular image
    .arrayProject([0])
    .arrayFlatten([independents]);

// Compute a detrended series.
var detrended = landsat8sr.map(function(image) {
    return image.select(dependent)
        .subtract(image.select(independents).multiply(
                coefficients)
            .reduce('sum'))
        .rename(dependent)
        .copyProperties(image, ['system:time_start']);
});
```

Now let us turn to autocovariance and autocorrelation. The autocovariance of a time series refers to the dependence of values in the time series at time t with values at time $h = t - lag$. The autocorrelation is the correlation between elements of a dataset at one time and elements of the same dataset at a different time. The autocorrelation is the autocovariance normalized by the standard deviations of the covariates. Specifically, we assume our time series is stationary, and define the autocovariance and autocorrelation following Shumway and Stoffer (2019). Comparing values at time t to the previous values is useful not only for computing autocovariance, but also for a variety of other time series analyzes as you will see shortly.

To combine image data with the previous values in Earth Engine, the first step is to join the previous values to the current values. To do that, we will use a ee.Join function to create what we will call a *lagged collection*. Copy and paste the code below to define a function that creates a lagged collection.

```
// Function that creates a lagged collection.
var lag = function(leftCollection, rightCollection, lagDays)
{
    var filter = ee.Filter.and(
        ee.Filter.maxDifference({
            difference: 1000 * 60 * 60 * 24 * lagDays,
            leftField: 'system:time_start',
            rightField: 'system:time_start'
        }),
        ee.Filter.greaterThan({
            leftField: 'system:time_start',
            rightField: 'system:time_start'
        }));

    return ee.Join.saveAll({
        matchesKey: 'images',
        measureKey: 'delta_t',
        ordering: 'system:time_start',
        ascending: false, // Sort reverse chronologically
    }).apply({
        primary: leftCollection,
        secondary: rightCollection,
        condition: filter
    });
};
```

This function joins a collection to itself, using a filter that gets all the images before each image's date that are within a specified time difference (in days) of each image. That list of the previous images within the lag time is stored in a property of the image called images, sorted reverse chronologically. For example, to create a lagged collection from the detrended Landsat imagery, copy and paste:

```
// Create a lagged collection of the detrended imagery.
var lagged17 = lag(detrended, detrended, 17);
```

Why 17 days? Recall that the temporal cadence of Landsat is 16 days. Specifying 17 days in the join gets one previous image, but no more.

Now, we will compute the autocovariance using a reducer that expects a set of one-dimensional arrays as input. So pixel values corresponding to time t need to be stacked with pixel values at time $t - lag$ as multiple bands in the same image. Copy and paste the code below to define a function to do so, and apply it to merge the bands from the lagged collection.

```
// Function to stack bands.
var merge = function(image) {
    // Function to be passed to iterate.
    var merger = function(current, previous) {
        return ee.Image(previous).addBands(current);
    };
    return
ee.ImageCollection.fromImages(image.get('images'))
        .iterate(merger, image);
};

// Apply merge function to the lagged collection.
var merged17 = ee.ImageCollection(lagged17.map(merge));
```

Now, the bands from time t and h are all in the same image. Note that the band name of a pixel at time h, p_h, was the same as time t, p_t (band name is "NDVI" in this case). During the merging process, it gets a '_1' appended to it (e.g., NDVI_1).

You can print the image collection to check the band names of one of the images. Copy and paste the code below to map a function to convert the merged bands to arrays with bands p_t and p_h, and then reduce it with the covariance reducer. We use a `parallelScale` factor of 8 in the reduce function to avoid the computation to run out of memory (this is not always needed). Note that the output of the covariance reducer is an array image, in which each pixel stores a 2×2 variance–covariance array. The off-diagonal elements are covariance, which you can map directly using the `arrayGet` function.

```
// Function to compute covariance.
var covariance = function(mergedCollection, band, lagBand) {
    return mergedCollection.select([band,
lagBand]).map(function(
        image) {
        return image.toArray();
    }).reduce(ee.Reducer.covariance(), 8);
};

// Concatenate the suffix to the NDVI band.
var lagBand = dependent.cat('_1');

// Compute covariance.
var covariance17 = ee.Image(covariance(merged17, dependent,
lagBand))
    .clip(roi);

// The output of the covariance reducer is an array image,
// in which each pixel stores a 2x2 variance-covariance
array.
// The off diagonal elements are covariance, which you can
map
// directly using:
Map.addLayer(covariance17.arrayGet([0, 1]),
    {
        min: 0,
        max: 0.02
    },
    'covariance (lag = 17 days)');
```

Inspect the pixel values of the resulting covariance image (Fig. 21.1). The covariance is positive when the greater values of one variable (at time *t*) mainly correspond to the greater values of the other variable (at time *h*), and the same holds for the lesser values; therefore, the values tend to show similar behavior. In the opposite case, when the greater values of a variable correspond to the lesser values of the other variable, the covariance is negative.

The diagonal elements of the variance–covariance array are variances. Copy and paste the code below to define and map a function to compute correlation (Fig. 21.2) from the variance–covariance array.

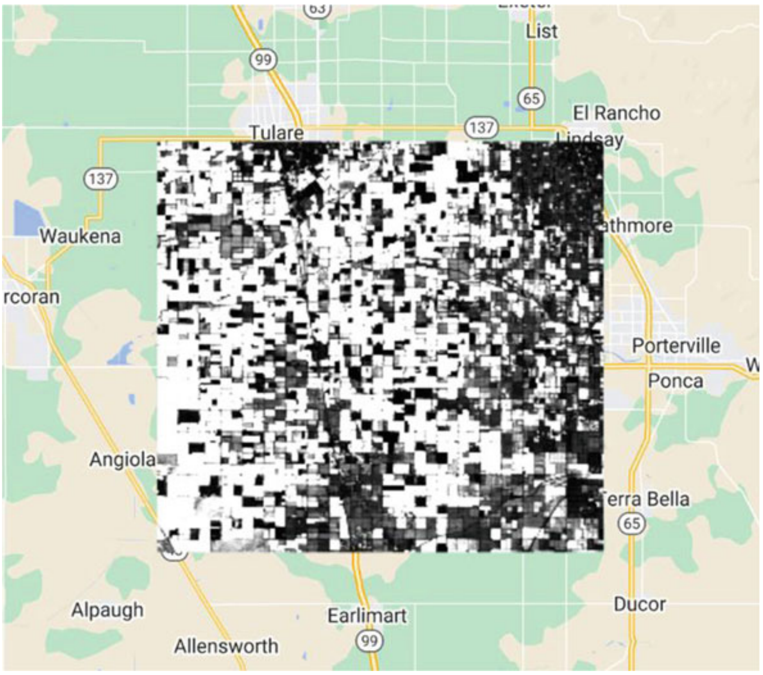


Fig. 21.1 Autocovariance image

```
// Define the correlation function.
var correlation = function(vcArrayImage) {
    var covariance = ee.Image(vcArrayImage).arrayGet([0, 1]);
    var sd0 = ee.Image(vcArrayImage).arrayGet([0, 0]).sqrt();
    var sd1 = ee.Image(vcArrayImage).arrayGet([1, 1]).sqrt();
    return covariance.divide(sd0).divide(sd1).rename(
        'correlation');
};

// Apply the correlation function.
var correlation17 = correlation(covariance17).clip(roi);
Map.addLayer(correlation17,
    {
        min: -1,
        max: 1
    },
    'correlation (lag = 17 days)');
```

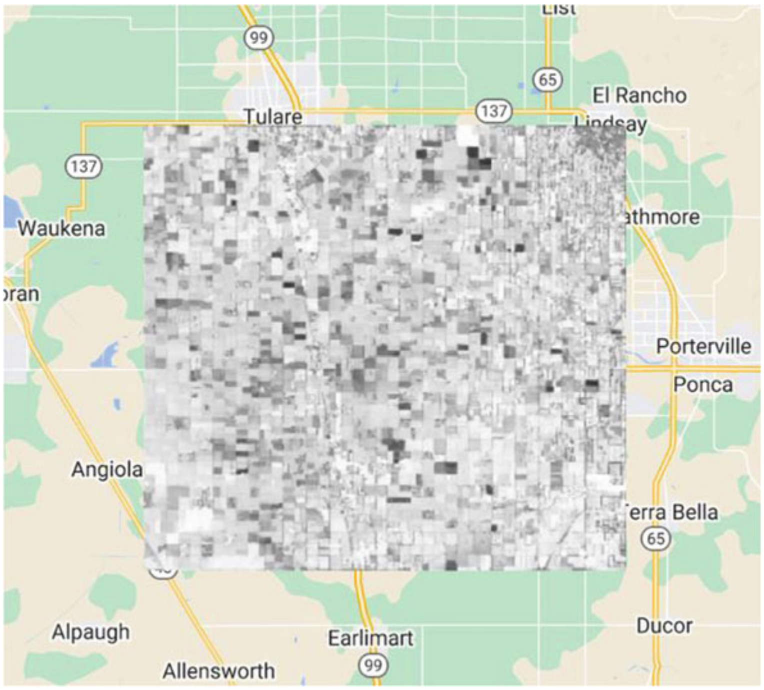


Fig. 21.2 Autocorrelation image

Higher positive values indicate higher correlation between the elements of the dataset, and lower negative values indicate the opposite.

It is worth noting that you can do this for longer lags as well. Of course, that `images` list will fill up with all the images that are within *lag* of *t*. Those other images are also useful—for example, in fitting auto-regressive models as described later.

Code Checkpoint F49a. The book's repository contains a script that shows what your code should look like at this point.

21.2.2 Section 2: Cross-Covariance and Cross-Correlation

Cross-covariance is analogous to autocovariance, except instead of measuring the correspondence between a variable and itself at a lag, it measures the correspondence between a variable and a covariate at a lag. Specifically, we will define the cross-covariance and cross-correlation according to Shumway and Stoffer (2019).

You already have all the code needed to compute cross-covariance and cross-correlation. But you do need a time series of another variable. Suppose, we postulate that NDVI is related in some way to the precipitation before the NDVI was observed. To estimate the strength of this relationship in every pixel, copy and paste the code below to the existing script to load precipitation, join, merge, and reduce as previously:

```
// Precipitation (covariate)
var chirps = ee.ImageCollection('UCSB-CHG/CHIRPS/PENTAD');

// Join the t-l (l=1 pentad) precipitation images to the
Landsat.
var lag1PrecipNDVI = lag(landsat8sr, chirps, 5);

// Add the precipitation images as bands.
var merged1PrecipNDVI =
ee.ImageCollection(lag1PrecipNDVI.map(merge));

// Compute and display cross-covariance.
var cov1PrecipNDVI = covariance(merged1PrecipNDVI, 'NDVI',
    'precipitation').clip(roi);
Map.addLayer(cov1PrecipNDVI.arrayGet([0, 1]), {},
    'NDVI - PRECIP cov (lag = 5)');

// Compute and display cross-correlation.
var corr1PrecipNDVI =
correlation(cov1PrecipNDVI).clip(roi);
Map.addLayer(corr1PrecipNDVI, {
    min: -0.5,
    max: 0.5
}, 'NDVI - PRECIP corr (lag = 5)');
```

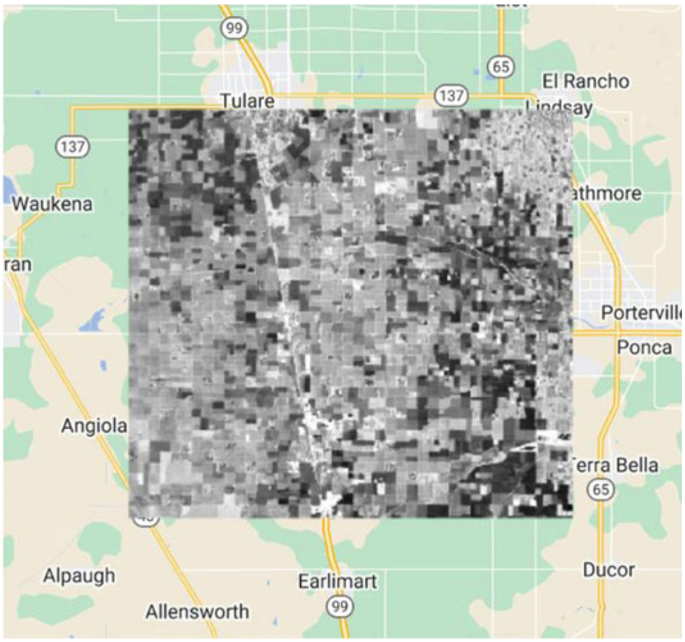


Fig. 21.3 Cross-correlation image of NDVI and precipitation with a five-day lag

What do you observe from this result? Looking at the cross-correlation image (Fig. 21.3), do you observe high values where you would expect high NDVI values (vegetated areas)? One possible drawback of this computation is that it is only based on five days of precipitation, whichever five days came right before the NDVI image.

Perhaps precipitation in the month before the observed NDVI is relevant? Copy and paste the code below to test the 30-day lag idea.

```
// Join the precipitation images from the previous month.
var lag30PrecipNDVI = lag(landsat8sr, chirps, 30);

var sum30PrecipNDVI =
ee.ImageCollection(lag30PrecipNDVI.map(function(
    image) {
    var laggedImages = ee.ImageCollection.fromImages(image
        .get('images'));
    return ee.Image(image).addBands(laggedImages.sum()
        .rename('sum'));
}));

// Compute covariance.
var cov30PrecipNDVI = covariance(sum30PrecipNDVI, 'NDVI',
'sum').clip(
    roi);
Map.addLayer(cov1PrecipNDVI.arrayGet([0, 1]), {},
    'NDVI - sum cov (lag = 30)');

// Correlation.
var corr30PrecipNDVI =
correlation(cov30PrecipNDVI).clip(roi);
Map.addLayer(corr30PrecipNDVI, {
    min: -0.5,
    max: 0.5
}, 'NDVI - sum corr (lag = 30)');
```

Observe that the only change is to the `merge` method. Instead of merging the bands of the NDVI image and the covariate (precipitation), the entire list of precipitation is summed and added as a band (eliminating the need for `iterate`).

Which changes do you notice between the cross-correlation images—5 days lag versus 30 days lag (Fig. 21.4)? You can use the **Inspector** tool to assess if the correlation increased or not at vegetated areas.

As long as there is sufficient temporal overlap between the time series, these techniques could be extended to longer lags and longer time series.

Code Checkpoint F49b. The book's repository contains a script that shows what your code should look like at this point.

21.2.3 Section 3: Auto-Regressive Models

The discussion of autocovariance preceded this section in order to introduce the concept of lag. Now that you have a way to get the previous values of a variable, it is worth considering auto-regressive models. Suppose that pixel values at time

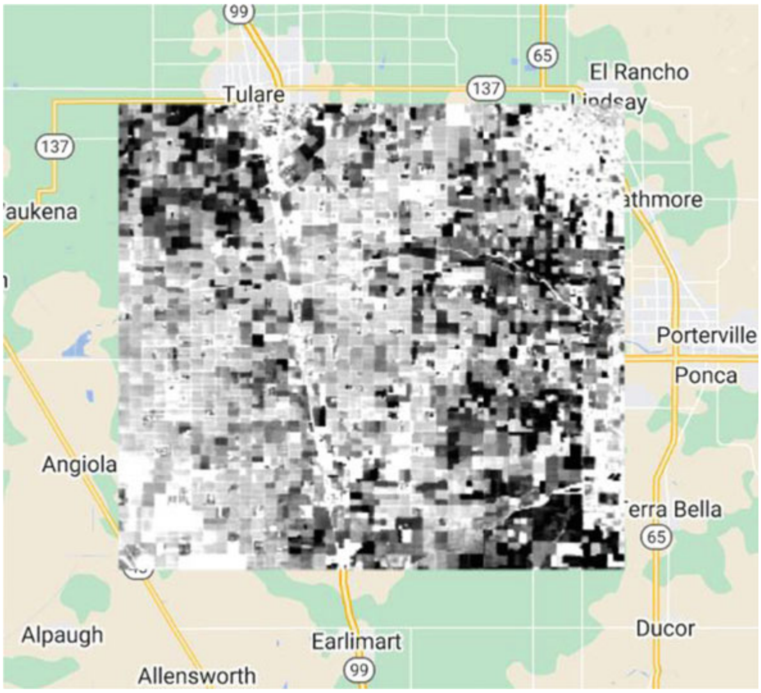


Fig. 21.4 Cross-correlation image of NDVI and precipitation with a 30-day lag

t depend in some way on the previous pixel values—auto-regressive models are time series models that use observations from the previous time steps as input to a regression equation to predict the value at the next time step. If you have observed significant, non-zero autocorrelations in a time series, this is a good assumption. Specifically, you may postulate a linear model such as the following, where p_t is a pixel at time t, and e_t is a random error (Chap. 18):

$$p_t = \beta_0 + \beta_1 p_{t-1} + \beta_2 p_{t-2} + e_t \tag{21.1}$$

To fit this model, you need a lagged collection as created previously, except with a longer lag (e.g., $lag = 34$ days). The next steps are to merge the bands then reduce with the linear regression reducer.

Copy and paste the line below to the existing script to create a lagged collection, where the `images` list stores the two previous images:

```
var lagged34 = ee.ImageCollection(lag(landsat8sr,
landsat8sr, 34));
```

Copy and paste the code below to merge the bands of the lagged collection such that each image has bands at time *t* and bands at times $t - 1, \ldots, t - lag$. Note that it is necessary to filter out any images that do not have two previous temporal neighbors.

```
var merged34 = lagged34.map(merge).map(function(image) {
    return image.set('n', ee.List(image.get('images'))
        .length());
}).filter(ee.Filter.gt('n', 1));
```

Now, copy and paste the code below to fit the regression model using the `linearRegression` reducer.

```
var arIndependents = ee.List(['constant', 'NDVI_1', 'NDVI_2']);

var ar2 = merged34
    .select(arIndependents.add(dependent))
    .reduce(ee.Reducer.linearRegression(arIndependents.length(),
1));

// Turn the array image into a multi-band image of coefficients.
var arCoefficients = ar2.select('coefficients')
    .arrayProject([0])
    .arrayFlatten([arIndependents]);
```

We can compute the fitted values using the `expression` function in Earth Engine. Because this model is a function of the previous pixel values, which may be masked, if any of the inputs to Eq. 21.1 are masked, the output of the equation will also be masked. That is why you should use an expression here, unlike the previous linear models of time. Copy and paste the code below to compute the fitted values.

```
// Compute fitted values.
var fittedAR = merged34.map(function(image) {
    return image.addBands(
        image.expression(
            'beta0 + beta1 * p1 + beta2 * p2', {
                p1: image.select('NDVI_1'),
                p2: image.select('NDVI_2'),
                beta0: arCoefficients.select('constant'),
                beta1: arCoefficients.select('NDVI_1'),
                beta2: arCoefficients.select('NDVI_2')
            }).rename('fitted'));
});
```

Finally, copy and paste the code below to plot the results (Fig. 21.5). We will use a specific point defined as `pt`. Note the missing values that result from masked data. If you run into computation errors, try commenting the `Map.addLayer` calls from the previous sections to save memory.

```
// Create an Earth Engine point object to print the time
series chart.
var pt = ee.Geometry.Point([-119.0955, 35.9909]);

print(ui.Chart.image.series(
        fittedAR.select(['fitted', 'NDVI']), pt, ee.Reducer
    .mean(), 30)
    .setSeriesNames(['NDVI', 'fitted'])
    .setOptions({
        title: 'AR(2) model: original and fitted values',
        lineWidth: 1,
        pointSize: 3,
    }));
```

At this stage, note that the missing data has become a real problem. Any data point for which at least one of the previous points is masked or missing is also masked.

Code Checkpoint F49c. The book's repository contains a script that shows what your code should look like at this point.

It may be possible to avoid this problem by substituting the output from Eq. 21.1 (the modeled value) for the missing or masked data. Unfortunately, the code to make that happen is not straightforward. You can check a solution in the following Code Checkpoint:

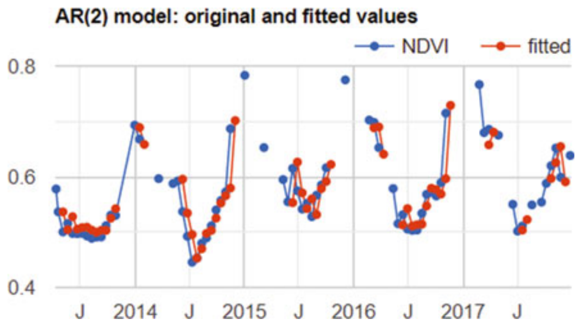


Fig. 21.5 Observed NDVI and fitted values at selected point

Code Checkpoint F49d. The book's repository contains a script that shows what your code should look like at this point.

21.3 Synthesis

Assignment 1. Analyze cross-correlation between NDVI and soil moisture, or precipitation and soil moisture, for example. Earth Engine contains different soil moisture datasets in its catalog (e.g., NASA-USDA SMAP, NASA-GLDAS). Try increasing the lagged time and see if it makes any difference. Alternatively, you can pick any other environmental variable/index (e.g., a different vegetation index: EVI instead of NDVI, for example) and analyze its autocorrelation.

21.4 Conclusion

In this chapter, we learned how to use autocovariance and autocorrelation to explore the relationship between elements of a time series at multiple time steps. We also explored how to use cross-covariance and cross-correlation to examine the relationship between elements of two time series at different points in time. Finally, we used auto-regressive models to regress the elements of a time series with elements of the same time series at a different point in time. With these skills, you can now examine how events in one time period impact your variable of interest in later time periods. While we have introduced the linear approach to lagged effects, these ideas can be expanded to more complex models.

References

Karthikeyan R, Rupner RN, Koti SR et al (2021) Spatio-temporal and time series analysis of bluetongue outbreaks with environmental factors extracted from Google Earth Engine (GEE) in Andhra Pradesh, India. Transbound Emerg Dis 68:3631–3642. https://doi.org/10.1111/tbed.13972

Sazib N, Bolten J, Mladenova I (2020) Exploring spatiotemporal relations between soil moisture, precipitation, and streamflow for a large set of watersheds using Google Earth Engine. Water (switzerland) 12:1371. https://doi.org/10.3390/w12051371

Shumway RH, Stoffer DS (2019) Time series: a data analysis approach using R. Chapman and Hall/CRC

Part V
Vectors and Tables

In addition to raster data processing, Earth Engine supports a rich set of vector processing tools. This part introduces you to the vector framework in Earth Engine, shows you how to create and to import your vector data, and how to combine vector and raster data for analyses.

Exploring Vectors

22

A. J. Purdy, Ellen Brock, and David Saah

Overview

In this chapter, you will learn about features and feature collections and how to use them in conjunction with images and image collections in Earth Engine. Maps are useful for understanding spatial patterns, but scientists often need to extract statistics to answer a question. For example, you may make a false-color composite showing which areas of San Francisco are more "green"—i.e., have more healthy vegetation—than others, but you will likely not be able to directly determine which block in a neighborhood is the most green. This tutorial will demonstrate how to do just that by utilizing vectors.

As described in Chap. 12, an important way to summarize and simplify data in Earth Engine is through the use of *reducers*. Reducers operating across space were used in Chap. 8, for example, to enable image regression between bands. More generally, chapters in Part III and Part IV used reducers mostly to summarize the values across bands or images on a pixel-by-pixel basis. What if you wanted to summarize information within the confines of given spatial elements—for example, within a set of polygons? In this chapter, we will illustrate and explore Earth Engine's method for doing that, which is through a `reduceRegions` call.

A. J. Purdy (✉) · D. Saah
University of San Francisco, San Francisco, CA, USA
e-mail: adamjpurdy@gmail.com; apurdy@usca.edu; adpurdy@csumb.edu

D. Saah
e-mail: dssaah@usfca.edu

A. J. Purdy
California State University, Monterey Bay Seaside, CA, USA

E. Brock
Mantle Labs Ltd., Bangalore, Karnataka, India

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_22

Learning Outcomes

- Uploading and working with a shapefile as an asset to use in Earth Engine.
- Creating a new feature using the geometry tools.
- Importing and filtering a feature collection in Earth Engine.
- Using a feature to clip and reduce image values within a geometry.
- Use `reduceRegions` to summarize an image in irregular neighborhoods.
- Exporting calculated data to tables with Tasks.

Assumes you know how to

- Import images and image collections, filter, and visualize (Part I).
- Calculate and interpret vegetation indices (Chap. 5).
- Use drawing tools to create points, lines, and polygons (Chap. 6).

22.1 Introduction to Theory

In the world of geographic information systems (GIS), data are typically thought of in one of two basic data structures: *raster* and *vector*. In the previous chapters, we have principally been focused on raster data—data using the remote sensing vocabulary of pixels, spatial resolution, images, and image collections. Working within the vector framework is also a crucial skill to master. If you do not know much about GIS, you can find any number of online explainers of the distinctions between these data types, their strengths and limitations, and analyzes using both data types. Being able to move fluidly between a raster conception and a vector conception of the world is powerful and is facilitated with specialized functions and approaches in Earth Engine.

For our purposes, you can think of vector data as information represented as points (e.g., locations of sample sites), lines (e.g., railroad tracks), or polygons (e.g., the boundary of a national park or a neighborhood). Line data and polygon data are built up from points: for example, the latitude and longitude of the sample sites, the points along the curve of the railroad tracks, and the corners of the park that form its boundary. These points each have a highly specific location on Earth's surface, and the vector data formed from them can be used for calculations with respect to other layers. As will be seen in this chapter, for example, a polygon can be used to identify which pixels in an image are contained within its borders. Point-based data have already been used in earlier chapters for filtering image collections by location (see Part I) and can also be used to extract values from an image at a point or a set of points (see Chap. 24). Lines possess the dimension of length and have similar capabilities for filtering image collections and accessing their values along a transect. In addition to using polygons to summarize values within a boundary, they can be used for other, similar purposes—for example, to clip an image.

As you have seen, raster features in Earth Engine are stored as an `Image` or as part of an `ImageCollection`. Using a similar conceptual model, vector data in Earth Engine are stored as a `Feature` or as part of a `FeatureCollection`. Features and feature collections provide useful data to filter images and image collections by their location, clip images to a boundary, or statistically summarize the pixel values within a region.

In the following example, you will use features and feature collections to identify which city block near the University of San Francisco (USF) campus is the most green.

22.2 Practicum

22.2.1 Section 1: Using Geometry Tools to Create Features in Earth Engine

To demonstrate how geometry tools in Earth Engine work, let us start by creating a point, and two polygons to represent different elements on the USF campus.

Click on the geometry tools in the top left of the **Map** pane and create a point feature. Place a new point where USF is located (see Fig. 22.1).

Use Google Maps to search for "Harney Science Center" or "Lo Schiavo Center for Science." Hover your mouse over the **Geometry Imports** to find the + **new layer** menu item and add a new layer to delineate the boundary of a building on campus.

Next, create another new layer to represent the entire campus as a polygon.

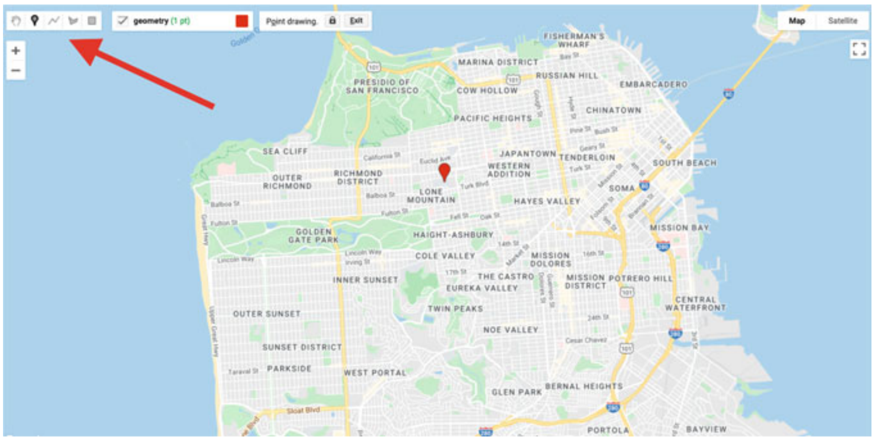
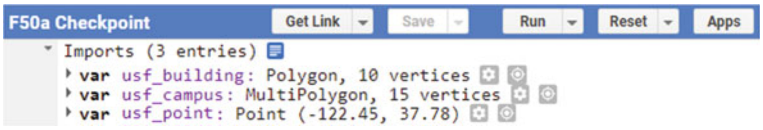


Fig. 22.1 Location of the USF campus in San Francisco, California. Your first point should be in this vicinity. The red arrow points to the geometry tools



```
Imports (3 entries)
 var usf_building: Polygon, 10 vertices
 var usf_campus: MultiPolygon, 15 vertices
 var usf_point: Point (-122.45, 37.78)
```

Fig. 22.2 Rename the default variable names for each layer in the **Imports** section of the code at the top of your script

After you create these layers, rename the geometry imports at the top of your script. Name the layers `usf_point`, `usf_building`, and `usf_campus`. These names are used within the script shown in Fig. 22.2.

Code Checkpoint F50a. The book's repository contains a script that shows what your code should look like at this point.

22.2.2 Section 2: Loading Existing Features and Feature Collections in Earth Engine

If you wish to have the exact same geometry imports in this chapter for the rest of this exercise, begin this section using the code at the Code Checkpoint above.

Next, you will load a city block dataset to determine the amount of vegetation on blocks near USF. The code below imports an existing feature dataset in Earth Engine. The Topologically Integrated Geographic Encoding and Referencing (TIGER) boundaries are census-designated boundaries that are a useful resource when comparing socioeconomic and diversity metrics with environmental datasets in the United States.

```
// Import the Census Tiger Boundaries from GEE.
var tiger = ee.FeatureCollection('TIGER/2010/Blocks');

// Add the new feature collection to the map, but do not
display.
Map.addLayer(tiger, {
    'color': 'black'
}, 'Tiger', false);
```

You should now have the geometry for USF's campus and a layer added to your map that is not visualized for census blocks across the United States. Next, we will use neighborhood data to spatially filter the TIGER feature collection for blocks near USF's campus.

22.2.3 Section 3: Importing Features into Earth Engine

There are many image collections loaded in Earth Engine, and they can cover a very large area that you might want to study. Borders can be quite intricate (for example, a detailed coastline), and fortunately, there is no need for you to digitize the intricate boundary of a large geographic area. Instead, we will show how to find a spatial dataset online, download the data, and load this into Earth Engine as an asset for use.

Find a Spatial Dataset of San Francisco Neighborhoods
Use your Internet searching skills to locate the “analysis neighborhoods” dataset covering San Francisco. This data might be located in a number of places, including DataSF, the City of San Francisco’s public-facing data repository.

After you find the analysis neighborhoods layer, click **Export** and select **Shapefile** (Fig. 22.3). Keep track of where you save the zipped file, as we will load this into Earth Engine. Shapefiles contain vector-based data—points, lines, polygons—and include a number of files, such as the location information, attribute information, and others.

Extract the folder to your computer. When you open the folder, you will see that there are actually many files. The extensions (*.shp*,*.dbf*,*.shx*,*.prj*) all provide a different piece of information to display vector-based data. The *.shp* file provides data on the geometry. The *.dbf* file provides data about the attributes. The *.shx* file

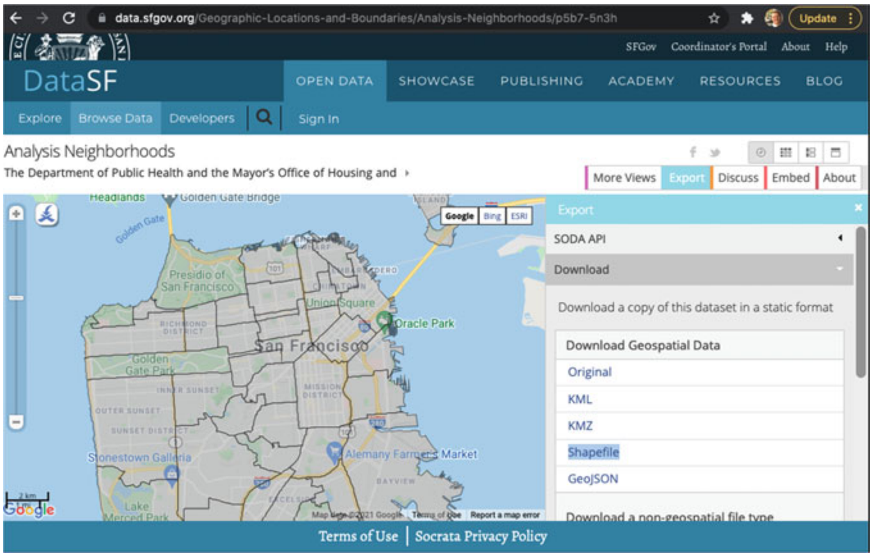


Fig. 22.3 DataSF Website neighborhood shapefile to download

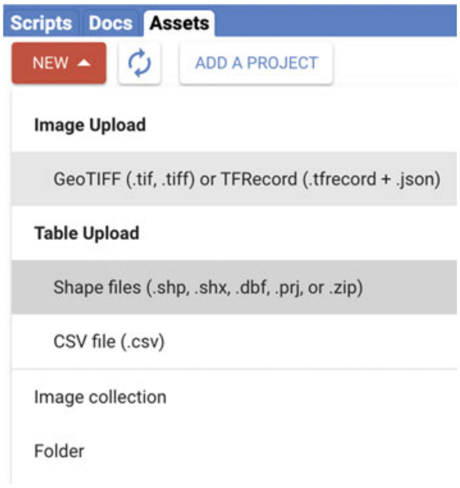

Scripts
Docs
Assets
NEW
ADD A PROJECT
Image Upload
GeoTIFF (.tif, .tiff) or TFRecord (.tfrecord + .json)
Table Upload
Shape files (.shp, .shx, .dbf, .prj, or .zip)
CSV file (.csv)
Image collection
Folder

Fig. 22.4 Import an asset as a zipped folder

is an index file. Lastly, the *.prj* file describes the map projection of the coordinate information for the shapefile. You will need to load all four files to create a new feature asset in Earth Engine.

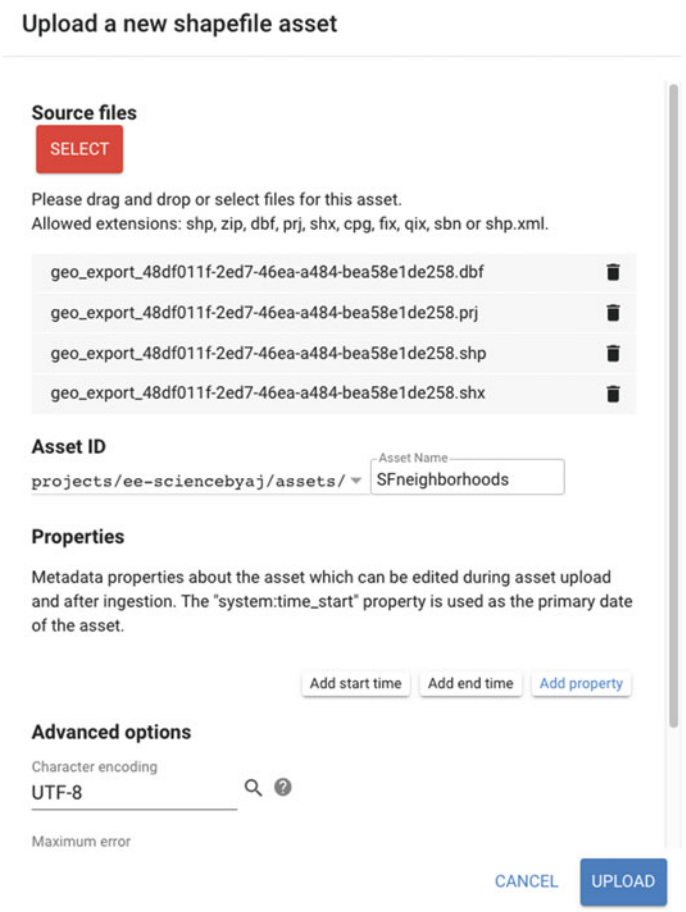
Upload SF Neighborhoods File as an Asset
Navigate to the **Assets** tab (near **Scripts**). Select **New** > **Table Upload** > **Shape files** (Fig. 22.4).

Select Files and Name Asset
Click the **Select** button and then use the file navigator to select the component files of the shapefile structure (i.e., *.shp*, *.dbf*, *.shx*, and *.prj*) (Fig. 22.5). Assign an **Asset Name** so you can recognize this asset.

Uploading the asset may take a few minutes. The status of the upload is presented under the **Tasks** tab. After your asset has been successfully loaded, click on the asset in the **Assets** folder and find the collection ID. Copy this text and use it to import the file into your Earth Engine analysis.

Assign the asset to the table (collection) ID using the script below. Note that you will need to replace `'path/to/your/asset/assetname'` with the actual path copied in the previous step.

```
// Assign the feature collection to the variable
sfNeighborhoods.
var sfNeighborhoods = ee.FeatureCollection(
    'path/to/your/asset/assetname');

// Print the size of the feature collection.
// (Answers the question how many features?)
print(sfNeighborhoods.size());
Map.addLayer(sfNeighborhoods, {
    'color': 'blue'
}, 'sfNeighborhoods');
```


Upload a new shapefile asset

Source files

SELECT

Please drag and drop or select files for this asset.
Allowed extensions: shp, zip, dbf, prj, shx, cpg, fix, qix, sbn or shp.xml.

geo_export_48df011f-2ed7-46ea-a484-bea58e1de258.dbf
geo_export_48df011f-2ed7-46ea-a484-bea58e1de258.prj
geo_export_48df011f-2ed7-46ea-a484-bea58e1de258.shp
geo_export_48df011f-2ed7-46ea-a484-bea58e1de258.shx

Asset ID

projects/ee-sciencebyaj/assets/ Asset Name: SFneighborhoods

Properties

Metadata properties about the asset which can be edited during asset upload and after ingestion. The "system:time_start" property is used as the primary date of the asset.

Add start time | Add end time | Add property

Advanced options

Character encoding: UTF-8

Maximum error

CANCEL UPLOAD

Fig. 22.5 Select the four files extracted from the zipped folder. Make sure each file has the same name and that there are no spaces in the file names of the component files of the shapefile structure

Note that if you have any trouble with loading the FeatureCollection using the technique above, you can follow directions in the Checkpoint script below to use an existing asset loaded for this exercise.

Code Checkpoint F50b. The book's repository contains a script that shows what your code should look like at this point.

22.2.4 Section 4: Filtering Feature Collections by Attributes

Filter by Geometry of Another Feature

First, let us find the neighborhood associated with USF. Use the first point you created to find the neighborhood that intersects this point; filterBounds is the tool that does that, returning a filtered feature.

```
// Filter sfNeighborhoods by USF.
var usfNeighborhood =
sfNeighborhoods.filterBounds(usf_point);
```

Now, filter the blocks layer by USF's neighborhood and visualize it on the map.

```
// Filter the Census blocks by the boundary of the
neighborhood layer.
var usfTiger = tiger.filterBounds(usfNeighborhood);
Map.addLayer(usfTiger, {}, 'usf_Tiger');
```

Filter by Feature (Attribute) Properties

In addition to filtering a FeatureCollection by the location of another feature, you can also filter it by its properties. First, let us print the usfTiger variable to the **Console** and inspect the object.

```
print(usfTiger);
```

You can click on the feature collection name in the **Console** to uncover more information about the dataset. Click on the columns to learn about what attribute information is contained in this dataset. You will notice this feature collection contains information on both housing ('housing10') and population ('pop10').

Now, you will filter for blocks with just the right amount of housing units. You do not want it too dense, nor do you want too few neighbors.

Filter the blocks to have fewer than 250 housing units.

```
// Filter for census blocks by housing units.
var housing10_l250 = usfTiger
    .filter(ee.Filter.lt('housing10', 250));
```

Now filter the already-filtered blocks to have more than 50 housing units.

```
var housing10_g50_l250 = housing10_l250.filter(ee.Filter.gt(
    'housing10', 50));
```

Now, let us visualize what this looks like.

```
Map.addLayer(housing10_g50_l250, {
    'color': 'Magenta'
}, 'housing');
```

We have combined spatial and attribute information to narrow the set to only those blocks that meet our criteria of having between 50 and 250 housing units.

Print Feature (Attribute) Properties to Console

We can print out attribute information about these features. The block of code below prints out the area of the resultant geometry in square meters.

```
var housing_area = housing10_g50_l250.geometry().area();
print('housing_area:', housing_area);
```

The next block of code reduces attribute information and prints out the mean of the `housing10` column.

```
var housing10_mean = usfTiger.reduceColumns({
    reducer: ee.Reducer.mean(),
    selectors: ['housing10']
});

print('housing10_mean', housing10_mean);
```

Both of the above sections of code provide meaningful information about each feature, but they do not tell us which block is the most green. The next section will address that question.

Code Checkpoint F50c. The book's repository contains a script that shows what your code should look like at this point.

22.2.5 Section 5: Reducing Images Using Feature Geometry

Now that we have identified the blocks around USF's campus that have the right housing density, let us find which blocks are the greenest.

The normalized difference vegetation index (NDVI), presented in detail in Chap. 5, is often used to compare the greenness of pixels in different locations. Values on land range from 0 to 1, with values closer to 1 representing healthier and greener vegetation than values near 0.

Create an NDVI Image

The code below imports the Landsat 8 `ImageCollection` as `landsat8`. Then, the code filters for images in 2021. Lastly, the code sorts the images from 2021 to find the least cloudy day.

```
// Import the Landsat 8 TOA image collection.
var landsat8 =
ee.ImageCollection('LANDSAT/LC08/C02/T1_TOA');

// Get the least cloudy image in 2015.
var image = ee.Image(
    landsat8
    .filterBounds(usf_point)
    .filterDate('2015-01-01', '2015-12-31')
    .sort('CLOUD_COVER')
    .first());
```

The next section of code assigns the near-infrared band (B5) to variable `nir` and assigns the red band (B4) to `red`. Then, the bands are combined together to compute NDVI as (`nir` − `red`)/(`nir` + `red`).

```
var nir = image.select('B5');
var red = image.select('B4');
var ndvi =
nir.subtract(red).divide(nir.add(red)).rename('NDVI');
```

Clip the NDVI Image to the Blocks Near USF

Next, you will clip the NDVI layer to only show NDVI over USF's neighborhood.

The first section of code provides visualization settings.

```
var ndviParams = {
    min: -1,
    max: 1,
    palette: ['blue', 'white', 'green']
};
```

The second block of code clips the image to our filtered housing layer.

```
var ndviUSFblocks = ndvi.clip(housing10_g50_l250);
Map.addLayer(ndviUSFblocks, ndviParams, 'NDVI image');
Map.centerObject(usf_point, 14);
```

The NDVI map for all of San Francisco is interesting and shows variability across the region. Now, let us compute mean NDVI values for each block of the city.

Compute NDVI Statistics by Block

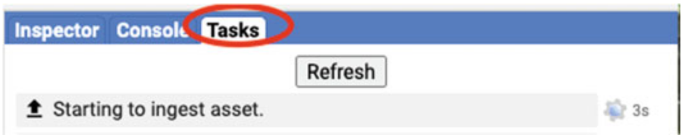
The code below uses the clipped image `ndviUSFblocks` and computes the mean NDVI value within each boundary. The scale provides a spatial resolution for the mean values to be computed on.

```
// Reduce image by feature to compute a statistic e.g.
mean, max, min etc.
var ndviPerBlock = ndviUSFblocks.reduceRegions({
    collection: housing10_g50_l250,
    reducer: ee.Reducer.mean(),
    scale: 30,
});
```

Now, we will use Earth Engine to find out which block is greenest.

Export Table of NDVI Data by Block from Earth Engine to Google Drive

Just as we loaded a feature into Earth Engine, we can export information from Earth Engine. Here, we will export the NDVI data, summarized by block, from Earth Engine to a Google Drive space so that we can interpret it in a program like Google Sheets or Excel.



Starting to ingest asset. 3s

Fig. 22.6 Under the **Tasks** tab, select **Run** to initiate download

```
// Get a table of data out of Google Earth Engine.
Export.table.toDrive({
    collection: ndviPerBlock,
    description: 'NDVI_by_block_near_USF'
});
```

When you run this code, you will notice that you have the **Tasks** tab highlighted on the top right of the Earth Engine Code Editor (Fig. 22.6). You will be prompted to name the directory when exporting the data.

After you run the task, the file will be saved to your Google Drive. You have now brought a feature into Earth Engine and also exported data from Earth Engine.

Code Checkpoint F50d. The book's repository contains a script that shows what your code should look like at this point.

22.2.6 Section 6: Identifying the Block in the Neighborhood Surrounding USF with the Highest NDVI

You are already familiar with filtering datasets by their attributes. Now, you will sort a table and select the first element of the table.

```
ndviPerBlock = ndviPerBlock.select(['blockid10', 'mean']);
print('ndviPerBlock', ndviPerBlock);
var ndviPerBlockSorted = ndviPerBlock.sort('mean', false);
var ndviPerBlockSortedFirst =
ee.Feature(ndviPerBlock.sort('mean',
        false) //Sort by NDVI mean in descending order.
    .first()); //Select the block with the highest NDVI.
print('ndviPerBlockSortedFirst', ndviPerBlockSortedFirst);
```

If you expand the feature of `ndviPerBlockSortedFirst` in the **Console,** you will be able to identify the `blockid10` value of the greenest block and the mean NDVI value for that area.

Another way to look at the data is by exporting the data to a table. Open the table using Google Sheets or Excel. You should see a column titled "mean." Sort the mean column in descending order from highest NDVI to lowest NDVI, then use the `blockid10` attribute to filter our feature collection one last time and display the greenest block near USF.

```
// Now filter by block and show on map!
var GreenHousing = usfTiger.filter(ee.Filter.eq('blockid10',
'#####')); //< Put your id here prepend a 0!
Map.addLayer(GreenHousing, {
    'color': 'yellow'
}, 'Green Housing!');
```

Code Checkpoint F50e. The book's repository contains a script that shows what your code should look like at this point.

22.3 Synthesis

Now it is your turn to use both feature classes and to reduce data using a geographic boundary. Create a new script for an area of interest and accomplish the following assignments.

Assignment 1. Create a study area map zoomed to a certain feature class that you made.

Assignment 2. Filter one feature collection using feature properties.

Assignment 3. Filter one feature collection based on another feature's location in space.

Assignment 4. Reduce one image to the geometry of a feature in some capacity; e.g., extract a mean value or a value at a point.

22.4 Conclusion

In this chapter, you learned how to import features into Earth Engine. In Sect. 22.2.1, you created new features using the geometry tools and loaded a feature from Earth Engine's Data Catalog. In Sect. 22.2.2, you loaded a shapefile to an Earth Engine asset. In Sect. 22.2.3, you filtered feature collections based on their properties and locations. Finally, in Sects. 22.2.4 and 22.2.5, you used a feature collection to reduce an image, then exported the data from Earth Engine. Now, you have all the tools you need to load, filter, and apply features to extract meaningful information from images using vector features in Earth Engine.

Raster/Vector Conversions

23

Keiko Nomura and Samuel Bowers

Overview
The purpose of this chapter is to review methods of converting between raster and vector data formats, and to understand the circumstances in which this is useful. By way of example, this chapter focuses on topographic elevation and forest cover change in Colombia, but note that these are generic methods that can be applied in a wide variety of situations.

Learning Outcomes

- Understanding raster and vector data in Earth Engine and their differing properties.
- Knowing how and why to convert from raster to vector.
- Knowing how and why to convert from vector to raster.
- Write a function and `map` it over a `FeatureCollection`.

Assumes you know how to

- Import images and image collections, filter, and visualize (Part I).
- Understand distinctions among `Image`, `ImageCollection`, `Feature`, and `FeatureCollection` Earth Engine objects (Part I, Part II, Part V).
- Perform basic image analysis: select bands, compute indices, and create masks (Part II).

K. Nomura (✉)
Climate Engine, 111 W. Proctor Street, Suite 203, Carson City, NV 89703, USA
e-mail: keiko@climateengine.com

S. Bowers
School of Geosciences, The University of Edinburgh, Crew Building, The King's Buildings, Alexander Crum Brown Road, Edinburgh EH9 3FF, UK
e-mail: sam.bowers@ed.ac.uk

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_23

- Perform image morphological operations (Chap. 10).
- Understand the `filter`, `map`, and `reduce` paradigm (Chap. 12).
- Write a function and `map` it over an `ImageCollection` (Chap. 12).
- Use `reduceRegions` to summarize an image in irregular shapes (Chap. 22).

23.1 Introduction to Theory

Raster data consist of regularly spaced pixels arranged into rows and columns, familiar as the format of satellite images. Vector data contain geometry features (i.e., points, lines, and polygons) describing locations and areas. Each data format has its advantages, and both will be encountered as part of GIS operations.

Raster data and vector data are commonly combined (e.g., extracting image information for a given location or clipping an image to an area of interest); however, there are also situations in which conversion between the two formats is useful. In making such conversions, it is important to consider the key advantages of each format. Rasters can store data efficiently where each pixel has a numerical value, while vector data can more effectively represent geometric features where homogenous areas have shared properties. Each format lends itself to distinctive analytical operations, and combining them can be powerful.

In this exercise, we'll use topographic elevation and forest change images in Colombia as well as a protected area feature collection to practice the conversion between raster and vector formats, and to identify situations in which this is worthwhile.

23.2 Practicum

23.2.1 Section 1: Raster to Vector Conversion

23.2.1.1 Section 1.1: Raster to Polygons

In this section, we will convert an elevation image (raster) to a feature collection (vector). We will start by loading the Global Multi-Resolution Terrain Elevation Data 2010 and the Global Administrative Unit Layers 2015 dataset to focus on Colombia. The elevation image is a raster at 7.5 arc-second spatial resolution containing a continuous measure of elevation in meters in each pixel.

```
// Load raster (elevation) and vector (colombia) datasets.
var elevation =
ee.Image('USGS/GMTED2010').rename('elevation');
var colombia = ee.FeatureCollection(
        'FAO/GAUL_SIMPLIFIED_500m/2015/level0')
    .filter(ee.Filter.equals('ADM0_NAME', 'Colombia'));

// Display elevation image.
Map.centerObject(colombia, 7);
Map.addLayer(elevation, {
    min: 0,
    max: 4000
}, 'Elevation');
```

When converting an image to a feature collection, we will aggregate the categorical elevation values into a set of categories to create polygon shapes of connected pixels with similar elevations. For this exercise, we will create four zones of elevation by grouping the altitudes to 0–100 m = 0, 100–200 m = 1, 200–500 m = 2, and > 500 m = 3.

```
// Initialize image with zeros and define elevation zones.
var zones = ee.Image(0)
    .where(elevation.gt(100), 1)
    .where(elevation.gt(200), 2)
    .where(elevation.gt(500), 3);

// Mask pixels below sea level (<= 0 m) to retain only
land areas.
// Name the band with values 0-3 as 'zone'.
zones = zones.updateMask(elevation.gt(0)).rename('zone');

Map.addLayer(zones, {
    min: 0,
    max: 3,
    palette: ['white', 'yellow', 'lime', 'green'],
    opacity: 0.7
}, 'Elevation zones');
```

We will convert this zonal elevation image in Colombia to polygon shapes, which is a vector format (termed a `FeatureCollection` in Earth Engine), using the `ee.Image.reduceToVectors` method. This will create polygons delineating connected pixels with the same value. In doing so, we will use the same projection and spatial resolution as the image. Please note that loading the vectorized image in the native resolution (232 m) takes time to execute. For faster visualization, we set a coarse scale of 1000 m (Fig. 23.1).

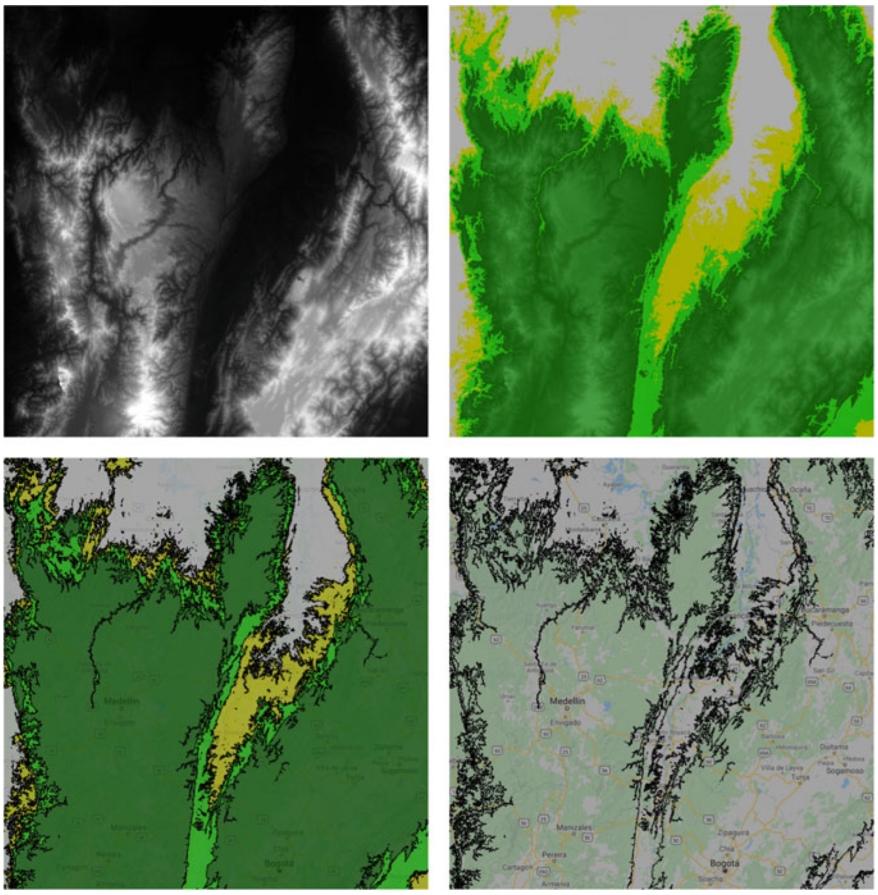


Fig. 23.1 Raster-based elevation (top left) and zones (top right), vectorized elevation zones overlaid on the raster (bottom-left), and vectorized elevation zones only (bottom-right)

```
var projection = elevation.projection();
var scale = elevation.projection().nominalScale();

var elevationVector = zones.reduceToVectors({
    geometry: colombia.geometry(),
    crs: projection,
    scale: 1000, // scale
    geometryType: 'polygon',
    eightConnected: false,
    labelProperty: 'zone',
    bestEffort: true,
    maxPixels: 1e13,
    tileScale: 3 // In case of error.
});

print(elevationVector.limit(10));

var elevationDrawn = elevationVector.draw({
    color: 'black',
    strokeWidth: 1
});
Map.addLayer(elevationDrawn, {}, 'Elevation zone polygon');
```

You may have realized that polygons consist of complex lines, including some small polygons with just one pixel. That happens when there are no surrounding pixels of the same elevation zone. You may not need a vector map with such details—if, for instance, you want to produce a regional or global map. We can use a morphological reducer `focalMode` to simplify the shape by defining a neighborhood size around a pixel. In this example, we will set the kernel radius as four pixels. This operation makes the resulting polygons look much smoother, but less precise (Fig. 23.2).

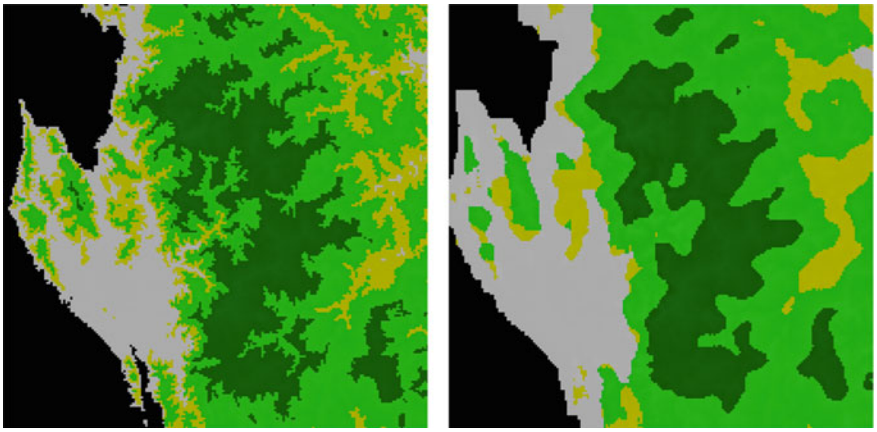


Fig. 23.2 Before (left) and after (right) applying `focalMode`

```
var zonesSmooth = zones.focalMode(4, 'square');

zonesSmooth =
zonesSmooth.reproject(projection.atScale(scale));

Map.addLayer(zonesSmooth, {
    min: 1,
    max: 3,
    palette: ['yellow', 'lime', 'green'],
    opacity: 0.7
}, 'Elevation zones (smooth)');

var elevationVectorSmooth = zonesSmooth.reduceToVectors({
    geometry: colombia.geometry(),
    crs: projection,
    scale: scale,
    geometryType: 'polygon',
    eightConnected: false,
    labelProperty: 'zone',
    bestEffort: true,
    maxPixels: 1e13,
    tileScale: 3
});
```

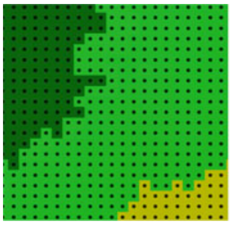
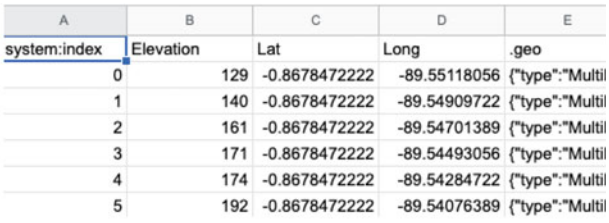
```
var smoothDrawn = elevationVectorSmooth.draw({
    color: 'black',
    strokeWidth: 1
});
Map.addLayer(smoothDrawn, {}, 'Elevation zone polygon
(smooth)');
```

We can see now that the polygons have more distinct shapes with many fewer small polygons in the new map (Fig. 23.2). It is important to note that when you use methods like `focalMode` (or other, similar methods such as `connectedComponents` and `connectedPixelCount`), you need to reproject according to the original image in order to display properly with zoom using the interactive Code Editor.

23.2.1.2 Section 1.2: Raster to Points

Lastly, we will convert a small part of this elevation image into a point vector dataset. For this exercise, we will use the same example and build on the code from the previous subsection. This might be useful when you want to use geospatial data in a tabular format in combination with other conventional datasets such as economic indicators (Fig. 23.3).

The easiest way to do this is to use `sample` while activating the geometries parameter. This will extract the points at the centroid of the elevation pixel.

A	B	C	D	E
system:index	Elevation	Lat	Long	.geo
0	129	-0.8678472222	-89.55118056	{"type":"Multil
1	140	-0.8678472222	-89.54909722	{"type":"Multil
2	161	-0.8678472222	-89.54701389	{"type":"Multil
3	171	-0.8678472222	-89.54493056	{"type":"Multil
4	174	-0.8678472222	-89.54284722	{"type":"Multil
5	192	-0.8678472222	-89.54076389	{"type":"Multil

Fig. 23.3 Elevation point values with latitude and longitude

```
var geometry = ee.Geometry.Polygon([
    [-89.553, -0.929],
    [-89.436, -0.929],
    [-89.436, -0.866],
    [-89.553, -0.866],
    [-89.553, -0.929]
]);

// To zoom into the area, un-comment and run below
// Map.centerObject(geometry,12);
Map.addLayer(geometry, {}, 'Areas to extract points');

var elevationSamples = elevation.sample({
    region: geometry,
    projection: projection,
    scale: scale,
    geometries: true,
});

Map.addLayer(elevationSamples, {}, 'Points extracted');

// Add three properties to the output table:
// 'Elevation', 'Longitude', and 'Latitude'.
elevationSamples = elevationSamples.map(function(feature) {
    var geom = feature.geometry().coordinates();
    return ee.Feature(null, {
        'Elevation': ee.Number(feature.get(
            'elevation')),
        'Long': ee.Number(geom.get(0)),
        'Lat': ee.Number(geom.get(1))
    });
});

// Export as CSV.
Export.table.toDrive({
    collection: elevationSamples,
    description: 'extracted_points',
    fileFormat: 'CSV'
});
```

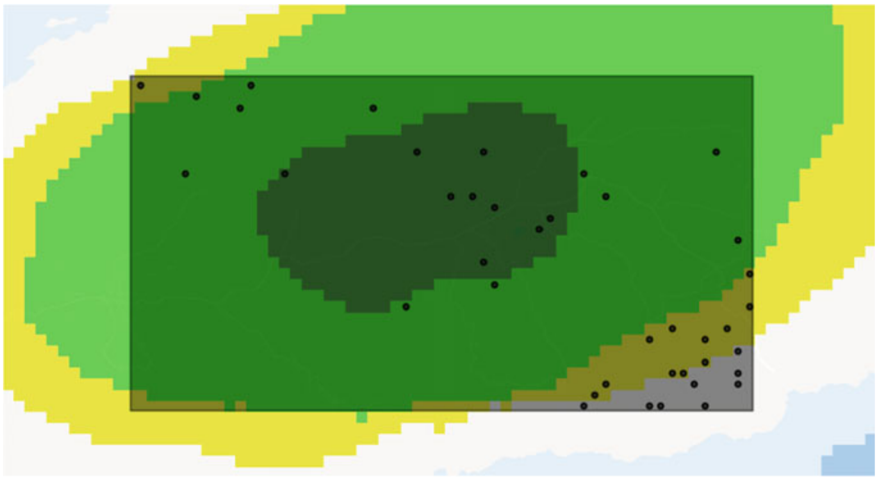


Fig. 23.4 Stratified random sampling over different elevation zones

We can also extract sample points per elevation zone. Below is an example of extracting 10 randomly selected points per elevation zone (Fig. 23.4). You can also set different values for each zone using `classValues` and `classPoints` parameters to modify the sampling intensity in each class. This may be useful, for instance, to generate point samples for a validation effort.

```
var elevationSamplesStratified = zones.stratifiedSample({
    numPoints: 10,
    classBand: 'zone',
    region: geometry,
    scale: scale,
    projection: projection,
    geometries: true
});

Map.addLayer(elevationSamplesStratified, {}, 'Stratified
samples');
```

Code Checkpoint F51a. The book's repository contains a script that shows what your code should look like at this point.

23.2.1.3 Section 1.3: A More Complex Example

In this section, we will use two global datasets, one to represent raster formats and the other vectors:

- The Global Forest Change (GFC) dataset: a raster dataset describing global tree cover and change for 2001–present.
- The World Protected Areas Database: a vector database of global protected areas.

The objective will be to combine these two datasets to quantify rates of deforestation in protected areas in the 'arc of deforestation' of the Colombian Amazon. The datasets can be loaded into Earth Engine with the following code:

```
// Read input data.
// Note: these datasets are periodically updated.
// Consider searching the Data Catalog for newer versions.
var gfc =
ee.Image('UMD/hansen/global_forest_change_2020_v1_8');
var wdpa =
ee.FeatureCollection('WCMC/WDPA/current/polygons');

// Print assets to show available layers and properties.
print(gfc);
print(wdpa.limit(10)); // Show first 10 records.
```

The GFC dataset (first presented in detail in Chap. 2) is a global set of rasters that quantify tree cover and change for the period beginning in 2001. We'll use a single image from this dataset:

- `'lossyear'`: a categorical raster of forest loss (1–20, corresponding to deforestation for the period 2001–2020), and 0 for no change

The World Database on Protected Areas (WDPA) is a harmonized dataset of global terrestrial and marine protected area locations, along with details on the classification and management of each. In addition to protected area outlines, we'll use two fields from this database:

- `'NAME'`: the name of each protected area
- `'WDPA_PID'`: a unique numerical ID for each protected area

To begin with, we'll focus on forest change dynamics in `'La Paya'`, a small protected area in the Colombian Amazon. We'll first visualize these data using the `paint` command, which is discussed in more detail in Chap. 25. This will display the boundary of the La Paya protected area and deforestation in the region (Fig. 23.5).

```
// Display deforestation.
var deforestation = gfc.select('lossyear');

Map.addLayer(deforestation, {
    min: 1,
    max: 20,
    palette: ['yellow', 'orange', 'red']
}, 'Deforestation raster');

// Display WDPA data.
var protectedArea = wdpa.filter(ee.Filter.equals('NAME',
'La Paya'));

// Display protected area as an outline (see F5.3 for
paint()).
var protectedAreaOutline = ee.Image().byte().paint({
    featureCollection: protectedArea,
    color: 1,
    width: 3
});

Map.addLayer(protectedAreaOutline, {
    palette: 'white'
}, 'Protected area');

// Set up map display.
Map.centerObject(protectedArea);
Map.setOptions('SATELLITE');
```

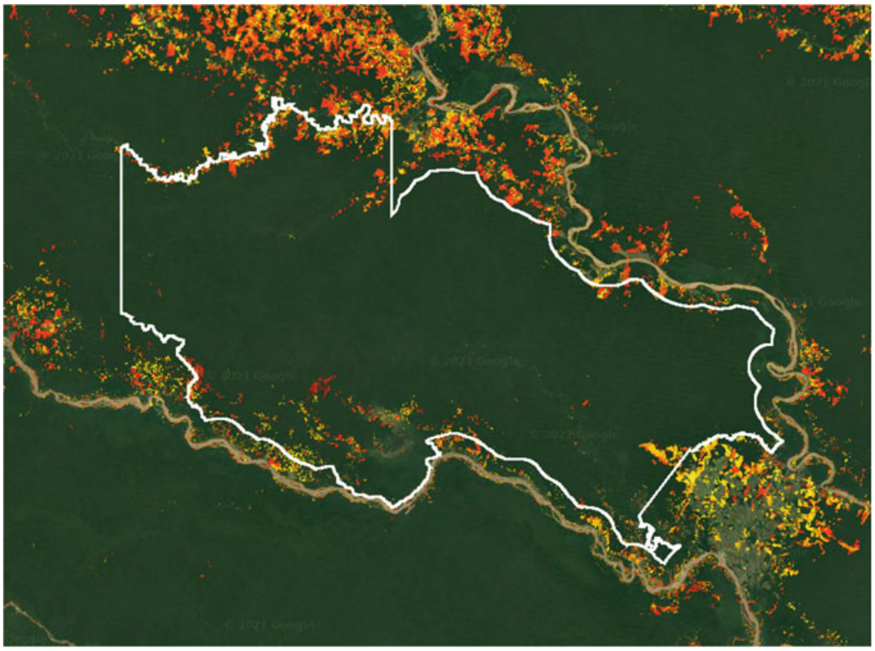


Fig. 23.5 View of the La Paya protected area in the Colombian Amazon (in white) and deforestation over the period 2001–2020 (in yellows and reds, with darker colors indicating more recent changes)

We can use Earth Engine to convert the deforestation raster to a set of polygons. The deforestation data are appropriate for this transformation as each deforestation event is labeled categorically by year, and change events are spatially contiguous. This is performed in Earth Engine using the `ee.Image.reduceToVectors` method, as described earlier in this section. Figure 23.6 shows a comparison of the raster versus vector representations of deforestation within the protected area.

```
// Convert from a deforestation raster to vector.
var deforestationVector = deforestation.reduceToVectors({
    scale: deforestation.projection().nominalScale(),
    geometry: protectedArea.geometry(),
    labelProperty: 'lossyear', // Label polygons with a
change year.
    maxPixels: 1e13
});

// Count the number of individual change events
print('Number of change events:',
deforestationVector.size());

// Display deforestation polygons. Color outline by change
year.
var deforestationVectorOutline = ee.Image().byte().paint({
    featureCollection: deforestationVector,
    color: 'lossyear',
    width: 1
});

Map.addLayer(deforestationVectorOutline, {
    palette: ['yellow', 'orange', 'red'],
    min: 1,
    max: 20
}, 'Deforestation vector');
```

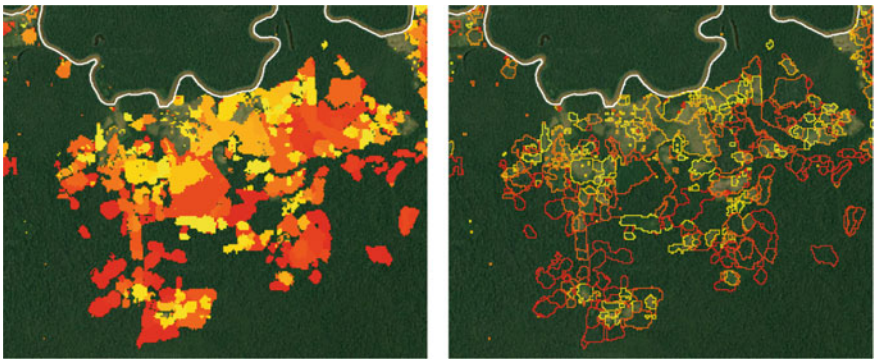


Fig. 23.6 Raster (left) versus vector (right) representations of deforestation data of the La Paya protected area

Having converted from raster to vector, a new set of operations becomes available for post-processing the deforestation data. We might, for instance, be interested in the number of individual change events each year (Fig. 23.7):

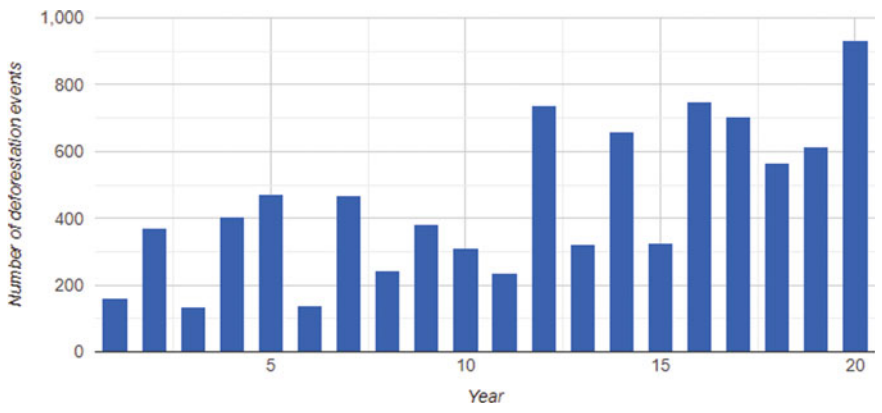


Fig. 23.7 Plot of the number of deforestation events in La Paya for the years 2001–2020

```
var chart = ui.Chart.feature
    .histogram({
        features: deforestationVector,
        property: 'lossyear'
    })
    .setOptions({
        hAxis: {
            title: 'Year'
        },
        vAxis: {
            title: 'Number of deforestation events'
        },
        legend: {
            position: 'none'
        }
    });

print(chart);
```

There might also be interest in generating point locations for individual change events (e.g., to aid a field campaign):

```
// Generate deforestation point locations.
var deforestationCentroids =
deforestationVector.map(function(feat) {
    return feat.centroid();
});

Map.addLayer(deforestationCentroids, {
    color: 'darkblue'
}, 'Deforestation centroids');
```

The vector format allows for easy filtering to only deforestation events of interest, such as only the largest deforestation events:

```
// Add a new property to the deforestation
FeatureCollection
// describing the area of the change polygon.
deforestationVector =
deforestationVector.map(function(feat) {
    return feat.set('area', feat.geometry().area({
        maxError: 10
    }).divide(10000)); // Convert m^2 to hectare.
});

// Filter the deforestation FeatureCollection for only
large-scale (>10 ha) changes
var deforestationLarge =
deforestationVector.filter(ee.Filter.gt(
    'area', 10));

// Display deforestation area outline by year.
var deforestationLargeOutline = ee.Image().byte().paint({
    featureCollection: deforestationLarge,
    color: 'lossyear',
    width: 1
});

Map.addLayer(deforestationLargeOutline, {
    palette: ['yellow', 'orange', 'red'],
    min: 1,
    max: 20
}, 'Deforestation (>10 ha)');
```

Code Checkpoint F51b. The book's repository contains a script that shows what your code should look like at this point.

23.2.1.4 Section 1.4: Raster Properties to Vector Fields

Sometimes we want to extract information from a raster to be included in an existing vector dataset. An example might be estimating a deforestation rate for a set of protected areas. Rather than performing this task on a case-by-case basis, we can attach information generated from an image as a property of a feature.

The following script shows how this can be used to quantify a deforestation rate for a set of protected areas in the Colombian Amazon.

```
// Load required datasets.
var gfc =
ee.Image('UMD/hansen/global_forest_change_2020_v1_8');
var wdpa =
ee.FeatureCollection('WCMC/WDPA/current/polygons');

// Display deforestation.
var deforestation = gfc.select('lossyear');

Map.addLayer(deforestation, {
    min: 1,
    max: 20,
    palette: ['yellow', 'orange', 'red']
}, 'Deforestation raster');

// Select protected areas in the Colombian Amazon.
var amazonianProtectedAreas = [
    'Cordillera de los Picachos', 'La Paya', 'Nukak',
    'Serrania de Chiribiquete',
    'Sierra de la Macarena', 'Tinigua'
];

var wdpaSubset = wdpa.filter(ee.Filter.inList('NAME',
    amazonianProtectedAreas));

// Display protected areas as an outline.
var protectedAreasOutline = ee.Image().byte().paint({
    featureCollection: wdpaSubset,
    color: 1,
    width: 1
});
```

```
Map.addLayer(protectedAreasOutline, {
    palette: 'white'
}, 'Amazonian protected areas');

// Set up map display.
Map.centerObject(wdpaSubset);
Map.setOptions('SATELLITE');

var scale = deforestation.projection().nominalScale();

// Use 'reduceRegions' to sum together pixel areas in each
protected area.
wdpaSubset = deforestation.gte(1)

.multiply(ee.Image.pixelArea().divide(10000)).reduceRegions
({
        collection: wdpaSubset,
        reducer: ee.Reducer.sum().setOutputs([
            'deforestation_area']),
        scale: scale
    });

print(wdpaSubset); // Note the new 'deforestation_area'
property.
```

The output of this script is an estimate of deforested area in hectares for each reserve. However, as reserve sizes vary substantially by area, we can normalize by the total area of each reserve to quantify rates of change.

```
// Normalize by area.
wdpaSubset = wdpaSubset.map(
    function(feat) {
        return feat.set('deforestation_rate',
            ee.Number(feat.get('deforestation_area'))
            .divide(feat.area().divide(10000)) // m2 to ha
            .divide(20) // number of years
            .multiply(100)); // to percentage points
    });

// Print to identify rates of change per protected area.
// Which has the fastest rate of loss?
print(wdpaSubset.reduceColumns({
    reducer: ee.Reducer.toList().repeat(2),
    selectors: ['NAME', 'deforestation_rate']
}));
```

Code Checkpoint F51c. The book's repository contains a script that shows what your code should look like at this point.

23.2.2 Section 2: Vector-To-Raster Conversion

In Sect. 23.2.1, we used the protected area feature collection as its original vector format. In this section, we will rasterize the protected area polygons to produce a mask and use this to assess rates of forest change.

23.2.2.1 Section 2.1: Polygons to a Mask

The most common operation to convert from vector to raster is the production of binary image masks, describing whether a pixel intersects a line or falls within a polygon. To convert from vector to a raster mask, we can use the `ee.FeatureCollection.reduceToImage` method. Let's continue with our example of the WDPA database and Global Forest Change data from the previous section:

```
// Load required datasets.
var gfc =
ee.Image('UMD/hansen/global_forest_change_2020_v1_8');
var wdpa =
ee.FeatureCollection('WCMC/WDPA/current/polygons');

// Get deforestation.
var deforestation = gfc.select('lossyear');

// Generate a new property called 'protected' to apply to
the output mask.
var wdpa = wdpa.map(function(feat) {
    return feat.set('protected', 1);
});

// Rasterize using the new property.
// unmask() sets areas outside protected area polygons to
0.
var wdpaMask = wdpa.reduceToImage(['protected'],
ee.Reducer.first())
    .unmask();
```

```
// Center on Colombia.
Map.setCenter(-75, 3, 6);

// Display on map.
Map.addLayer(wdpaMask, {
    min: 0,
    max: 1
}, 'Protected areas (mask)');
```

We can use this mask to, for example, highlight only deforestation that occurs within a protected area using logical operations:

```
// Set the deforestation layer to 0 where outside a
protected area.
var deforestationProtected =
deforestation.where(wdpaMask.eq(0), 0);

// Update mask to hide where deforestation layer = 0
var deforestationProtected = deforestationProtected
    .updateMask(deforestationProtected.gt(0));

// Display deforestation in protected areas
Map.addLayer(deforestationProtected, {
    min: 1,
    max: 20,
    palette: ['yellow', 'orange', 'red']
}, 'Deforestation protected');
```

In the above example, we generated a simple binary mask, but `reduceToImage` can also preserve a numerical property of the input polygons. For example, we might want to be able to determine which protected area each pixel represents. In this case, we can produce an image with the unique ID of each protected area:

```
// Produce an image with unique ID of protected areas.
var wdpaId = wdpa.reduceToImage(['WDPAID'],
ee.Reducer.first());

Map.addLayer(wdpaId, {
    min: 1,
    max: 100000
}, 'Protected area ID');
```

This output can be useful when performing large-scale raster operations, such as efficiently calculating deforestation rates for multiple protected areas.

Code Checkpoint F51d. The book's repository contains a script that shows what your code should look like at this point.

23.2.2.2 Section 2.2: A More Complex Example

The `reduceToImage` method is not the only way to convert a feature collection to an image. We will create a distance image layer from the boundary of the protected area using `distance`. For this example, we return to the La Paya protected area explored in Sect. 23.2.1.

```
// Load required datasets.
var gfc =
ee.Image('UMD/hansen/global_forest_change_2020_v1_8');
var wdpa =
ee.FeatureCollection('WCMC/WDPA/current/polygons');

// Select a single protected area.
var protectedArea = wdpa.filter(ee.Filter.equals('NAME',
'La Paya'));

// Maximum distance in meters is set in the brackets.
var distance = protectedArea.distance(1000000);

Map.addLayer(distance, {
    min: 0,
    max: 20000,
    palette: ['white', 'grey', 'black'],
    opacity: 0.6
}, 'Distance');

Map.centerObject(protectedArea);
```

We can also show the distance inside and outside of the boundary by using the rasterized protected area (Fig. 23.8).

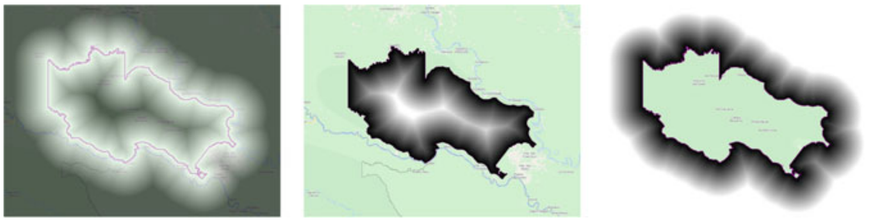


Fig. 23.8 Distance from the La Paya boundary (left), distance within the La Paya (middle), and distance outside the La Paya (right)

```
// Produce a raster of inside/outside the protected area.
var protectedAreaRaster = protectedArea.map(function(feat)
{
    return feat.set('protected', 1);
}).reduceToImage(['protected'], ee.Reducer.first());

Map.addLayer(distance.updateMask(protectedAreaRaster), {
    min: 0,
    max: 20000
}, 'Distance inside protected area');

Map.addLayer(distance.updateMask(protectedAreaRaster.unmask
()
.not()), {
    min: 0,
    max: 20000
}, 'Distance outside protected area');
```

Sometimes it makes sense to work with objects in raster imagery. This is an unusual case of vector-like operations conducted with raster data. There is a good reason for this where the vector equivalent would be computationally burdensome.

An example of this is estimating deforestation rates by distance to the edge of the protected area, as it is common that rates of change will be higher at the boundary of a protected area. We will create a distance raster with three zones from the La Paya boundary (>1 km, > 2 km, > 3 km, and > 4 km) and to estimate the deforestation by distance from the boundary (Fig. 23.9).

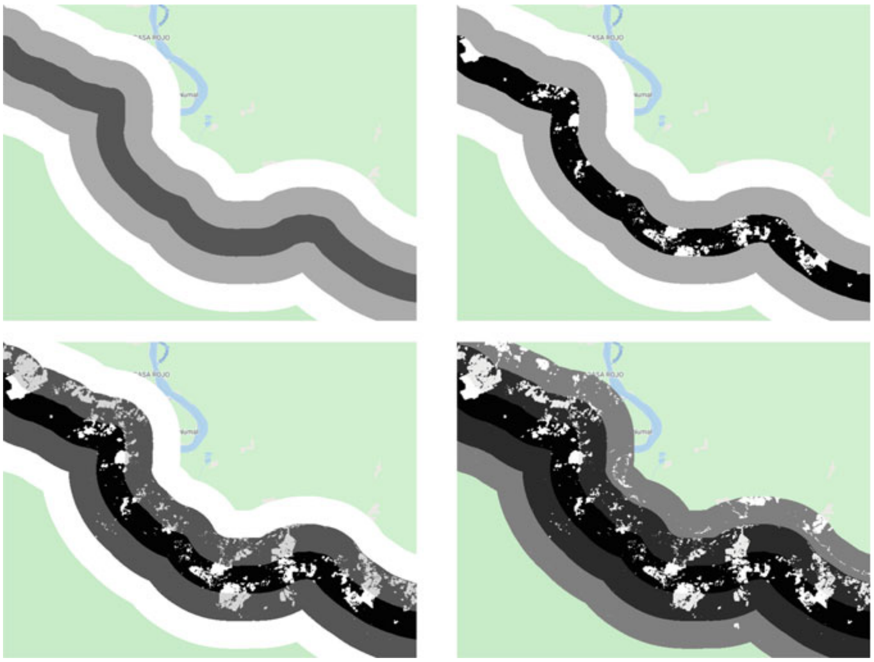


Fig. 23.9 Distance zones (top left) and deforestation by zone (<1 km, < 3 km, and < 5 km)

```
var distanceZones = ee.Image(0)
    .where(distance.gt(0), 1)
    .where(distance.gt(1000), 2)
    .where(distance.gt(3000), 3)
    .updateMask(distance.lte(5000));

Map.addLayer(distanceZones, {}, 'Distance zones');

var deforestation = gfc.select('loss');
var deforestation1km =
deforestation.updateMask(distanceZones.eq(1));
var deforestation3km =
deforestation.updateMask(distanceZones.lte(2));
var deforestation5km =
deforestation.updateMask(distanceZones.lte(3));
```

```
Map.addLayer(deforestation1km, {
    min: 0,
    max: 1
}, 'Deforestation within a 1km buffer');
Map.addLayer(deforestation3km, {
    min: 0,
    max: 1,
    opacity: 0.5
}, 'Deforestation within a 3km buffer');
Map.addLayer(deforestation5km, {
    min: 0,
    max: 1,
    opacity: 0.5
}, 'Deforestation within a 5km buffer');
```

Lastly, we can estimate the deforestation area within 1 km of the protected area but only outside of the boundary.

```
var deforestation1kmOutside = deforestation1km
    .updateMask(protectedAreaRaster.unmask().not());

// Get the value of each pixel in square meters
// and divide by 10000 to convert to hectares.
var deforestation1kmOutsideArea =
deforestation1kmOutside.eq(1)
    .multiply(ee.Image.pixelArea()).divide(10000);

// We need to set a larger geometry than the protected area
// for the geometry parameter in reduceRegion().
var deforestationEstimate = deforestation1kmOutsideArea
    .reduceRegion({
        reducer: ee.Reducer.sum(),
        geometry: protectedArea.geometry().buffer(1000),
        scale: deforestation.projection().nominalScale()
    });

print('Deforestation within a 1km buffer outside the
protected area (ha)',
    deforestationEstimate);
```

Code Checkpoint F51e. The book’s repository contains a script that shows what your code should look like at this point.

23.3 Synthesis

Question 1. In this lab, we quantified rates of deforestation in La Paya. There is another protected area in the Colombian Amazon named Tinigua. By modifying the existing scripts, determine how the dynamics of forest change in Tinigua compare to those in La Paya with respect to:

- the number of deforestation events;
- the year with the greatest number of change events;
- the mean average area of change events;
- the total area of loss.

Question 2. In Sect. 23.2.1.4, we only considered losses of tree cover, but many protected areas will also have increases in tree cover from regrowth (which is typical of shifting agriculture). Calculate growth in hectares using the Global Forest Change dataset's gain layer for the six protected areas in Sect. 23.2.1.4 by extracting the raster properties and adding them to vector fields. Which has the greatest area of regrowth? Is this likely to be sufficient to balance out the rates of forest loss? Note: The gain layer shows locations where tree cover has increased for the period 2001–2012 (0 = no gain, 1 = tree cover increase), so for comparability use deforestation between the same time period of 2001–2012.

Question 3. In Sect. 23.2.2.2, we considered rates of deforestation in a buffer zone around La Paya. Estimate the deforestation rates inside of La Paya using buffer zones. Is forest loss more common close to the boundary of the reserve?

Question 4. Sometimes it's advantageous to perform processing using raster operations, particularly at large scales. It is possible to perform many of the tasks in Sects. 23.2.1.3 and 23.2.1.4 by first converting the protected area vector to raster and then using only raster operations. As an example, can you display only deforestation events > 10 ha in La Paya using only raster data? (Hint: Consider using `ee.Image.connectedPixelCount`. You may also want to also look at Sect. 23.2.2.1).

23.4 Conclusion

In this chapter, you learned how to convert raster to vector and vice versa. More importantly, you now have a better understanding of why and when such conversions are useful. The examples should give you practical applications and ideas for using these techniques.

Zonal Statistics

24

Sara Winsemius and Justin Braaten

Overview
The purpose of this chapter is to extract values from rasters for intersecting points or polygons. We will lay out the process and a function to calculate zonal statistics, which includes optional parameters to modify the function, and then apply the process to three examples using different raster datasets and combinations of parameters.

Learning Outcomes

- Buffering points as square or circular regions.
- Writing and applying functions with optional parameters.
- Learning what zonal statistics are and how to use reducers.
- Exporting computation results to a table.
- Copying properties from one image to another.

Assumes you know how to

- Recognize similarities and differences among Landsat 5, 7, and 8 spectral bands (Part I, Part II, Part III).
- Understand distinctions among `Image`, `ImageCollection`, `Feature`, and `FeatureCollection` Earth Engine objects (Part I, Part II, Part V).
- Use drawing tools to create points, lines, and polygons (Chap. 6).

S. Winsemius (✉)
Department of Land, Air and Water Resources, University of California, One Shields Ave, Davis, CA 95616-8627, USA
e-mail: swinsemius@ucdavis.edu

J. Braaten
Google Inc., 1600 Amphitheater Parkway, Mountain View, CA 94043, USA
e-mail: braaten@google.com

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_24

- Write a function and `map` it over an `ImageCollection` (Chap. 12).
- Mask cloud, cloud shadow, snow/ice, and other undesired pixels (Chap. 15).
- Export calculated data to tables with Tasks (Chap. 22).
- Understand the differences between raster and vector data (Chaps. 22 and 23).
- Write a function and `map` it over a `FeatureCollection` (Chap. 23).

24.1 Introduction to Theory

Anyone working with field data collected at plots will likely need to summarize raster-based data associated with those plots. For instance, they need to know the Normalized Difference Vegetation Index (NDVI), precipitation, or elevation for each plot (or surrounding region). Calculating statistics from a raster within given regions is called zonal statistics. Zonal statistics were calculated in Chaps. 22 and 23 using `ee.Image.ReduceRegions`. Here, we present a more general approach to calculating zonal statistics with a custom function that works for both `ee.Image` and `ee.ImageCollection` objects. In addition to its flexibility, the reduction method used here is less prone to "Computed value is too large" errors that can occur when using `ReduceRegions` with very large or complex `ee.FeatureCollection` object inputs.

The zonal statistics function in this chapter works for an `Image` or an `ImageCollection`. Running the function over an `ImageCollection` will produce a table with values from each image in the collection per point. Image collections can be processed before extraction as needed—for example, by masking clouds from satellite imagery or by constraining the dates needed for a particular research question. In this tutorial, the data extracted from rasters are exported to a table for analysis, where each row of the table corresponds to a unique point-image combination.

In fieldwork, researchers often work with plots, which are commonly recorded as polygon files or as a center point with a set radius. It is rare that plots will be set directly in the center of pixels from your desired raster dataset, and many field GPS units have positioning errors. Because of these issues, it may be important to use a statistic of adjacent pixels (as described in Chap. 10) to estimate the central value in what's often called a neighborhood mean or focal mean (Cansler and McKenzie 2012; Miller and Thode 2007).

To choose the size of your neighborhood, you will need to consider your research questions, the spatial resolution of the dataset, the size of your field plot, and the error from your GPS. For example, the raster value extracted for randomly placed 20 m diameter plots would likely merit use of a neighborhood mean when using Sentinel-2 or Landsat 8—at 10 m and 30 m spatial resolution, respectively—while using a thermal band from MODIS (Moderate Resolution Imaging Spectroradiometer) at 1000 m may not. While much of this tutorial is written with plot points and buffers in mind, a polygon asset with predefined regions will serve the same purpose.

Table 24.1 Parameters for `bufferPoints`

Parameter	Type	Description
radius	Number	buffer radius (m)
[bounds = false]	Boolean	An optional flag indicating whether to transform buffered point (i.e., a circle) to square bounds

24.2 Practicum

24.2.1 Section 1: Functions

Two functions are provided; copy and paste them into your script:

- A function to generate circular or square regions from buffered points.
- A function to extract image pixel neighborhood statistics for a given region.

24.2.1.1 Section 1.1: Function: BufferPoints(Radius, Bounds)

Our first function, `bufferPoints`, returns a function for adding a buffer to points and optionally transforming to rectangular bounds (see Table 24.1).

```
function bufferPoints(radius, bounds) {
    return function(pt) {
        pt = ee.Feature(pt);
        return bounds ? pt.buffer(radius).bounds() :
pt.buffer(radius);
    };
}
```

24.2.1.2 Section 1.2: Function: ZonalStats(Fc, Params)

The second function, `zonalStats`, reduces images in an `ImageCollection` by regions defined in a `FeatureCollection`. Note that reductions can return null statistics that you might want to filter out of the resulting feature collection. Null statistics occur when there are no valid pixels intersecting the region being reduced. This situation can be caused by points that are outside of an image or in regions that are masked for quality or clouds.

This function is written to include many optional parameters (see Table 24.2). Look at the function carefully and note how it is written to include defaults that make it easy to apply the basic function while allowing customization.

Table 24.2 Parameters for `zonalStats`

Parameter	Type	Description
ic	ee.ImageCollection	Image collection from which to extract values
fc	ee.FeatureCollection	Feature collection that provides regions/zones by which to reduce image pixels
[params]	Object	An optional Object that provides function arguments
[params.reducer = ee.Reducer.mean()]	ee.Reducer	The reducer to apply. Optional
[params.scale = null]	Number	A nominal scale in meters of the projection to work in. If null, the native nominal image scale is used. Optional
[params.crs = null]	String	The projection to work in. If null, the native image Coordinate Reference System (CRS) is used. Optional
[params.bands = null]	Array	A list of image band names for which to reduce values. If null, all bands will be reduced. Band names define column names in the resulting reduction table. Optional
[params.bandsRename = null]	Array	A list of desired image band names. The length and order must correspond to the params.bands list. If null, band names will be unchanged. Band names define column names in the resulting reduction table. Optional
[params.imgProps = null]	Array	A list of image properties to include in the table of region reduction results. If null, all image properties are included. Optional
[params.imgPropsRename = null]	Array	A list of image property names to replace those provided by params.imgProps. The length and order must match the params.imgProps entries. Optional
[params.datetimeName = ‘datetime]	String	The desired name of the datetime field. The datetime refers to the ‘system:time_start’ value of the ee.Image being reduced. Optional
[params.datetimeFormat = ‘YYYY-MM-dd HH:mm:ss]	String	The desired datetime format. Use ISO 8601 data string standards. The datetime string is derived from the ‘system:time_start’ value of the ee.Image being reduced. Optional

```
function zonalStats(ic, fc, params) {
    // Initialize internal params dictionary.
    var _params = {
        reducer: ee.Reducer.mean(),
        scale: null,
        crs: null,
        bands: null,
        bandsRename: null,
        imgProps: null,

        imgPropsRename: null,
        datetimeName: 'datetime',
        datetimeFormat: 'YYYY-MM-dd HH:mm:ss'
    };

    // Replace initialized params with provided params.
    if (params) {
        for (var param in params) {
            _params[param] = params[param] ||
_params[param];
        }
    }

    // Set default parameters based on an image
representative.
    var imgRep = ic.first();
    var nonSystemImgProps = ee.Feature(null)
        .copyProperties(imgRep).propertyNames();
    if (!_params.bands) _params.bands = imgRep.bandNames();
    if (!_params.bandsRename) _params.bandsRename =
_params.bands;
    if (!_params.imgProps) _params.imgProps =
nonSystemImgProps;
    if (!_params.imgPropsRename) _params.imgPropsRename =
_params
        .imgProps;

    // Map the reduceRegions function over the image
collection.
    var results = ic.map(function(img) {
        // Select bands (optionally rename), set a datetime
& timestamp property.
        img = ee.Image(img.select(_params.bands, _params
                .bandsRename))
            // Add datetime and timestamp features.
            .set(_params.datetimeName, img.date().format(
                _params.datetimeFormat))

            .set('timestamp',
img.get('system:time_start'));

        // Define final image property dictionary to set in
output features.
        var propsFrom = ee.List(_params.imgProps)
            .cat(ee.List([_params.datetimeName,
            'timestamp']));
        var propsTo = ee.List(_params.imgPropsRename)
            .cat(ee.List([_params.datetimeName,
            'timestamp']));
        var imgProps = img.toDictionary(propsFrom).rename(
            propsFrom, propsTo);

        // Subset points that intersect the given image.
        var fcSub = fc.filterBounds(img.geometry());

        // Reduce the image by regions.
        return img.reduceRegions({
                collection: fcSub,
                reducer: _params.reducer,
                scale: _params.scale,
                crs: _params.crs
            })
            // Add metadata to each feature.
            .map(function(f) {
                return f.set(imgProps);
            });

        // Converts the feature collection of feature
collections to a single
        //feature collection.
    }).flatten();

    return results;
}
```

24.2.2 Section 2: Point Collection Creation

Below, we create a set of points that form the basis of the zonal statistics calculations. Note that a unique `plot_id` property is added to each point. A unique plot or point ID is important to include in your vector dataset for future filtering and joining.

```
var pts = ee.FeatureCollection([
    ee.Feature(ee.Geometry.Point([-118.6010, 37.0777]), {
        plot_id: 1
    }),
    ee.Feature(ee.Geometry.Point([-118.5896, 37.0778]), {
        plot_id: 2
    }),
    ee.Feature(ee.Geometry.Point([-118.5842, 37.0805]), {
        plot_id: 3
    }),
    ee.Feature(ee.Geometry.Point([-118.5994, 37.0936]), {
        plot_id: 4
    }),
    ee.Feature(ee.Geometry.Point([-118.5861, 37.0567]), {
        plot_id: 5
    })
]);

print('Points of interest', pts);
```

Code Checkpoint F52a. The book's repository contains a script that shows what your code should look like at this point.

24.2.3 Section 3: Neighborhood Statistic Examples

The following examples demonstrate extracting raster neighborhood statistics for the following:

- A single raster with elevation and slope bands.
- A multiband MODIS time series.
- A multiband Landsat time series.

In each example, the points created in the previous section will be buffered and then used as regions to extract zonal statistics for each image in the image collection.

24.2.3.1 Section 3.1: Topographic Variables

This example demonstrates how to calculate zonal statistics for a single multiband image. This Digital Elevation Model (DEM) contains a single topographic band representing elevation.

Section 3.1.1: Buffer the Points

Next, we will apply a 45 m radius buffer to the points defined previously by mapping the `bufferPoints` function over the feature collection. The radius is set to 45 m to correspond to the 90 m pixel resolution of the DEM. In this case, circles are used instead of squares (set the second argument as false, i.e., do not use bounds).

```
// Buffer the points.
var ptsTopo = pts.map(bufferPoints(45, false));
```

Section 3.1.2: Calculate Zonal Statistics

There are two important things to note about the `zonalStats` function that this example addresses:

- It accepts only an `ee.ImageCollection`, not an `ee.Image`; single images must be wrapped in an `ImageCollection`.
- It expects every image in the input image collection to have a timestamp property named `'system:time_start'` with values representing milliseconds from 00:00:00 UTC on 1 January 1970. Most datasets should have this property, if not, one should be added.

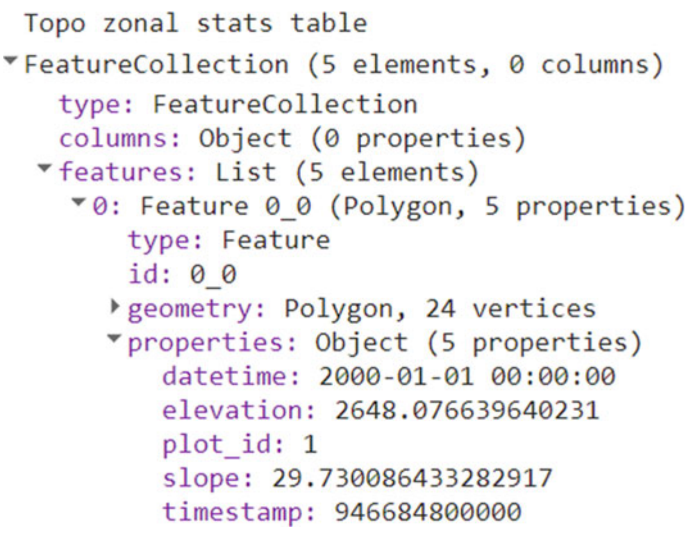
```
// Import the MERIT global elevation dataset.
var elev = ee.Image('MERIT/DEM/v1_0_3');

// Calculate slope from the DEM.
var slope = ee.Terrain.slope(elev);

// Concatenate elevation and slope as two bands of an
image.
var topo = ee.Image.cat(elev, slope)
    // Computed images do not have a 'system:time_start'
property; add one based
    // on when the data were collected.
    .set('system:time_start', ee.Date('2000-01-
01').millis());
```

```
// Wrap the single image in an ImageCollection for use in
the
// zonalStats function.
var topoCol = ee.ImageCollection([topo]);
```

Define arguments for the `zonalStats` function and then run it. Note that we are accepting defaults for the reducer, scale, Coordinate Reference System (CRS), and image properties to copy over to the resulting feature collection. Refer to the function definition above for defaults.

```
// Define parameters for the zonalStats function.
var params = {
    bands: [0, 1],
    bandsRename: ['elevation', 'slope']
};

// Extract zonal statistics per point per image.
var ptsTopoStats = zonalStats(topoCol, ptsTopo, params);
print('Topo zonal stats table', ptsTopoStats);

// Display the layers on the map.
Map.setCenter(-118.5957, 37.0775, 13);
Map.addLayer(topoCol.select(0), {
    min: 2400,
    max: 4200
}, 'Elevation');
Map.addLayer(topoCol.select(1), {
    min: 0,
    max: 60
}, 'Slope');
Map.addLayer(pts, {
    color: 'purple'
}, 'Points');
Map.addLayer(ptsTopo, {
    color: 'yellow'
}, 'Points w/ buffer');
```

The result is a copy of the buffered point feature collection with new properties added for the region reduction of each selected image band according to the given reducer. A part of the `FeatureCollection` is shown in Fig. 24.1. The data in that `FeatureCollection` corresponds to a table containing the information of Table 24.3. See Fig. 24.2 for a graphical representation of the points and the topographic data being summarized.



```
Topo zonal stats table
▾FeatureCollection (5 elements, 0 columns)
   type: FeatureCollection
   columns: Object (0 properties)
  ▾features: List (5 elements)
    ▾0: Feature 0_0 (Polygon, 5 properties)
        type: Feature
        id: 0_0
       ▸geometry: Polygon, 24 vertices
       ▾properties: Object (5 properties)
          datetime: 2000-01-01 00:00:00
          elevation: 2648.076639640231
          plot_id: 1
          slope: 29.730086433282917
          timestamp: 946684800000
```

Fig. 24.1 A part of the FeatureCollection produced by calculating the zonal statistics

Table 24.3 Example output from zonalStats organized as a table

plot_id	timestamp	Datetime	elevation	slope
1	946684800000	2000-01-01 00:00:00	2648.1	29.7
2	946684800000	2000-01-01 00:00:00	2888.2	33.9
3	946684800000	2000-01-01 00:00:00	3267.8	35.8
4	946684800000	2000-01-01 00:00:00	2790.7	25.1
5	946684800000	2000-01-01 00:00:00	2559.4	29.4

Rows correspond to collection features and columns are feature properties. Note that elevation and slope values in this table are rounded to the nearest tenth for brevity

24.2.3.2 Section 3.2: MODIS Time Series

A time series of MODIS eight-day surface reflectance composites demonstrates how to calculate zonal statistics for a multiband ImageCollection that requires no preprocessing, such as cloud masking or computation. Note that there is no built-in function for performing region reductions on ImageCollection objects. The zonalStats function that we are using for reduction is mapping the reduceRegions function over an ImageCollection.

Section 3.2.1: Buffer the Points

In this example, suppose the point collection represents center points for field plots that are 100 × 100 m, and apply a 50 m radius buffer to the points to match the size of the plot. Since we want zonal statistics for square plots, set the second

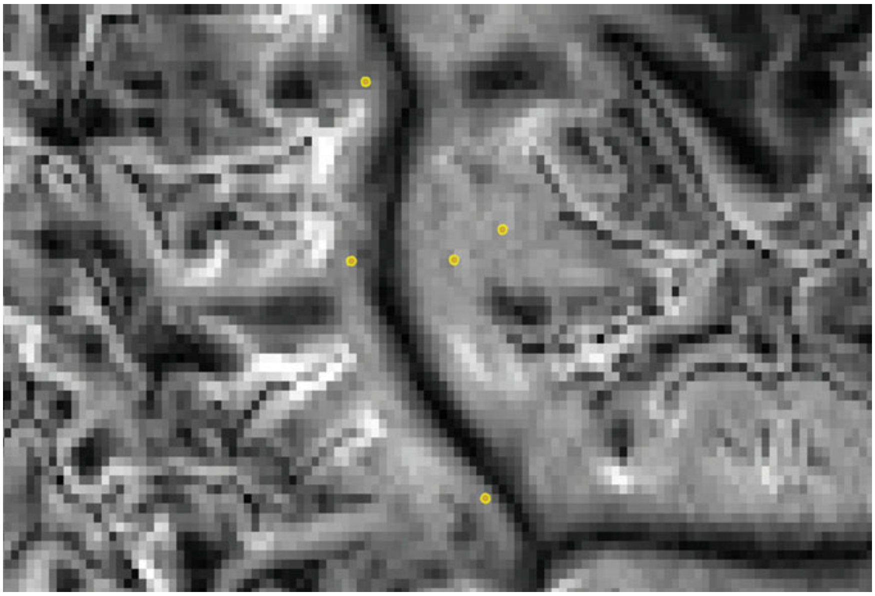


Fig. 24.2 Sample points and topographic slope. Elevation and slope values for regions intersecting each buffered point are reduced and attached as properties of the points

argument of the `bufferPoints` function to true, so that square bounds of the buffered points are returned.

```
var ptsModis = pts.map(bufferPoints(50, true));
```

Section 3.2.2: Calculate Zonal Statistic

Import the MODIS 500 m global eight-day surface reflectance composite collection and filter the collection to include data for July, August, and September from 2015 through 2019.

```
var modisCol = ee.ImageCollection('MODIS/006/MOD09A1')
    .filterDate('2015-01-01', '2020-01-01')
    .filter(ee.Filter.calendarRange(183, 245,
'DAY_OF_YEAR'));
```

Reduce each image in the collection by each plot according to the following parameters. Note that this time the reducer is defined as the neighborhood median (`ee.Reducer.median`) instead of the default mean, and that scale, CRS, and properties for the datetime are explicitly defined.

```
// Define parameters for the zonalStats function.
var params = {
    reducer: ee.Reducer.median(),
    scale: 500,
    crs: 'EPSG:5070',
    bands: ['sur_refl_b01', 'sur_refl_b02',
'sur_refl_b06'],
    bandsRename: ['modis_red', 'modis_nir', 'modis_swir'],
    datetimeName: 'date',
    datetimeFormat: 'YYYY-MM-dd'
};

// Extract zonal statistics per point per image.
var ptsModisStats = zonalStats(modisCol, ptsModis, params);
print('Limited MODIS zonal stats table',
ptsModisStats.limit(50));
```

The result is a feature collection with a feature for all combinations of plots and images. Interpreted as a table, the result has 200 rows (5 plots times 40 images) and as many columns as there are feature properties. Feature properties include those from the plot asset and the image, and any associated non-system image properties. Note that the printed results are limited to the first 50 features for brevity.

24.2.3.3 Section 3.3: Landsat Time Series

This example combines Landsat surface reflectance imagery across three instruments: Thematic Mapper (TM) from Landsat 5, Enhanced Thematic Mapper Plus (ETM +) from Landsat 7, and Operational Land Imager (OLI) from Landsat 8.

The following section prepares these collections so that band names are consistent and cloud masks are applied. Reflectance among corresponding bands are roughly congruent for the three sensors when using the surface reflectance product; therefore, the processing steps that follow do not address inter-sensor harmonization. Review the current literature on inter-sensor harmonization practices if you'd like to apply a correction.

Section 3.3.1: Prepare the Landsat Image Collection

First, define the function to mask cloud and shadow pixels (See Chap. 15 for more detail on cloud masking).

```
// Mask clouds from images and apply scaling factors.
function maskScale(img) {
    var qaMask =
img.select('QA_PIXEL').bitwiseAnd(parseInt('11111',
        2)).eq(0);
    var saturationMask = img.select('QA_RADSAT').eq(0);

    // Apply the scaling factors to the appropriate bands.
    var getFactorImg = function(factorNames) {
        var factorList =
img.toDictionary().select(factorNames)
            .values();
        return ee.Image.constant(factorList);
    };
    var scaleImg =
getFactorImg(['REFLECTANCE_MULT_BAND_.']);
    var offsetImg =
getFactorImg(['REFLECTANCE_ADD_BAND_.']);
    var scaled =
img.select('SR_B.').multiply(scaleImg).add(
    offsetImg);

    // Replace the original bands with the scaled ones and
apply the masks.
    return img.addBands(scaled, null, true)
        .updateMask(qaMask)
        .updateMask(saturationMask);
}
```

Next, define functions to select and rename the bands of interest for the Operational Land Imager (OLI) aboard Landsat 8, and for the TM/ETM + imagers aboard earlier Landsats. This is important because the band numbers are different for OLI and TM/ETM+, and it will make future index calculations easier.

```
// Selects and renames bands of interest for Landsat OLI.
function renameOli(img) {
    return img.select(
        ['SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6',
'SR_B7'],
        ['Blue', 'Green', 'Red', 'NIR', 'SWIR1', 'SWIR2']);
}

// Selects and renames bands of interest for TM/ETM+.
function renameEtm(img) {
    return img.select(
        ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5',
'SR_B7'],
        ['Blue', 'Green', 'Red', 'NIR', 'SWIR1', 'SWIR2']);
}
```

Combine the cloud mask and band renaming functions into preparation functions for OLI and TM/ETM+. Add any other sensor-specific preprocessing steps that you'd like to the functions below.

```
// Prepares (cloud masks and renames) OLI images.
function prepOli(img) {
    img = maskScale(img);
    img = renameOli(img);
    return img;
}

// Prepares (cloud masks and renames) TM/ETM+ images.
function prepEtm(img) {
    img = maskScale(img);
    img = renameEtm(img);
    return img;
}
```

Get the Landsat surface reflectance collections for OLI, ETM+, and TM sensors. Filter them by the bounds of the point feature collection and apply the relevant image preparation function.

```
var ptsLandsat = pts.map(bufferPoints(15, true));

var oliCol = ee.ImageCollection('LANDSAT/LC08/C02/T1_L2')
    .filterBounds(ptsLandsat)
    .map(prepOli);

var etmCol = ee.ImageCollection('LANDSAT/LE07/C02/T1_L2')
    .filterBounds(ptsLandsat)
    .map(prepEtm);

var tmCol = ee.ImageCollection('LANDSAT/LT05/C02/T1_L2')
    .filterBounds(ptsLandsat)
    .map(prepEtm);
```

Merge the prepared sensor collections.

```
var landsatCol = oliCol.merge(etmCol).merge(tmCol);
```

Section 3.3.2: Calculate Zonal Statistics

Reduce each image in the collection by each plot according to the following parameters. Note that this example defines the `imgProps` and `imgPropsRename` parameters to copy over and rename just two selected image properties: Landsat image ID and the satellite that collected the data. It also uses the `max` reducer, which, as an unweighted reducer, will return the maximum value from pixels that have their centroid within the buffer (see Sect. 24.2.4.1 below for more details).

```
// Define parameters for the zonalStats function.
var params = {
    reducer: ee.Reducer.max(),
    scale: 30,
    crs: 'EPSG:5070',
    bands: ['Blue', 'Green', 'Red', 'NIR', 'SWIR1',
'SWIR2'],
    bandsRename: ['ls_blue', 'ls_green', 'ls_red',
'ls_nir',
        'ls_swir1', 'ls_swir2'
    ],
    imgProps: ['SENSOR_ID', 'SPACECRAFT_ID'],
    imgPropsRename: ['img_id', 'satellite'],
    datetimeName: 'date',
    datetimeFormat: 'YYYY-MM-dd'
};
```

```
// Extract zonal statistics per point per image.
var ptsLandsatStats = zonalStats(landsatCol, ptsLandsat,
params)
    // Filter out observations where image pixels were all
masked.
    .filter(ee.Filter.notNull(params.bandsRename));
print('Limited Landsat zonal stats table',
ptsLandsatStats.limit(50));
```

The result is a feature collection with a feature for all combinations of plots and images.

Section 3.3.3: Dealing with Large Collections

If your browser times out, try exporting the results (as described in Chap. 29). It's likely that point feature collections that cover a large area or contain many points (point-image observations) will need to be exported as a batch task by either exporting the final feature collection as an asset or as a CSV/shapefile/GeoJSON to Google Drive or GCS.

Here is how you would export the above Landsat image-point feature collection to an asset and to Google Drive. Run the following code, activate the Code Editor **Tasks** tab, and then click the **Run** button. If you don't specify your own existing folder in Drive, the folder "EEFA_outputs" will be created.

```
Export.table.toAsset({
    collection: ptsLandsatStats,
    description: 'EEFA_export_Landsat_to_points',
    assetId: 'EEFA_export_values_to_points'
});

Export.table.toDrive({
    collection: ptsLandsatStats,
    folder: 'EEFA_outputs', // this will create a new
folder if it doesn't exist
    description: 'EEFA_export_values_to_points',
    fileFormat: 'CSV'
});
```

Code Checkpoint F52b. The book's repository contains a script that shows what your code should look like at this point.

24.2.4 Section 4: Additional Notes

24.2.4.1 Section 4.1: Weighted Versus Unweighted Region Reduction

A region used for calculation of zonal statistics often bisects multiple pixels. Should partial pixels be included in zonal statistics? Earth Engine lets you decide by allowing you to define a reducer as either weighted or unweighted (or you can provide per-pixel weight specification as an image band). A *weighted* reducer will include partial pixels in the zonal statistic calculation by weighting each pixel's contribution according to the fraction of the area intersecting the region. An *unweighted* reducer, on the other hand, gives equal weight to all pixels whose cell center intersects the region; all other pixels are excluded from calculation of the statistic.

For aggregate reducers like `ee.Reducer.mean` and `ee.Reducer.median`, the default mode is weighted, while identifier reducers such as `ee.Reducer.min` and `ee.Reducer.max` are unweighted. You can adjust the behavior of weighted reducers by calling `unweighted` on them, as in `ee.Reducer.mean.unweighted`. You may also specify the weights by modifying the reducer with `splitWeights`; however, that is beyond the scope of this book.

24.2.4.2 Section 4.2: Copy Properties to Computed Images

Derived, computed images do not retain the properties of their source image, so be sure to copy properties to computed images if you want them included in the region reduction table. For instance, consider the simple computation of unscaling Landsat SR data:

```
// Define a Landsat image.
var img =
ee.ImageCollection('LANDSAT/LC08/C02/T1_L2').first();

// Print its properties.
print('All image properties', img.propertyNames());

// Subset the reflectance bands and unscale them.
var computedImg =
img.select('SR_B.').multiply(0.0000275).add(-0.2);

// Print the unscaled image's properties.
print('Lost original image properties',
computedImg.propertyNames());
```

Notice how the computed image does not have the source image's properties and only retains the bands information. To fix this, use the `copyProperties` function to add desired source properties to the derived image. It is best practice to copy only the properties you really need because some properties, such as those containing geometry objects, lists, or feature collections, can significantly increase the computational burden for large collections.

```
// Subset the reflectance bands and unscale them, keeping
selected
// source properties.
var computedImg =
img.select('SR_B.').multiply(0.0000275).add(-0.2)
    .copyProperties(img, ['system:time_start',
'LANDSAT_PRODUCT_ID']);

// Print the unscaled image's properties.
print('Selected image properties retained', computedImg
.propertyNames());
```

Now selected properties are included. Use this technique when returning computed, derived images in a mapped function, and in single-image operations.

24.2.4.3 Section 4.3: Understanding Which Pixels Are Included in Polygon Statistics

If you want to visualize what pixels are included in a polygon for a region reducer, you can adapt the following code to use your own region (by replacing geometry), dataset, desired scale, and CRS parameters. The important part to note is that the image data you are adding to the map is reprojected using the same scale and CRS as that used in your region reduction (see Fig. 24.3).

```
// Define polygon geometry.
var geometry = ee.Geometry.Polygon(
    [
        [
            [-118.6019835717645, 37.079867782687884],
            [-118.6019835717645, 37.07838698844939],
            [-118.60036351751951, 37.07838698844939],
            [-118.60036351751951, 37.079867782687884]
        ]
    ], null, false);

// Import the MERIT global elevation dataset.
var elev = ee.Image('MERIT/DEM/v1_0_3');
```

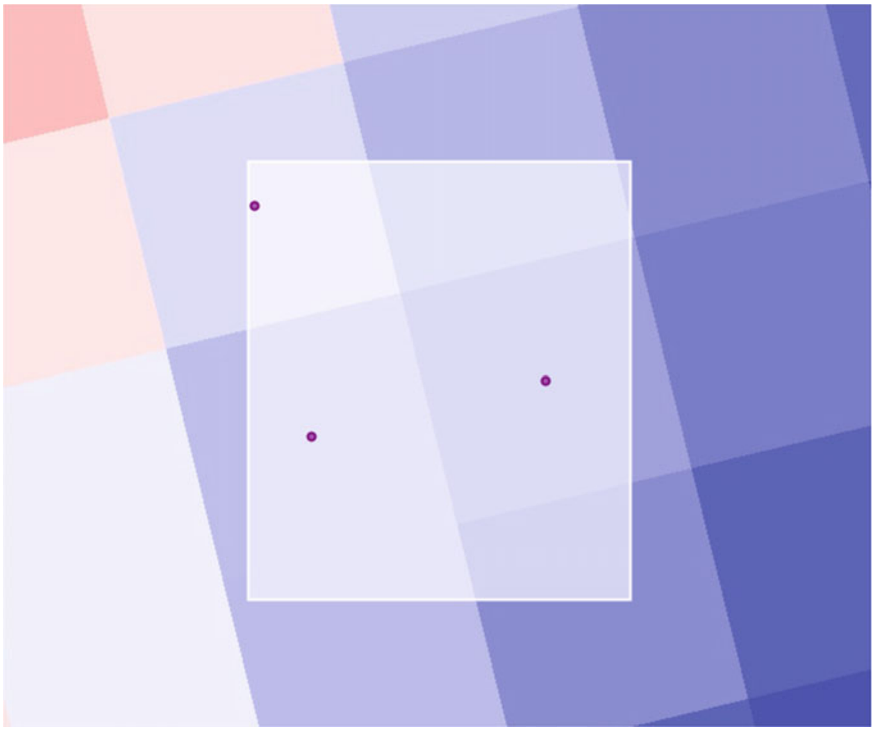


Fig. 24.3 Identifying pixels used in zonal statistics. By mapping the image and vector together, you can see which pixels are included in the unweighted statistic. For this example, three pixels would be included in the statistic because the polygon covers the center point of three pixels

```
// Define desired scale and crs for region reduction (for
image display too).
var proj = {
    scale: 90,
    crs: 'EPSG:5070'
};
```

The `count` reducer will return how many pixel centers are overlapped by the polygon region, which would be the number of pixels included in any unweighted reducer statistic. You can also visualize which pixels will be included in the reduction by using the `toCollection` reducer on a latitude/longitude image and adding resulting coordinates as feature geometry. Be sure to specify CRS and scale for both the region reducers and the reprojected layer added to the map (see bullet list below for more details).

```
// A count reducer will return how many pixel centers are
overlapped by the
// polygon region.
var count = elev.select(0).reduceRegion({
    reducer: ee.Reducer.count(),
    geometry: geometry,
    scale: proj.scale,
    crs: proj.crs
});
print('n pixels in the reduction', count.get('dem'));

// Make a feature collection of pixel center points for
those that are
// included in the reduction.
var pixels = ee.Image.pixelLonLat().reduceRegion({
    reducer: ee.Reducer.toCollection(['lon', 'lat']),
    geometry: geometry,
    scale: proj.scale,
    crs: proj.crs
});
var pixelsFc =
ee.FeatureCollection(pixels.get('features')).map(
    function(f) {
        return
f.setGeometry(ee.Geometry.Point([f.get('lon'), f
            .get('lat')
        ]));
    });
```

```
// Display layers on the map.
Map.centerObject(geometry, 18);
Map.addLayer(
    elev.reproject({
        crs: proj.crs,
        scale: proj.scale
    }),
    {
        min: 2500,
        max: 3000,
        palette: ['blue', 'white', 'red']
            }, 'Image');
        Map.addLayer(geometry, {

            color: 'white'
        }, 'Geometry');
        Map.addLayer(pixelsFc, {
            color: 'purple'
        }, 'Pixels in reduction');
```

Code Checkpoint F52c. The book's repository contains a script that shows what your code should look like at this point.

Finally, here are some notes on CRS and scale:

- Earth Engine runs `reduceRegion` using the projection of the image's first band if the CRS is unspecified in the function. For imagery spanning multiple UTM zones, for example, this would lead to different origins. For some functions Earth Engine uses the default EPSG:4326. Therefore, when the opportunity is presented, such as by the `reduceRegion` function, it is important to specify the scale and CRS explicitly.
- The `Map` default CRS is EPSG:3857. When looking closely at pixels on the map, the data layer scale and CRS should also be set explicitly. Note that zooming out after setting a relatively small scale when reprojecting may result in memory and/or timeout errors because optimized pyramid layers for each zoom level will not be used.
- Specifying the CRS and scale in both the `reduceRegion` and `addLayer` functions allows the map visualization to align with the information printed in the **Console**.
- The Earth Engine default, WGS 84 lat long (EPSG:4326), is a generic CRS that works worldwide. The code above reprojects to EPSG:5070, North American Equal Albers, which is a CRS that preserves area for North American locations. Use the CRS that is best for your use case when adapting this to your own project or maintain (and specify) the CRS of the image using, for example, `crs: 'img.projection().crs()'`.

24.3 Synthesis

Question 1. Look at the MODIS example (Sect. 24.2.3.2), which uses the median reducer. Try modifying the reducer to be unweighted, either by specifying `unweighted` or by using an identifier reducer like `max`. What happens and why?

Question 2. Calculate zonal statistics for your own buffered points or polygons using a raster and reducer of interest. Be sure to consider the spatial scale of the raster and whether a weighted or unweighted reducer would be more appropriate for your interests.

If the point or polygon file is stored in a local shapefile or CSV file, first upload the data to your Earth Engine assets. All columns in your vector file, such as the plot name, will be retained through this process. Once you have an Earth Engine table asset ready, import the asset into your script by hovering over the name of the asset and clicking the arrow at the right side, or by calling it in your script with the following code.

```
var pts = ee.FeatureCollection('users/yourUsername/yourAsset');
```

If you prefer to define points or polygons dynamically rather than loading an asset, you can add them to your script using the geometry tools. See Chap. 6 and 22 for more detail on adding and creating vector data.

Question 3. Try the code from Sect. 24.2.4.3 using the MODIS data and the first point from the `pts` variable. Among other modifications, you will need to create a buffer for the point, take a single MODIS image from the collection, and change visualization parameters.

- Think about the CRS in the code: The code reprojects to EPSG:5070, but MODIS is collected in the sinusoidal projection SR-ORG:6974. Try that CRS and describe how the image changes.
- Is the count reducer weighted or unweighted? Give an example of a circumstance to use a weighted reducer and an example for an unweighted reducer. Specify the buffer size you would use and the spatial resolution of your dataset.

Question 4. In the examples above, only a single `ee.Reducer` is passed to the `zonalStats` function, which means that only a single statistic is calculated (e.g., zonal mean or median or maximum). What if you want multiple statistics—can you alter the code in Sect. 24.2.3.1 to (1) make the point buffer 500 instead of 45; (2) add the `reducer` parameter to the `params` dictionary; and (3) as its argument, supply a combined `ee.Reducer` that will calculate minimum, maximum, standard deviation, and mean statistics?

To achieve this you'll need to chain several `ee.Reducer.combine` functions together. Note that if you accept all the individual `ee.Reducer` and `ee.Reducer.combine` function defaults, you'll run into two problems related to reducer weighting differences, and whether or not the image inputs are shared among the combined set of reducers. How can you manipulate the individual `ee.Reducer` and `ee.Reducer.combine` functions to achieve the goal of calculating multiple zonal statistics in one call to the `zonalStats` function?

24.4 Conclusion

In this chapter, you used functions containing optional parameters to extract raster values for collocated points. You also learned how to buffer points and apply weighted and unweighted reducers to get different types of zonal statistics. These functions were applied to three examples that differed by raster dataset, reducer, spatial resolution, and scale. Lastly, you covered related topics like weighting of reducers and buffer visualization. Now you're ready to apply these ideas to your own work!

References

Cansler CA, McKenzie D (2012) How robust are burn severity indices when applied in a new region? Evaluation of alternate field-based and remote-sensing methods. Remote Sens 4:456–483. https://doi.org/10.3390/rs4020456

Miller JD, Thode AE (2007) Quantifying burn severity in a heterogeneous landscape with a relative version of the delta Normalized Burn Ratio (dNBR). Remote Sens Environ 109:66–80. https://doi.org/10.1016/j.rse.2006.12.006

Advanced Vector Operations

25

Ujaval Gandhi

Overview

This chapter covers advanced techniques for visualizing and analyzing vector data in Earth Engine. There are many ways to visualize feature collections, and you will learn how to pick the appropriate method to create visualizations, such as a choropleth map. We will also cover geoprocessing techniques involving multiple vector layers, such as selecting features in one layer by their proximity to features in another layer and performing spatial joins.

Learning Outcomes

- Visualizing any vector dataset and creating a thematic map.
- Understanding joins in Earth Engine.
- Carrying out geoprocessing tasks with vector layers in Earth Engine.

Assumes you know how to

- Filter a `FeatureCollection` to obtain a subset (Chaps. 22 and 23).
- Write a function and `map` it over a `FeatureCollection` (Chaps. 23 and 24).

U. Gandhi (✉)
Spatial Thoughts, Ahmedabad, India
e-mail: ujaval@spatialthoughts.com

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_25

25.1 Practicum

25.1.1 Section 1: Visualizing Feature Collections

There is a distinct difference between how rasters and vectors are visualized. While images are typically visualized based on pixel values, vector layers use feature properties (i.e., attributes) to create a visualization. Vector layers are rendered on the **Map** by assigning a value to the red, green, and blue channels for each pixel on the screen based on the geometry and attributes of the features. The functions used for vector data visualization in Earth Engine are listed below in increasing order of complexity.

- `Map.addLayer`: As with raster layers, you can add a `FeatureCollection` to the **Map** by specifying visualization parameters. This method supports only one visualization parameter: `color`. All features are rendered with the specified color.
- `draw`: This function supports the parameters `pointRadius` and `strokeWidth` in addition to `color`. It renders all features of the layer with the specified parameters.
- `paint`: This is a more powerful function that can render each feature with a different `color` and `width` based on the values in the specified property.
- `style`: This is the most versatile function. It can apply a different style to each feature, including `color`, `pointSize`, `pointShape`, `width`, `fillColor`, and `lineType`.

In the exercises below, we will learn how to use each of these functions and see how they can generate different types of maps.

25.1.1.1 Section 1.1: Creating a Choropleth Map

We will use the TIGER: US Census Blocks layer, which stores census block boundaries and their characteristics within the United States, along with the San Francisco neighborhoods layer from Chap. 22 to create a population density map for the city of San Francisco.

We start by loading the census blocks and San Francisco neighborhoods layers. We use `ee.Filter.bounds` to filter the census blocks layer to the San Francisco boundary.

```
var blocks = ee.FeatureCollection('TIGER/2010/Blocks');
var roads = ee.FeatureCollection('TIGER/2016/Roads');
var sfNeighborhoods = ee.FeatureCollection(
    'projects/gee-book/assets/F5-0/SFneighborhoods');

var geometry = sfNeighborhoods.geometry();
Map.centerObject(geometry);

// Filter blocks to the San Francisco boundary.
var sfBlocks = blocks.filter(ee.Filter.bounds(geometry));
```

The simplest way to visualize this layer is to use `Map.addLayer` (Fig. 25.1). We can specify a `color` value in the `visParams` parameter of the function. Each census block polygon will be rendered with stroke and fill of the specified color. The fill color is the same as the stroke color but has a 66% opacity.

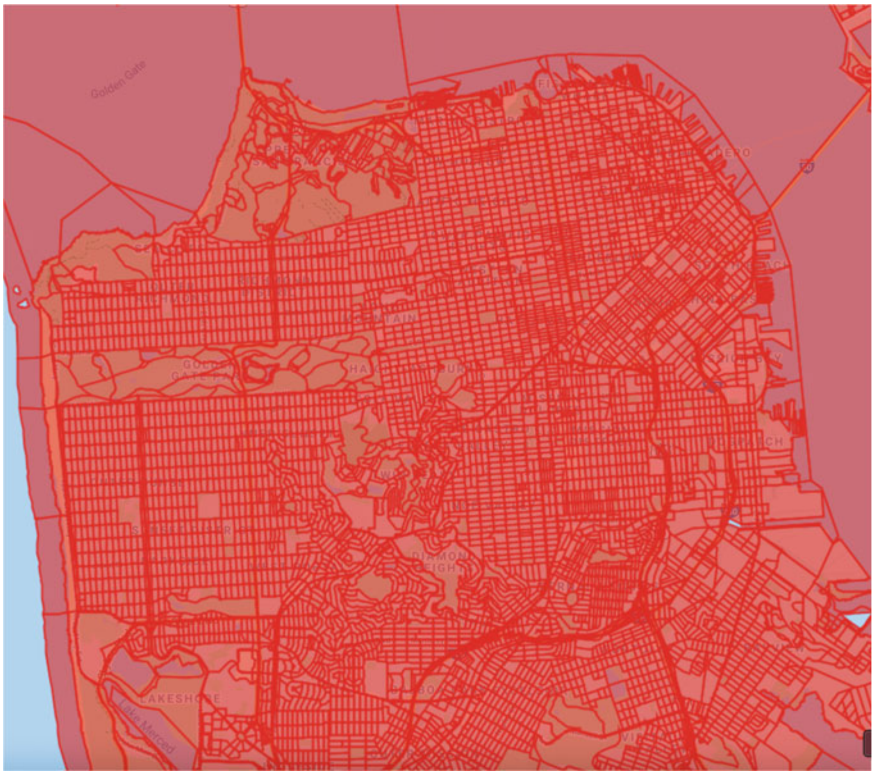


Fig. 25.1 San Francisco census blocks

```
// Visualize with a single color.
Map.addLayer(sfBlocks, {
    color: '#de2d26'
}, 'Census Blocks (single color)');
```

The census blocks table has a property named `'pop10'` containing the population totals as of the 2010 census. We can use this to create a choropleth map showing population density. We first need to compute the population density for each feature and add it as a property. To add a new property to each feature, we can `map` a function over the `FeatureCollection` and calculate the new property called `'pop_density'`. Earth Engine provides the `area` function, which can calculate the area of a feature in square meters. We convert it to square miles and calculate the population density per square mile.

```
// Add a pop_density column.
var sfBlocks = sfBlocks.map(function(f) {
    // Get the polygon area in square miles.
    var area_sqmi = f.area().divide(2.59e6);
    var population = f.get('pop10');
    // Calculate population density.
    var density = ee.Number(population).divide(area_sqmi);
    return f.set({
        'area_sqmi': area_sqmi,
        'pop_density': density
    });
});
```

Now we can use the `paint` function to create an image from this `FeatureCollection` using the `pop_density` property. The `paint` function needs an empty image that needs to be cast to the appropriate data type. Let's use the `aggregate_stats` function to calculate basic statistics for the given column of a `FeatureCollection`.

```
// Calculate the statistics of the newly computed column.
var stats = sfBlocks.aggregate_stats('pop_density');
print(stats);
```

You will see that the population density values have a large range. We also have values that are greater than 100,000, so we need to make sure we select a data type that can store values of this size. We create an empty image and cast it to `int32`, which is able to hold large integer values.

```
// Create an empty image into which to paint the features.
// Cast to 32-bit integer which supports storing values
// up to 2,147,483,647.

var empty = ee.Image().int32();
```

Now we can use the `paint` function, seen briefly in Chap. 23, to assign each pixel's value based on the `pop_density` property.

```
var sfBlocksPaint = empty.paint({
    featureCollection: sfBlocks,
    color: 'pop_density',
});
```

The result is an image with pixel values representing the population density of the polygons. We can now use the standard image visualization method to add this layer to the **Map** (Fig. 25.2). Then, we need to determine minimum and maximum values for the visualization parameters. A reliable technique to produce a good visualization is to find minimum and maximum values that are within one standard deviation. From the statistics that we calculated earlier, we can estimate good minimum and maximum values to be 0 and 50,000, respectively.

```
var palette = ['fee5d9', 'fcae91', 'fb6a4a', 'de2d26',
'a50f15'];
var visParams = {
    min: 0,
    max: 50000,
    palette: palette
};
Map.addLayer(sfBlocksPaint.clip(geometry), visParams,
    'Population Density');
```

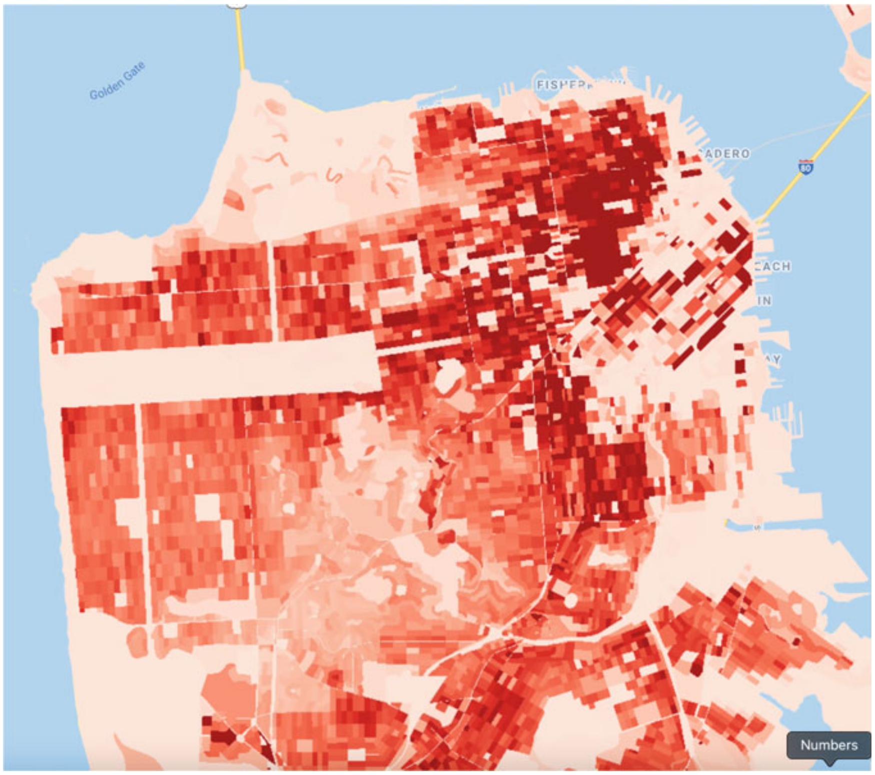


Fig. 25.2 San Francisco population density

25.1.1.2 Section 1.2: Creating a Categorical Map

Continuing the exploration of styling methods, we will now learn about `draw` and `style`. These are the preferred methods of styling for points and line layers. Let's see how we can visualize the TIGER: US Census Roads layer to create a categorical map.

We start by filtering the roads layer to the San Francisco boundary and using `Map.addLayer` to visualize it.

```
// Filter roads to San Francisco boundary.
var sfRoads = roads.filter(ee.Filter.bounds(geometry));

Map.addLayer(sfRoads, {
    color: 'blue'
}, 'Roads (default)');
```

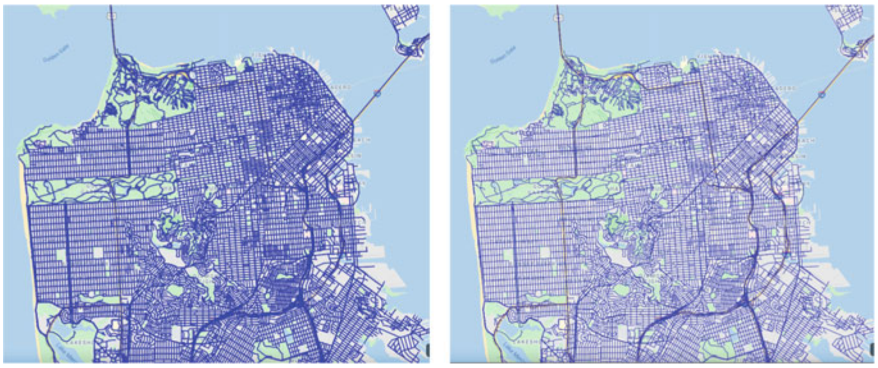


Fig. 25.3 San Francisco roads rendered with a line width of 2 pixels (left) and a line width of 1 pixel (right)

The default visualization renders each line using a width of 2 pixels. The `draw` function provides a way to specify a different line width. Let's use it to render the layer with the same color as before but with a line width of 1 pixel (Fig. 25.3).

```
// Visualize with draw().
var sfRoadsDraw = sfRoads.draw({
    color: 'blue',
    strokeWidth: 1
});
Map.addLayer(sfRoadsDraw, {}, 'Roads (Draw)');
```

The road layer has a column called "MTFCC" (standing for the MAF/TIGER Feature Class Code). This contains the road priority codes, representing the various types of roads, such as primary and secondary. We can use this information to render each road segment according to its priority. The `draw` function doesn't allow us to specify different styles for each feature. Instead, we need to make use of the `style` function.

The column contains string values indicating different road types as indicated in Table 25.1. This full list is available at the MAF/TIGER Feature Class Code Definitions page on the US Census Bureau website.

Let's say we want to create a map with rules based on the MTFCC values shown in Table 25.2.

Table 25.1 Census Bureau road priority codes

MTFCC	Feature class
S1100	Primary road
S1200	Secondary road
S1400	Local neighborhood road, rural road, city street
S1500	Vehicular trail
S1630	Ramp
S1640	Service drive
S1710	Walkway/pedestrian trail
S1720	Stairway
S1730	Alley
S1740	Private road for service vehicles
S1750	Internal US Census Bureau use
S1780	Parking lot road
S1820	Bike path or trail
S1830	Bridle path
S2000	Road median

Table 25.2 Styling parameters for road priority codes

MTFCC	Color	Line width
S1100	Blue	3
S1200	Green	2
S1400	Orange	1
All other classes	Gray	1

Let's define a dictionary containing the styling information.

```
var styles = ee.Dictionary({
    'S1100': {
        'color': 'blue',
        'width': 3
    },
    'S1200': {
        'color': 'green',
        'width': 2
    },
    'S1400': {
        'color': 'orange',
        'width': 1
    }
});
var defaultStyle = {
    color: 'gray',
    'width': 1
};
```

The `style` function needs a property in the `FeatureCollection` that contains a dictionary with the style parameters. This allows you to specify a different style for each feature. To create a new property, we `map` a function over the `FeatureCollection` and assign an appropriate style dictionary to a new property named `'style'`. Note the use of the `get` function, which allows us to fetch the value for a key in the dictionary. It also takes a default value in case the specified key does not exist. We make use of this to assign different styles to the three road classes specified in Table 25.2 and a default style to all others.

```
var sfRoads = sfRoads.map(function(f) {
    var classcode = f.get('mtfcc');
    var style = styles.get(classcode, defaultStyle);
    return f.set('style', style);
});
```

Our collection is now ready to be styled. We call the `style` function to specify the property that contains the dictionary of style parameters. The output of the `style` function is an RGB image rendered from the `FeatureCollection` (Fig. 25.4).

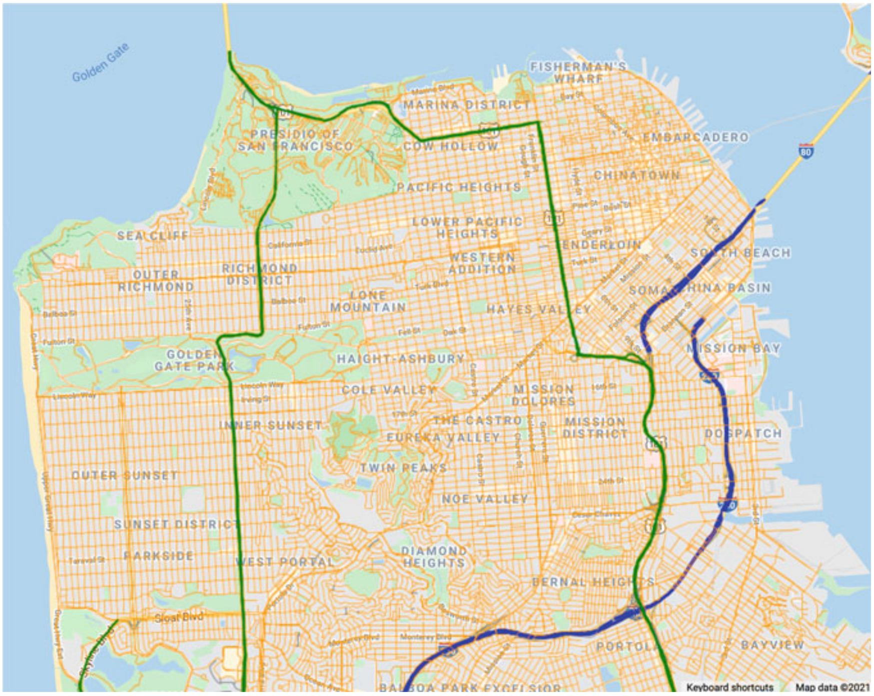


Fig. 25.4 San Francisco roads rendered according to road priority

```
var sfRoadsStyle = sfRoads.style({
    styleProperty: 'style'
});
Map.addLayer(sfRoadsStyle.clip(geometry), {}, 'Roads
(Style)');
```

Code Checkpoint F53a. The book's repository contains a script that shows what your code should look like at this point.

Save your script for your own future use, as outlined in Chap. 1. Then, refresh the Code Editor to begin with a new script for the next section.

25.1.2 Section 2: Joins with Feature Collections

Earth Engine was designed as a platform for processing raster data, and that is where it shines. Over the years, it has acquired advanced vector data processing capabilities, and users are now able to carry out complex geoprocessing tasks within Earth Engine. You can leverage the distributed processing power of Earth Engine to process large vector layers in parallel.

This section shows how you can do spatial queries and spatial joins using multiple large feature collections. This requires the use of joins. As described for Image Collections in Chap. 21, a join allows you to match every item in a collection with items in another collection based on certain conditions. While you can achieve similar results using `map` and `filter`, joins perform better and give you more flexibility. We need to define the following items to perform a join on two collections.

1. **Filter**: A filter defines the condition used to select the features from the two collections. There is a suite of filters in the `ee.Filters` module that work on two collections, such as `ee.Filter.equals` and `ee.Filter.withinDistance`.
2. **Join type**: While the filter determines which features will be joined, the join type determines how they will be joined. There are many join types, including *simple join*, *inner join*, and *save-all join*.

Joins are one of the harder skills to master, but doing so will help you perform many complex analysis tasks within Earth Engine. We will go through practical examples that will help you understand these concepts and the workflow better.

25.1.2.1 Section 2.1: Selecting by Location

In this section, we will learn how to select features from one layer that are within a specified distance from features in another layer. We will continue to work with the San Francisco census blocks and roads datasets from the previous section. We will implement a join to select all blocks in San Francisco that are within 1 km of an interstate highway.

We start by loading the census blocks and roads collections and filtering the roads layer to the San Francisco boundary.

```
var blocks = ee.FeatureCollection('TIGER/2010/Blocks');
var roads = ee.FeatureCollection('TIGER/2016/Roads');
var sfNeighborhoods = ee.FeatureCollection(
    'projects/gee-book/assets/F5-0/SFneighborhoods');

var geometry = sfNeighborhoods.geometry();
Map.centerObject(geometry);

// Filter blocks and roads to San Francisco boundary.
var sfBlocks = blocks.filter(ee.Filter.bounds(geometry));
var sfRoads = roads.filter(ee.Filter.bounds(geometry));
```

As we want to select all blocks within 1 km of an interstate highway, we first filter the `sfRoads` collection to select all segments with the `rttyp` property value of `I`.

```
var interstateRoads = sfRoads.filter(ee.Filter.eq('rttyp',
'I'));
```

We use the `draw` function to visualize the `sfBlocks` and `interstateRoads` layers (Fig. 25.5).

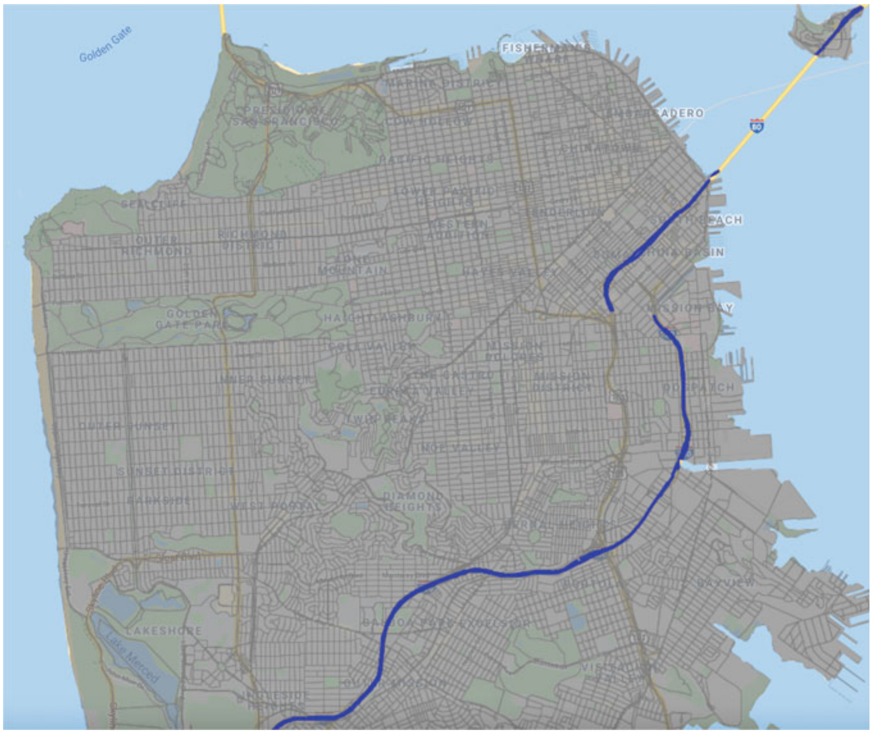


Fig. 25.5 San Francisco blocks and interstate highways

```
var sfBlocksDrawn = sfBlocks.draw({
        color: 'gray',
        strokeWidth: 1
    })
    .clip(geometry);
Map.addLayer(sfBlocksDrawn, {}, 'All Blocks');
var interstateRoadsDrawn = interstateRoads.draw({
        color: 'blue',
        strokeWidth: 3
    })
    .clip(geometry);
Map.addLayer(interstateRoadsDrawn, {}, 'Interstate Roads');
```

Let’s define a join that will select all the features from the `sfBlocks` layer that are within 1 km of any feature from the `interstateRoads` layer. We start by defining a filter using the `ee.Filter.withinDistance` filter. We want to compare the geometries of features in both layers, so we use a special property called `'.geo'` to compare the collections. By default, the filter will work with exact distances between the geometries. If your analysis does not require a very precise tolerance of spatial uncertainty, specifying a small non-zero `maxError` distance value will help speed up the spatial operations. A larger tolerance also helps when testing or debugging code so you can get the result quickly instead of waiting longer for a more precise output.

```
var joinFilter = ee.Filter.withinDistance({
    distance: 1000,
    leftField: '.geo',
    rightField: '.geo',
    maxError: 10
});
```

We will use a *simple join* as we just want features from the first (primary) collection that match the features from the other (secondary) collection.

```
var closeBlocks = ee.Join.simple().apply({
    primary: sfBlocks,
    secondary: interstateRoads,
    condition: joinFilter
});
```

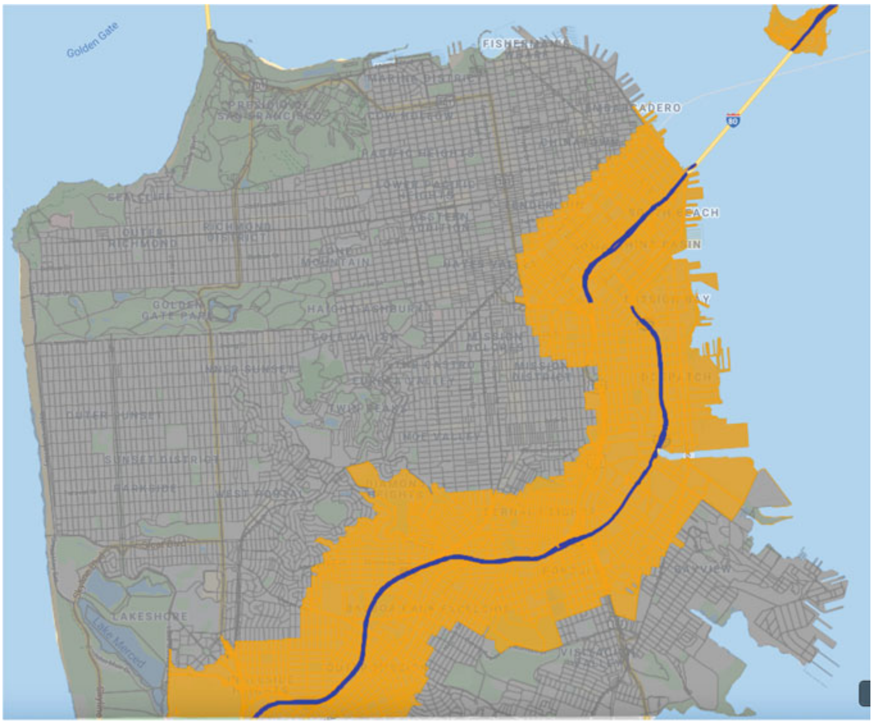


Fig. 25.6 Selected blocks within 1 km of an interstate highway

We can visualize the results in a different color and verify that the join worked as expected (Fig. 25.6).

```
var closeBlocksDrawn = closeBlocks.draw({
        color: 'orange',
        strokeWidth: 1
    })
    .clip(geometry);
Map.addLayer(closeBlocksDrawn, {}, 'Blocks within 1km');
```

25.1.2.2 Section 2.2: Spatial Joins

A *spatial join* allows you to query two collections based on the spatial relationship. We will now implement a spatial join to count points in polygons. We will work with a dataset of tree locations in San Francisco and polygons of neighborhoods to produce a CSV file with the total number of trees in each neighborhood.

The San Francisco Open Data Portal maintains a street tree map dataset that has a list of street trees with their latitude and longitude. We will also use the San Francisco neighborhood dataset from the same portal. We downloaded, processed, and uploaded these layers as Earth Engine assets for use in this exercise. We start by loading both layers and using the `paint` and `style` functions, covered in Sect. 25.1.1, to visualize them (Fig. 25.7).

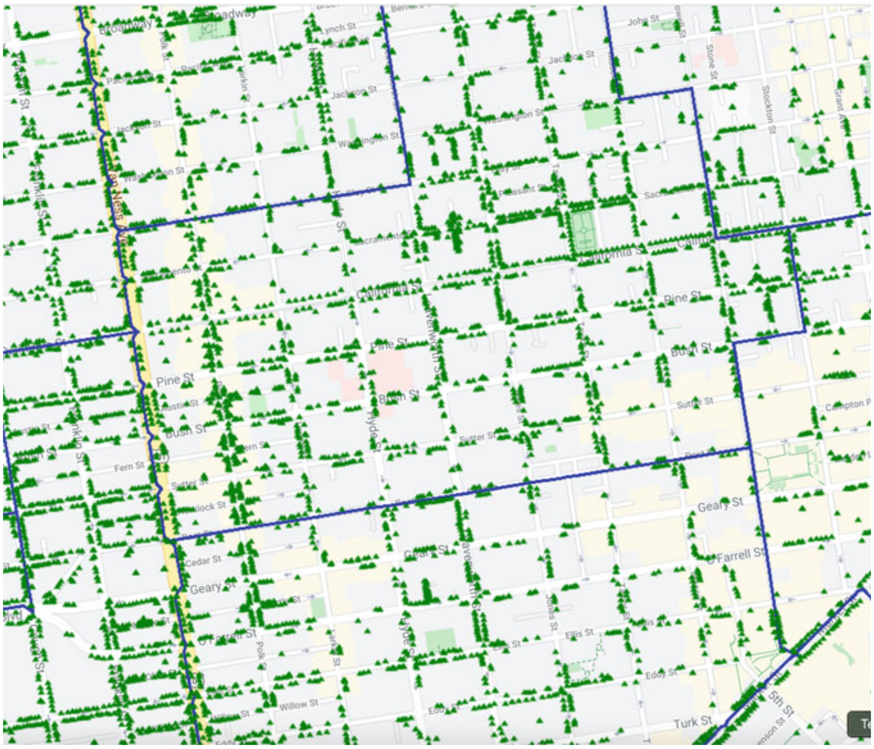


Fig. 25.7 San Francisco neighborhoods and trees

```
var sfNeighborhoods = ee.FeatureCollection(
    'projects/gee-book/assets/F5-0/SFneighborhoods');
var sfTrees = ee.FeatureCollection(
    'projects/gee-book/assets/F5-3/SFTrees');

// Use paint() to visualize the polygons with only outline
var sfNeighborhoodsOutline = ee.Image().byte().paint({
    featureCollection: sfNeighborhoods,
    color: 1,
    width: 3
});
Map.addLayer(sfNeighborhoodsOutline, {
        palette: ['blue']
    },
    'SF Neighborhoods');

// Use style() to visualize the points
var sfTreesStyled = sfTrees.style({
    color: 'green',
    pointSize: 2,
    pointShape: 'triangle',
    width: 2
});
Map.addLayer(sfTreesStyled, {}, 'SF Trees');
```

To find the tree points in each neighborhood polygon, we will use an `ee.Filter.intersects` filter.

```
var intersectFilter = ee.Filter.intersects({
    leftField: '.geo',
    rightField: '.geo',
    maxError: 10
});
```

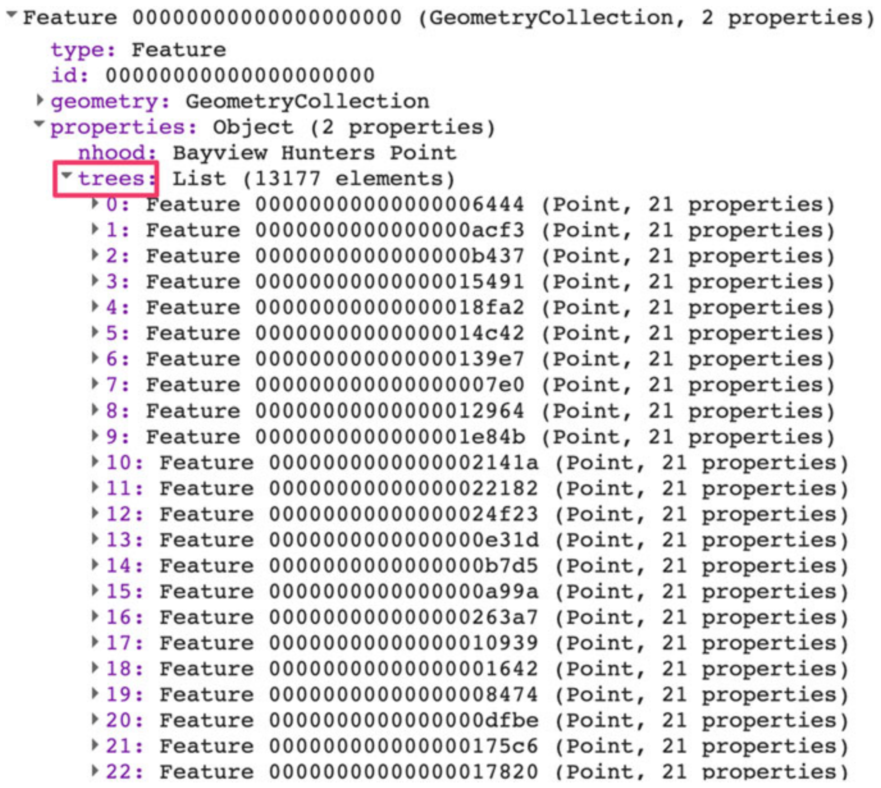
We need a join that can give us a list of all tree features that intersect each neighborhood polygon, so we need to use a *saving join*. A saving join will find all the features from the secondary collection that match the filter and store them in a property in the primary collection. Once you apply this join, you will get a version of the primary collection with an additional property that has the matching features from the secondary collection. Here we use the `ee.Join.saveAll` join, since we want to store all matching features. We specify the `matchesKey` property that will be added to each feature with the results.

```
var saveAllJoin = ee.Join.saveAll({
    matchesKey: 'trees',
});
```

Let's apply the join and print the first feature of the resulting collection to verify (Fig. 25.8).

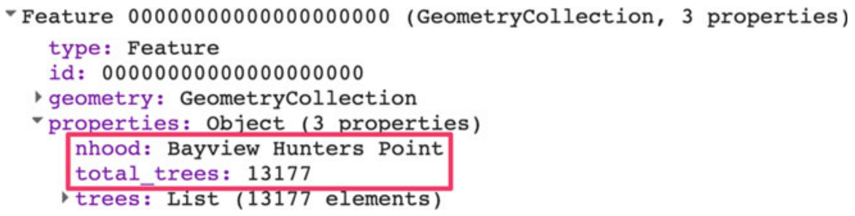
```
var joined = saveAllJoin
    .apply(sfNeighborhoods, sfTrees, intersectFilter);
print(joined.first());
```

You will see that each feature of the `sfNeighborhoods` collection now has an additional property called `trees`. This contains all the features from the `sfTrees` collection that were matched using the `intersectFilter`. We can



```
▾Feature 00000000000000000000 (GeometryCollection, 2 properties)
    type: Feature
    id: 00000000000000000000
   ▸geometry: GeometryCollection
   ▾properties: Object (2 properties)
      nhood: Bayview Hunters Point
     ▾trees: List (13177 elements)
        ▸0: Feature 0000000000000006444 (Point, 21 properties)
        ▸1: Feature 000000000000000acf3 (Point, 21 properties)
        ▸2: Feature 000000000000000b437 (Point, 21 properties)
        ▸3: Feature 0000000000000015491 (Point, 21 properties)
        ▸4: Feature 00000000000000018fa2 (Point, 21 properties)
        ▸5: Feature 00000000000000014c42 (Point, 21 properties)
        ▸6: Feature 000000000000000139e7 (Point, 21 properties)
        ▸7: Feature 000000000000000007e0 (Point, 21 properties)
        ▸8: Feature 0000000000000012964 (Point, 21 properties)
        ▸9: Feature 000000000000001e84b (Point, 21 properties)
        ▸10: Feature 000000000000002141a (Point, 21 properties)
        ▸11: Feature 0000000000000022182 (Point, 21 properties)
        ▸12: Feature 0000000000000024f23 (Point, 21 properties)
        ▸13: Feature 000000000000000e31d (Point, 21 properties)
        ▸14: Feature 000000000000000b7d5 (Point, 21 properties)
        ▸15: Feature 000000000000000a99a (Point, 21 properties)
        ▸16: Feature 00000000000000263a7 (Point, 21 properties)
        ▸17: Feature 0000000000000010939 (Point, 21 properties)
        ▸18: Feature 0000000000000001642 (Point, 21 properties)
        ▸19: Feature 0000000000000008474 (Point, 21 properties)
        ▸20: Feature 000000000000000dfbe (Point, 21 properties)
        ▸21: Feature 00000000000000175c6 (Point, 21 properties)
        ▸22: Feature 0000000000000017820 (Point, 21 properties)
```

Fig. 25.8 Result of the save-all join



```
▾Feature 00000000000000000000 (GeometryCollection, 3 properties)
    type: Feature
    id: 00000000000000000000
   ▸geometry: GeometryCollection
   ▾properties: Object (3 properties)
      nhood: Bayview Hunters Point
      total_trees: 13177
     ▸trees: List (13177 elements)
```

Fig. 25.9 Final `FeatureCollection` with the new property

now `map` a function over the results and post-process the collection. As our analysis requires the computation of the total number of trees in each neighborhood, we extract the matching features and use the `size` function to get the count (Fig. 25.9).

```
// Calculate total number of trees within each feature.
var sfNeighborhoods = joined.map(function(f) {
    var treesWithin = ee.List(f.get('trees'));
    var totalTrees =
ee.FeatureCollection(treesWithin).size();
    return f.set('total_trees', totalTrees);
});

print(sfNeighborhoods.first());
```

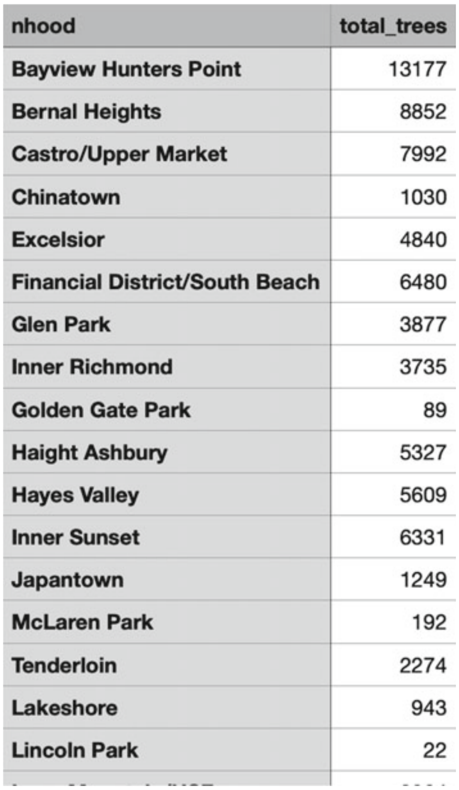
The results now have a property called `total_trees` containing the count of intersecting trees in each neighborhood polygon.

The final step in the analysis is to export the results as a CSV file using the `Export.table.toDrive` function. Note that as described in detail in Chap. 29, you should output only the columns you need to the CSV file. Suppose we do not need all the properties to appear in the output; imagine that we do not need the `trees` property, for example, in the output. In that case, we can create only those columns we want in the manner below, by specifying the other `selectors` parameters with the list of properties to export.

```
// Export the results as a CSV.
Export.table.toDrive({
    collection: sfNeighborhoods,
    description: 'SF_Neighborhood_Tree_Count',
    folder: 'earthengine',
    fileNamePrefix: 'tree_count',
    fileFormat: 'CSV',
    selectors: ['nhood', 'total_trees']
});
```

The final result is a CSV file with the neighborhood names and total numbers of trees counted using the join (Fig. 25.10).

Code Checkpoint F53b. The book's repository contains a script that shows what your code should look like at this point.



nhood	total_trees
Bayview Hunters Point	13177
Bernal Heights	8852
Castro/Upper Market	7992
Chinatown	1030
Excelsior	4840
Financial District/South Beach	6480
Glen Park	3877
Inner Richmond	3735
Golden Gate Park	89
Haight Ashbury	5327
Hayes Valley	5609
Inner Sunset	6331
Japantown	1249
McLaren Park	192
Tenderloin	2274
Lakeshore	943
Lincoln Park	22
[illegible]	[illegible]

Fig. 25.10 Exported CSV file with tree counts for San Francisco neighborhoods

25.2 Synthesis

Assignment 1. What join would you use if you wanted to know which neighborhood each tree belongs to? Modify the code above to do a join and post-process the result to add a neighborhood property to each tree point. Export the results as a shapefile.

25.3 Conclusion

This chapter covered visualization and analysis using vector data in Earth Engine. You should now understand different functions for `FeatureCollection` visualization and be able to create thematic maps with vector layers. You also learned techniques for doing spatial queries and spatial joins within Earth Engine. Earth Engine is capable of handling large feature collections and can be effectively used for many spatial analysis tasks.

GEEDiT—Digitizing from Satellite Imagery

26

James Lea

Overview

Earth surface margins and features are often of key interest to environmental scientists. A coastline, the terminus of a glacier, the outline of a landform, and many other examples can help illustrate past, present, and potential future environmental change. The information gained from these features can be used to achieve greater understanding of the underlying processes that are controlling these systems, to monitor their responses to ongoing environmental changes, and to assess and inform wider socio-economic impacts at local to global scales.

While it is common practice in remote sensing to automate identification of margins from imagery, these attempts are not always successful or transferable between seasons or locations. Furthermore, in some circumstances, the nature of the margins that need to be identified means that implementing automated approaches are not always necessary, desirable, or even possible. In such cases, user-led manual digitization of margins may be the only appropriate way to generate accurate, user-verified data. However, users who wish to undertake this analysis at scale often face the time-consuming challenge of having to download and process large volumes of satellite imagery. Furthermore, issues such as internet connection speed, local storage capacity, and computational processing power availability can potentially limit or even preclude this style of analysis, leading to a fundamental inequality in the ability of users to generate datasets of comparable standards.

The Google Earth Engine Digitisation Tool (GEEDiT) is built to allow any researcher to rapidly access satellite imagery and directly digitize margins as lines or polygons. With a simple user interface and no need to download enormous images, GEEDiT leverages Earth Engine's cloud computing power and its access to satellite

J. Lea (✉)
Department of Geography and Planning, School of Environmental Sciences, University of Liverpool, Liverpool, UK
e-mail: j.lea@liverpool.ac.uk

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_26

image repositories (Lea 2018). When the delineated vector margins are exported, they are also appended with standardized metadata from the source imagery. In doing so, users can easily trace back to which images associate with which digitized margins, facilitating reproducibility, independent verification, and building of easily shareable and transferable datasets (e.g., Goliber et al. 2022). Since GEEDiT is used through a graphical user interface, it requires no coding ability in Earth Engine to use it.

GEEDiT has been used for research in glaciology (e.g., Tuckett et al. 2019; Fahrner et al. 2021), hydrology (e.g., Boothroyd et al. 2021; Scheingross et al. 2021), and lake extent (e.g., Kochtitzky et al. 2020; Field et al. 2021) in addition to other fields. Version 2 of GEEDiT was released in early 2022 with improved functionality, usability, and access to new datasets.

Learning Outcomes

- Understanding how to use the GEEDiT v2 interface and export data from it.
- Becoming aware of some of the new functionality introduced in GEEDiT v2.
- Learning about potential options to customize GEEDiT v2 visualization parameters.

Assumes you know how to

- Use drawing tools to create points, lines, and polygons (Chap. 6).
- Recognize similarities and differences among satellite spectral bands (Part I, Part II, Part III).

26.1 Introduction to Theory

As a tool, GEEDiT came about due to the author's frustration with the requirement to download, process, and visualize significant volumes of satellite imagery for simple margin analysis, in addition to the tedious steps required for appending comprehensive and consistent metadata to them. GEEDiT was built to take advantage of Earth Engine's capability for rapid cloud-based data access, processing and visualization, and potential for fully automated assignment of metadata.

GEEDiT allows users to visualize, review, and analyze an image in potentially a few seconds, whereas previously downloading and processing a single image would have taken considerably longer in a desktop GIS environment. Consequently, GEEDiT users can obtain margin data from significantly more imagery than was previously possible in the time available to them. Users can create larger, more spatially and temporally comprehensive datasets compared to traditional approaches, with automatically appended metadata also allowing easier dataset filtering, and tracing of data back to original source imagery. Where coding-based analysis of GEEDiT output is applied, the standardization of metadata also facilitates the transferability of code between datasets, and assists in the collation of homogenized margin and feature datasets that have been generated by multiple users (e.g., Goliber et al. 2022).

GEEDiT version 2 (hereafter GEEDiT) was released in January 2022 to provide improvements to the user experience of the tool, and the imagery data that it can access. Similar to version 1, it is a graphical user interface (GUI) built entirely within Earth Engine's JavaScript API. This chapter provides a walkthrough of GEEDiT's user interface, alongside information on how users can get the most out of its functionality.

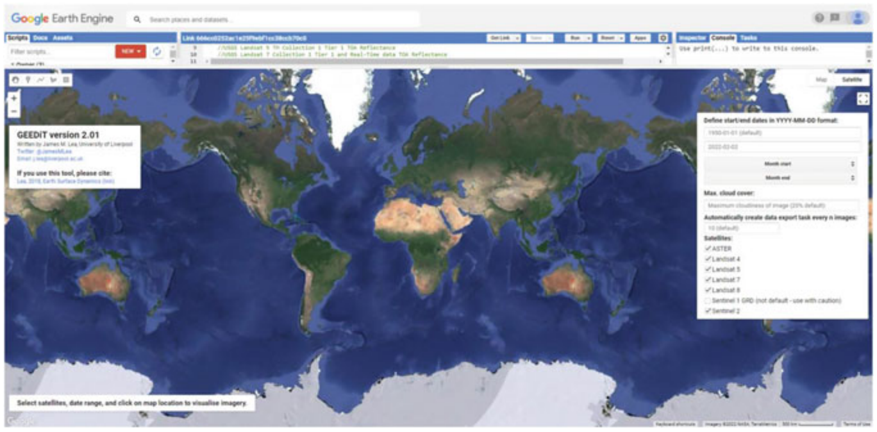
26.2 Practicum

Code Checkpoint F54a. The book's repository contains information about accessing the interface for GEEDiT.

Below, each step required to extract data from GEEDiT is described, using an example of digitizing the margin of a Greenlandic marine terminating glacier.

26.2.1 Section 1: Loading GEEDiT and Selecting Imagery Options and a Location

Click on the link to GEEDiT v2.02 Tier 1, to open the GUI in a new tab as shown in Fig. 26.1. If you are interested in the underlying code, you can view this in the Code Editor. Resize the view to maximize the map portion of the Earth Engine interface. If the interface does not appear, click the **Run** button above the Code Editor at the top of the page.



```
//USGS Landsat 5 TM Collection 1 Tier 1 TOA Reflectance
//USGS Landsat 7 Collection 1 Tier 1 and Real-Time data TOA Reflectance
```

```
Use print(...) to write to this console.
```

GEEDiT version 2.01
Written by James M. Lea, University of Liverpool
Twitter: @JamesMLea
Email: j.lea@liverpool.ac.uk

If you use this tool, please cite:
Lea, 2018, Earth Surface Dynamics (link)

Define start/end dates in YYYY-MM-DD format:
1950-01-01 (default)
2022-02-02
Month start
Month end
Max. cloud cover:
Maximum cloudiness of image (20% default)
Automatically create data export task every n images:
10 (default)
Satellites:
✓ ASTER
✓ Landsat 4
✓ Landsat 5
✓ Landsat 7
✓ Landsat 8
☐ Sentinel 1 GRD (not default - use with caution)
✓ Sentinel 2

Select satellites, date range, and click on map location to visualise imagery.

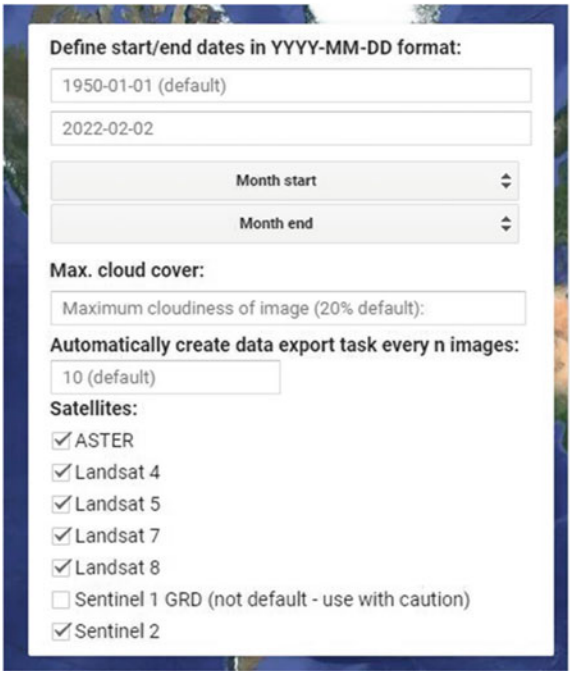
Fig. 26.1 Starting screen of GEEDiT

26.2.1.1 Section 1.1: Description of Starting Page Interface

Brief instructions on how to use the GUI are given at the bottom left of the interface, while image selection options are provided on the right (Fig. 26.2).

In this panel, you can define the date range, month range, maximum allowable cloud cover, export task frequency, and satellite platforms for the images you want to analyze. Each of these is explained in turn below:

- **Define start/end dates in YYYY-MM-DD format**: These fields provide text boxes for you to define the date range of interest. The default start date is given as 1950-01-01, and the end date is the date the tool is being run. In this example, we use the default settings, so you do not need to make any changes to the default values. However, common issues that arise using this functionality include:
  - Not entering a date that is compatible with the YYYY-MM-DD format, including: placing days, months, and years in the wrong order (e.g., 02-22-2022); and not using '-' as a separator (e.g., 2022/22/02).
  - Entering a date that cannot exist (e.g., 2022-09-31).
  - Not including a leading 0 for single digit days or months (e.g., 2022-2-2).
  - Not entering the full four-digit year (e.g., 22-02-02).
  - Typographical errors.


Define start/end dates in YYYY-MM-DD format:

1950-01-01 (default)

2022-02-02

Max. cloud cover:

Maximum cloudiness of image (20% default):

Automatically create data export task every n images:

10 (default)

Satellites:

✓ ASTER
✓ Landsat 4
✓ Landsat 5
✓ Landsat 7
✓ Landsat 8
☐ Sentinel 1 GRD (not default - use with caution)
✓ Sentinel 2

Fig. 26.2 Options for which images to include for visualization

- **Month start and month end**: These are optional dropdown menus that allow users to obtain imagery only from given ranges of months within the previously defined date range. If users do not define these, imagery for the full year will be returned. In some cases (e.g., analysis of imagery during the austral summer), users may wish to obtain imagery from a month range that may span the year end (e.g., November–February). For this month range, this can be achieved by selecting a start month of 11, and a month end of 2. Given that glacier margins are often clearest and easiest to identify during the summer months, set the start month to July (7) and end month to September (9).
- **Max. cloud cover**: This field filters imagery by cloud cover metadata values that are appended to imagery obtained from optical satellite platforms, such as Landsat, Sentinel-2, and ASTER satellites. The names of the metadata that hold this information vary between these satellite platforms and are 'CLOUD_COVER', 'CLOUDY_PIXEL_PERCENTAGE', and 'CLOUDCOVER' respectively. Values associated with these image metadata represent automatically derived estimates of the percentage cloud cover of an entire image scene. These are calculated in different ways for the different satellites depending on the processing chain performed by NASA/ESA when the imagery was acquired.

  Also, note that Sentinel-2 and ASTER imagery footprints are smaller than those of Landsat, meaning that 1% cloud cover for a Landsat image will cover a larger geographic area than 1% of a Sentinel-2 or ASTER image. Here we use the default setting (maximum 20% cloud cover), so you can leave this field unchanged.
- **Automatically create data export task every *n* images**: This will generate a data export task in Earth Engine for every *n* images viewed, where *n* is the value entered. The reason for including this as an option is that manual digitization features from imagery can be time-consuming depending on the level of detail required or number of features on an image. While rare, data losses can result from internet connection dropout, computer crashes, or the Earth Engine server going offline, so exporting digitized data frequently can help guard against this.
- **Satellites**: This menu provides a list of checkboxes of all the satellite platforms from which imagery is available (Table 26.1). Descriptions of aspects of each satellite that users should be aware of when using these data for digitizing margins are described below.
  - **ASTER L1T Radiance**: The Advanced Spaceborne Thermal Emission and Reflection Radiometer (ASTER) collects both daytime and nighttime imagery. (Some images may appear black.) The lack of a blue spectral band on ASTER means that it is not possible to visualize imagery as "true color" (i.e., red–green–blue), though it does allow imagery to be visualized at up to 15 m pixel resolution. The default color channel visualization for ASTER within GEEDiT is: B3N, near infrared in the red channel; B02, visible red in the green channel; B01, visible green/yellow in the blue channel.
  - **Landsat 4 Top of Atmosphere (TOA)**: The default image collection for Landsat 4 and all Landsat satellites listed below are taken from the TOA

Table 26.1 Key information for each satellite

Satellite	No. of bands	True color?	Optical/SAR	Date range	Resolution (dependent on bands used)
ASTER	14	No	Optical	03/00–present	Up to 15 m
Landsat 4	7	Yes	Optical	08/82–11/93	Up to 30 m
Landsat 5	7	Yes	Optical	03/84–05/12	Up to 30 m
Landsat 7	9	Yes	Optical	05/99–present	Up to 15 m
Landsat 8	11	Yes	Optical	03/13–present	Up to 15 m
Sentinel-1	Up to 4	No	SAR	10/14–present	Up to 10 m
Sentinel-2	14	Yes	Optical	06/15–present	Up to 10 m

collections where the imagery has been calibrated for top of atmosphere reflectance (see Chander et al. 2009). The density of global coverage for Landsat 4 is variable, meaning that some regions may not possess any available images.

– **Landsat 5 TOA**: Similar to Landsat 4, Landsat 5 has variable global coverage, meaning that some remote regions may have very few or no images.
– **Landsat 7 Real-Time Data, TOA Reflectance**: Beginning in 2003, a failure in the satellite's scan-line corrector (SLC) mirror has resulted in data gaps on Landsat 7 imagery. The impact of this instrument failure is that stripes of missing data of increasing width emanate from near the center of the image scene, which can lead to challenges in continuous mapping of features or margins. The orbit of Landsat has also been drifting since 2017 meaning that the time of day when imagery is acquired for a given location has been getting earlier.
– **Landsat 8 Real-Time Data, TOA Reflectance**: Landsat 8 is automatically updated with new imagery as it becomes available. Note that the red, green, and blue bands are different for Landsat 8 compared to Landsat 4, 5, and 7 due to the addition of the coastal aerosol band as band 1.
– **Sentinel-1 SAR GRD**: Unlike other satellites that are listed as options in GEEDiT, Sentinel-1 is a synthetic aperture radar (SAR) satellite (i.e., it is an active source satellite operating in the radio frequency range and does not collect visible spectrum imagery). A notable advantage of SAR imagery is that it is unaffected by cloud cover (cloud is transparent at radio frequencies) or sun illumination, providing full scene coverage for each image acquisition. Sentinel-1 transmits and receives reflected radio waves in either the vertical (V) or horizontal (H) polarizations, providing up to four bands available in Sentinel-1 data: VV, VH, HH, and HV. In practice, a subset of these bands are usually acquired for each Sentinel-1 scene, with GEEDiT automatically visualizing the first available band. However, care needs to be taken when using this imagery for digitizing margins or features. This arises

from the satellite being "side-looking," which results in the location accuracy and ground resolution of imagery being variable, especially in areas of steep topography. The Sentinel-1 imagery available in Earth Engine is the ground range detected (GRD) product, meaning that a global digital elevation model (DEM) has been used to ortho-correct imagery. However, where there is a mismatch between the DEM used for ortho-correction and the true surface elevation (e.g., arising from glacier thinning), this may result in apparent shifts in topography between images that are non-negligible for analysis. This is especially the case where vectors are digitized from imagery obtained from both ascending and descending parts of its orbit. The magnitude of these apparent shifts will be dependent on the satellite look angle and mismatch between the DEM and true surface elevation, meaning that applying corrections without up-to-date DEMs is unfortunately non-trivial. For these reasons, it is suggested that where Sentinel-1 imagery is used to delineate margins or features that they are used for quantifying *relative* rather than absolute change, and significant care should be taken in the interpretation of margin and feature data generated from this imagery. As a result of these caveats, Sentinel-1 is not selected as a default option for visualization.

– **Sentinel-2 MSI: Level 1C**: The image collection included in GEEDiT is the Level 1C TOA data that provides global coverage. Also note that the ground control and ortho-correction for Sentinel-2 imagery are performed using a different DEM compared to Landsat satellites, meaning there may be small differences in geolocation consistency between imagery from these different platforms. Where margin or feature data from both these platform types are combined, users should, therefore, check the level of accuracy it is possible to derive, especially where fine scale differences between margins or features (e.g., a few pixels) are important (e.g., Brough et al. 2019).

26.2.1.2 Section 1.2: Visualizing Imagery at a Given Location

To visualize imagery once the desired options have been defined, zoom into your area of interest (in this case, southwest Greenland) and simply click on the map where imagery should be loaded. Once this is done, an "Imagery loading" panel will appear while GEEDiT collates all imagery for the clicked location that satisfies the image selection options. If lots of imagery is returned, then it may take 30 s to a minute for the list of images to be obtained. Once GEEDiT has loaded this, the vector digitization screen will appear, and the earliest image available from the list will be visualized automatically (Fig. 26.3).

26.2.2 Section 2: GEEDiT Digitisation Interface

26.2.2.1 Section 2.1: Digitizing and Editing Margins and Features

The GEEDiT digitization interface will load and the first image will appear, providing you with multiple options to interact with vectors and imagery. However,

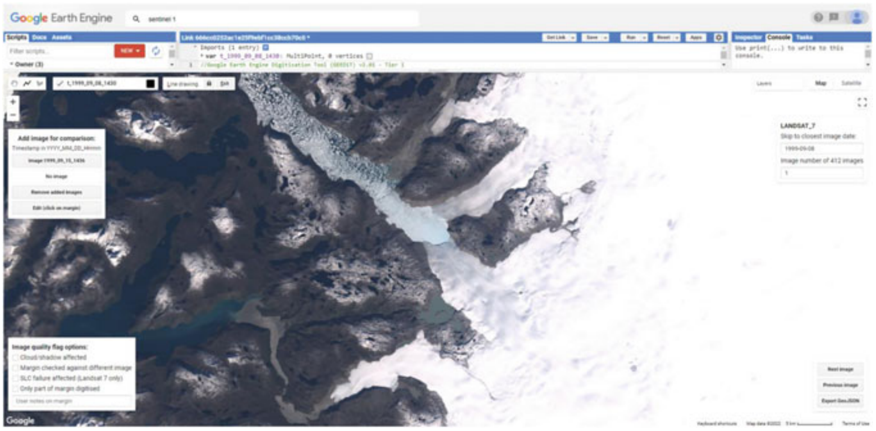


Fig. 26.3 GEEDiT vector digitization screen with first available image for a glacier in southwest Greenland visualized

if you wish to begin digitizing straight away, you can do so by clicking directly on the image, and double clicking to finish the final vertex of the feature or margin. By default, vectors will be digitized as a line. However, you can also draw polygons by selecting this option from the top left of the screen.

Once vectors have been digitized, you may wish to modify or delete vectors where required. To do this, either click the edit button on the top left panel before clicking on the digitized margin that is to be changed, or press the keyboard escape key and click to select the margin (Fig. 26.4). The selected vector will show all digitized vertices as solid white circles, and the midpoints between vertices as semi-transparent circles. Any of these can be dragged using the cursor to modify the vector. If you want to delete the entire vector, press your keyboard's Delete key.

In order to begin digitizing again, hover your cursor over the geometry layer at the top left of the screen, and click on the relevant layer so that it changes from standard to bold typeface (Fig. 26.5). Do not use the "+new layer" option.

You can then digitize multiple lines and/or polygons to individual images, and they will be saved in the geometry layer selected. The name of the geometry layer is given in the format *t_YYYY_MM_DD_HHmm*, also providing you with information as to the precise time and date that the source image was acquired.

26.2.2.2 Section 2.2: Moving Between Images

There are multiple options in GEEDiT for how you can move from the current image to the next. Options to move between images are contained in the two panels on the right-hand side of the screen (Fig. 26.6). Note that when a new image is loaded, the interface creates a new geometry layer for that image, and sets previous layers to be not visible on the map (though these can be easily visualized where

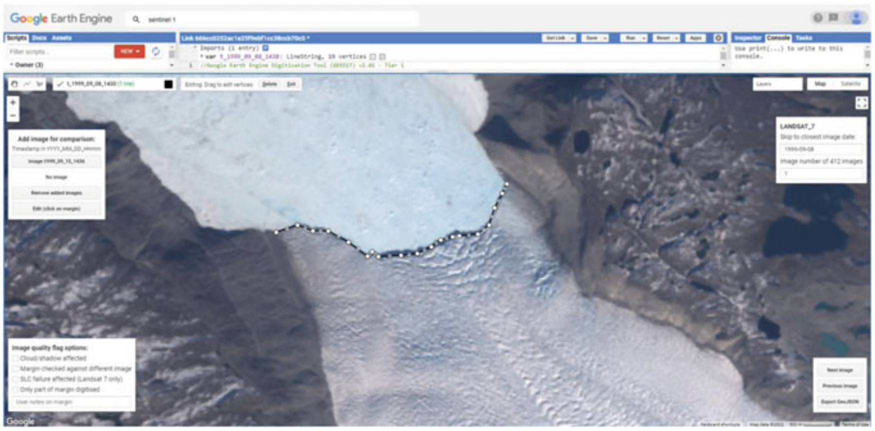

Google Earth Engine
Add image for comparison:
No image
Remove added images
LANDSAT_7
Skip to closest image date
1999-09-08
Image number of 412 images
Image quality flag options:
Cloud/shadow affected
Margin checked against different image
SLC failure affected (Landsat 7 only)
Only part of margin digitised
Next image
Previous image
Export GeoJSON

Fig. 26.4 Digitized vector along a glacier terminus that has been selected for editing

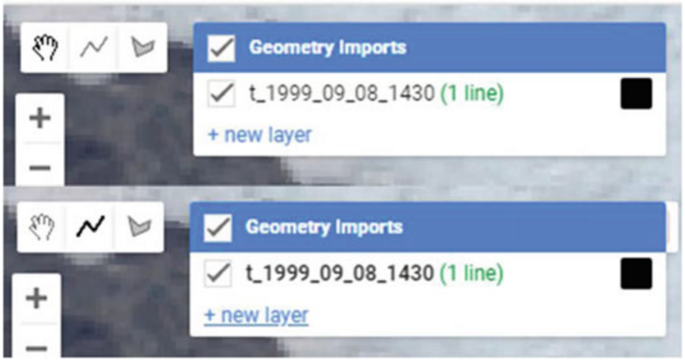


Fig. 26.5 To continue adding vectors to the image after editing, select geometry layer to add vectors to so that the layer name appears in bold typeface

comparison may be needed). The geometry layer related to the current image will be black, while vectors digitized from other imagery will be blue.

To move between images, the first option is to click **Next image**; if you do, the user interface will load image number 2 in the list. Similarly, the **Previous image** button will load the preceding image in the list where one is available. Try clicking **Next image** to bring up image number 2.

Other options to move between images are provided in the panel on the upper right of the screen (Fig. 26.6, right). This panel contains information on which satellite the image was acquired by, the date that the image was acquired, and the image number. The date and image number are contained within editable textboxes and provide two different ways for moving to a new image.

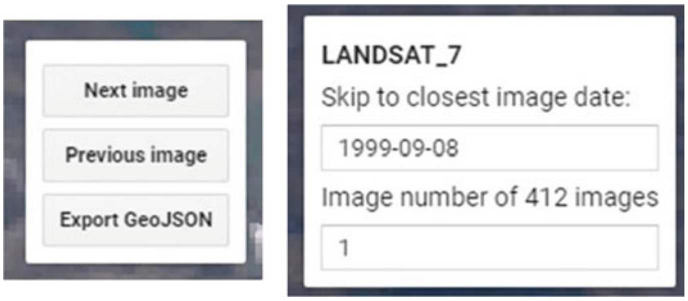


Fig. 26.6 Options for how to move between images are contained in the two panels on the right-hand side of the digitization interface

The first option is to edit the text box containing the date of the current image (in *YYYY-MM-DD* format) and allows users to skip to the image in the list that is closest in time to the date that they define. Once the new date has been entered, the user should press the keyboard enter key to trigger the image to load. GEEDiT will then visualize the image that is closest to the user defined date and update the value of the text box with the actual date of the image. The second option is to change the image number text box value, press enter, and then GEEDiT will load that image number from the list of images available. Try each of these options for yourself.

26.2.2.3 Section 2.3: Visualizing Previously Digitized Margins or Features on the Current Image

The geometries of each vector digitized from each image are retained as geometry layers in the digitization interface (Fig. 26.5). Consequently, you can re-visualize previously digitized vectors on the current image by toggling the checkboxes next to each geometry layer, providing a means to briefly check how or if a feature has changed between one date or another. In the user interface the margin relating to the current image will appear as black, while previously digitized vectors will appear as blue (Fig. 26.7). This functionality is not intended as a substitute for analysis, but a means for you to quickly check whether change has occurred, thus facilitating potential study site identification.

26.2.2.4 Section 2.4: Extra Functionality

GEEDiT provides the ability to compare between images, for interpreting subtle features in the current image. This functionality is contained in the top left panel of the digitization interface (Fig. 26.8).

The options in this panel allow users to easily add the images immediately preceding and following the current image. Try clicking on these buttons to add imagery to the map and use the **Remove added images** button to remove any imagery that has been added to leave only the original image visualized. To aid comparison between images, try using the image layer transparency slider bars

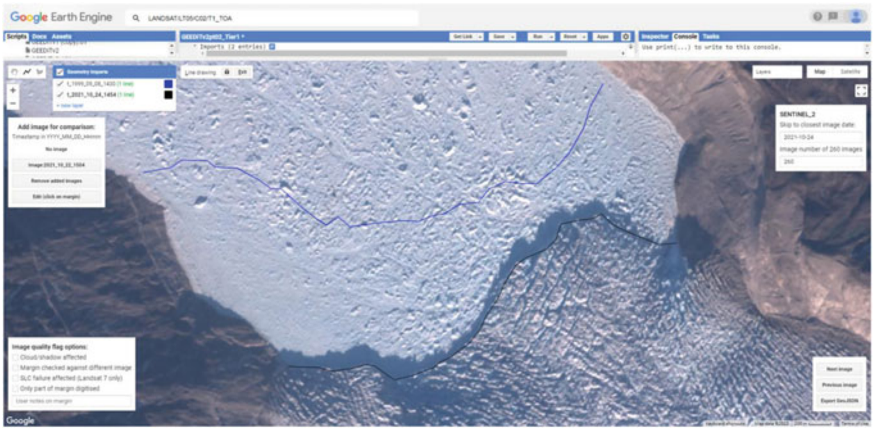
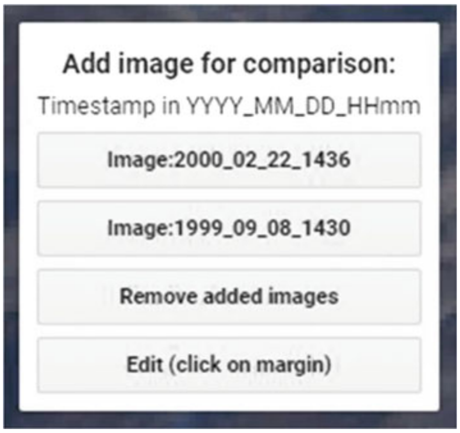


Fig. 26.7 Digitization interface showing a previously digitized margin (blue line, 1999-09-08) and the currently visualized image (black line, 2021-10-24)



Add image for comparison:

Timestamp in YYYY_MM_DD_HHmm

Image:2000_02_22_1436

Image:1999_09_08_1430

Remove added images

Edit (click on margin)

Fig. 26.8 Top left panel of digitization interface showing options to visualize subsequent and previous imagery

contained in the map **Layers** dropdown in the top right of the digitization interface (Fig. 26.9).

When using this option take care to ensure that vectors are digitized directly from the current image only (in the case above, 1999-09-08) to ensure data consistency.

26.2.2.5 Section 2.5: Image Quality Flags

For some margins and features, it may be necessary to note problems that have impacted the quality of vector digitization. To aid in this, GEEDiT includes several

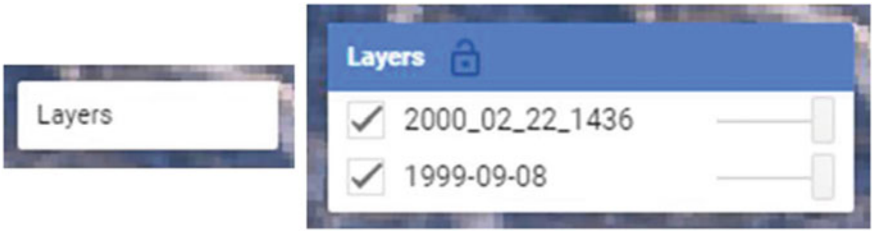


Fig. 26.9 The layers menu as it appears on the digitization interface (left) and how it appears when a user hovers the cursor over it (right). The slider bars can be used to toggle image transparency

quality flags for common issues. These are contained in the panel in the bottom left of the screen with information from these flags added to vector metadata upon export (Fig. 26.10).

These quality flags help you to subsequently filter any vectors impacted by imagery issues that may affect the quality or accuracy of the vector digitization. These options are:

- **Cloud/shadow affected**: The user may wish to flag a vector as having been negatively impacted by cloud or shadow.
- **Margin checked against different image**: This flag is automatically checked when you add an image for comparison (see above and Fig. 26.8).
- **SLC failure affected**: This flag is automatically checked if margins from an image that is impacted by the Landsat 7 SLC failure are used (see Sect. 26.2.1, Landsat 7 description). You may wish to uncheck this box if vectors that are digitized from SLC failure affected imagery fall within areas that are not impacted by image data gaps.
- **Only part of margin digitized**: When it is only possible for you to digitize part of the full feature of interest, then this flag should be checked.
- **Text box for user notes**: This text box provides the opportunity for you to provide brief notes (up to 256 characters) on the vectors digitized. These notes will be preserved in the exported data.

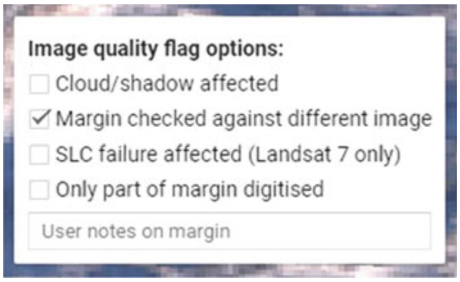


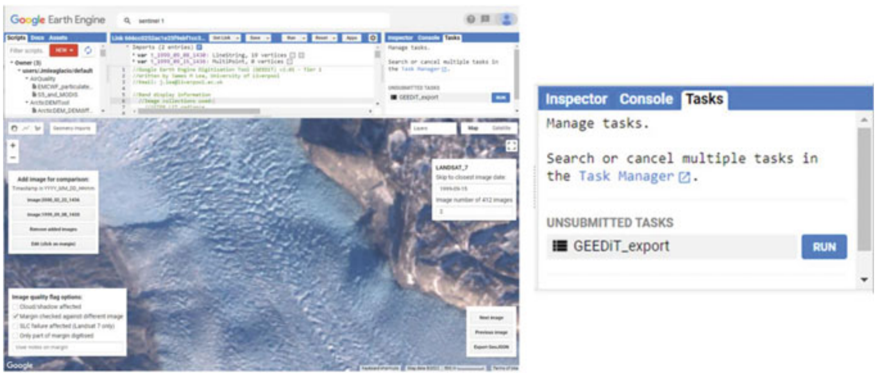
Image quality flag options:
Cloud/shadow affected
✓ Margin checked against different image
SLC failure affected (Landsat 7 only)
Only part of margin digitised
User notes on margin

Fig. 26.10 Image quality flag options in lower left panel of digitization interface

26.2.2.6 Section 2.6: Exporting Data

Once you have digitized all the vectors that you wish to, or you have reviewed enough imagery to automatically trigger an export task (see Sect. 26.2.1 and Fig. 26.2), you can export your digitized vectors to your Google Drive or to Earth Engine as an asset. To manually trigger an export task, try clicking the **Export GeoJSON** button in the bottom right panel shown on the screen (Fig. 26.6). It is important to note that just this step alone will not result in data being exported. To execute the export task, go to the **Tasks** tab next to the Earth Engine Code Editor. To access this, you may need to resize the map from the top of the screen to make this visible (Fig. 26.11).

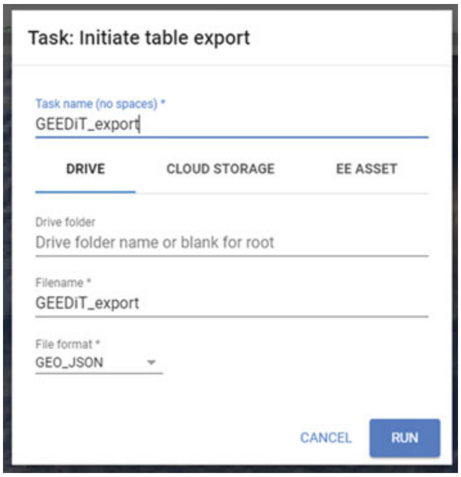
To trigger the export, click the blue **Run** button in the **Tasks** tab. This will bring up the Earth Engine data export menu (Fig. 26.12). In this menu, the user has the option to define the task name (note that this is not how the export will be saved, but how the name of the export task will appear in the **Tasks** tab), the folder in Google Drive where the exported file will be saved, the name of the exported file, and the format of the exported file. As default, the GeoJSON file format is selected, due to its capability of retaining both line and polygon vectors within the same file. If users have digitized only lines or only polygons, it is possible to export these as shapefiles. However, if a user attempts to export a combination of lines and polygons as a single shapefile, this will fail. GeoJSON files can be easily imported into GIS platforms, such as QGIS or ArcGIS, while their consistent file structure also assists coding-based analysis.



```
Manage tasks.

Search or cancel multiple tasks in
the Task Manager.
```

GEEDiT_export

Fig. 26.11 Resizing the map to show the Code Editor (left), and the Tasks tab (right) that can be found to the right of the Code Editor panel


Task: Initiate table export

Task name (no spaces) *
GEEDiT_export

Drive folder
Drive folder name or blank for root

Filename *
GEEDiT_export

File format *
GEO_JSON

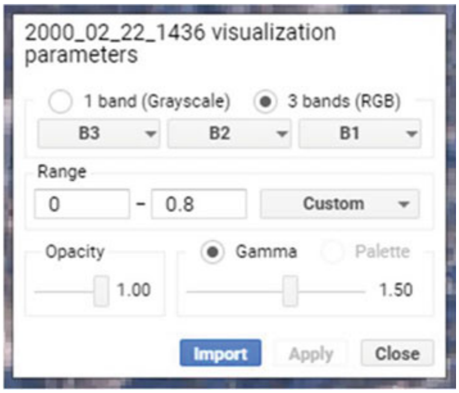
Fig. 26.12 Earth Engine data export menu

26.2.3 Section 3: Making GEEDiT Fit Your Own Purposes (Advanced)

26.2.3.1 Section 3.1: Changing Default Image Band Combinations and Visualization Ranges

GEEDiT has been designed to visualize, where possible, true color satellite imagery at the highest possible spatial resolution. However, for some applications, margins or features may be identified more easily using different satellite image band combinations or minimum/maximum image data ranges. Where this is necessary, there are two options available to you:

1. After an image has been visualized, try manually editing the image bands and data range by hovering over the map **Layers** menu and the image to be edited (Fig. 26.9), before clicking the Gear icon that appears next to the image name. This will bring up image visualization options that allow editing of the image band combination, in addition to a variety of options to define the minimum/maximum data values. Try exploring some of the options that this provides, and to implement the changes, click the **Apply** button to re-visualize the image using the new band and minimum/maximum visualization options (Fig. 26.13). However, note that this will not change the default visualization options in GEEDiT, and if you go to the next image, the default visualization options will be restored (see below).
2. Changing GEEDiT's default image band combinations and/or minimum/maximum values requires direct editing GEEDiT's code within the GEE code editor and making changes to the *imVisParams* variable (Fig. 26.14; L420-428 in GEEDiT v2.02). This will allow you to change the default band combinations and other image visualization options for each satellite. Once changes have been made, click the **Run** button above the Code Editor for the


2000_02_22_1436 visualization parameters

1 band (Grayscale) [✓ 3 bands (RGB)]

B3 B2 B1

Range 0 - 0.8 Custom

Opacity 1.00

[✓ Gamma] Palette 1.50

Fig. 26.13 Menu for editing image visualization properties

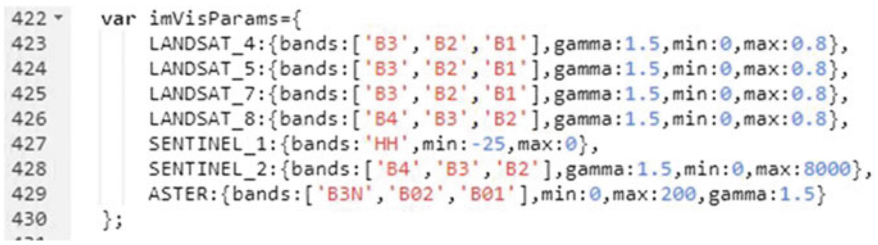

```
var imVisParams={
    LANDSAT_4:{bands:['B3','B2','B1'],gamma:1.5,min:0,max:0.8},
    LANDSAT_5:{bands:['B3','B2','B1'],gamma:1.5,min:0,max:0.8},
    LANDSAT_7:{bands:['B3','B2','B1'],gamma:1.5,min:0,max:0.8},
    LANDSAT_8:{bands:['B4','B3','B2'],gamma:1.5,min:0,max:0.8},
    SENTINEL_1:{bands:'HH',min:-25,max:0},
    SENTINEL_2:{bands:['B4','B3','B2'],gamma:1.5,min:0,max:8000},
    ASTER:{bands:['B3N','B02','B01'],min:0,max:200,gamma:1.5}
};
```

Fig. 26.14 Default image visualization parameters for GEEDiT

changes to take effect. If you are unfamiliar with Earth Engine or JavaScript, any changes in GEEDiT's code should be undertaken with care, given that any small formatting errors could result in the tool failing to run or resulting in unexpected behavior. It is suggested that users take care not to delete any brackets, colons, commas, or inverted commas when editing these visualization options. Note that while Sentinel-1 has the HH band selected as default, in practice this is not frequently used, as some Sentinel-1 images do not contain this band.

26.2.3.2 Section 3.2: Filtering Image Lists by Other Image Metadata

The GEEDiT user interface is intended to provide a simple means by which margins and features can be digitized from satellite imagery. It is purposely not designed to provide comprehensive image filtering options so as to retain an intuitive, straightforward user experience. However, it is possible to make additions to GEEDiT's code that allow users to filter image collections by other image metadata.

For example, in order to find imagery that highlights margins or features using shadows, users may wish to only digitize from optical imagery where the sun angle at the time of image acquisition does not exceed a given value. This can be achieved by adding lines that will filter imagery by metadata values to L326-387 (GEEDiT v2.02). If you wish to do this, then care should be taken to ensure that the correct image metadata names are used, and also note that these metadata names can vary between satellite platforms. More information on the metadata available for each satellite platform can be found in the relevant links in Sect. 26.2.1 where the satellites are described. For more information on how to filter image collections, see Chap. 13.

26.2.3.3 Section 3.3: Changing Image Collections Used in GEEDiT

For optical satellites, GEEDiT uses real-time TOA image collections for Landsat and Sentinel-2, as these provide the highest numbers of images (see Sect. 26.2.1). However, in some circumstances, it may be more appropriate to use different image collections that are available in Earth Engine (e.g., surface reflectance imagery that has been calibrated for atmospheric conditions at the time of acquisition). If this is the case, you can change GEEDiT's code to access this imagery as default by updating the image collection paths used to access satellite data (contained between L326-387 in GEEDiT v2.02).

The different image collections available within Earth Engine can be explored using the search bar at the top of the screen above the user interface, and users should read information relating to each image collection carefully to ensure that the data will be appropriate for the task that they wish to undertake. Users should also replace paths in a "like for like" fashion (i.e., Landsat 8 TOA replaced with Landsat 8 SR), as otherwise this will result in incorrect metadata being appended to digitized vectors when they are exported. To do this, try replacing the Landsat 8 TOA collection with the path name for a Landsat 8 Surface Reflectance image collection that can be found using the search bar. Once you have made the changes, click **Run** for them to take effect.

26.2.3.4 Section 3.4: Saving Changes to GEEDiT Defaults for Future Use

If you have edited GEEDiT to implement your own defaults, then you can save this for future use rather than needing to change these settings each time you open GEEDiT. This can be easily done by clicking the **Save** button at the top of the Code Editor, allowing you to access your edited GEEDiT version via the **Scripts** tab to the left of the Code Editor. For more information on the Earth Engine interface, see Chap. 1.

26.3 Synthesis

You should now be familiar with the GEEDiT user interface, its functionality, and options to modify default visualization parameters of the tool.

Assignment 1. Visualize imagery from your own area of interest and digitize multiple features or margins using polygons and lines.

Assignment 2. Export these data and visualize them in an offline environment (e.g., QGIS, ArcMap, Matplotlib), and view the metadata associated with each vector. Think how these metadata may prove useful when post-processing your data.

Assignment 3. Try modifying the GEEDiT code to change the tool's default visualization parameters for Landsat 8 imagery to highlight the features that you are interested in digitizing (e.g., band combinations and value ranges to highlight water, vegetation, etc.).

Assignment 4. Using information about the wavelengths detected by different bands for other optical satellites, change the default visualization parameters for Landsat 4, 5, and 7, and Sentinel-2 so that they are consistent with the modified Landsat 8 visualization parameters.

26.4 Conclusion

This chapter has covered background information to the Earth surface margin and feature digitization tool GEEDiT. It provides an introduction to the imagery used within GEEDiT and some of the considerations to take into account when digitizing from these, in addition to a walkthrough of how to use GEEDiT and its functionality. The final section provides some information on how to edit GEEDiT for bespoke purposes, though you are encouraged to do this with care. You will be able to confidently use the interface and conceptually understand how the tool operates.

References

Boothroyd RJ, Williams RD, Hoey TB et al (2021) Applications of Google Earth Engine in fluvial geomorphology for detecting river channel change. Wiley Interdiscip Rev Water 8:e21496. https://doi.org/10.1002/wat2.1496

Brough S, Carr JR, Ross N, Lea JM (2019) Exceptional retreat of Kangerlussuaq Glacier, East Greenland, between 2016 and 2018. Front Earth Sci 7:123. https://doi.org/10.3389/feart.2019.00123

Chander G, Markham BL, Helder DL (2009) Summary of current radiometric calibration coefficients for Landsat MSS, TM, ETM+, and EO-1 ALI sensors. Remote Sens Environ 113:893–903. https://doi.org/10.1016/j.rse.2009.01.007

Fahrner D, Lea JM, Brough S et al (2021) Linear response of the Greenland ice sheet's tidewater glacier terminus positions to climate. J Glaciol 67:193–203. https://doi.org/10.1017/jog.2021.13

Field HR, Armstrong WH, Huss M (2021) Gulf of Alaska ice-marginal lake area change over the Landsat record and potential physical controls. Cryosphere 15:3255–3278. https://doi.org/10.5194/tc-15-3255-2021

Goliber S, Black T, Catania G, Lea JM, Olsen H, Cheng D, Bevan S, Bjørk A, Bunce C, Brough S, Carr JR (2022) TermPicks: a century of Greenland glacier terminus data for use in scientific and machine learning applications. Cryosphere 16(8):3215-3233. https://doi.org/10.5194/tc-16-3215-2022

Kochtitzky W, Copland L, Painter M, Dow C (2020) Draining and filling of ice-dammed lakes at the terminus of surge-type Dań Zhùr (Donjek) Glacier, Yukon, Canada. Can J Earth Sci 57:1337–1348. https://doi.org/10.1139/cjes-2019-0233

Lea JM (2018) The Google Earth Engine Digitisation Tool (GEEDiT) and the Margin change Quantification Tool (MaQiT)—simple tools for the rapid mapping and quantification of changing Earth surface margins. Earth Surf Dyn 6:551–561. https://doi.org/10.5194/esurf-6-551-2018

Scheingross JS, Repasch MN, Hovius N et al (2021) The fate of fluvially-deposited organic carbon during transient floodplain storage. Earth Planet Sci Lett 561:116822. https://doi.org/10.1016/j.epsl.2021.116822

Tuckett PA, Ely JC, Sole AJ et al (2019) Rapid accelerations of Antarctic Peninsula outlet glaciers driven by surface melt. Nat Commun 10:1–8. https://doi.org/10.1038/s41467-019-12039-2

Part VI
Advanced Topics

Although you now know the most basic fundamentals of Earth Engine, there is still much more that can be done. The part presents some advanced topics that can help expand your skill set for doing larger and more complex projects. These include tools for sharing code among users, scaling up with efficient project design, creating apps for non-expert users, and combining R with other information processing platforms.

27 Advanced Raster Visualization

Gennadii Donchyts and 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, e.g., 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 I).

G. Donchyts (✉)
Google, Mountain View, CA, USA
e-mail: dgena@google.com

F. Baart
Deltares, Delft, Netherlands
e-mail: fedor.baart@deltares.nl

Delft University of Technology , Delft, Netherlands

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_27

- Write a function and `map` it over an `ImageCollection` (Chap. 12).
- Inspect an `Image` and an `ImageCollection`, as well as their properties (Chap. 13).

27.1 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 toward 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 (e.g., 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 (2005). 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 et al. 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.

27.2 Practicum

27.2.1 Section 1: Palettes

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. 27.1) 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—e.g., the *turku* colormap from *Crameri* (Fig. 27.1). You can use this for variables like population count or gross domestic product.

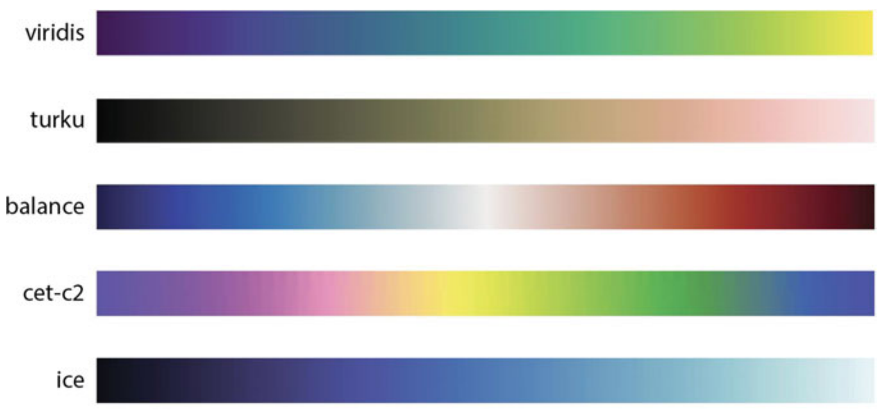


Fig. 27.1 Examples of colormaps from a variety of packages: *viridis* from *matplotlib*, *turku* from *Crameri*, *balance* from *cmocean*, *cet-c2* from colorcet and *ice* from *cmocean*

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. 27.1) 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. 27.1) 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. 27.1) from the *cmocean* package for our ice example.

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. 27.2), the `Balance` colormap has a more elegant and professional feel to it, because it uses a perceptual uniform palette and both saturation and value.

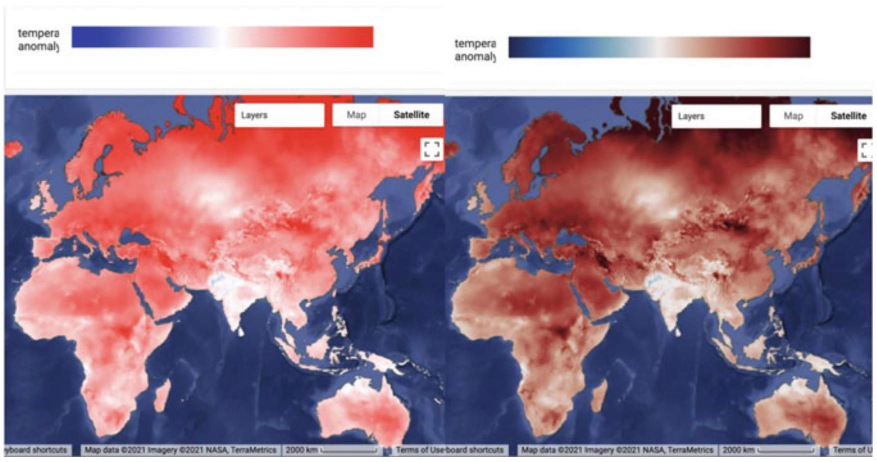


Fig. 27.2 Temperature difference of ERA5 (2011–2020, 1981–1990) using the *balance* colormap from *cmocean* (right) versus a basic *blue-white-red* colormap (left)

```
// 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');
```

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 us 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_2021121
6T042029_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 et al. 2020). We will use the `Ice` colormap from the `cmocean` package (Thyng et al. 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 Fig. 27.3, 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 et al. 2022) of the glacier.

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

27.2.2 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 us access the NLCD dataset, name it `nlcd`, and view it (Fig. 27.4) with its built-in palette.

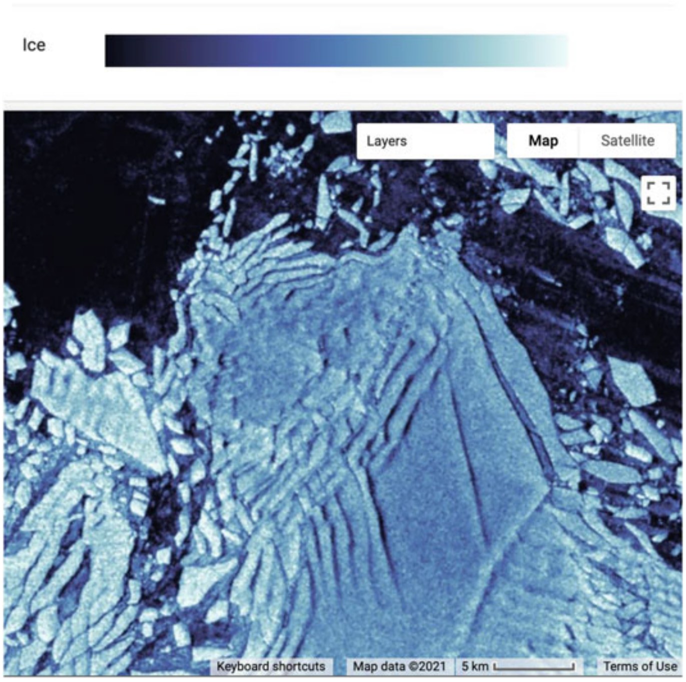


Fig. 27.3 Ice observed in Antarctica by the Sentinel-1 satellite. The image is rendered using the *ice* color palette stretched to backscatter amplitude values [− 15; 1]

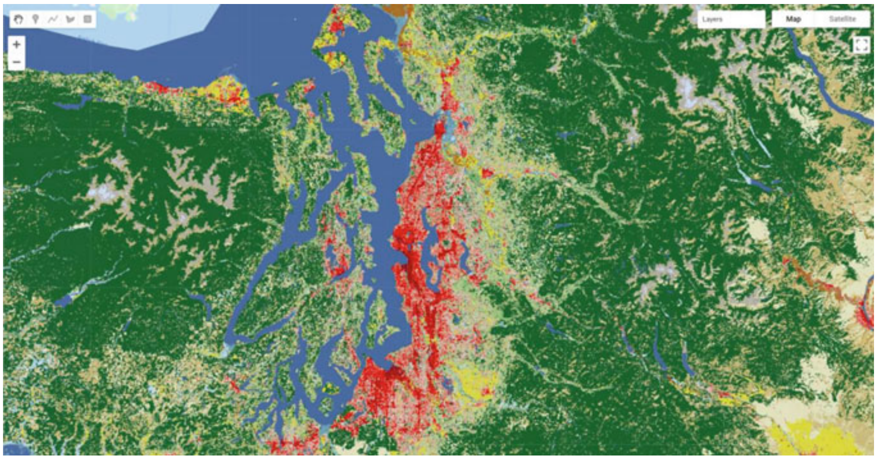


Fig. 27.4 NLCD visualized with default colors for each class

```
// 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');
```

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 us 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. 27.5).

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. 27.6).

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

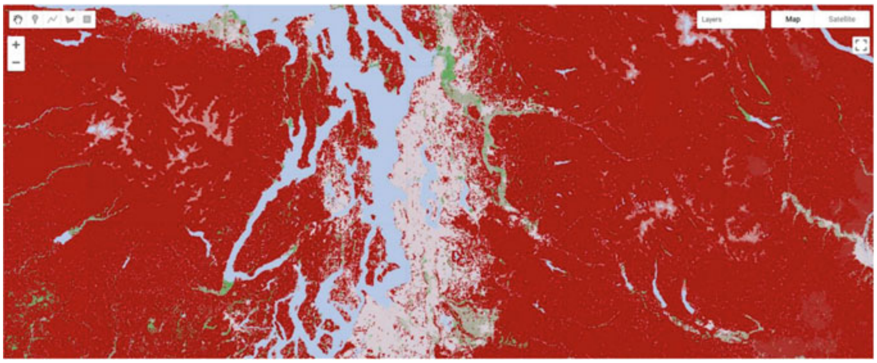


Fig. 27.5 Applying a new palette to a multi-class layer has some unexpected results

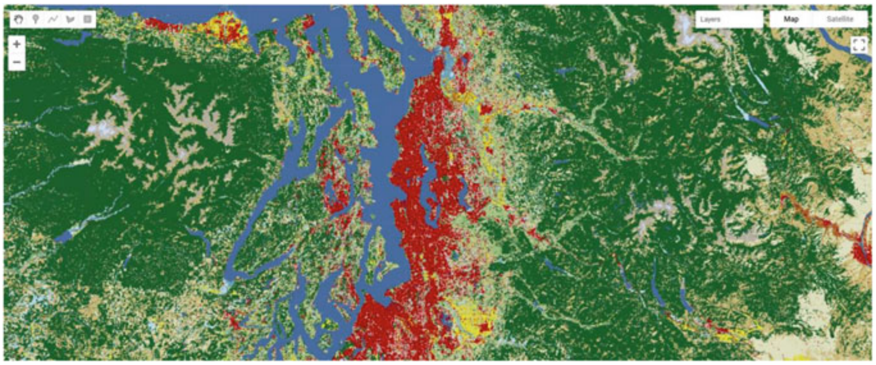


Fig. 27.6 Expected results of the new color palette. All urban areas are now correctly showing as dark red and the other land cover types remain their original color

27.2.3 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, e.g., 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 us 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. 27.7.

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

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

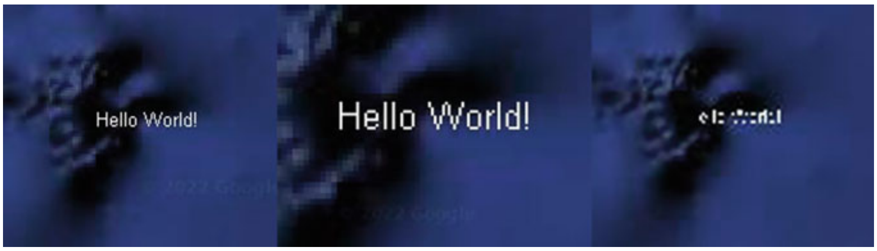


Fig. 27.7 Results of the `text.draw` call, scaled to 1×: `var scale = Map.getScale() * 1` (left); 2×: `var scale = Map.getScale() * 2` (center); and 0.5×: `var scale = Map.getScale() * 0.5` (right)

be visible. Multiple parameters listed in the above API documentation can be used to adjust text rendering. For example, let us 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. 27.8):

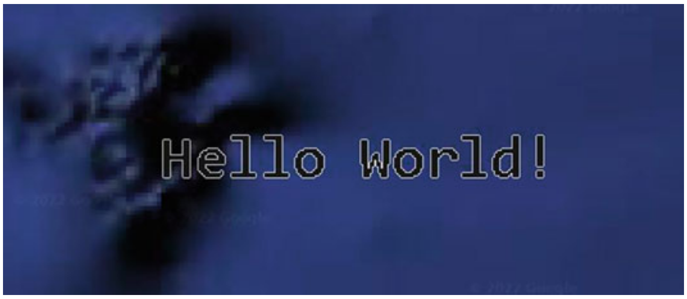


Fig. 27.8 Rendering text with adjusted parameters (font type: Consolas, fontSize: 32, textColor: 'black', outlineWidth: 1, outlineColor: 'white', outlineOpacity: 0.8)

```
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.

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 us 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 define the code to annotate an image as a function. We will then map that function (as described in Chap. 12) over the filtered `ImageCollection`. The code is as follows:

```
var text = require('users/gena/packages:text');

var geometry = ee.Geometry.Polygon(
    [
        [
            [-109.248, 43.3913],
            [-109.248, 33.2689],
            [-86.5283, 33.2689],
            [-86.5283, 43.3913]
        ]
    ], null, false);

Map.centerObject(geometry, 6);

function annotate(image) {
    // Annotates an image by adding outline border and
cloudiness
    // Cloudiness is shown as a text string rendered at the
image center.

    // Add an edge around the image.
    var edge = ee.FeatureCollection([image])
        .style({
            color: 'cccc00cc',
            fillColor: '00000000'
        });

    // Draw cloudiness as text.
    var props = {
        textColor: '0000aa',
        outlineColor: 'ffffff',
        outlineWidth: 2,
        outlineOpacity: 0.6,
        fontSize: 24,
        fontType: 'Consolas'
    };
    var center = image.geometry().centroid(1);
    var str =
ee.Number(image.get('CLOUD_COVER')).format('%.2f');
    var scale = Map.getScale();
        var textCloudiness = text.draw(str, center, scale,
    props);
    // Shift left 25 pixels.
    textCloudiness = textCloudiness
        .translate(-scale * 25, 0, 'meters', 'EPSG:3857');

    // Merge results.
    return ee.ImageCollection([edge,
textCloudiness]).mosaic();
}

// Select images.
var images =
ee.ImageCollection('LANDSAT/LC08/C02/T1_RT_TOA')
    .select([5, 4, 2])
    .filterBounds(geometry)
    .filterDate('2018-01-01', '2018-01-7');

// dim background.
Map.addLayer(ee.Image(1), {
    palette: ['black']
}, 'black', true, 0.5);

// Show images.
Map.addLayer(images, {
    min: 0.05,
    max: 1,
    gamma: 1.4
}, 'images');

// Show annotations.
var labels = images.map(annotate);
var labelsLayer = ui.Map.Layer(labels, {}, 'annotations');
Map.layers().add(labelsLayer);
```

The result of defining and mapping this function over the filtered set of images is shown in Fig. 27.9. Notice that by adding an outline around the text, we can ensure the text is visible for both dark and light images. Earth Engine requires casting properties to their corresponding value type, which is why we've used `ee.Number` (as described in Chap. 1) before generating a formatted string. Also, we have shifted the resulting text image 25 pixels to the left. This was necessary to ensure that the text is positioned properly. In more complex text rendering applications, users may be required to compute the text position in a different way using `ee.Geometry` calls from the Earth Engine API: for example, by positioning text labels somewhere near the corners.

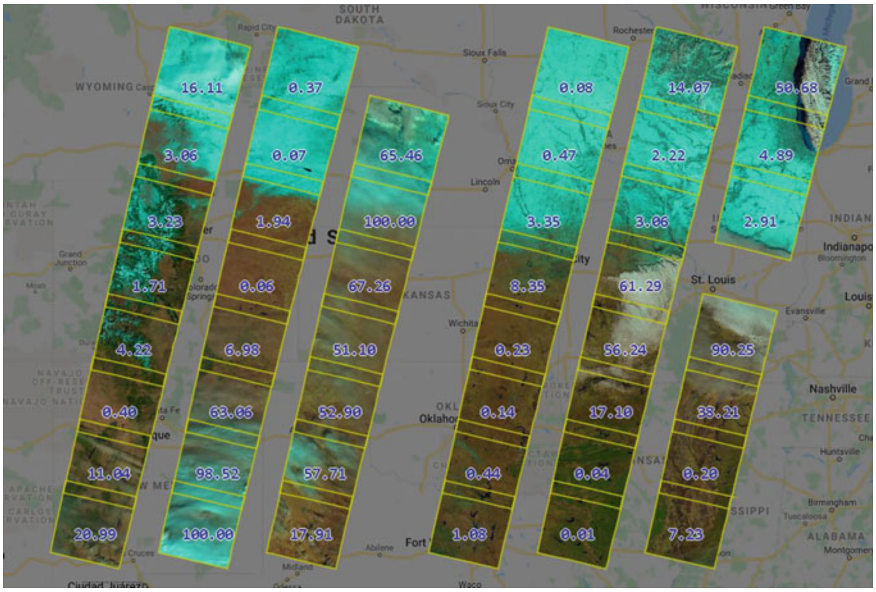


Fig. 27.9 Annotating Landsat 8 images with image boundaries, border, and text strings indicating cloudiness

Because we render text labels using the Earth Engine raster API, they are not automatically scaled depending on map zoom size. This may cause unwanted artifacts; to avoid that, the text labels image needs to be updated every time the map zoom changes. To implement this in a script, we can make use of the Map API—in particular, the `Map.onChangeZoom` event handler. The following code snippet shows how the image containing text annotations can be re-rendered every time the map zoom changes. Add it to the end of your script.

```
// re-render (rescale) annotations when map zoom changes.
Map.onChangeZoom(function(zoom) {
    labelsLayer.setEeObject(images.map(annotate));
});
```

Code Checkpoint F60f. The book's repository contains a script that shows what your code should look like at this point.

Try commenting that event handler and observe how annotation rendering changes when you zoom in or zoom out.

27.2.4 Section 4: Animations

Visualizing raster images as animations is a useful technique to explore changes in time-dependent datasets, but also, to render short animations to communicate how changing various parameters affects the resulting image—for example, varying thresholds of spectral indices resulting in different binary maps or the changing geometry of vector features.

Animations are very useful when exploring satellite imagery, as they allow viewers to quickly comprehend dynamics of changes of earth surface or atmospheric properties. Animations can also help to decide what steps should be taken next to designing a robust algorithm to extract useful information from satellite image time series. Earth Engine provides two standard ways to generate animations: as animated GIFs, and as AVI video clips. Animation can also be rendered from a sequence of images exported from Earth Engine, using numerous tools such as *ffmpeg* or *moviepy*. However, in many cases, it is useful to have a way to quickly explore image collections as animation without requiring extra steps.

In this section, we will generate animations in three different ways:

1. Generate animated GIF
2. Export video as an AVI file to Google Drive
3. Animate image collection interactively using UI controls and map layers.

We will use an image collection showing sea ice as an input dataset to generate animations with visualization parameters from earlier. However, instead of querying a single Sentinel-1 image, let us generate a filtered image collection with all images intersecting with our area of interest. After importing some packages and palettes and defining a point and rectangle, we'll build the image collection. Here, we will use point geometry to define the location where the image date label will be rendered and the rectangle geometry to indicate the area of interest for the animation. To do this, we will build the following logic in a new script. Open a new script and paste the following code into it:

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

var point = /* color: #98ff00 */ ee.Geometry.Point([-
    106.15944300895228, -74.58262940096245
]);

var rect = /* color: #d63000 */
    ee.Geometry.Polygon(
        [
            [
                [-106.19789515738981, -74.56509549360152],
                [-106.19789515738981, -74.78071448733921],
                [-104.98115931754606, -74.78071448733921],
                [-104.98115931754606, -74.56509549360152]
            ]
        ], null, false);

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

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

// Center map on geometry.
Map.centerObject(point, 9);

// Select S1 images for the Thwaites glacier.
var images = ee.ImageCollection('COPERNICUS/S1_GRD')
    .filterBounds(rect)
    .filterDate('2021-01-01', '2021-03-01')
    .select('HH')
    // Make sure we include only images which fully contain
the region geometry.
    .filter(ee.Filter.isContained({
        leftValue: rect,
        rightField: '.geo'
    }))
    .sort('system:time_start');

    // Print number of images.
    print(images.size());
```

As you see from the last lines of the above code, it is frequently useful to print the number of images in an image collection: an example of what's often known as a "sanity check."

Here, we have used two custom geometries to configure animations: the green pin named `point`, used to filter image collection and to position text labels drawn on top of the image, and the blue rectangle `rect`, used to define a bounding box for the exported animations. To make sure that the point and rectangle geometries are shown under the Geometry Imports in the Code Editor, you need to click on these variables in the code and then select the **Convert** link.

Notice that in addition to the bounds and date filter, we have also used a less known `isContained` filter to ensure that we get only images that fully cover our region. To better understand this filter, you could try commenting out the filter and compare the differences, observing images with empty (masked) pixels in the resulting image collection.

Code Checkpoint F60g. The book's repository contains a script that shows what your code should look like at this point.

Next, to simplify the animation API calls, we will generate a composite RGB image collection out of satellite images and draw the image's acquisition date as a label on every image, positioned within our region geometry.

```
// Render images.
var vis = {
    palette: palette,
    min: -15,
    max: 1
};

var scale = Map.getScale();
var textProperties = {
    outlineColor: '000000',
    outlineWidth: 3,
    outlineOpacity: 0.6
};
```

```
var imagesRgb = images.map(function(i) {
    // Use the date as the label.
    var label = i.date().format('YYYY-MM-dd');
    var labelImage = text.draw(label, point, scale,
        textProperties);

    return i.visualize(vis)
        .blend(labelImage) // Blend label image on top.
        .set({
            label: label
        }); // Keep the text property.
});
Map.addLayer(imagesRgb.first());
Map.addLayer(rect, {color:'blue'}, 'rect', 1, 0.5);
```

In addition to printing the size of the `ImageCollection`, we also often begin by adding a single image to the map from a mapped collection to see that everything works as expected—another example of a sanity check. The resulting map layer will look like Fig. 27.10.

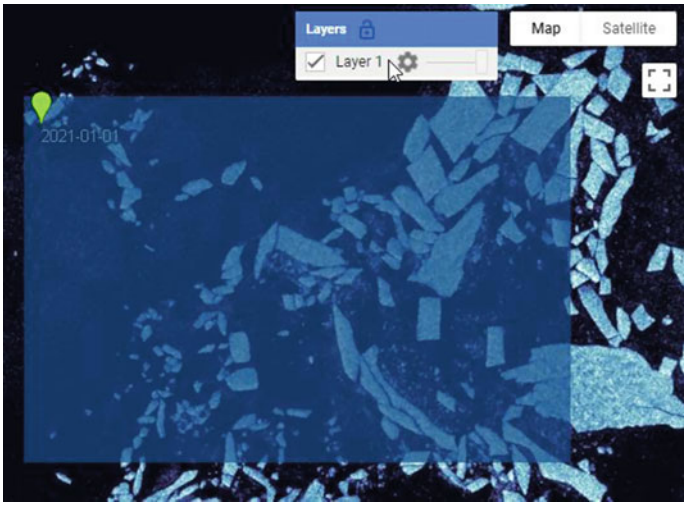


Fig. 27.10 Results of adding the first layer from the RGB composite image collection showing Sentinel-1 images with a label blended on top at a specified location. The blue geometry is used to define the bounds for the animation to be exported

Code Checkpoint F60h. The book's repository contains a script that shows what your code should look like at this point.

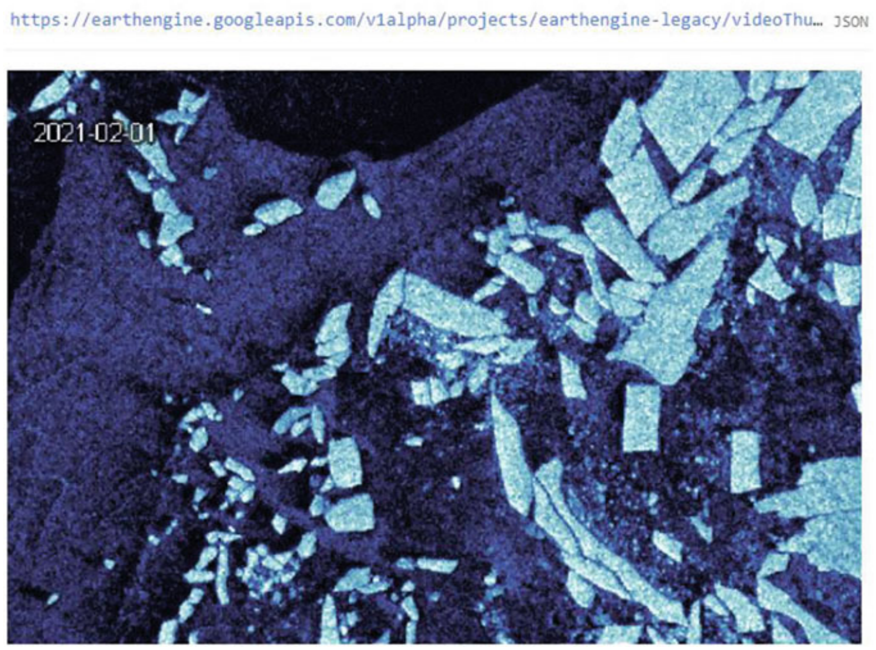

https://earthengine.googleapis.com/v1alpha/projects/earthengine-legacy/videoThu... JSON

Fig. 27.11 **Console** output after running the animated GIF code snippet, showing the GIF URL and an animation shown directly in the **Console**

Animation 1: Animated GIF with ui.Thumbnail

The quickest way to generate an animation in Earth Engine is to use the animated GIF API and either print it to the **Console** or print the URL to download the generated GIF. The following code snippet will result in an animated GIF as well as the URL to the animated GIF printed to **Console**. This is as shown in Fig. 27.11:

```
// Define GIF visualization parameters.
var gifParams = {
    region: rect,
    dimensions: 600,
    crs: 'EPSG:3857',
    framesPerSecond: 10
};

// Print the GIF URL to the console.
print(imagesRgb.getVideoThumbURL(gifParams));

// Render the GIF animation in the console.
print(ui.Thumbnail(imagesRgb, gifParams));
```

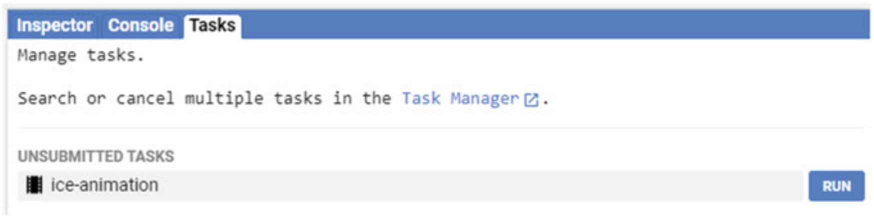
Earth Engine provides multiple options to specify the size of the resulting video. In this example, we specify 600 as the size of the maximum dimension. We also specify the number of frames per second for the resulting animated GIF as well as the target projected coordinate system to be used (EPSG:3857 here, which is the projection used in web maps such as Google Maps and the Code Editor background).

Animation 2: Exporting an Animation with Export.video.toDrive
Animated GIFs can be useful to generate animations quickly. However, they have several limitations. In particular, they are limited to 256 colors, become large for larger animations, and most web players do not provide play controls when playing animated GIFs. To overcome these limitations, Earth Engine provides export of animations as video files in MP4 format. Let us use the same RGB image collection we have used for the animated GIF to generate a short video. We can ask Earth Engine to export the video to the Google Drive using the following code snippet:

```
Export.video.toDrive({
    collection: imagesRgb,
    description: 'ice-animation',
    fileNamePrefix: 'ice-animation',
    framesPerSecond: 10,
    dimensions: 600,
    region: rect,
    crs: 'EPSG:3857'
});
```

Here, many arguments to the `Export.video.toDrive` function resemble the ones we've used in the `ee.Image.getVideoThumbURL` code above. Additional arguments include description and fileNamePrefix, which are required to configure the name of the task and the target file of the video file to be saved to Google Drive. Running the above code will result in a new task created under the Tasks tab in the Code Editor. Starting the export video task (Fig. 27.12) will result in a video file saved in the Google Drive once completed.

Animation 3: The Custom Animation Package
For the last animation example, we will use the custom package `'users/gena/packages:animation'`, built using the Earth Engine User Interface API. The main difference between this package and the above examples is that it generates an interactive animation by adding Map layers individually to the layer set, and providing UI controls that allow users to play animations or interactively switch between frames. The `animate` function in that package generates an interactive animation of an `ImageCollection`, as described below. This function has a number of optional arguments allowing you



Manage tasks.

Search or cancel multiple tasks in the Task Manager.

UNSUBMITTED TASKS

ice-animation RUN

Fig. 27.12 New export video tasks in the Tasks panel of the Code Editor

to configure, for example, the number of frames to be animated, the number of frames to be preloaded, or a few others. The optional parameters to control the function are the following:

- *maxFrames*: maximum number of frames to show (default: 30)
- *vis*: visualization parameters for every frame (default: {})
- *Label*: text property of images to show in the animation controls (default: undefined)
- *width*: width of the animation panel (default: '600px')
- *compact*: show only play control and frame slider (default: false)
- *position*: position of the animation panel (default: 'top-center')
- *timeStep*: time step (ms) used when playing animation (default: 100)
- *preloadCount*: number of frames (map layers) to preload (default: all).

Let us call this function to add interactive animation controls to the current Map:

```
// include the animation package
var animation = require('users/gena/packages:animation');

// show animation controls
animation.animate(imagesRgb, {
  label: 'label',
  maxFrames: 50
});
```

Before using the interactive animation API, we need to include the corresponding package using `require`. Here, we provide our pre-rendered image collection and two optional parameters (`label` and `maxFrames`). The first optional parameter label indicates that every image in our image collection has the `'label'` text property. The `animate` function uses this property to name map layers as well as to visualize in the animation UI controls when switching between frames. This can be useful when inspecting image collections. The second optional parameter, `maxFrames`, indicates that the maximum number of animation frames that

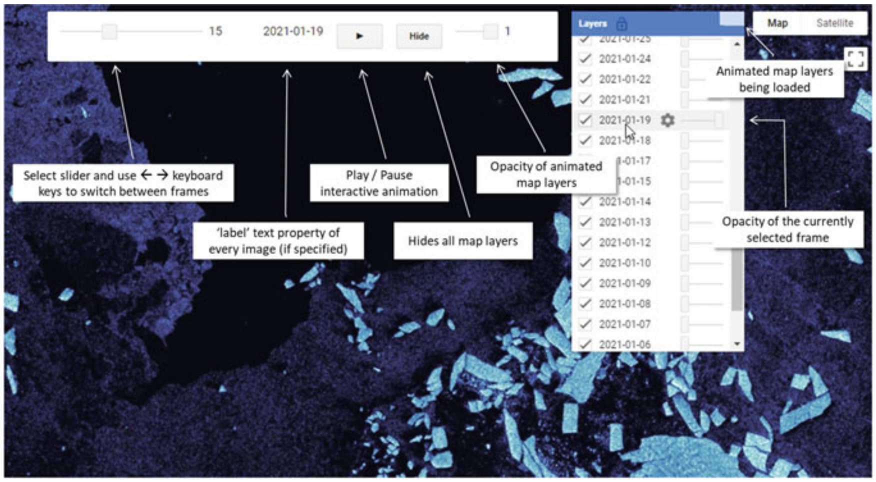

15
2021-01-19
Hide
1
Layers
Map
Satellite
2021-01-24
2021-01-22
2021-01-21
2021-01-19
2021-01-18
2021-01-14
2021-01-13
2021-01-12
2021-01-10
2021-01-09
2021-01-08
2021-01-07
2021-01-06
Select slider and use ← → keyboard keys to switch between frames
'label' text property of every image (if specified)
Play / Pause interactive animation
Hides all map layers
Opacity of animated map layers
Animated map layers being loaded
Opacity of the currently selected frame

Fig. 27.13 Interactive animation controls when using custom animation API

we would like to visualize is 50. To prevent the Code Editor from crashing, this parameter should not be too large: it is best to keep it below 100. For a much larger number of frames, it is better to use the Export video or animated GIF API. Running this code snippet will result in the animation control panel added to the map as shown in Fig. 27.13.

It is important to note that the animation API uses asynchronous UI calls to make sure that the Code Editor does not hang when running the script. The drawback of this is that for complex image collections, a large amount of processing is required. Hence, it may take some time to process all images and to visualize the interactive animation panel. The same is true for map layer names: they are updated once the animation panel is visualized. Also, map layers used to visualize individual images in the provided image collection may require some time to be rendered.

The main advantage of the interactive animation API is that it provides a way to explore image collections at frame-by-frame basis, which can greatly improve our visual understanding of the changes captured in sets of images.

Code Checkpoint F60i. The book's repository contains a script that shows what your code should look like at this point.

27.2.5 Section 5: Terrain Visualization

This section introduces several raster visualization techniques useful to visualize terrain data such as:

- Basic hillshading and parameters (light azimuth, elevation)

- Combining elevation data and colors using HSV transform (Wikipedia, 2022).
- Adding shadows.

One special type of raster data is data that represents height. Elevation data can include topography, bathymetry, but also other forms of height, such as sea surface height can be presented as a terrain.

Height is often visualized using the concept of directional light with a technique called hillshading. Because height is such a common feature in our environment, we also have an expectancy of how height is visualized. If height is visualized using a simple grayscale colormap, it looks very unnatural (Fig. 27.14, top left). By using hillshading, data immediately looks more natural (Fig. 27.14, top middle).

We can further improve the visualization by including shadows (Fig. 27.14, top right). A final step is to replace the simple grayscale colormap with a **perceptual uniform topographic** colormap and mix this with the hillshading and shadows (Fig. 27.14, bottom). This section explains how to apply these techniques.

We'll focus on elevation data stored in raster form. Elevation data is not always stored in raster formats. Other data formats include Triangulated Irregular Network

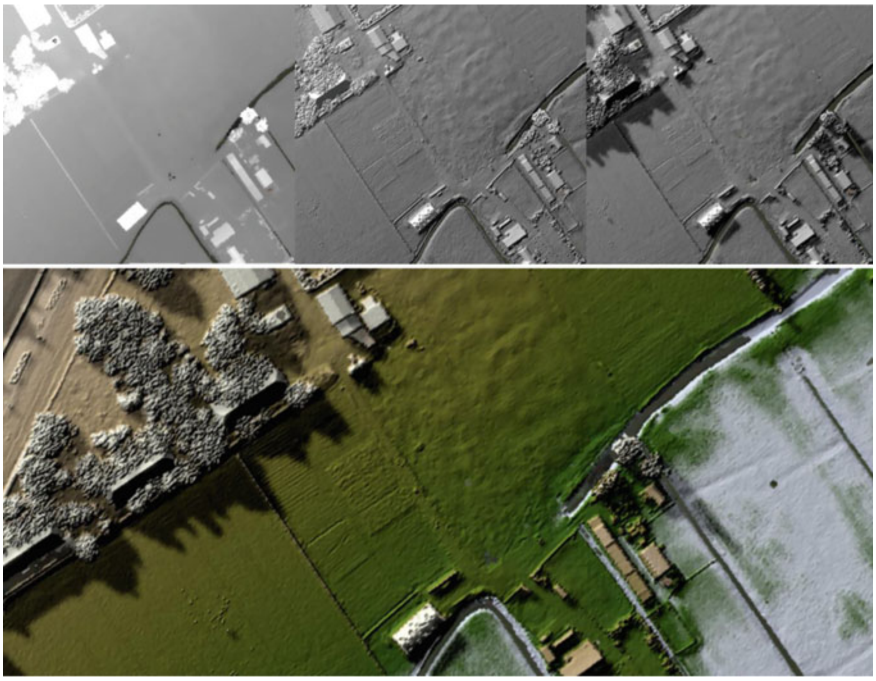


Fig. 27.14 Hillshading with shadows. Steps in visualizing a topographic dataset: (1) top left, topography with grayscale colormap; (2) top middle, topography with grayscale colormap and hillshading; (3) top right, topography with grayscale colormap, hillshading, and shadows; (4) bottom, topography with topographic colormap, hillshading, and shadows

(TIN), which allows storing information at varying resolutions and as 3D objects. This format allows one to have overlapping geometries, such as bridges with a road below it. In raster-based digital elevation models, in contrast, there can only be one height recorded for each pixel.

Let us start by loading data from a digital elevation model. This loads a topographic dataset from the Netherlands (Algemeen Hoogtebestand Nederland). It is a Digital Surface Model, based on airborne LIDAR measurements regridded to 0.5 m resolution. Enter the following code in a new script.

```
var dem = ee.Image('AHN/AHN2_05M_RUW');
```

We can visualize this dataset using a sequential gradient colormap from black to white. This results in Fig. 27.14. One can infer which areas are lower and which are higher, but the visualization does not quite "feel" like a terrain.

```
// Change map style to HYBRID and center map on the
Netherlands
Map.setOptions('HYBRID');
Map.setCenter(4.4082, 52.1775, 18);

// Visualize DEM using black-white color palette
var palette = ['black', 'white'];
var demRGB = dem.visualize({
    min: -5,
    max: 5,
    palette: palette
});
Map.addLayer(demRGB, {},'DEM');
```

An important step to visualize terrain is to add shadows created by a distant point source of light. This is referred to as hillshading or a shaded relief map. This type of map became popular in the 1940s through the work of Edward Imhof, who also used grayscale colormaps (Imhof 2015). Here, we'll use the `'gena/packages:utils'` library to combine the colormap image with the shadows. That Earth Engine package implements a `hillshadeRGB` function to simplify rendering of images enhanced with hillshading and shadow effects. One important argument this function takes is the light azimuth—an angle from the image plane upward to the light source (the Sun). This should always be set to the top left to avoid bistable perception artifacts, in which the DEM can be misperceived as inverted.

```
var utils = require('users/gena/packages:utils');

var weight =
    0.4; // Weight of Hillshade vs RGB (0 - flat, 1 -
hillshaded).
var exaggeration = 5; // Vertical exaggeration.
var azimuth = 315; // Sun azimuth.
var zenith = 20; // Sun elevation.
var brightness = -0.05; // 0 - default.
var contrast = 0.05; // 0 - default.
var saturation = 0.8; // 1 - default.
var castShadows = false;

var rgb = utils.hillshadeRGB(
    demRGB, dem, weight, exaggeration, azimuth, zenith,
    contrast, brightness, saturation, castShadows);

Map.addLayer(rgb, {}, 'DEM (no shadows)');
```

Standard hillshading only determines per pixel if it will be directed to the light or not. One can also project shadows on the map. That is done using the `ee.Algorithms.HillShadow` algorithm. Here, we'll turn on `castShadows` in the `hillshadeRGB` function. This results in a more realistic map, as can be seen in Fig. 27.14.

```
var castShadows = true;

var rgb = utils.hillshadeRGB(
    demRGB, dem, weight, exaggeration, azimuth, zenith,
    contrast, brightness, saturation, castShadows);

Map.addLayer(rgb, {}, 'DEM (with shadows)');
```

The final step is to add a topographic colormap. To visualize topographic information, one often uses special topographic colormaps. Here, we'll use the `oleron` colormap from `Crameri`. The colors get mixed with the shadows using the `hillshadeRGB` function. As you can see in Fig. 27.14, this gives a nice overview of the terrain. The area colored in blue is located below sea level.

```
var palettes = require('users/gena/packages:palettes');
var palette = palettes.crameri.oleron[50];

var demRGB = dem.visualize({
    min: -5,
    max: 5,
    palette: palette
});

var castShadows = true;

var rgb = utils.hillshadeRGB(
    demRGB, dem, weight, exaggeration, azimuth, zenith,
    contrast, brightness, saturation, castShadows);

Map.addLayer(rgb, {}, 'DEM colormap');
```

Steps to further improve a terrain visualization include using light sources from multiple directions. This allows the user to render terrain to appear more natural. In the real world, light is often scattered by clouds and other reflections.

One can also use lights to emphasize certain regions. To use even more advanced lighting techniques, one can use a raytracing engine, such as the R *rayshader* library, as discussed earlier in this chapter. The raytracing engine in the Blender 3D program is also capable of producing stunning terrain visualizations using physical-based rendering, mist, environment lights, and camera effects such as depth of field.

Code Checkpoint F60j. The book's repository contains a script that shows what your code should look like at this point.

27.3 Synthesis

To synthesize what you have learned in this chapter, you can do the following assignments.

Assignment 1. Experiment with different color palettes from the `palettes` library. Try combining palettes with image opacity (using `ee.Image.updateMask` call) to visualize different physical features (for example, hot or cold areas using temperature and elevation).

Assignment 2. Render multiple text annotations when generating animations using image collection. For example, show other image properties in addition to date or image statistics generated using regional reducers for every image.

Assignment 3. In addition to text annotations, try blending geometry elements (lines, polygons) to highlight specific areas of rendered images.

27.4 Conclusion

In this chapter, we have learned about several techniques that can greatly improve visualization and analysis of images and image collections. Using predefined palettes can help to better comprehend and communicate Earth observation data, and combining with other visualization techniques such as hillshading and annotations can help to better understand processes studied with Earth Engine. When working with image collections, it is often very helpful to analyze their properties through time by visualizing them as animations. Usually, this step helps to better understand dynamics of the changes that are stored in image collections and to develop a proper algorithm to study these changes.

References

Crameri F, Shephard GE, Heron PJ (2020) The misuse of colour in science communication. Nat Commun 11:1–10. https://doi.org/10.1038/s41467-020-19160-7
Imhof E (2015) Cartographic relief presentation. Walter de Gruyter GmbH & Co KG
Lhermitte S, Sun S, Shuman C et al (2020) Damage accelerates ice shelf instability and mass loss in Amundsen Sea Embayment. Proc Natl Acad Sci USA 117:24735–24741. https://doi.org/10.1073/pnas.1912890117
Thyng KM, Greene CA, Hetland RD et al (2016) True colors of oceanography. Oceanography 29:9–13
Wikipedia (2022) HSL and HSV. https://en.wikipedia.org/wiki/HSL_and_HSV. Accessed 1 Apr 2022
Wild CT, Alley KE, Muto A et al (2022) Weakening of the pinning point buttressing Thwaites Glacier, West Antarctica. Cryosphere 16:397–417. https://doi.org/10.5194/tc-16-397-2022
Wilkinson L (2005) The grammar of graphics. Springer Verlag

Collaborating in Earth Engine with Scripts and Assets

28

Sabrina H. Szeto

Overview
Many users find themselves needing to collaborate with others in Earth Engine at some point. Students may need to work on a group project, people from different organizations might want to collaborate on research together, or people may want to share a script or an asset they created with others. This chapter will show you how to collaborate with others and share your work.

Learning Outcomes

- Understanding when it is important to share a script or asset.
- Understanding what roles and permission options are available.
- Sharing a script with others.
- Sharing an asset with others.
- Sharing an asset so it can be displayed in an app.
- Sharing a repository with others.
- Seeing who made changes to a script and what changes were made.
- Reverting to a previous version of a script.
- Using the `require` function to load modules.
- Creating a script to share as a module.

Assumes you know how to:

- Sign up for an Earth Engine account, open the Code Editor, and save your script (Chap. 1).

S. H. Szeto (✉)
Thrive GEO GmbH, Vogtland, Germany
e-mail: sabrina@thrivegeo.com

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_28

28.1 Introduction to Theory

Many people find themselves needing to share a script when they encounter a problem; they wish to share the script with someone else so they can ask a question. When this occurs, sharing a link to the script often suffices. The other person can then make comments or changes before sending a new link to the modified script.

If you have included any assets from the Asset Manager in your script, you will also need to share these assets in order for your script to work for your colleague. The same goes for sharing assets to be displayed in an app.

Another common situation involves collaborating with others on a project. They may have some scripts they have written that they want to reuse or modify for the new project. Alternatively, several people might want to work on the same script together. For this situation, sharing a repository would be the best way forward; team members will be able to see who made what changes to a script and even revert to a previous version.

If you or your group members find yourselves repeatedly reusing certain functions for visualization or for part of your analysis, you could use the `require` module to call that function instead of having to copy and paste it into a new script each time. You could even make this function or module available to others to use via `require`.

28.2 Practicum

Let’s get started. For this lab, you will need to work in small groups or pairs.

28.2.1 Section 1: Using Get Link to Share a Script

Copy and paste the following code into the Code Editor.

```
print('The author of this script is MyName.');
```

Replace `MyName` with your name, then click on **Save** to save the script in your home repository. Next, click on the **Get Link** button and copy the link to this script onto your clipboard. Using your email program of choice, send this script to one of your group members.

Now add the following code below the line of code that you pasted earlier.

```
print('I just sent this script to GroupMemberName.');
```

Replace `GroupMemberName` with the name of the person you sent this script to, then save the script again. Next, click on the **Get Link** button and copy the link to this script onto your clipboard. Using your email program of choice, send this script to the same person.

Question 1. You should also have received two emails from someone in your group who is also doing this exercise. Open the first and second links in your Code Editor by clicking on them. Is the content of both scripts the same?

Answer: No, the scripts will be different, because **Get Link** sends a snapshot of the script at a particular point in time. Thus, even though the script was updated, the first link does not reflect that change.

Question 2. What happens when you check the box for **Hide code panel** or **Disable auto-run** before sharing the script?

Answer: **Hide code panel** will minimize the code panel so the person you send the script to will see the **Map** maximized. This is useful when you want to draw the person's attention to the results rather than to the code. To expand the code panel, they have to click on the **Show code** button. **Disable auto-run** is helpful when you do not want the script to start running when the person you sent it to opens it. Perhaps your script takes very long to run or requires particular user inputs and you just want to share the code with the person.

28.2.2 Section 2: Sharing Assets from Your Asset Manager

When you clicked the **Get Link** button earlier, you may have noticed a note in the popup reading: "To give others access to assets in the code snapshot, you may need to share them." If your script uses an asset that you have uploaded into your Asset Manager, you will need to share that asset as well. If not, an error message will appear when the person you shared the script with tries to run it.

Before sharing an asset, think about whether you have permission to share it. Is this some data that is owned by you, or did you get it from somewhere else? Do you need permission to share this asset? Make sure you have the permission to share an asset before doing so.

Now, let's practice sharing assets. First, navigate to your Asset Manager by clicking on the **Assets** tab in the left panel. If you already have some assets uploaded, pick one that you have permission to share. If not, upload one to your Asset Manager. If you do not have a shapefile or raster to upload, you can upload a small text file. Consult the Earth Engine documentation for how to do this; it will take only a few steps.

Hover your cursor over that asset in your Asset Manager. The asset gets highlighted in gray, and three buttons appear to the right of the asset. Click on the first button from the left (outlined in red in Fig. 28.1). This icon means "share."

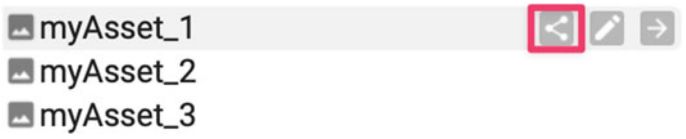
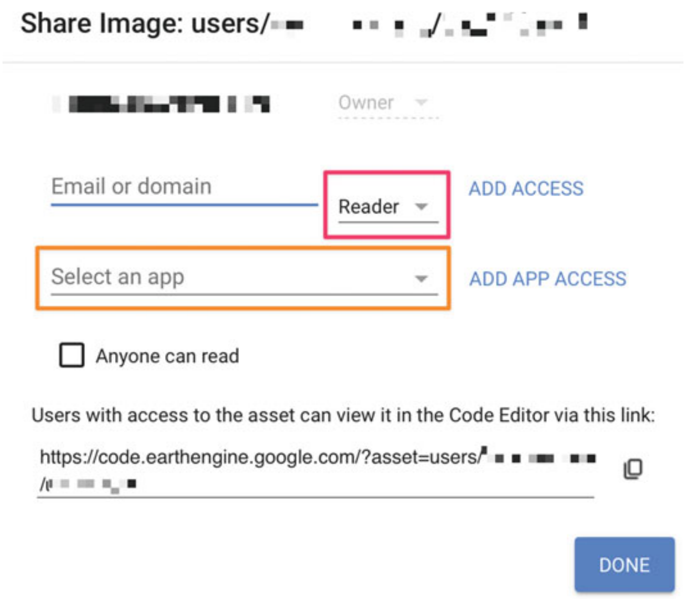


Fig. 28.1 Three assets in the Asset Manager

After you click the share button, a **Share Image** popup will appear (Fig. 28.2). This popup contains information about the path of the asset and the email address of the owner. The owner of the asset can decide who can view and edit the asset.

Click on the dropdown menu whose default option is "Reader" (outlined in red in Fig. 28.2). You will see two options for permissions: **Reader** and **Writer**. A Reader can view the asset, while a Writer can both view and make changes to it. For example, a Writer could add a new image to an `ImageCollection`. A Writer can also add other people to view or edit the asset, and a Writer can delete the asset. When in doubt, give someone the Reader role rather than the Writer role.

To share an asset with someone, you can type their email address into the **Email or domain** text field, choose **Reader** or **Writer** in the dropdown menu, and then click on **Add Access**. You can also share an asset with everyone with a certain email domain, which is useful if you want to share an asset with everyone in your organization, for instance.


Share Image: users/[redacted]

[redacted] Owner

Email or domain | Reader | ADD ACCESS

Select an app | ADD APP ACCESS

☐ Anyone can read

Users with access to the asset can view it in the Code Editor via this link:

https://code.earthengine.google.com/?asset=users/[redacted]

DONE

Fig. 28.2 **Share Image** popup window

If you want to share reading access publicly, then check the box that says **Anyone can read**. Note that you still need to share the link to the asset in order for others to access it. The only exceptions to this are when you are using the asset in a script and sharing that script using the **Get Link** button or when you share the asset with an Earth Engine app. To do the latter, use the **Select an app** dropdown menu (outlined in orange in Fig. 28.2) and click **Add App Access**. When you have completed making changes, click on the blue **Done** button to save these changes.

Question 3. Share an asset with a group member and give them reader access. Send them the link to that asset. You will also receive a link from someone else in your group. Open that link. What can you do with that asset? What do you need to do to import it into a script?

Answer: You can view details about the asset and import it for use in a script in the Code Editor. To import the asset, click on the blue **Import** button.

Question 4. Share an asset with a group member and give them writer access. Send them the link to that asset. You will also receive a link from someone else in your group. Open that link. What can you do with that asset? Try sharing the asset with a different group member.

Answer: You can view details about the asset and import it for use in a script in the Code Editor. You can also share the asset with others and delete the asset.

28.2.3 Section 3: Working with Shared Repositories

Now that you know how to share assets and scripts, let's move on to sharing repositories. In this section, you will learn about different types of repositories and how to add a repository that someone else shared with you. You will also learn how to view previous versions of a script and how to revert back to an earlier version.

Earlier, we learned how to share a script using the **Get Link** button. This link shares a code snapshot from a script. This snapshot does not reflect any changes made to the script after the time the link was shared. If you want to share a script that updates to reflect the most current version when it is opened, you need to share a repository with that script instead.

If you look under the **Scripts** tab of the leftmost panel in the Code Editor, you will see that the first three categories are labeled **Owner**, **Reader**, and **Writer**.

- Repositories categorized under **Owner** are created and owned by you. No one else has access to view or make changes to them until you share these repositories.
- Repositories categorized under **Reader** are repositories to which you have reader access. You can view the scripts but not make any changes to them. If you want to make any changes, you will need to save the script as a new file in a repository that you own.

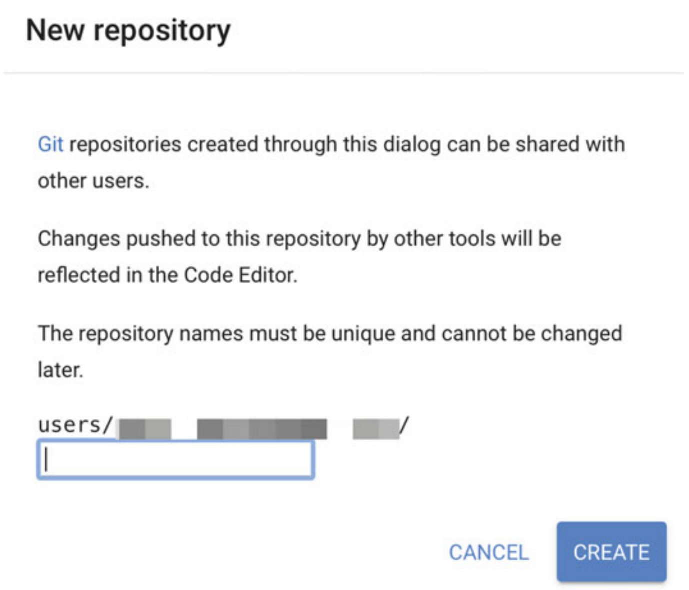
- Repositories categorized under **Writer** are repositories to which you have writer access. This means you can view and make changes to the scripts.

Let's practice creating and sharing repositories. We will start by making a new repository. Click on the red **New** button located in the left panel. Select **Repository** from the dropdown menu. A **New repository** popup window will open (Fig. 28.3).

In the popup window's text field, type a name for your new repository, such as "ForSharing1," then click on the blue **Create** button. You will see the new repository appear under the **Owner** category in the **Scripts** tab (Fig. 28.4).

Now, share this new repository with your group members: Hover your cursor over the repository you want to share. The repository gets highlighted in gray, and three buttons appear. Click on the Gear icon (outlined in red in Fig. 28.4).

A **Share Repo** popup window appears (Fig. 28.5) which is very similar to the **Share Image** popup window we saw in Fig. 28.2. The method for sharing a



New repository

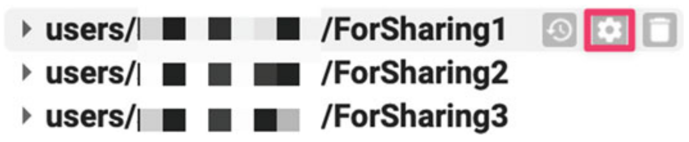
Git repositories created through this dialog can be shared with other users.

Changes pushed to this repository by other tools will be reflected in the Code Editor.

The repository names must be unique and cannot be changed later.

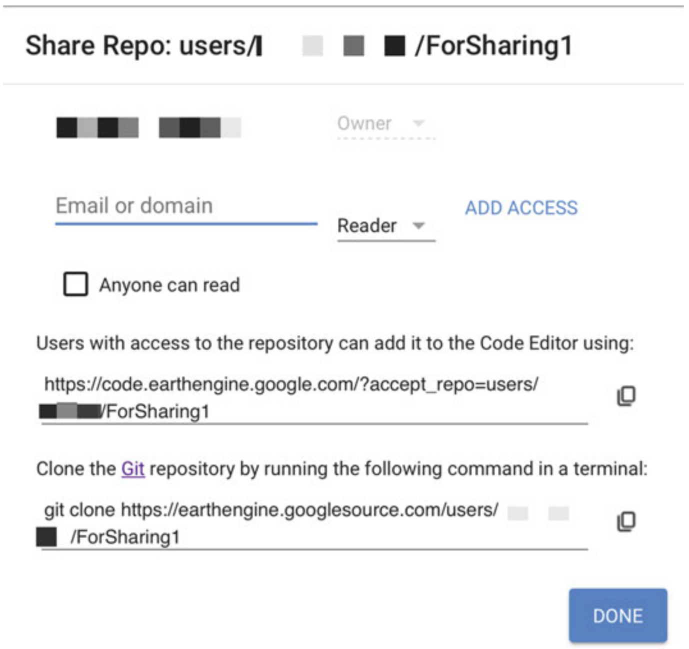
users/[redacted]/

Fig. 28.3 **New repository** popup window



users/[redacted]/ForSharing1
users/[redacted]/ForSharing2
users/[redacted]/ForSharing3

Fig. 28.4 Three repositories under the **Owner** category


Share Repo: users/[redacted]/ForSharing1

[redacted] Owner

Email or domain Reader ADD ACCESS

☐ Anyone can read

Users with access to the repository can add it to the Code Editor using:

```
https://code.earthengine.google.com/?accept_repo=users/
[redacted]/ForSharing1
```

Clone the Git repository by running the following command in a terminal:

```
git clone https://earthengine.googlesource.com/users/[redacted]
/ForSharing1
```

DONE

Fig. 28.5 **Share Repo** popup window

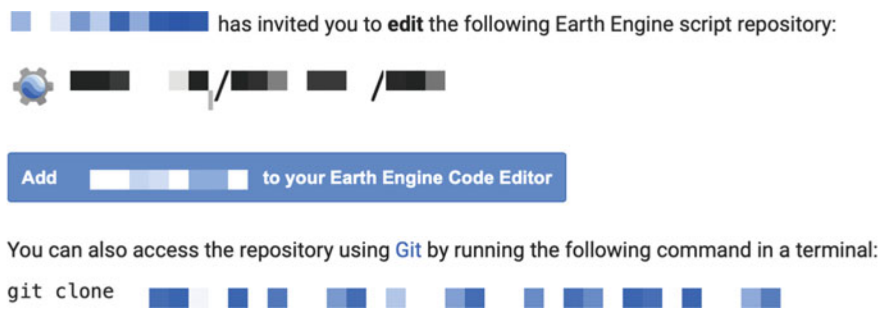
repository with a specific user or the general public is the same as for sharing assets.

Type the email address of a group member in the **Email or domain** text field and give this person a writer role by selecting **Writer** in the dropdown menu, then click on **Add Access**. When you have completed making changes, click on the blue **Done** button to save your changes.

Your group member should receive an email inviting them to edit the repository. Check your email inbox for the repository that your group member has shared with you. When you open that email, you will see content similar to what is shown in Fig. 28.6.

Now, click on the blue button that says **Add [repository path] to your Earth Engine Code Editor**. You will find the new repository added to the **Writer** category in your **Scripts** tab. The repository path will contain the username of your group member, such as `users/username/sharing`.

Now, let's add a script to the empty repository. Click on the red **New** button in the **Scripts** tab and select **File** from the dropdown menu. A **Create file** popup will appear, as shown in Fig. 28.7. Click on the gray arrow beside the default path to open a dropdown menu that will allow you to choose the path of the repository that your group member shared with you. Type a new **File Name** in the text field, such as "exercise," then click on the blue **OK** button to create the file.


[redacted] has invited you to **edit** the following Earth Engine script repository:

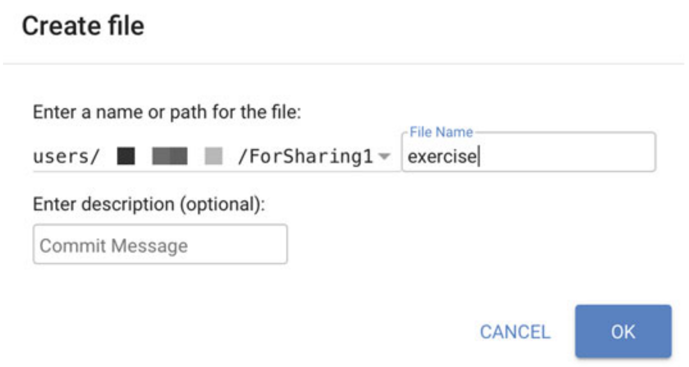
[redacted]

Add [redacted] to your Earth Engine Code Editor

You can also access the repository using Git by running the following command in a terminal:

```
git clone [redacted]
```

Fig. 28.6 “Invitation to edit” email


Create file

Enter a name or path for the file:

users/ [redacted] /ForSharing1 File Name: exercise

Enter description (optional):

Commit Message

Fig. 28.7 **Create file** popup window

A new file should now appear in the shared repository in the **Writer** category. If you do not see it, click on the Refresh icon, which is to the right of the red **New** button in the **Scripts** tab.

Double-click on the new script in the shared repository to open it. Then, copy and paste the following code to your Code Editor.

```
print('The owner of this repository is GroupMemberName.');
```

Replace `GroupMemberName` with the name of your group member, then click **Save** to save the script in the shared repository, which is under the **Writer** category.

Now, navigate to the repository under **Owner** which you shared with your group member. Open the new script which they just created by double-clicking it.

Add the following code below the line of code that you pasted earlier.

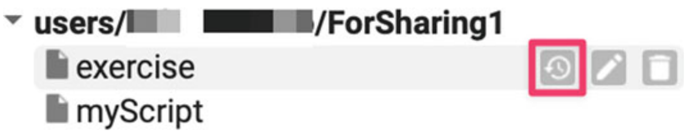
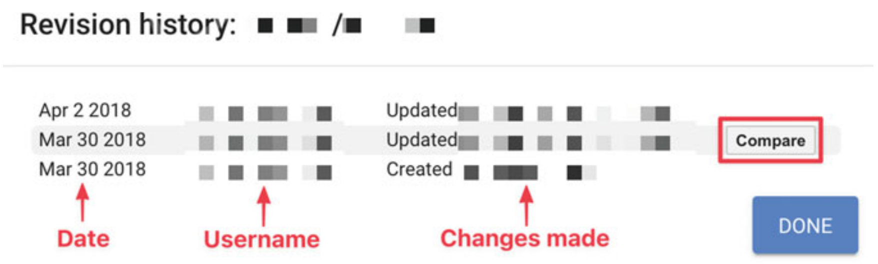


Fig. 28.8 Changes made and previous versions of the script


Revision history: [redacted]

Apr 2 2018 [redacted] Updated [redacted]
Mar 30 2018 [redacted] Updated [redacted]
Mar 30 2018 [redacted] Created [redacted]

Fig. 28.9 **Revision history** popup window

```
print('This script is shared with MyName.');
```

Replace `MyName` with your name, then save the script.

Next, we will compare changes made to the script. Click on the Versions icon (outlined in red in Fig. 28.8).

A popup window will appear, titled **Revision history**, followed by the path of the script (Fig. 28.9). There are three columns of information below the title.

- The left column contains the dates on which changes have been made.
- The middle column contains the usernames of the people who made changes.
- The right column contains information about what changes were made.

The most recent version of the script is shown in the first row, while previous versions are listed in subsequent rows. (More advanced users may notice that this is actually a Git repository.)

If you hover your cursor over a row, the row will be highlighted in gray and a button labeled **Compare** will appear. Clicking on this button allows you to compare differences between the current version of the script and a previous version in a **Version comparison** popup window (Fig. 28.10).

In the **Version comparison** popup, you will see text highlighted in two different colors. Text highlighted in red shows code that was present in the older version but is absent in the current version (the “latest commit”). Text highlighted in green shows code that is present in the current version but that was absent in the older

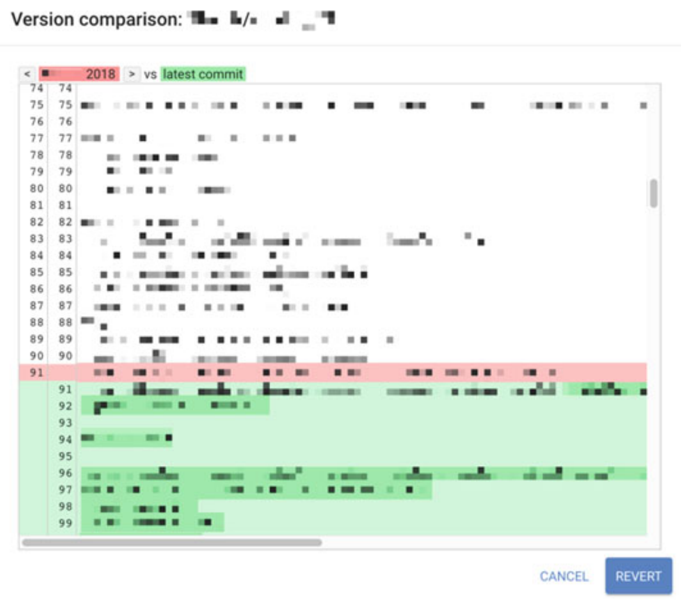


Fig. 28.10 **Version comparison** popup window

version. Generally speaking, text highlighted in red has been removed in the current version and text highlighted in green has been added to the current version. Text that is not highlighted shows code that is present in both versions.

Question 5. What text, if any, is highlighted in red when you click on **Compare** in your “exercise” script?

Answer: No text is highlighted in red, because none was removed between the previous and current versions of the script.

Question 6. What text, if any, is highlighted in green when you click on **Compare** in your “exercise” script?

Answer: `print(‘This script is shared with MyName.’);`

Question 7. What happens when you click on the blue **Revert** button?

Answer: The script reverts to the previous version, in which the only line of code is

```
print('The owner of this repository is GroupMemberName.');
```

28.2.4 Section 4: Using the Require Function to Load a Module

In earlier chapters, you may have noticed that the `require` function allows you to reuse code that has already been written without having to copy and paste it into your current script. For example, you might have written a function for cloud masking that you would like to use in multiple scripts. Saving this function as a module enables you to share the code across your own scripts and with other people. Or you might discover a new module with capabilities you need written by other authors. This section will show you how to use the `require` function to create and share your own module or to load a module that someone else has shared.

The module we will use is `ee-palettes`, which enables users to visualize raster data using common specialized color palettes (Donchyts et al. 2019). (If you would like to learn more about using these color palettes, the `ee-palettes` module is described and illustrated in detail in Chap. F6.0.) The first step is to go to this link to accept access to the repository as a reader: https://code.earthengine.google.com/?accept_repo=users/gena/packages.

Now, if you navigate to your **Reader** directory in the Code Editor, you should see a new repository called `'users/gena/packages'` listed. Look for a script called `'palettes'` and click on it to load it in your Code Editor.

If you scroll down, you will see that the script contains a nested series of dictionaries with lists of hexadecimal color specifications (as described in Chap. F2.1) that describe a color palette, as shown in the code block below. For example, the color palette named "Algae" stored in the `cmocean` variable consists of seven colors, ranging from dark green to light green (Fig. 28.11).

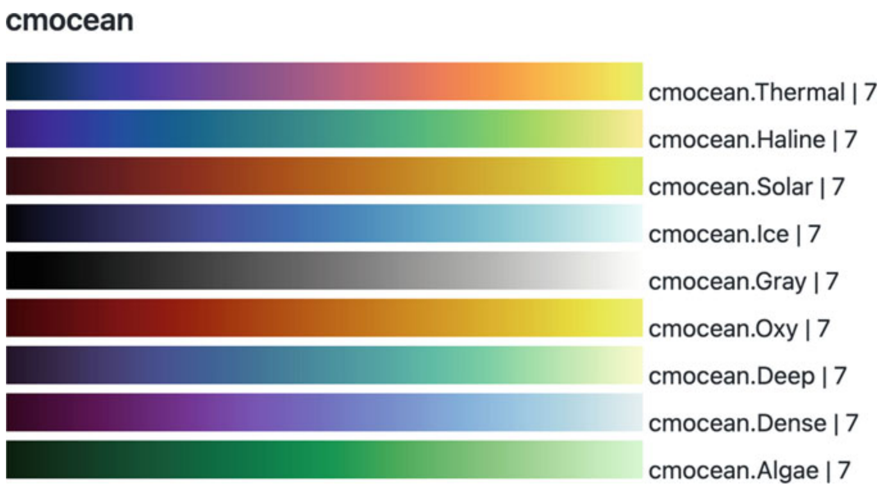


Fig. 28.11 Some of the color palettes from the `ee-palettes` GitHub repository

```
exports.cmocean = {
    Thermal: {
        7: ['042333', '2c3395', '744992', 'b15f82',
            'eb7958', 'fbb43d', 'e8fa5b'
        ]
    },
    Haline: {
        7: ['2a186c', '14439c', '206e8b', '3c9387',
            '5ab978', 'aad85c', 'fdef9a'
        ]
    },
    Solar: {
        7: ['331418', '682325', '973b1c', 'b66413',
            'cb921a', 'dac62f', 'e1fd4b'
        ]
    },
    Ice: {
        7: ['040613', '292851', '3f4b96', '427bb7',
            '61a8c7', '9cd4da', 'eafdfd'
        ]
    },
    Gray: {
        7: ['000000', '232323', '4a4a49', '727171',
            '9b9a9a', 'cacac9', 'fffffd'
        ]
    },
    Oxy: {
        7: ['400505', '850a0b', '6f6f6e', '9b9a9a',
            'cbcac9', 'ebf34b', 'ddaf19'
        ]
    },
    Deep: {
        7: ['fdfecc', 'a5dfa7', '5dbaa4', '488e9e',
            '3e6495', '3f396c', '281a2c'
        ]
    },
    Dense: {
        7: ['e6f1f1', 'a2cee2', '76a4e5', '7871d5',
            '7642a5', '621d62', '360e24'
        ]
    },
    Algae: {
        7: ['d7f9d0', 'a2d595', '64b463', '129450',
            '126e45', '1a482f', '122414'
        ]
    },
    ...
}
```

Notice that the variable is named `exports.cmocean`. Adding `exports` to the name of a function or variable makes it available to other scripts to use, as it gets added to a special global variable (Chang 2017).

To see all the color palettes available in this module, go to https://github.com/gee-community/ee-palettes.

Now let’s try using the `ee-palettes` module. Look for a script in the same repository called `'palettes-test'` and click on it to load it in your Code Editor. When you run the script, you will see digital elevation data from the National Aeronautics and Space Administration Shuttle Radar Topography Mission satellite visualized using two palettes, `colorbrewer.Blues` and `cmocean.Algae`. The map will have two layers that show the same data with different palettes.

The script first imports the digital elevation model data in the **Imports** section of the Code Editor.

```
var dem = ee.Image('USGS/SRTMGL1_003');
```

The script then loads the `ee-palettes` module by using the `require` function. The path to the module, `'users/gena/packages:palettes'`, is passed to the function. The `require` function is then stored in a variable named `'palettes'`, which will be used later to obtain the palettes for data visualization.

```
var palettes = require('users/gena/packages:palettes');
```

As described by Donchyts et al. (2019), “Each palette is defined by a group and a name, which are separated by a period (JS object dot notation), and a color level. To retrieve a desired palette, use JS object notation to specify the group, name, and number of color levels.” We define the color palette `Algae` as `palettes.cmocean.Algae[7]` because it is part of the group `cmocean` and has `7` color levels. In the next code block, you can see that the palettes (i.e., lists of hex colors) have been defined for use by setting them as the value for the `palette` key in the `visParams` object supplied to the `Map.addLayer` function.

```
// colorbrewer
Map.addLayer(dem, {
    min: 0,
    max: 3000,
    palette: palettes.colorbrewer.Blues[9]
}, 'colorbrewer Blues[9]');

// cmocean
Map.addLayer(dem, {
    min: 0,
    max: 3000,
    palette: palettes.cmocean.Algae[7]
}, 'cmocean Algae[7]');
```

Question 8. Try adding a third layer to the **Map** with a different color palette from `ee-palettes`. How easy was it to do?

Now that you have loaded and used a module shared by someone else, you can try your hand at creating your own module and sharing it with someone else in your group. First, go to the shared repository that you created in Sect. 28.2.3, create a new script in that repository, and name it "cloudmasking."

Then, go to the **Examples** repository at the bottom of the **Scripts** tab and select a function from the **Cloud Masking** repository. Let's use the `Landsat8 Surface Reflectance` cloud masking script as an example. In that script, you will see the code shown in the block below. Copy and paste all of it into your empty script.

```
// This example demonstrates the use of the Landsat 8
Collection 2, Level 2
// QA_PIXEL band (CFMask) to mask unwanted pixels.
function maskL8sr(image) {
    // Bit 0 - Fill
    // Bit 1 - Dilated Cloud
    // Bit 2 - Cirrus
    // Bit 3 - Cloud
    // Bit 4 - Cloud Shadow
    var qaMask =
image.select('QA_PIXEL').bitwiseAnd(parseInt('11111',
        2)).eq(0);
    var saturationMask = image.select('QA_RADSAT').eq(0);
    // Apply the scaling factors to the appropriate bands.
    var opticalBands =
image.select('SR_B.').multiply(0.0000275).add(-
        0.2);
    var thermalBands =
image.select('ST_B.*').multiply(0.00341802)
        .add(149.0);

    // Replace the original bands with the scaled ones and
apply the masks.
    return image.addBands(opticalBands, null, true)
        .addBands(thermalBands, null, true)
        .updateMask(qaMask)
        .updateMask(saturationMask);
}

// Map the function over one year of data.
var collection =
ee.ImageCollection('LANDSAT/LC08/C02/T1_L2')
    .filterDate('2020-01-01', '2021-01-01')
    .map(maskL8sr);

var composite = collection.median();

// Display the results.
Map.setCenter(-4.52, 40.29, 7); // Iberian Peninsula

// Display the results.
Map.setCenter(-4.52, 40.29, 7); // Iberian Peninsula
Map.addLayer(composite, {
    bands: ['SR_B4', 'SR_B3', 'SR_B2'],
    min: 0,
    max: 0.3
});
```

Note that this code is well-commented and has a header that describes what the script does. Do not forget to comment your code and describe what you are doing each step of the way. This is a good practice for collaborative coding and for your own future reference.

Imagine that you want to make the `maskL8sr` function available to other users and scripts. To do that, you can turn the function into a module. Copy and paste the code from the example code into the new script you created called "cloudmasking." (Hint: Store the function in a variable starting with `exports`. Be careful that you do not accidentally use `Export`, which is used to export datasets.)

Your script should be similar to the following code.

```
exports.maskL8sr = function(image) {
    // Bit 0 - Fill
    // Bit 1 - Dilated Cloud
    // Bit 2 - Cirrus
    // Bit 3 - Cloud
    // Bit 4 - Cloud Shadow
    var qaMask =
image.select('QA_PIXEL').bitwiseAnd(parseInt(
        '11111', 2)).eq(0);
    var saturationMask = image.select('QA_RADSAT').eq(0);

    // Apply the scaling factors to the appropriate bands.
    var opticalBands =
image.select('SR_B.').multiply(0.0000275)
        .add(-0.2);
    var thermalBands =
image.select('ST_B.*').multiply(0.00341802)
        .add(149.0);

    // Replace the original bands with the scaled ones and
apply the masks.
    return image.addBands(opticalBands, null, true)
        .addBands(thermalBands, null, true)
        .updateMask(qaMask)
        .updateMask(saturationMask);
}
```

Next, you will create a test script that makes use of the cloud masking module you just made. Begin by creating a new script in your shared repository called “cloudmasking-test.” You can modify the last part of the example cloud masking script shown in the code block below to use your module.

```
// Map the function over one year of data.
var collection =
ee.ImageCollection('LANDSAT/LC08/C02/T1_L2')
    .filterDate('2020-01-01', '2021-01-01')
    .map(maskL8sr);

var composite = collection.median();

// Display the results.
Map.setCenter(-4.52, 40.29, 7); // Iberian Peninsula
Map.addLayer(composite, {
    bands: ['SR_B4', 'SR_B3', 'SR_B2'],
    min: 0,
    max: 0.3
});
```

Question 9. How will you modify the cloud masking script to use your module? What does the script look like?

Answer: Your code might look something like the code block below.

```
// Load the module
var myCloudFunctions = require(
    'users/myusername/my-shared-repo:cloudmasking');

// Map the function over one year of data.
var collection =
ee.ImageCollection('LANDSAT/LC08/C02/T1_L2')
    .filterDate('2020-01-01', '2021-01-01')
    .map(myCloudFunctions.maskL8sr);

var composite = collection.median();

// Display the results.
Map.setCenter(-4.52, 40.29, 7); // Iberian Peninsula
Map.addLayer(composite, {
    bands: ['SR_B4', 'SR_B3', 'SR_B2'],
    min: 0,
    max: 0.3
});
```

28.3 Synthesis

Apply what you learned in this chapter by setting up a shared repository for your project, lab group, or organization. What scripts would you share? What permissions should different users have? Are there any scripts that you would turn into modules?

28.4 Conclusion

In this chapter, you learned how to collaborate with others in the Earth Engine Code Editor through sharing scripts, assets, and repositories. You learned about different roles and permissions available for sharing and when it is appropriate to use each. In addition, you are now able to see what changes have been made to a script and revert to a previous version. Lastly, you loaded and used a module that was shared with you and created your own module for sharing. You are now ready to start collaborating and developing scripts with others.

References

Chang A (2017) Making it easier to reuse code with Earth Engine script modules. In: Google Earth and Earth Engine. https://medium.com/google-earth/making-it-easier-to-reuse-code-with-earth-engine-script-modules-2e93f49abb13. Accessed 24 Feb 2022

Donchyts G, Baart F, Braaten J (2019) ee-palettes. https://github.com/gee-community/ee-palettes. Accessed 24 Feb 2022

Scaling up in Earth Engine

29

Jillian M. Deines, Stefania Di Tommaso, Nicholas Clinton, and Noel Gorelick

Overview
Commonly, when Earth Engine users move from tutorials to developing their own processing scripts, they encounter the dreaded error messages, "computation timed out" or "user memory limit exceeded." Computational resources are never unlimited, and the team at Earth Engine has designed a robust system with built-in checks to ensure that server capacity is available to everyone. This chapter will introduce general tips for creating efficient Earth Engine workflows that accomplish users' ambitious research objectives within the constraints of the Earth Engine ecosystem. We use two example case studies: (1) extracting a daily climate time series for many locations across two decades and (2) generating a regional, cloud-free median composite from Sentinel-2 imagery.

Learning Outcomes

- Understanding constraints on Earth Engine resource use.
- Becoming familiar with multiple strategies to scale Earth Engine operations.

J. M. Deines (✉)
Pacific Northwest National Laboratory, Earth Systems Predictability and Resiliency Group, Seattle, WA, USA
e-mail: jill.deines@pnnl.gov

J. M. Deines · S. Di Tommaso
Center for Food Security and the Environment, Stanford University, Stanford, CA, USA
e-mail: sditom@stanford.edu

N. Clinton
Google LLC, Mountain View, CA 94043, USA

N. Gorelick
Google Switzerland, Zürich, Switzerland
e-mail: gorelick@google.com

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_29

- Managing large projects and multistage workflows.
- Recognizing when using the Python API may be advantageous to execute large batches of tasks.

Assumes you know how to:

- Import images and image collections, filter, and visualize (Part 1).
- Write a function and map it over an `ImageCollection` (Chap. 12).
- Export and import results as Earth Engine assets (Chap. 22).
- Understand distinctions among `Image`, `ImageCollection`, `Feature`, and `FeatureCollection` Earth Engine objects (Parts 1, 2 and 5).
- Use the `require` function to load code from existing modules (Chap. 28).

29.1 Introduction to Theory

Parts 1–5 of this book have covered key remote sensing concepts and demonstrated how to implement them in Earth Engine. Most exercises have used local-scale examples to enhance understanding and complete tasks within a class-length time period. But Earth Engine's power comes from its scalability—the ability to apply geospatial processing across large areas and many years.

How we go from small to large scales is influenced by Earth Engine's design. Earth Engine runs on many individual computer servers, and its functions are designed to split up processing onto these servers. This chapter focuses on common approaches to implement large jobs within Earth Engine's constraints. To do so, we first discuss Earth Engine's underlying infrastructure to provide context for existing limits. We then cover four core concepts for scaling:

1. Using best coding practices.
2. Breaking up jobs across time.
3. Breaking up jobs across space.
4. Building a multipart workflow and exporting intermediate assets.

Earth Engine: Under the Hood

As you use Earth Engine, you may begin to have questions about how it works and how you can use that knowledge to optimize your workflow. In general, the inner workings are opaque to users. Typical fixes and approaches that data scientists use to manage memory constraints often do not apply. It is helpful to know what users can and cannot control and how your scripts translate to Earth Engine's server operations.

Earth Engine is a parallel, distributed system (see Gorelick et al. 2017), which means that when you submit tasks, it breaks up pieces of your query onto different processors to complete them more efficiently. It then collects the results and returns them to you. For many users, not having to manually design this parallel, distributed processing is a huge benefit. For some advanced users, it can be frustrating to not have better control. We would argue that leaving the details up to Earth Engine is a huge time-saver for most cases, and learning to work within a few constraints is a good time investment.

One core concept useful to master is the relationship between client-side and server-side operations. Client-side operations are performed within your browser (for the JavaScript API Code Editor) or local system (for the Python API). These include things such as manipulating strings or numbers in JavaScript. Server-side operations are executed on Google's servers and include all of the `ee.*` functions. By using the Earth Engine APIs—JavaScript or Python—you are building a chain of commands to send to the servers and later receive the result back. As much as possible, you want to structure your code to send all the heavy lifting to Google and keep processing off of your local resources.

In other words, your work in the Code Editor is making a description of a computation. All `ee` objects are just placeholders for server-side objects—their actual value does not exist locally on your computer. To see or use the actual value, it has to be evaluated by the server. If you `print` an Earth Engine object, it calls **`getInfo`** to evaluate and return the value. In contrast, you can also work with JavaScript/Python lists or numbers locally and do basic JavaScript/Python things to them, like add numbers together or loop over items. These are client-side objects. Whenever you bring a server-side object into your local environment, there is a computational cost.

Table 29.1 describes some nuts and bolts about Earth Engine and their implications. Table 29.2 provides some of the existing limits on individual tasks.

The Importance of Coding Best Practices

Good code scales better than bad code. But what is good code? Generally, for Earth Engine, good code means (1) using Earth Engine's server-side operators; (2) avoiding multiple passes through the same image collection; (3) avoiding unnecessary conversions; and (4) setting the processing scale or sample numbers appropriate for your use case, i.e., avoid using very fine scales or large samples without reason.

We encourage readers to become familiar with the "Coding Best Practices" page in the online Earth Engine User Guide. This page provides examples for avoiding mixing client- and server-side functions, unnecessary conversions, costly algorithms, combining reducers, and other helpful tips. Similarly, the "Debugging Guide–Scaling Errors" page of the online Earth Engine User Guide covers some common problems and solutions.

In addition, some Earth Engine functions are more efficient than others. For example, `Image.reduceRegions` is more efficient than `Image.sampleRegions`, because `sampleRegions` regenerates the geometries under the hood. These types of best practices are trickier to enumerate and somewhat idiosyncratic. We encourage users to learn about and make use of the **Profiler** tab, which will track and display the resources used for each operation within your script. This can help identify areas to focus efficiency improvements. Note that the profiler itself increases resource use, so only use it when necessary to develop a script and remove it for production-level execution. Other ways to discover best practices include following/posting questions

Table 29.1 Characteristics of Google Earth engine and implications for running large jobs

Earth engine characteristics	Implications
A parallel, distributed system	Occasionally, doing the exact same thing in two different orders can result in different processing distributions, impacting the ability to complete the task within system limits
Most processing is done per tile (generally a square that is 256 × 256 pixels)	Tasks that require many tiles are the most memory intensive. Some functions have a `tileScale` argument that reduces tile size, allowing processing-intensive jobs to succeed (at the cost of reduced speed)
Export mode has higher memory and time allocations than interactive mode	It is better to export large jobs. You can export to your Earth Engine assets, your Google Drive, or Google Cloud Storage
Some operations are cached temporarily	Running the same job twice could result in different run times. Occasionally, tasks may run successfully on a second try
Underlying infrastructure is composed of clusters of low-end servers	There is a hard limit on data size for any individual server; large computations need to be done in parallel using Earth Engine functions
The image processing domain, scale, and projection are defined by the specified output and applied backward throughout the processing chain	There are not many cases when you will need to manually reproject images, and these operations are costly. Similarly, manually "clipping" images is typically unnecessary

Table 29.2 Size limits for Earth engine tasks

Earth engine component	Limits
Interactive mode	Can print up to 5000 records. Computations must finish within five minutes
Export mode	Jobs have no time limit as long as they continue to make reasonable progress (defined roughly as 600 s per feature, 2 h per aggregation, and 10 min per tile). If any one tile, feature, or aggregation takes too long, the whole job will get canceled. Any jobs that take longer than one week to run will likely fail due to Earth Engine's software update release cycles
Table assets	Maximum of 100 million features, 1000 properties (columns), and 100,000 vertices for a geometry

to GIS StackExchange or the Earth Engine Developer's Discussion Group, swapping code with others, and experimentation.

29.2 Practicum

29.2.1 Topic 1: Scaling Across Time

In this section, we use an example of extracting climate data at features (points or polygons) to demonstrate how to scale an operation across many features (Sect. 29.2.1.1) and how to break up large jobs by time units when necessary (e.g., by years; Sect. 29.2.1.2).

29.2.1.1 Scaling up with Earth Engine Operators: Annual Daily Climate Data

Earth Engine's operators are designed to parallelize queries on the backend without user intervention. In many cases, they are sufficient to accomplish a scaling operation.

As an example, we will extract a daily time series of precipitation, maximum temperature, and minimum temperature for county polygons in the USA. We will use the GRIDMET Climate Reanalysis product (Abatzoglou 2013), which provides daily, 4000 m resolution gridded meteorological data from 1979 to the present across the contiguous USA. To save time for this practicum, we will focus on the states of Indiana, Illinois, and Iowa in the central USA, which together include 293 counties (Fig. 29.1).

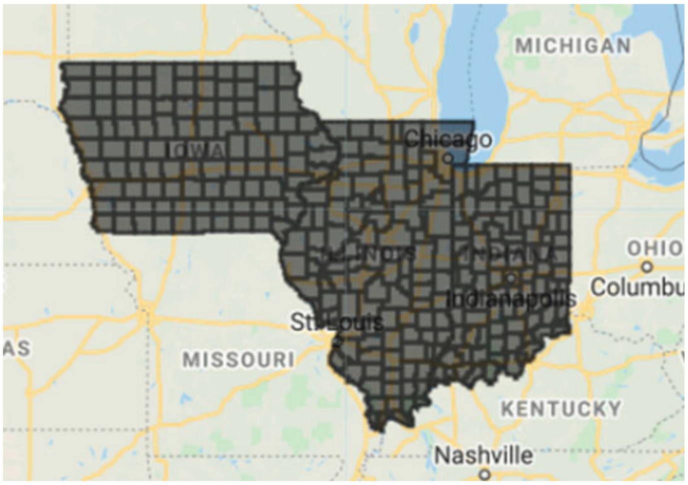


Fig. 29.1 Map of study area, showing 293 county features within the states of Iowa, Illinois, and Indiana in the USA

This example uses the ee.Image.reduceRegions operator, which extracts statistics from an Image for each Feature (point or polygon) in a FeatureCollection. We will map the reduceRegions operator over each daily image in an ImageCollection, thus providing us with the daily climate information for each county of interest.

Note that although our example uses a climate ImageCollection, this approach transfers to any ImageCollection, including satellite imagery, as well as image collections that you have already processed, such as cloud masking (Chap. 15) or time series aggregation (Chap. 14).

First, define the FeatureCollection, ImageCollection, and time period:

```
// Load county dataset.
// Filter counties in Indiana, Illinois, and Iowa by state
FIPS code.
// Select only the unique ID column for simplicity.
var countiesAll =
ee.FeatureCollection('TIGER/2018/Counties');
var states = ['17', '18', '19'];
var uniqueID = 'GEOID';
var featColl =
countiesAll.filter(ee.Filter.inList('STATEFP', states))
    .select(uniqueID);

print(featColl.size());
print(featColl.limit(1));

// Visualize target features (create Figure F6.2.1).
Map.centerObject(featColl, 5);
Map.addLayer(featColl);

// specify years of interest
var startYear = 2020;
var endYear = 2020;

// climate dataset info
var imageCollectionName = 'IDAHO_EPSCOR/GRIDMET';
var bandsWanted = ['pr', 'tmmn', 'tmmx'];
var scale = 4000;
```

Printing the size of the FeatureCollection indicates that there are 293 counties in our subset. Since we want to pull a daily time series for one year, our final dataset will have 106,945 rows—one for each county day.

Note that from our county FeatureCollection, we select only the GEOID column, which represents a unique identifier for each record in this dataset. We do this here to simplify print outputs; we could also specify which properties to include in the export function (see below).

Next, load and filter the climate data. Note we adjust the end date to January 1 of the following year, rather than December 31 of the specified year, since the filterDate function has an inclusive start date argument and an exclusive end date argument; without this modification, the output would lack data for December 31.

```
// Load and format climate data.
var startDate = startYear + '-01-01';

var endYear_adj = endYear + 1;
var endDate = endYear_adj + '-01-01';

var imageCollection =
ee.ImageCollection(imageCollectionName)
    .select(bandsWanted)
    .filterBounds(featColl)
    .filterDate(startDate, endDate);
```

Now, get the mean value for each climate attribute within each county feature. Here, we map the ee.Image.reduceRegions call over the ImageCollection, specifying an ee.Reducer.mean reducer. The reducer will apply to each band in the image, and it returns the FeatureCollection with new properties. We also add a 'date_ymd' time property extracted from the image to correctly associate daily values with their date. Finally, we flatten the output to reform a single FeatureCollection with one feature per county day.

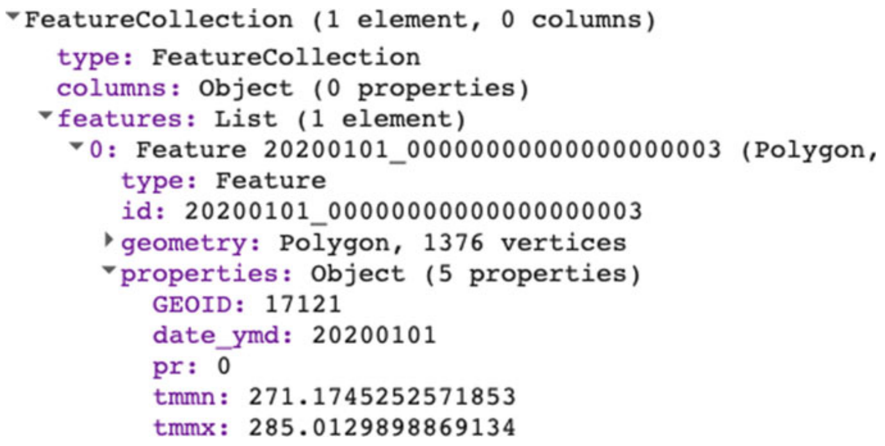
```
// get values at features
var sampledFeatures = imageCollection.map(function(image) {
    return image.reduceRegions({
            collection: featColl,
            reducer: ee.Reducer.mean(),
            scale: scale
        }).filter(ee.Filter.notNull(
        bandsWanted)) // drop rows with no data
        .map(function(f) { // add date property
            var time_start = image.get(
                'system:time_start');
            var dte = ee.Date(time_start).format(
                'YYYYMMdd');
            return f.set('date_ymd', dte);
        });
}).flatten();

print(sampledFeatures.limit(1));
```

Note that we include a filter to remove feature-day rows that lacked data. While this is less common when using gridded climate products, missing data can be common when reducing satellite images. This is because satellite collections come in scene tiles, and each image tile likely does not overlap all of our features unless it has first been aggregated temporally. It can also occur if a cloud mask has been applied to an image prior to the reduction. By filtering out **`null`** values, we can reduce empty rows.

Now, explore the result. If we simply `print(sampledFeatures)` we get our first error message: “User memory limit exceeded.” This is because we have created a `FeatureCollection` that exceeds the size limits set for interactive mode. How many are there? We could try `print(sampledFeatures.size())`, but due to the larger size, we receive a “Computation timed out” message—it is unable to tell us. Of course, we know that we expect 293 counties $\times$ 365 days = 106,945 features. We can, however, check that our reducer has worked as expected by asking Earth Engine for just one feature: `print(sampledFeatures.limit(1))`.

Here, we can see the precipitation, minimum temperature, and maximum temperature for the county with GEOID = 17,121 on January 1, 2020 (Fig. 29.2; note, temperature is in Kelvin units).

Next, export the full `FeatureCollection` as a CSV to a folder in your Google Drive. Specify the names of properties to include. Build part of the filename dynamically based on arguments used for year and data scale, so we do not need to manually modify the filenames.



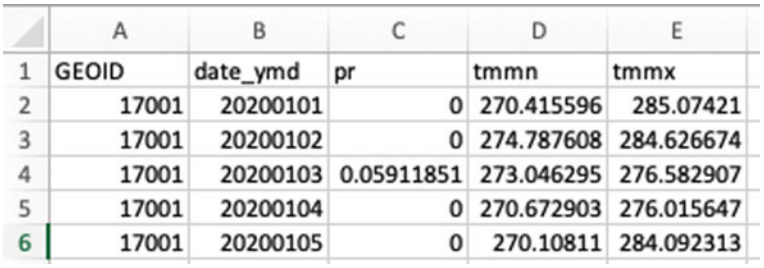
```
▾FeatureCollection (1 element, 0 columns)
    type: FeatureCollection
    columns: Object (0 properties)
  ▾features: List (1 element)
    ▾0: Feature 20200101_00000000000000000003 (Polygon,
        type: Feature
        id: 20200101_00000000000000000003
      ▸geometry: Polygon, 1376 vertices
      ▾properties: Object (5 properties)
          GEOID: 17121
          date_ymd: 20200101
          pr: 0
          tmmn: 271.1745252571853
          tmmx: 285.0129898869134
```

Fig. 29.2 Screenshot of the `print` output for one feature after the `reduceRegions` call

```
// export info
var exportFolder = 'GEE_scalingUp';
var filename = 'Gridmet_counties_IN_IL_IA_' + scale + 'm_'
+
    startYear + '-' + endYear;

// prepare export: specify properties/columns to include
var columnsWanted = [uniqueID].concat(['date_ymd'],
bandsWanted);
print(columnsWanted);

Export.table.toDrive({
    collection: sampledFeatures,
    description: filename,
    folder: exportFolder,
    fileFormat: 'CSV',
    selectors: columnsWanted
});
```

Code Checkpoint F62a. The book's repository contains a script that shows what your code should look like at this point.

On our first export, this job took about eight minutes to complete, producing a dataset 6.8 MB in size. The data is ready for downstream use but may need formatting to suit the user's goals. You can see what the exported CSV looks like in Fig. 29.3.



	A	B	C	D	E
1	GEOID	date_ymd	pr	tmmn	tmmx
2	17001	20200101	0	270.415596	285.07421
3	17001	20200102	0	274.787608	284.626674
4	17001	20200103	0.05911851	273.046295	276.582907
5	17001	20200104	0	270.672903	276.015647
6	17001	20200105	0	270.10811	284.092313

Fig. 29.3 Top six rows of the exported CSV viewed in Microsoft Excel and sorted by county GEOID

Using the Selectors Argument

There are two excellent reasons to use the `selectors` argument in your `Export.table.toDrive` call. First, if the argument is not specified, Earth Engine will generate the column names for the exported CSV from the first feature in your `FeatureCollection`. If that feature is missing properties, those properties will be dropped from the export for all features.

Perhaps even more important if you are seeking to scale up an analysis, including unnecessary columns can greatly increase file size and even processing time. For example, Earth Engine includes a ".geo" field that contains a GeoJSON description of each spatial feature. For non-simple geometries, the field can be quite large, as it lists coordinates for each polygon vertex. For many purposes, it is not necessary to include this information for each daily record (here, 365 daily rows per feature).

For example, when we ran the same job as above but did not use the `selectors` argument, the output dataset was 5.7 GB (versus 6.8 MB!) and the runtime was slower. This is a cumbersomely large file, with no real benefit. We generally recommend dropping the ".geo" column and other unnecessary properties. To retain spatial information, a unique identifier for each feature can be used for downstream joins with the spatial data or other properties. If working with point data, latitude and longitude columns can be added prior to export to maintain easily accessible geographic information, although the.geo column for point data is far smaller than for irregularly shaped polygon features.

29.2.1.2 Scaling Across Time by Batching: Get 20 Years of Daily Climate Data

Above, we extracted one year of daily data for our 293 counties. Let us say we want to do the same thing, but for 2001–2020. We have already written our script to flexibly specify years, so it is fairly adaptable to this new use case:

```
// specify years of interest
var startYear = 2020;
var endYear = 2020;
```

If we only wanted a few years for a small number of features, we could just modify the `startYear` or `endYear` and proceed. Indeed, our current example is modest in size and number of features, and we were able to run 2001–2020 in one export job that took about 2 h, with an output file size of 299 MB. However, with larger feature collections, or hourly data, we will again start to bump up against Earth Engine's limits. Generally, jobs of this sort do not fail quickly—exports are allowed to run as long as they continue making progress (see Table 29.2). It is not uncommon, however, for a large job to take well over 24 h to run, or even to fail after more than 24 h of run time, as it accumulates too many records or a single aggregation fails. For users, this can be frustrating.

We generally find it simpler to run several small jobs rather than one large job. Outputs can then be combined in external software. This avoids any frustration with long-running jobs or delayed failures, and it allows parts of the task to be run simultaneously. Earth Engine generally executes from 2 to 20 jobs per user at a time, depending on overall user load (although 20 is rare). As a counterpoint, there is some overhead for generating separate jobs.

Important note: When running a batch of jobs, it may be tempting to use multiple accounts to execute subsets of your batch and thus get your shared results faster. However, doing so is a direct violation of the Earth Engine terms of service and can result in your account(s) being terminated.

For-Loops: They are Sometimes OK

Batching jobs in time is a great way to break up a task into smaller units. Other options include batching jobs by spatial regions defined by polygons (see Sect. 29.2.2) or for computationally heavy tasks, batching by both space and time.

Because `Export` functions are client-side functions, however, you cannot create an export within an Earth Engine `map` command. Instead, we need to loop over the variable that will define our batches and create a set of export tasks.

But wait! Are not we supposed to avoid for-loops at all costs? Yes, within a computational chain. Here, we are using a loop to send multiple computational chains to the server.

First, we will start with the same script as in Sect. 29.2.1.1, but we will modify the start year. We will also modify the desired output filename to be a generic base filename, to which we will append the year for each task within the loop (in the next step).

```
// Load county dataset.
var countiesAll =
ee.FeatureCollection('TIGER/2018/Counties');
var states = ['17', '18', '19'];
var uniqueID = 'GEOID';
var featColl =
countiesAll.filter(ee.Filter.inList('STATEFP', states))
    .select(uniqueID);

print(featColl.size());
print(featColl.limit(1));
Map.addLayer(featColl);

// Specify years of interest.
var startYear = 2001;
var endYear = 2020;

// Climate dataset info.
var imageCollectionName = 'IDAHO_EPSCOR/GRIDMET';
var bandsWanted = ['pr', 'tmmn', 'tmmx'];
var scale = 4000;

// Export info.
var exportFolder = 'GEE_scalingUp';
var filenameBase = 'Gridmet_counties_IN_IL_IA_' + scale +
'm_';
```

Now, modify the code in Sect. 29.2.1.1 to use a looping variable, `i`, to represent each year. Here, we are using standard JavaScript looping syntax, where `i` will take on each value between our `startYear` (2001) and our `endYear` (2020) for each loop through this section of code, thus creating 20 queries to send to Earth Engine's servers.

```
// Initiate a loop, in which the variable i takes on values
of each year.
for (var i = startYear; i <= endYear; i++) {        // for
each year....

  // Load climate collection for that year.
  var startDate = i + '-01-01';

  var endYear_adj = i + 1;
  var endDate = endYear_adj + '-01-01';

  var imageCollection =
ee.ImageCollection(imageCollectionName)
      .select(bandsWanted)
      .filterBounds(featColl)
      .filterDate(startDate, endDate);

  // Get values at feature collection.
  var sampledFeatures = imageCollection.map(function(image)
{
    return image.reduceRegions({
      collection: featColl,
      reducer: ee.Reducer.mean(),
      tileScale: 1,
      scale: scale
    }).filter(ee.Filter.notNull(bandsWanted))  // remove
rows without data
      .map(function(f) {                  // add date
property
        var time_start = image.get('system:time_start');
        var dte = ee.Date(time_start).format('YYYYMMdd');
        return f.set('date_ymd', dte);
    });
  }).flatten();

  // Prepare export: specify properties and filename.
  var columnsWanted = [uniqueID].concat(['date_ymd'],
bandsWanted);
  var filename = filenameBase + i;

    Export.table.toDrive({
      collection: sampledFeatures,
      description: filename,
      folder: exportFolder,
      fileFormat: 'CSV',
      selectors: columnsWanted
    });

  }
```

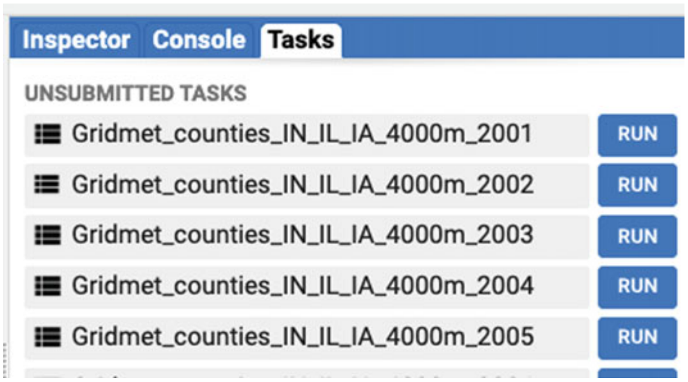
Code Checkpoint F62b. The book's repository contains a script that shows what your code should look like at this point.

When we run this script, it builds our computational query for each year, creating a batch of 20 individual jobs that will show up in the **Task** pane (Fig. 29.4). Each task name includes the year, since we used our looping variable `i` to modify the base filename we specified.

We now encounter a downside to creating batch tasks within the JavaScript Code Editor: we need to click **Run** to execute each job in turn. Here, we made this easier by programmatically assigning each job the filename we want, so we can hold the Cmd/Ctrl key and click **Run** to avoid the export task option window and only need to click once per task. Still, one can imagine that at some number of tasks, one's patience for clicking **Run** will be exceeded. We assume that number is different for everyone.

Note: If at any time you have submitted several tasks to the server but want to cancel them all, you can do so more easily from the **Earth Engine Task Manager** that is linked at the top of the Task pane. You can read about that task manager in the Earth Engine User Guide.

In order to auto-execute jobs in batch mode, we would need to use the Python API. Interested users can see the Earth Engine User Guide Python API tutorial for further details about the Python API.



Gridmet_counties_IN_IL_IA_4000m_2001
Gridmet_counties_IN_IL_IA_4000m_2002
Gridmet_counties_IN_IL_IA_4000m_2003
Gridmet_counties_IN_IL_IA_4000m_2004
Gridmet_counties_IN_IL_IA_4000m_2005

Fig. 29.4 Creation of batch tasks for each year

29.2.2 Topic 2: Scaling Across Space via Spatial Tiling

Breaking up jobs in space is another key strategy for scaling operations in Earth Engine. Here, we will focus on making a cloud-free composite from the Sentinel-2 Level 2A Surface Reflectance product. The approach is similar to that in Chap. 15, which explores cloud-free compositing. The main difference is that Landsat scenes come with a reliable quality band for each scene, whereas the process for Sentinel-2 is a bit more complicated and computationally intense (see below).

Our region of interest is the state of Washington in the USA for demonstration purposes, but the method will work at much larger continental scales as well.

Cloud Masking Approach

While we do not intend to cover the theory behind Sentinel-2 cloud masking, we do want to include a brief description of the process to convey the computational needs of this approach.

The Sentinel-2 Level 2A collection does not come with a robust cloud mask. Instead, we will build one from related products that have been developed for this purpose. Following the existing Sentinel-2 cloud masking tutorials in the Earth Engine guides, this approach requires three Sentinel-2 image collections:

- The Sentinel-2 Level 2A Surface Reflectance product. This is the dataset we want to use to build our final composite.
- The Sentinel-2 Cloud Probability Dataset, an `ImageCollection` that contains cloud probabilities for each Sentinel-2 scene.
- The Sentinel-2 Level 1C top-of-atmosphere product. This collection is needed to run the Cloud Displacement Index to identify cloud shadows, which is calculated using `ee.Algorithms.Sentinel2.CDI` (see Frantz et al. 2018 for algorithm description).

These three image collections all contain 10 m resolution data for every Sentinel-2 scene. We will join them based on their `'system:index'` property, so we can relate each Level 2A scene with the corresponding cloud probability and cloud displacement index. Furthermore, there are two `ee.Image.projection` steps to control the scale when calculating clouds and their shadows.

To sum up, the cloud masking approach is computationally costly, thus requiring some thought when applying it at scale.

29.2.2.1 Generate a Cloud-Free Satellite Composite: Limits to On-the-Fly Computing

Note: Our focus here is on code structure for implementing spatial tiling. Below, we import existing tested functions for cloud masking using the `require` command.

First, define our region and time of interest; then, load the module containing the cloud functions.

```
// Set the Region of Interest:Seattle, Washington, United
States
var roi = ee.Geometry.Point([-122.33524518034544,
47.61356183942883]);

// Dates over which to create a median composite.
var start = ee.Date('2019-03-01');
var end = ee.Date('2019-09-01');

// Specify module with cloud mask functions.
var s2mask_tools = require(
    'projects/gee-edu/book:Part F - Fundamentals/F6 -
Advanced Topics/F6.2 Scaling Up/modules/s2cloudmask.js'
);
```

Next, load and filter our three Sentinel-2 image collections.

```
// Sentinel-2 surface reflectance data for the composite.
var s2Sr = ee.ImageCollection('COPERNICUS/S2_SR')
    .filterDate(start, end)
    .filterBounds(roi)
    .select(['B2', 'B3', 'B4', 'B5']);

// Sentinel-2 Level 1C data (top-of-atmosphere).
// Bands B7, B8, B8A and B10 needed for CDI and the cloud
mask function.
var s2 = ee.ImageCollection('COPERNICUS/S2')
    .filterBounds(roi)
    .filterDate(start, end)
    .select(['B7', 'B8', 'B8A', 'B10']);

// Cloud probability dataset - used in cloud mask function
var s2c =
ee.ImageCollection('COPERNICUS/S2_CLOUD_PROBABILITY')
    .filterDate(start, end)
    .filterBounds(roi);
```

Now, apply the cloud mask:

```
// Join the cloud probability dataset to surface
reflectance.
var withCloudProbability = s2mask_tools.indexJoin(s2Sr,
s2c,
    'cloud_probability');

// Join the L1C data to get the bands needed for CDI.
var withS2L1C =
s2mask_tools.indexJoin(withCloudProbability, s2,
    'l1c');

// Map the cloud masking function over the joined
collection.
// Cast output to ImageCollection
var masked = ee.ImageCollection(withS2L1C.map(s2mask_tools
.maskImage));
```

Next, generate and visualize the median composite:

```
// Take the median, specifying a tileScale to avoid memory
errors.
var median = masked.reduce(ee.Reducer.median(), 8);

// Display the results.
Map.centerObject(roi, 12);
Map.addLayer(roi);

var viz = {
    bands: ['B4_median', 'B3_median', 'B2_median'],
    min: 0,
    max: 3000
};
Map.addLayer(median, viz, 'median');
```

Code Checkpoint F62c. The book’s repository contains a script that shows what your code should look like at this point.

After about 1–3 min, Earth Engine returns our composite to us on the fly (Fig. 29.5). Note that panning and zooming to a new area require that Earth Engine must again issue the compositing request to calculate the image for new areas. Given the delay, this is not a very satisfying way to explore our composite.

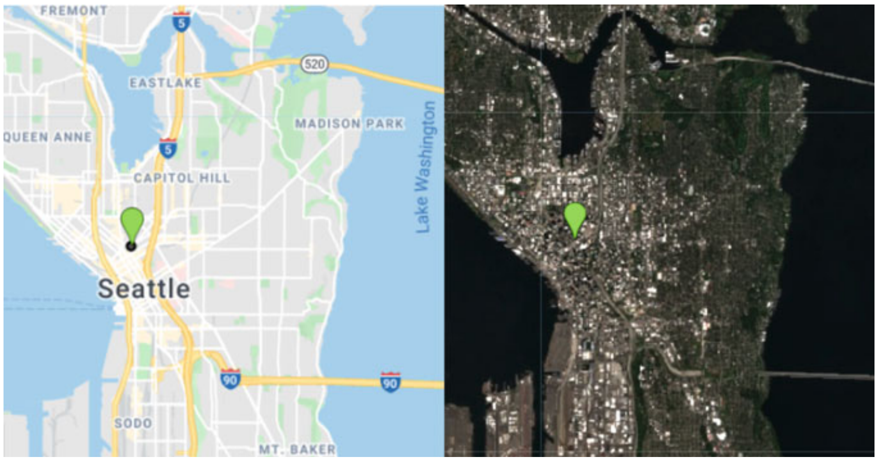
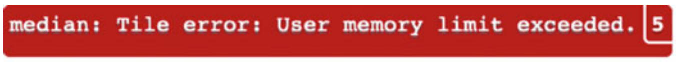


Fig. 29.5 Map view of Seattle, Washington, USA (left), and the corresponding Sentinel-2 composite (right)

```
Map.centerObject(roi, 9);
Map.addLayer(roi);
Map.addLayer(median, viz, 'median');
```



```
median: Tile error: User memory limit exceeded. 5
```

Fig. 29.6 Error message for exceeding memory limits in interactive mode

Next, expand our view (set zoom to 9) to exceed the limits of on-the-fly computation (Fig. 29.6).

As you can see, this is an excellent candidate for an export task rather than running in "on-the-fly" interactive mode, as above.

29.2.2.2 Generate a Regional Composite Through Spatial Tiling

Our goal is to apply the cloud masking method in Sect. 29.2.2.1 to the state of Washington, USA. In our testing, we successfully exported one Sentinel-2 composite for this area in about 9 h, but for this tutorial, let us presume we need to split the task up to be successful.

Essentially, we want to split our region of interest up into a regular grid. For each grid, we will export a composite image into a new `ImageCollection` asset. We can then load and `mosaic` our composite for use in downstream scripts (see below).

First, generate a spatial polygon grid (`FeatureCollection`) of desired size over your region of interest (see Fig. 29.7).

```
// Specify helper functions.
var s2mask_tools = require(
    'projects/gee-edu/book:Part F - Fundamentals/F6 -
Advanced Topics/F6.2 Scaling Up/modules/s2cloudmask.js'
);

// Set the Region of Interest: Washington, USA.
var roi = ee.FeatureCollection('TIGER/2018/States')
    .filter(ee.Filter.equals('NAME', 'Washington'));

// Specify grid size in projection, x and y units (based on
projection).
var projection = 'EPSG:4326'; // WGS84 lat lon
var dx = 2.5;
var dy = 1.5;

// Dates over which to create a median composite.
var start = ee.Date('2019-03-01');
var end = ee.Date('2019-09-01');

// Make grid and visualize.
var proj = ee.Projection(projection).scale(dx, dy);
var grid = roi.geometry().coveringGrid(proj);

Map.addLayer(roi, {}, 'roi');
Map.addLayer(grid, {}, 'grid');
```

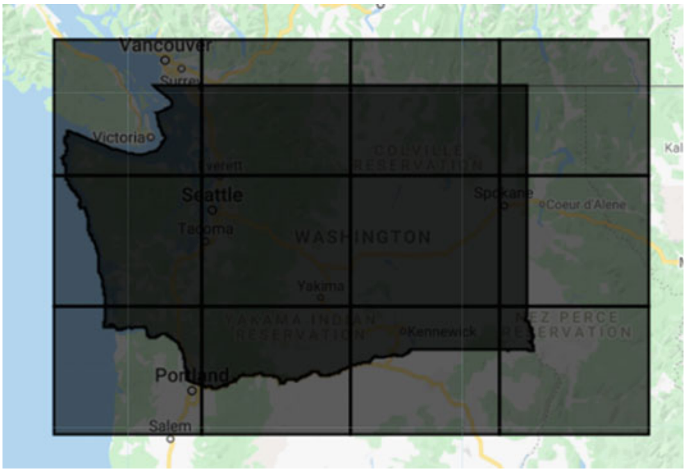


Fig. 29.7 Visualization of the regular spatial grid generated for use in spatial batch processing

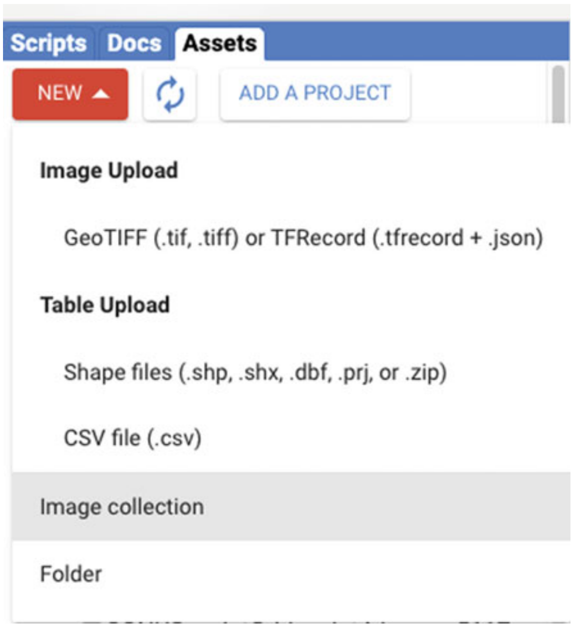

Scripts
Docs
Assets
NEW
ADD A PROJECT
Image Upload
GeoTIFF (.tif, .tiff) or TFRecord (.tfrecord + .json)
Table Upload
Shape files (.shp, .shx, .dbf, .prj, or .zip)
CSV file (.csv)
Image collection
Folder

Fig. 29.8 The "create new image collection asset" menu in the Code Editor

Next, create a new, empty `ImageCollection` asset to use as our export destination (**Assets** > **New** > **Image Collection**; Fig. 29.8). Name the image collection 'S2_composite_WA' and specify the asset location in your user folder (e.g., "path/to/your/asset/s2_composite_WA").

Specify the `ImageCollection` to export to, along with a base name for each image (the tile number will be appended in the batch export).

```
// Export info.
var assetCollection = 'path/to/your/asset/s2_composite_WA';
var imageBaseName = 'S2_median_';
```

Extract grid numbers to use as looping variables. Note that there is one **`getInfo`** call here, which should be used sparingly and never within a for-loop if you can help it. We use it to bring the number of grid cells we have generated onto the client side to set up the for-loop over grids. Note that if your grid has too many elements, you may need a different strategy.

```
// Get a list based on grid number.
var gridSize = grid.size().getInfo();
var gridList = grid.toList(gridSize);
```

Batch generate a composite image task export for each grid via looping:

```
// In each grid cell, export a composite
for (var i = 0; i < gridSize; i++) {

    // Extract grid polygon and filter S2 datasets for this
region.
    var gridCell = ee.Feature(gridList.get(i)).geometry();

    var s2Sr = ee.ImageCollection('COPERNICUS/S2_SR')
        .filterDate(start, end)
        .filterBounds(gridCell)
        .select(['B2', 'B3', 'B4', 'B5']);

    var s2 = ee.ImageCollection('COPERNICUS/S2')
        .filterDate(start, end)
        .filterBounds(gridCell)
        .select(['B7', 'B8', 'B8A', 'B10']);

    var s2c =
ee.ImageCollection('COPERNICUS/S2_CLOUD_PROBABILITY')
        .filterDate(start, end)
        .filterBounds(gridCell);

    // Apply the cloud mask.
    var withCloudProbability = s2mask_tools.indexJoin(s2Sr,
s2c,
        'cloud_probability');
    var withS2L1C =
s2mask_tools.indexJoin(withCloudProbability, s2,
        'l1c');
    var masked =
ee.ImageCollection(withS2L1C.map(s2mask_tools
        .maskImage));

    // Generate a median composite and export.
    var median = masked.reduce(ee.Reducer.median(), 8);

    // Export.
    var imagename = imageBaseName + 'tile' + i;
    Export.image.toAsset({
        image: median,
        description: imagename,
        assetId: assetCollection + '/' + imagename,
        scale: 10,
          region: gridCell,
          maxPixels: 1e13
     });
}
```

Code Checkpoint F62d. The book's repository contains a script that shows what your code should look like at this point.

Similar to Sect. 29.2.1.2, we now have a list of tasks to execute. We can hold the Cmd/Ctrl key and click **Run** to execute each task (Fig. 29.9). Again, users with applications requiring large batches may want to explore the Earth Engine Python API, which is well-suited to batching work. The output `ImageCollection` is 35.3 GB, so you may not want to execute all (or any) of these tasks but can access our pre-generated image, as discussed below.

In addition to being necessary for very large regions, batch processing can speed things up for moderate scales. In our tests, tiles averaged about 1 h to complete. Because three jobs in our queue were running simultaneously, we covered the full state of Washington in about 4 h (compared to about 9 h when tested for the full state of Washington at once). Users should note, however, that there is also an overhead to spinning up each batch task. Finding the balance between task size and task number is a challenge for most Earth Engine users that becomes easier with experience.

In a new script, load the exported `ImageCollection` and `mosaic` for use.

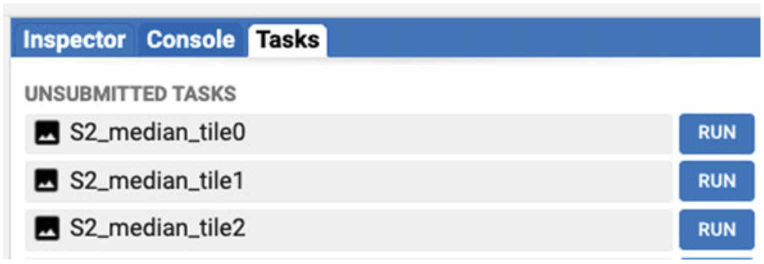


Fig. 29.9 Spatial batch tasks have been generated and are ready to run

```
// load image collection and mosaic into single image
var assetCollection = 'projects/gee-book/assets/F6-
2/s2_composite_WA';
var composite =
ee.ImageCollection(assetCollection).mosaic();

// Display the results
var geometry = ee.Geometry.Point([-120.5873563817392,
    47.39035206888694
]);
Map.centerObject(geometry, 6);
var vizParams = {
    bands: ['B4_median', 'B3_median', 'B2_median'],
    min: 0,
    max: 3000
};
Map.addLayer(composite, vizParams, 'median');
```

Code Checkpoint F62e. The book's repository contains a script that shows what your code should look like at this point.

Note the ease, speed, and joy of panning and zooming to explore the pre-computed composite asset (Fig. 29.10) compared to the on-the-fly version discussed in Sect. 29.2.2.1.

29.2.3 Topic 3: Multistep Workflows and Intermediate Assets

Often, our goals require several processing steps that cannot be completed within one Earth Engine computational chain. In these cases, the best strategy becomes breaking down tasks into individual pieces that are created, stored in assets, and used across several scripts. Each sequential script creates an intermediate output, and this intermediate output becomes the input to the next script.

As an example, consider the land use classification task of identifying irrigated agricultural lands. This type of classification can benefit from several types of evidence, including satellite composites, aggregated weather information, soil information, and/or crop type locations. Individual steps for this type of work might include:

- Generating satellite composites of annual or monthly vegetation indices.
- Processing climate data into monthly or seasonal values.
- Generating random point locations from a ground truth layer for use as a feature training dataset and accuracy validation, and extracting composite and weather values at these features.

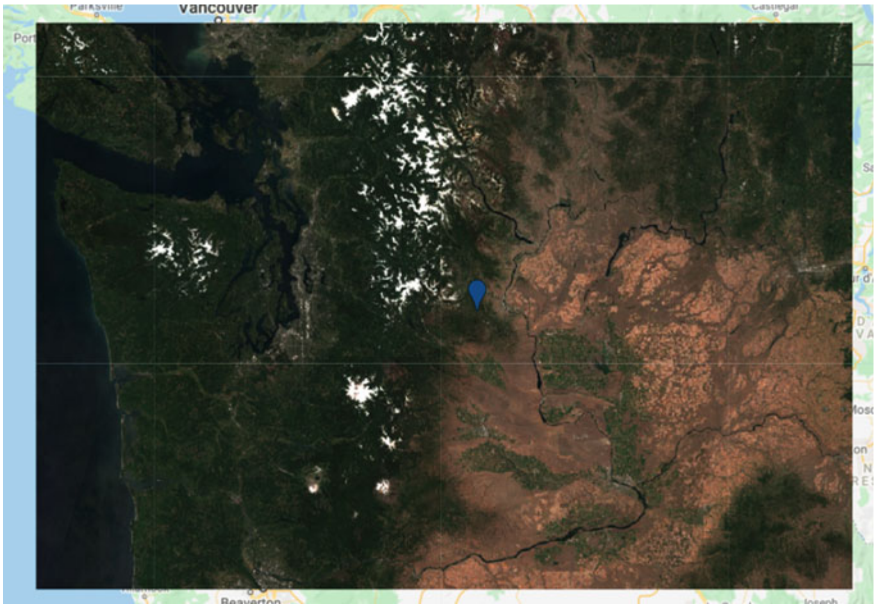


Fig. 29.10 Sentinel-2 composite covering the state of Washington, loaded from asset. The remaining white colors are snow-capped mountains, not clouds

- Training a classifier and applying it, possibly across multiple years; researchers will often implement multiple classifiers and compare the performance of different methods.
- Implementing post-classification cleaning steps, such as removing “speckle”.
- Evaluating accuracy at ground truth validation points and against government statistics using total area per administrative boundary.
- Exporting your work as spatial layers, visualizations, or other formats.

Multipart workflows can become unwieldy to manage, particularly if there are multiple collaborators or the project has a long timeline; it can be difficult to remember why each script was developed and where it fits in the overall workflow.

Here, we provide tips for managing multipart workflows. These are somewhat opinionated and based largely on concepts from “Good Enough Practices in Scientific Computing” (Wilson et al. 2017). Ultimately, your personal workflow practices will be a combination of what works for you, what works for your larger team and organization, and hopefully, what works for good documentation and reproducibility.

Tip 1. Create A Repository For Each Project

The repository can be considered the fundamental project unit. In Earth Engine, sharing permissions are set for each individual repository, so this allows you to share a specific project with others (see Chap. 28).

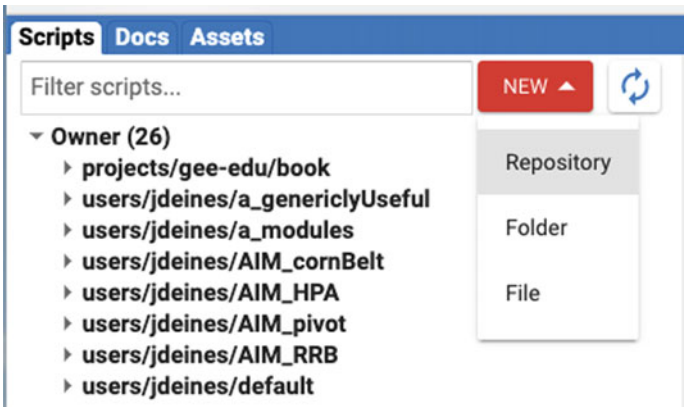
By default, Earth Engine saves new scripts in a "default" repository specific for each user (users/ < username > /default). You can create new repositories on the **Scripts** tab of the Code Editor (Fig. 29.11).

To adjust permissions for each repository, click on the Gear icon (Fig. 29.12).

For users familiar with version control, Earth Engine uses a git-based script manager, so each repository can also be managed, edited, and/or synced with your local copy or collaborative spaces like GitHub.

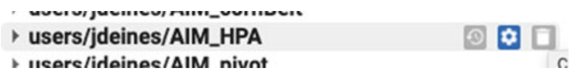
Tip 2. Make a Separate Script for Each Step, and Make Script File Names Informative and Self-sorting

Descriptive, self-sorting filenames are an excellent "good enough" way to keep your projects organized. We recommend starting script names with zero-padded numeric values to take advantage of default ordering. Because we are generating assets in early scripts that are used in later scripts, it is important to preserve the order of your workflow. The name should also include short descriptions of what the script does (Fig. 29.13).



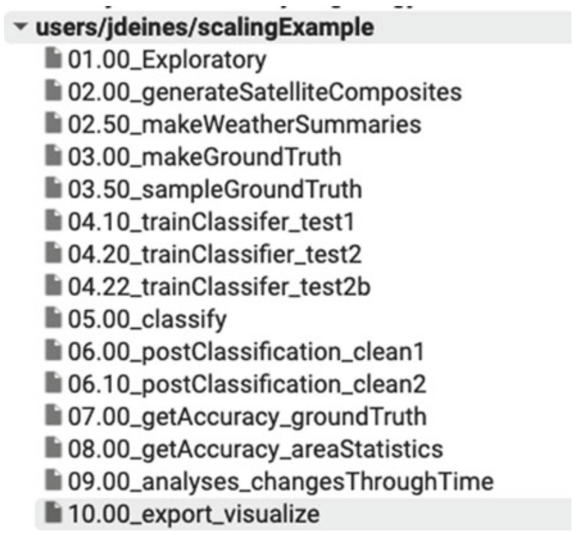
Owner (26)
- projects/gee-edu/book
- users/jdeines/a_genericlyUseful
- users/jdeines/a_modules
- users/jdeines/AIM_cornBelt
- users/jdeines/AIM_HPA
- users/jdeines/AIM_pivot
- users/jdeines/AIM_RRB
- users/jdeines/default

Fig. 29.11 Code Editor menu for creating new repositories



users/jdeines/AIM_HPA

Fig. 29.12 Access the sharing and permissions' menu for each repository by clicking the Gear icon



users/jdeines/scalingExample
- 01.00_Exploratory
- 02.00_generateSatelliteComposites
- 02.50_makeWeatherSummaries
- 03.00_makeGroundTruth
- 03.50_sampleGroundTruth
- 04.10_trainClassifer_test1
- 04.20_trainClassifier_test2
- 04.22_trainClassifer_test2b
- 05.00_classify
- 06.00_postClassification_clean1
- 06.10_postClassification_clean2
- 07.00_getAccuracy_groundTruth
- 08.00_getAccuracy_areaStatistics
- 09.00_analyses_changesThroughTime
- 10.00_export_visualize

Fig. 29.13 Example project repository with multiple scripts. Using leading numbers when naming scripts allows you to order them by their position in the workflow

Leaving some decimal places between successive scripts gives you the ability to easily insert any additional steps you did not originally anticipate. And, zero-padding means that your self-sorting still works once you move into double-digit numbers.

Other script organization strategies might involve including subfolders to collect scripts related to main steps. For example, one could have a subfolder "04_classifiers" to keep alternative classification scripts in one place, using a more tree-based file structure. Again, each user or group will develop a system that works for them. The important part is to have an organizational system.

Tip 3. Consider Data Types and File Sizes When Storing Intermediates
Images and image collections are common intermediate file types, since generating satellite composites or creating land use classifications tends to be computationally intensive. These assets can also be quite large, depending on the resolution and region size. Recall that our single-year, subregional Sentinel-2 composite in Sect. 29.2.2 was about 23 GB.

Image values can be stored from 8-bit integers to 64-bit double floats (numbers with decimals). Higher bits allow for more precision, but have much larger file sizes and are not always necessary. For example, if we are generating a land use map with five classes, we can convert that to a signed or unsigned 8-bit integer using `toInt8` or `toUint8` prior to exporting to asset, which can accommodate 256 unique values. This results in a smaller file size. Selectively retaining only bands of interest is also helpful to reduce size.

For cases requiring decimals and precision, consider whether a 32-bit float will do the job instead of a 64-bit double—`toFloat` will convert an image to a 32-bit float. If you find that you need to conserve storage, you can also scale float values and store as an integer image (`image.multiply(100).toInt16()`, for example).

This would retain precision to the second decimal place and reduce file size by a factor of two. Note that this may require you to unscale the values in downstream use. Ultimately, the appropriate data type will be specific to your needs.

And of course, as mentioned above under "The Importance of Best Coding Practices," be aware of the scale resolution you are working at, and avoid using unnecessarily high resolution when it is not supported by either the input imagery or your research goals.

Tip 4. Consider Google Cloud Platform for Hosting Larger Intermediates
If you are working with very large or very many files, you can link Earth Engine with Cloud Projects on Google Cloud Platform. See the Earth Engine documentation on "Setting Up Earth Engine Enabled Cloud Projects" for more information.

29.3 Synthesis and Conclusion

Earth Engine is built to be scaled. Scaling up working scripts, however, can present challenges when the computations take too long or return results that are too large or numerous. We have covered some key strategies to use when you encounter memory or computational limits. Generally, they involve (1) optimizing your code based on Earth Engine's functions and infrastructure; (2) working at scales appropriate for your data, question, and region of interest and not at higher resolutions than necessary; and (3) breaking up tasks into discrete units.

References

Abatzoglou JT (2013) Development of gridded surface meteorological data for ecological applications and modelling. Int J Climatol 33:121–131. https://doi.org/10.1002/joc.3413

Frantz D, Haß E, Uhl A et al (2018) Improvement of the Fmask algorithm for Sentinel-2 images: separating clouds from bright surfaces based on parallax effects. Remote Sens Environ 215:471–481. https://doi.org/10.1016/j.rse.2018.04.046

Gorelick N, Hancher M, Dixon M et al (2017) Google Earth engine: planetary-scale geospatial analysis for everyone. Remote Sens Environ 202:18–27. https://doi.org/10.1016/j.rse.2017.06.031

Wilson G, Bryan J, Cranston K et al (2017) Good enough practices in scientific computing. PLoS Comput Biol 13:e1005510. https://doi.org/10.1371/journal.pcbi.1005510

Sharing Work in Earth Engine: Basic UI and Apps

30

Qiusheng Wu

Overview
The purpose of this chapter is to demonstrate how to design and publish Earth Engine Apps using both JavaScript and Python. You will be introduced to the Earth Engine User Interface JavaScript API and the *geemap* Python package. Upon completion of this chapter, you will be able to publish an Earth Engine App with a split-panel map for visualizing land cover change.

Learning Outcomes

- Designing a user interface for an Earth Engine App using JavaScript.
- Publishing an Earth Engine App for visualizing land cover change.
- Developing an Earth Engine App using Python and *geemap*.
- Deploying an Earth Engine App using a local computer as a web server.
- Publishing an Earth Engine App using Python and free cloud platforms.
- Creating a *conda* environment using Anaconda/Miniconda.
- Installing Python packages and using Jupyter Notebook.
- Committing changes to a GitHub repository.

Q. Wu (✉)
University of Tennessee, Knoxville, USA
e-mail: qwu18@utk.edu

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_30

Assumes you know how to

- Import images and image collections, filter, and visualize (Part 1).
- Use the basic functions and logic of Python.

30.1 Introduction to Theory

Earth Engine has a user interface API that allows users to build and publish interactive web apps directly from the JavaScript Code Editor. Many readers will have encountered a call to ui.Chart in other chapters, but much more interface functionality is available. In particular, users can utilize the `ui` functions to construct an entire graphical user interface (GUI) for their Earth Engine script. The GUI may include simple widgets (e.g., labels, buttons, checkboxes, sliders, text boxes) as well as more complex widgets (e.g., charts, maps, panels) for controlling the GUI layout. A complete list of the `ui` widgets and more information about panels can be found at the links below. Once a GUI is constructed, users can publish the App from the JavaScript Code Editor by clicking the **Apps** button above the script panel in the Code Editor.

- Widgets: https://developers.google.com/earth-engine/guides/ui_widgets
- Panels: https://developers.google.com/earth-engine/guides/ui_panels.

Unlike the Earth Engine JavaScript API, the Earth Engine Python API does not provide functionality for building interactive user interfaces. Fortunately, the Jupyter ecosystem has *ipywidgets*, an architecture for creating interactive user interface controls (e.g., buttons, sliders, checkboxes, text boxes, dropdown lists) in Jupyter notebooks that communicate with Python code. The integration of graphical widgets into the Jupyter Notebook workflow allows users to configure ad hoc control panels to interactively sweep over parameters using graphical widget controls. One very powerful widget is the *output* widget, which can be used to display rich output generated by IPython, such as text, images, charts, and videos. A complete list of widgets and more information about the output widget can be found at the links below. By integrating *ipyleaflet* (for creating interactive maps) and *ipywidgets* (for designing interactive user interfaces), the *geemap* Python package (https://geemap.org) makes it much easier to explore and analyze massive Earth Engine datasets via a web browser in a Jupyter environment suitable for interactive exploration, teaching, and sharing. Users can build interactive Earth Engine Apps using *geemap* with minimal coding (Fig. 30.1).

- Widgets: https://ipywidgets.readthedocs.io/en/latest/examples/Widget%20List.html
- Output: https://ipywidgets.readthedocs.io/en/latest/examples/Output%20Widget.html.

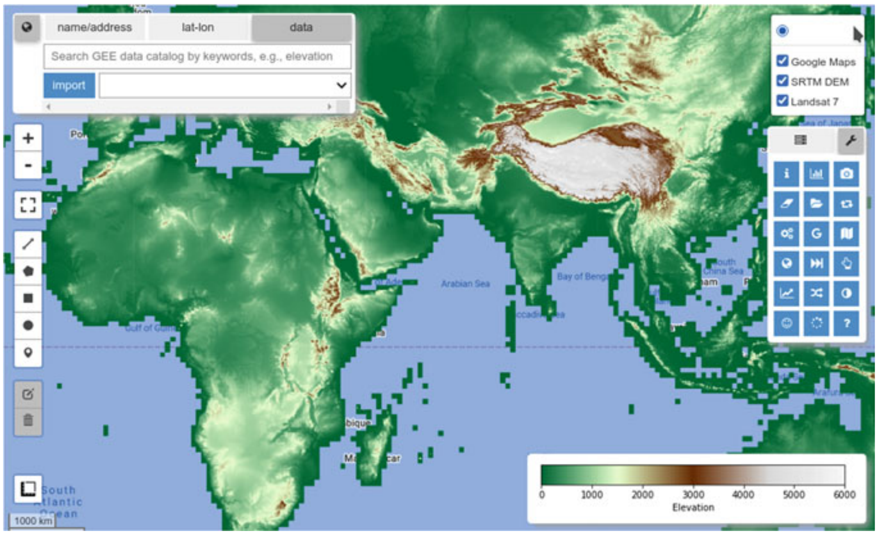


Fig. 30.1 GUI of *geemap* in a Jupyter environment

30.2 Practicum

30.2.1 Section 1. Building an Earth Engine App Using JavaScript

In this section, you will learn how to design a user interface for an Earth Engine App using JavaScript and the Earth Engine User Interface API. Upon completion of this section, you will have an Earth Engine App with a split-panel map for visualizing land cover change using the Landsat-based United States Geological Survey National Land Cover Database (NLCD).

First, let's define a function for filtering the NLCD `ImageCollection` by year and select the `landcover` band. The function returns an Earth Engine `ui.Map.Layer` of the `landcover` band of the selected NLCD image. Note that as of this writing, NLCD spans nine epochs: 1992, 2001, 2004, 2006, 2008, 2011, 2013, 2016, and 2019. The 1992 data are primarily based on unsupervised classification of Landsat data, while the rest of the images rely on the imperviousness data layer for the urban classes and on a decision-tree classification for the rest. The 1992 image is not directly comparable to any later editions of NLCD (see the Earth Engine Data Catalog for more details, if needed). Therefore, we will use only the eight epochs after 2000 in this lab.

```
// Get an NLCD image by year.
var getNLCD = function(year) {
    // Import the NLCD collection.
    var dataset = ee.ImageCollection(
        'USGS/NLCD_RELEASES/2019_REL/NLCD');

    // Filter the collection by year.
    var nlcd = dataset.filter(ee.Filter.eq('system:index',
year))
        .first();

    // Select the land cover band.
    var landcover = nlcd.select('landcover');
    return ui.Map.Layer(landcover, {}, year);
};
```

Our intention is to create a dropdown list so that when a particular epoch is selected, the corresponding NLCD image layer will be displayed on the map. We will define a dictionary with each NLCD epoch as the key and its corresponding NLCD image layer as the value. The keys of the dictionary (i.e., the eight NLCD epochs) will be used as the input to the dropdown lists (`ui.Select`) on the split-level map.

```
// Create a dictionary with each year as the key
// and its corresponding NLCD image layer as the value.
var images = {
    '2001': getNLCD('2001'),
    '2004': getNLCD('2004'),
    '2006': getNLCD('2006'),
    '2008': getNLCD('2008'),
    '2011': getNLCD('2011'),
    '2013': getNLCD('2013'),
    '2016': getNLCD('2016'),
    '2019': getNLCD('2019'),
};
```

The split-panel map is composed of two individual maps, `leftMap` and `rightMap`. The map controls (e.g., `zoomControl`, `scaleControl`, `mapTypeControl`) will be shown only on `rightMap`. A control panel (`ui.Panel`) composed of a label (`ui.Label`) and a dropdown list (`ui.Select`) is added to each map. When an NLCD epoch is selected from a dropdown list, the function `updateMap` will be called to show the corresponding image layer of the selected epoch.

```
// Create the left map, and have it display the first
layer.
var leftMap = ui.Map();
leftMap.setControlVisibility(false);
var leftSelector = addLayerSelector(leftMap, 0, 'top-
left');

// Create the right map, and have it display the last
layer.
var rightMap = ui.Map();
rightMap.setControlVisibility(true);
var rightSelector = addLayerSelector(rightMap, 7, 'top-
right');

// Adds a layer selection widget to the given map, to allow
users to
// change which image is displayed in the associated map.
function addLayerSelector(mapToChange, defaultValue,
position) {
    var label = ui.Label('Select a year:');

    // This function changes the given map to show the
selected image.
    function updateMap(selection) {
        mapToChange.layers().set(0, images[selection]);
    }
    // Configure a selection dropdown to allow the user to
choose
    // between images, and set the map to update when a
user
    // makes a selection.
    var select = ui.Select({
        items: Object.keys(images),
        onChange: updateMap
    });
    select.setValue(Object.keys(images)[defaultValue],
    true);
    var controlPanel =
        ui.Panel({
            widgets: [label, select],
            style: {
                position: position
            }
        });

    mapToChange.add(controlPanel);
}
```

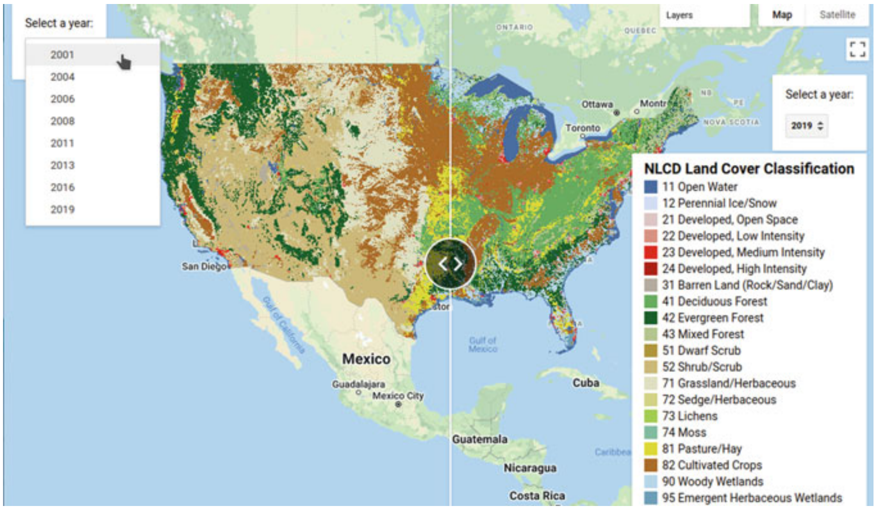
When displaying a land cover classification image on the **Map**, it would be useful to add a legend to make it easier for users to interpret the land cover type associated with each color. Let's define a dictionary that will be used to construct the legend. The dictionary contains two keys: `names` (a list of land cover types) and `colors` (a list of colors associated with each land cover type). The legend will be placed in the bottom right of the **Map**.

```
// Set the legend title.
var title = 'NLCD Land Cover Classification';

// Set the legend position.
var position = 'bottom-right';

// Define a dictionary that will be used to make a legend
var dict = {
    'names': [
        '11   Open Water',
        '12   Perennial Ice/Snow',
        '21   Developed, Open Space',
        '22   Developed, Low Intensity',
        '23   Developed, Medium Intensity',
        '24   Developed, High Intensity',
        '31   Barren Land (Rock/Sand/Clay)',
        '41   Deciduous Forest',
        '42   Evergreen Forest',
        '43   Mixed Forest',
        '51   Dwarf Scrub',
        '52   Shrub/Scrub',
        '71   Grassland/Herbaceous',
        '72   Sedge/Herbaceous',
        '73   Lichens',
        '74   Moss',
        '81   Pasture/Hay',
        '82   Cultivated Crops',
        '90   Woody Wetlands',
        '95   Emergent Herbaceous Wetlands',
    ],
    'colors': [
        '#466b9f', '#d1def8', '#dec5c5', '#d99282',
        '#eb0000', '#ab0000',

        '#b3ac9f', '#68ab5f', '#1c5f2c', '#b5c58f',
        '#af963c', '#ccb879',

        '#dfdfc2', '#d1d182', '#a3cc51', '#82ba9e',
        '#dcd939', '#ab6c28',

        '#b8d9eb', '#6c9fb8',
    ]
};
```

With the legend dictionary defined above, we can now create a panel to hold the legend widget and add it to the **Map**. Each row on the legend widget is composed of a color box followed by its corresponding land cover type.

```
// Create a panel to hold the legend widget.
var legend = ui.Panel({
    style: {
        position: position,
        padding: '8px 15px'
    }
});

// Function to generate the legend.
function addCategoricalLegend(panel, dict, title) {
    // Create and add the legend title.
    var legendTitle = ui.Label({
        value: title,
        style: {
            fontWeight: 'bold',
            fontSize: '18px',
            margin: '0 0 4px 0',
            padding: '0'
        }
    });
    panel.add(legendTitle);

    var loading = ui.Label('Loading legend...', {
        margin: '2px 0 4px 0'
    });
    panel.add(loading);

    // Creates and styles 1 row of the legend.
    var makeRow = function(color, name) {
        // Create the label that is actually the colored
box.
        var colorBox = ui.Label({
            style: {
                backgroundColor: color,
                // Use padding to give the box height and
width.
                padding: '8px',
                margin: '0 0 4px 0'
            }
        });

        // Create the label filled with the description
text.
        var description = ui.Label({
            value: name,
            style: {
                margin: '0 0 4px 6px'
            }
        });

        return ui.Panel({
            widgets: [colorBox, description],
            layout: ui.Panel.Layout.Flow('horizontal')
        });
    };
    // Get the list of palette colors and class names from
the image.
    var palette = dict.colors;
    var names = dict.names;
    loading.style().set('shown', false);

    for (var i = 0; i < names.length; i++) {
        panel.add(makeRow(palette[i], names[i]));
    }

    rightMap.add(panel);

}
```

The last step is to create a split-panel map to hold the linked maps (`leftMap` and `rightMap`) and tie everything together. When users pan and zoom one map, the other map will also be panned and zoomed to the same extent automatically. When users select a year from a dropdown list, the image layer will be updated accordingly. Users can use the slider to swipe through and visualize land cover change easily (Fig. 30.2). Please make sure you minimize the Code Editor and maximize the **Map** so that you can see the dropdown widget in the upper-right corner of the map.



NLCD Land Cover Classification
- 🟦 11 Open Water
- ⬜ 12 Perennial Ice/Snow
- 🟫 21 Developed, Open Space
- 🟥 22 Developed, Low Intensity
- 🟥 23 Developed, Medium Intensity
- 🟥 24 Developed, High Intensity
- ⬜ 31 Barren Land (Rock/Sand/Clay)
- 🟩 41 Deciduous Forest
- 🟩 42 Evergreen Forest
- 🟩 43 Mixed Forest
- 🟫 51 Dwarf Scrub
- 🟫 52 Shrub/Scrub
- ⬜ 71 Grassland/Herbaceous
- 🟨 72 Sedge/Herbaceous
- 🟩 73 Lichens
- 🟩 74 Moss
- 🟨 81 Pasture/Hay
- 🟫 82 Cultivated Crops
- 🟦 90 Woody Wetlands
- 🟦 95 Emergent Herbaceous Wetlands

Fig. 30.2 Split-panel map for visualizing land cover change using NLCD

```
addCategoricalLegend(legend, dict, title);

// Create a SplitPanel to hold the adjacent, linked maps.
var splitPanel = ui.SplitPanel({
    firstPanel: leftMap,
    secondPanel: rightMap,
    wipe: true,
    style: {
        stretch: 'both'
    }
});

// Set the SplitPanel as the only thing in the UI root.
ui.root.widgets().reset([splitPanel]);
var linker = ui.Map.Linker([leftMap, rightMap]);
leftMap.setCenter(-100, 40, 4);
```

Code Checkpoint F63a. The book's repository contains a script that shows what your code should look like at this point.

30.2.2 Section 2. Publishing an Earth Engine App from the Code Editor

The goal of this section is to publish the Earth Engine App that we created in Sect. 30.2.1. The look and feel of interfaces changes often; if the exact windows described below change over time, the concepts should remain stable to help you to publish your App. First, load the script (see the Code Checkpoint in Sect. 30.2.1) into the Code Editor. Then, open the **Manage Apps** panel by clicking the **Apps** button above the script panel in the Code Editor (Fig. 30.3).

Now click on the **New App** button (Fig. 30.4).

In the **Publish New App** dialog (Fig. 30.5), choose a name for the App (e.g., NLCD Land Cover Change), select a Google Cloud Project, provide a thumbnail to

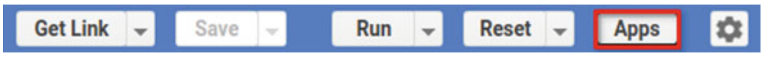


Fig. 30.3 **Apps** button in the JavaScript Code Editor

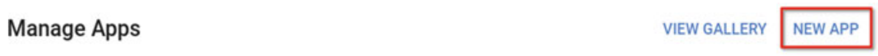
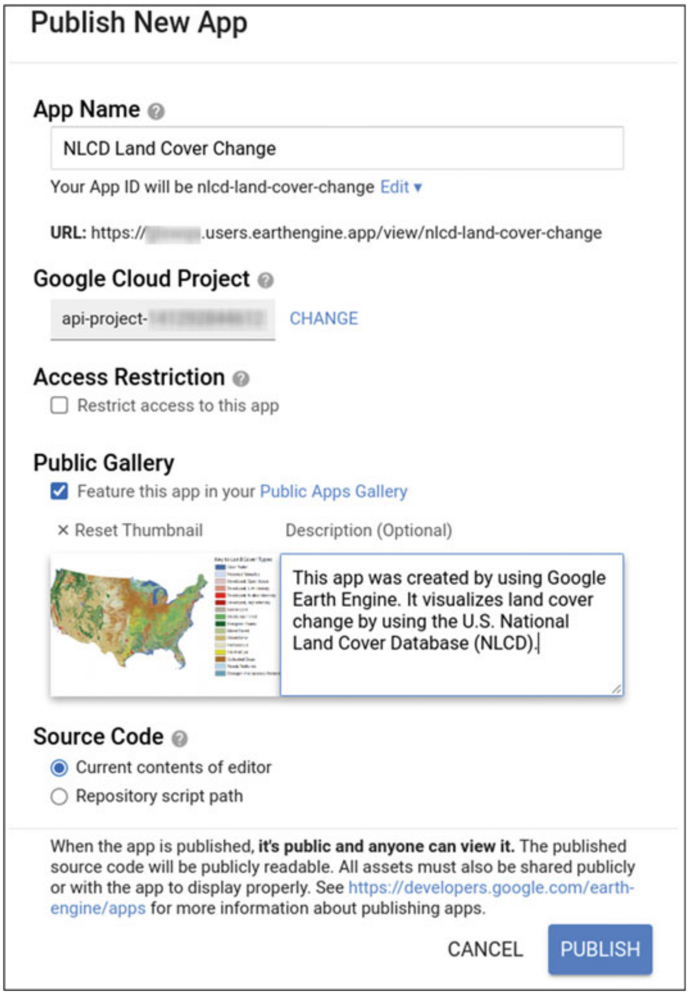


Fig. 30.4 **New App** button

be shown in the Public Apps Gallery, and specify the location of the App's source code. You may restrict access to the App to a particular Google Group or make it publicly accessible. Check **Feature this app in your Public Apps Gallery** if you would like this App to appear in your public gallery of Apps available at https://USERNAME.users.earthengine.app. When all fields are filled out and validated, the **Publish** button will be enabled; click it to complete publishing the App.

To manage an App from the Code Editor, open the **Manage Apps** panel (Fig. 30.6) by clicking the **Apps** button above the script panel in the Code Editor (Fig. 30.3). There, you can update your App's configuration or delete the App.



Publish New App

App Name

NLCD Land Cover Change

Your App ID will be nlcd-land-cover-change Edit ▾

URL: https://[redacted].users.earthengine.app/view/nlcd-land-cover-change

Google Cloud Project

api-project-[redacted] CHANGE

Access Restriction

☐ Restrict access to this app

Public Gallery

☑ Feature this app in your Public Apps Gallery

× Reset Thumbnail Description (Optional)

This app was created by using Google Earth Engine. It visualizes land cover change by using the U.S. National Land Cover Database (NLCD).

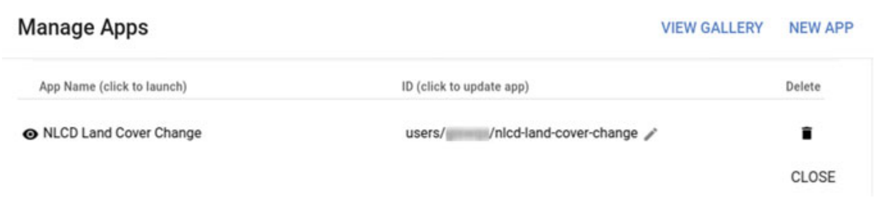
Source Code

◉ Current contents of editor

○ Repository script path

When the app is published, **it's public and anyone can view it.** The published source code will be publicly readable. All assets must also be shared publicly or with the app to display properly. See https://developers.google.com/earth-engine/apps for more information about publishing apps.

Fig. 30.5 **Publish New App** dialog



Manage Apps

App Name (click to launch) | ID (click to update app) | Delete
NLCD Land Cover Change | users/[redacted]/nlcd-land-cover-change

Fig. 30.6 **Manage Apps** panel

30.2.3 Section 3. Developing an Earth Engine App Using *geemap*

In this section, you will learn how to develop an Earth Engine App using the *geemap* Python package and Jupyter Notebook. The *geemap* package is available on both PyPI (pip) and conda-forge. It is highly recommended that you create a fresh conda environment to install *geemap*.

Code Checkpoint F63b. The book's repository contains information about setting up a conda environment and installing *geemap*.

Once you have launched a Jupyter Notebook in your browser, you can continue with the next steps of the lab.

On the Jupyter Notebook interface, click the **New** button in the upper-right corner and select **Python 3** to create a new notebook. Change the name of the notebook from "Untitled" to something meaningful (e.g., "nlcd_app"). With the newly created notebook, we can start writing and executing Python code.

First, let's import the Earth Engine and *geemap* libraries. Press Alt + Enter to execute the code and create a new cell below.

```
import ee
import geemap
```

Create an interactive map by specifying the map center (latitude, longitude) and zoom level (1–18). If this is the first time you use *geemap* and the Earth Engine Python API, a new tab will open in your browser asking you to authenticate Earth Engine. Follow the on-screen instructions to authenticate Earth Engine.

```
Map = geemap.Map(center=[40, -100], zoom=4)
Map
```

Retrieve the NLCD 2019 image by filtering the NLCD `ImageCollection` and selecting the `landcover` band. Display the NLCD 2019 image on the interactive **Map** by using `Map.addLayer`.

```
# Import the NLCD collection.
dataset =
ee.ImageCollection('USGS/NLCD_RELEASES/2019_REL/NLCD')

# Filter the collection to the 2019 product.
nlcd2019 = dataset.filter(ee.Filter.eq('system:index',
'2019')).first()

# Select the land cover band.
landcover = nlcd2019.select('landcover')

# Display land cover on the map.
Map.addLayer(landcover, {}, 'NLCD 2019')
Map
```

Next, add the NLCD legend to the **Map**. The *geemap* package has several built-in legends, including the NLCD legend. Therefore, you can add the NLCD legend to the **Map** by using just one line of code (`Map.add_legend`).

```
title = 'NLCD Land Cover Classification'
Map.add_legend(legend_title=title, builtin_legend='NLCD')
```

Alternatively, if you want to add a custom legend, you can define a legend dictionary with labels as keys and colors as values, then you can use `Map.add_legend` to add the custom legend to the **Map**.

```
legend_dict = {
    '11 Open Water': '466b9f',
    '12 Perennial Ice/Snow': 'd1def8',
    '21 Developed, Open Space': 'dec5c5',
    '22 Developed, Low Intensity': 'd99282',
    '23 Developed, Medium Intensity': 'eb0000',
    '24 Developed High Intensity': 'ab0000',
    '31 Barren Land (Rock/Sand/Clay)': 'b3ac9f',
    '41 Deciduous Forest': '68ab5f',
    '42 Evergreen Forest': '1c5f2c',
    '43 Mixed Forest': 'b5c58f',
    '51 Dwarf Scrub': 'af963c',
    '52 Shrub/Scrub': 'ccb879',
    '71 Grassland/Herbaceous': 'dfdfc2',
    '72 Sedge/Herbaceous': 'd1d182',
    '73 Lichens': 'a3cc51',
    '74 Moss': '82ba9e',
    '81 Pasture/Hay': 'dcd939',
    '82 Cultivated Crops': 'ab6c28',
    '90 Woody Wetlands': 'b8d9eb',
    '95 Emergent Herbaceous Wetlands': '6c9fb8'
}
title = 'NLCD Land Cover Classification'
Map.add_legend(legend_title=title, legend_dict=legend_dict)
```

The **Map** with the NLCD 2019 image and legend should look like Fig. 30.7.

The **Map** above shows only the NLCD 2019 image. To create an Earth Engine App for visualizing land cover change, we need a stack of NLCD images. Let's print the list of system IDs of all available NLCD images.

```
dataset.aggregate_array('system:id').getInfo()
```

The output should look like this.

```
['USGS/NLCD_RELEASES/2019_REL/NLCD/2001',
 'USGS/NLCD_RELEASES/2019_REL/NLCD/2004',
 'USGS/NLCD_RELEASES/2019_REL/NLCD/2006',
 'USGS/NLCD_RELEASES/2019_REL/NLCD/2008',
 'USGS/NLCD_RELEASES/2019_REL/NLCD/2011',
 'USGS/NLCD_RELEASES/2019_REL/NLCD/2013',
 'USGS/NLCD_RELEASES/2019_REL/NLCD/2016',
 'USGS/NLCD_RELEASES/2019_REL/NLCD/2019']
```

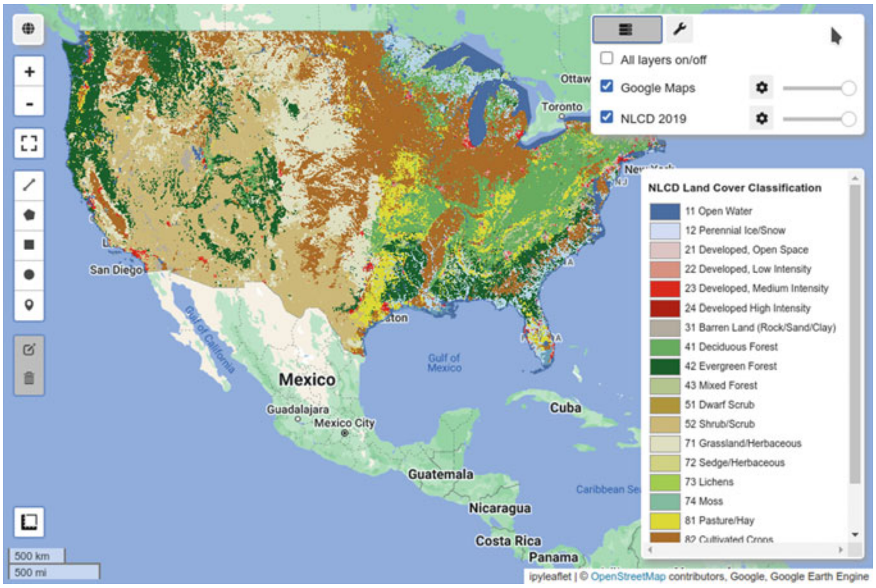


Fig. 30.7 NLCD 2019 image layer displayed in *geemap*

Select the eight NLCD epochs after 2000.

```
years = ['2001', '2004', '2006', '2008', '2011', '2013',
'2016', '2019']
```

Define a function for filtering the NLCD `ImageCollection` by year and select the `'landcover'` band.

```
# Get an NLCD image by year.
def getNLCD(year):
    # Import the NLCD collection.
    dataset =
ee.ImageCollection('USGS/NLCD_RELEASES/2019_REL/NLCD')

    # Filter the collection by year.
    nlcd = dataset.filter(ee.Filter.eq('system:index',
year)).first()

    # Select the land cover band.
    landcover = nlcd.select('landcover');
    return landcover
```

Create an NLCD `ImageCollection` to be used in the split-panel map.

```
# Create an NLCD image collection for the selected years.
collection = ee.ImageCollection(ee.List(years).map(lambda
year: getNLCD(year)))
```

Print out the list of system IDs of the selected NLCD images covering the contiguous United States.

```
collection.aggregate_array('system:id').getInfo()
```

The output should look like this.

```
['USGS/NLCD_RELEASES/2019_REL/NLCD/2001',
 'USGS/NLCD_RELEASES/2019_REL/NLCD/2004',
 'USGS/NLCD_RELEASES/2019_REL/NLCD/2006',
 'USGS/NLCD_RELEASES/2019_REL/NLCD/2008',
 'USGS/NLCD_RELEASES/2019_REL/NLCD/2011',
 'USGS/NLCD_RELEASES/2019_REL/NLCD/2013',
 'USGS/NLCD_RELEASES/2019_REL/NLCD/2016',
 'USGS/NLCD_RELEASES/2019_REL/NLCD/2019']
```

Next, create a list of labels to populate the dropdown list.

```
labels = [f'NLCD {year}' for year in years]
labels
```

The output should look like this.

```
['NLCD 2001',
 'NLCD 2004',
 'NLCD 2006',
 'NLCD 2008',
 'NLCD 2011',
 'NLCD 2013',
 'NLCD 2016',
 'NLCD 2019']
```

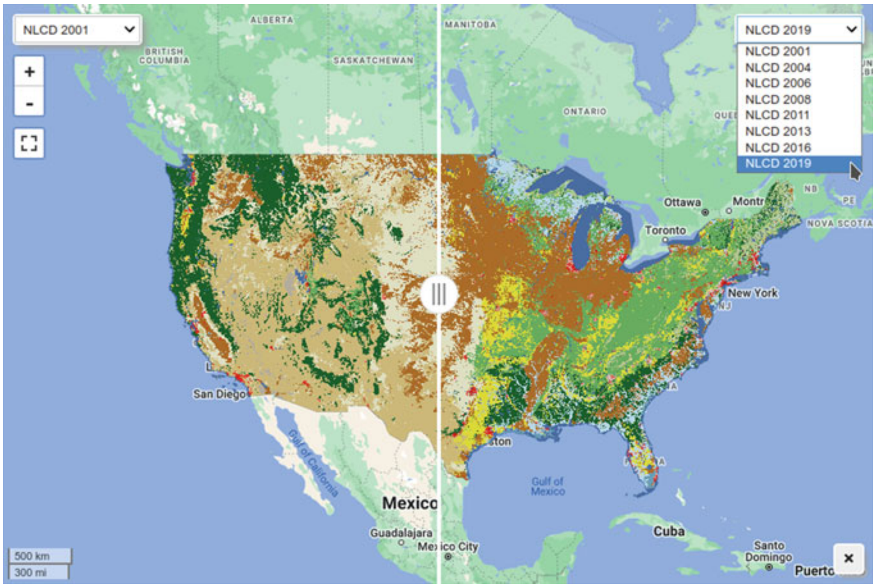


Fig. 30.8 Split-panel map for visualizing land cover change with *geemap*

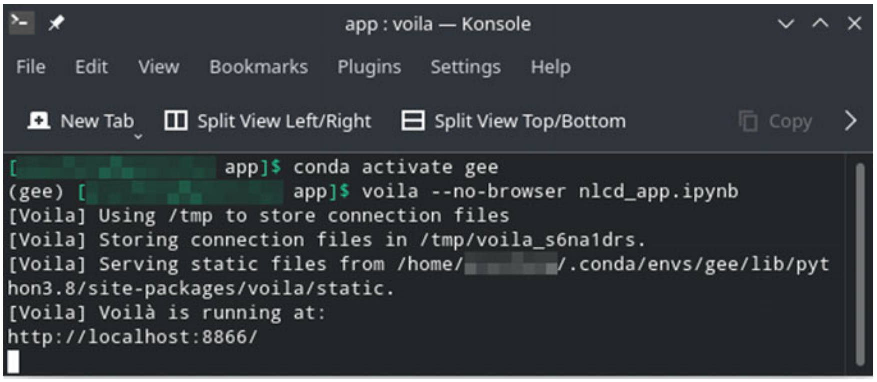
The last step is to create a split-panel map by passing the NLCD `ImageCollection` and list of labels to `Map.ts_inspector`.

```
Map.ts_inspector(left_ts=collection, right_ts=collection,
left_names=labels, right_names=labels)
Map
```

The split-panel map should look like Fig. 30.8.

To visualize land cover change, choose one NLCD image from the left drop-down list and another image from the right dropdown list, then use the slider to swipe through to visualize land cover change interactively. Click the **close** button in the bottom-right corner to close the split-panel map and return to the NLCD 2019 image shown in Fig. 30.7.

Code Checkpoint F63c. The book's repository contains information about what your code should look like at this point.



```
[[redacted] app]$ conda activate gee
(gee) [[redacted] app]$ voila --no-browser nlcd_app.ipynb
[Voila] Using /tmp to store connection files
[Voila] Storing connection files in /tmp/voila_s6na1drs.
[Voila] Serving static files from /home/[redacted]/.conda/envs/gee/lib/pyt
hon3.8/site-packages/voila/static.
[Voila] Voilà is running at:
http://localhost:8866/
```

Fig. 30.9 Output of the terminal running Voilà

30.2.4 Section 4. Publishing an Earth Engine App Using a Local Web Server

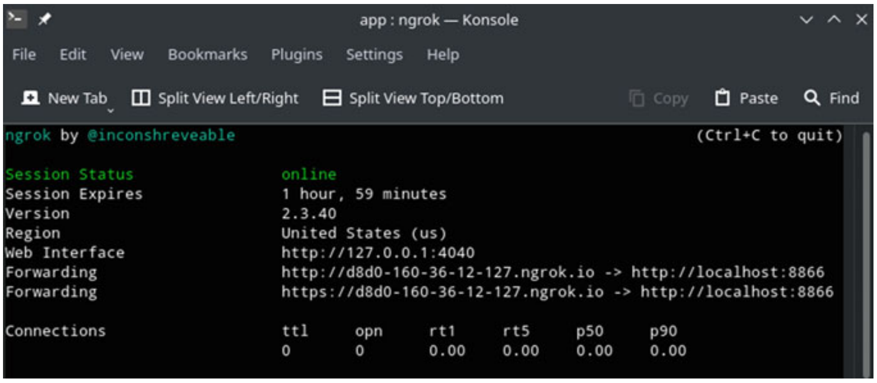
In this section, you will learn how to deploy an Earth Engine App using a local computer as a web server. Assume that you have completed Sect. 30.2.3 and created a Jupyter Notebook named `nlcd_app.ipynb`. First, you need to download *ngrok*, a program that can turn your computer into a secure web server and connect it to the *ngrok* cloud service, which accepts traffic on a public address. Download *ngrok* from https://ngrok.com and unzip it to a directory on your computer, then copy `nlcd_app.ipynb` to the same directory. Open the Anaconda Prompt (on Windows) or the Terminal (on macOS/Linux) and enter the following commands. Make sure you change `/path/to/ngrok/dir` to your computer directory where the *ngrok* executable is located, e.g., `~/Downloads`.

```
cd /path/to/ngrok/dir
conda activate gee
voila --no-browser nlcd_app.ipynb
```

The output of the terminal should look like this.

Voilà can be used to run, convert, and serve a Jupyter Notebook as a standalone app. Click the link (e.g., http://localhost:8866) shown in the terminal window to launch the interactive dashboard. Note that the port number is 8866, which is needed in the next step to launch *ngrok*. Open another terminal and enter the following command.

```
cd /path/to/ngrok/dir
ngrok http 8866
```



```
ngrok by @inconshreveable                                        (Ctrl+C to quit)

Session Status                online
Session Expires               1 hour, 59 minutes
Version                       2.3.40
Region                        United States (us)
Web Interface                 http://127.0.0.1:4040
Forwarding                    http://d8d0-160-36-12-127.ngrok.io -> http://localhost:8866
Forwarding                    https://d8d0-160-36-12-127.ngrok.io -> http://localhost:8866

Connections                   ttl     opn     rt1     rt5     p50     p90
                              0       0       0.00    0.00    0.00    0.00
```

Fig. 30.10 Output of the terminal running *ngrok*

The output of the terminal should look like Fig. 30.10. Click the link shown in the terminal window to launch the interactive dashboard. The link should look like https://random-string.ngrok.io, which is publicly accessible. Anyone with the link will be able to launch the interactive dashboard and use the split-panel map to visualize NLCD land cover change. Keep in mind that the dashboard might take several seconds to load, so please be patient.

To stop the web server, press Ctrl+C on both terminal windows. See below for some optional settings for running Voilà and *ngrok*.

To show code cells from your App, run the following from the terminal.

```
voila --no-browser --strip_sources=False nlcd_app.ipynb
```

To protect your App with a password, run the following.

```
ngrok http -auth="username:password" 8866
```

30.2.5 Section 5. Publish an Earth Engine App Using Cloud Platforms

In this section, you will learn how to deploy an Earth Engine App on cloud platforms, such as Heroku and Google App Engine. Heroku is a "platform as a service" that enables developers to build, run, and operate applications entirely in the cloud. It has a free tier with limited computing hours, which would be sufficient for this lab. Follow the steps below to deploy the Earth Engine App on Heroku.

First, go to https://github.com/signup to sign up for a GitHub account if you do not have one already. Once your GitHub account has been created, log into your account and navigate to the sample app repository: https://github.com/giswqs/earthengine-apps. Click the **Fork** button in the top-right corner to fork this repository into your account. Two important files in the repository are worth mentioning here: requirements.txt lists the required packages (e.g., *geemap*) to run the App, while Procfile specifies the commands that are executed by the App on startup. The content of Procfile should look like this.

```
web: voila --port=$PORT --no-browser --strip_sources=True -
-enable_nbextensions=True --
MappingKernelManager.cull_interval=60 --
MappingKernelManager.cull_idle_timeout=120 notebooks/
```

The above command instructs the server to hide the source code and periodically check for idle kernels—in this example, every 60s—and cull them if they have been idle for more than 120s. This can avoid idle kernels using up the server computing resources. The page served by Voilà will contain a list of any notebooks in the notebooks directory.

Next, go to https://signup.heroku.com to sign up for a Heroku account if you do not have one already. Log into your account and click the **New** button in the top-right corner, then choose **Create new app** from the dropdown list (Fig. 30.11).

Choose a name for your App (Fig. 30.12). Note that the App name must be unique; if an App name has already been taken, you will not be able to use it.

Once the Heroku App has been created, click the **Deploy** tab and choose **GitHub** as the deployment method. Connect to your GitHub account and enter `earthengine-apps` in the search box. The repository should be listed beneath the search box. Click the **Connect** button to connect the repository to Heroku (Fig. 30.13).

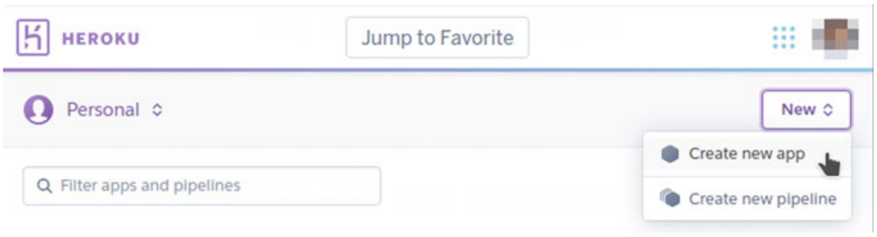


Fig. 30.11 Creating a new app in Heroku

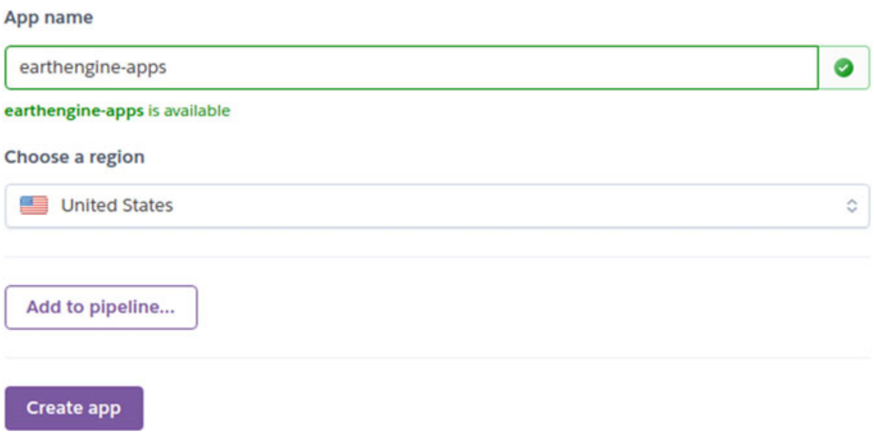


Fig. 30.12 Choosing an App name

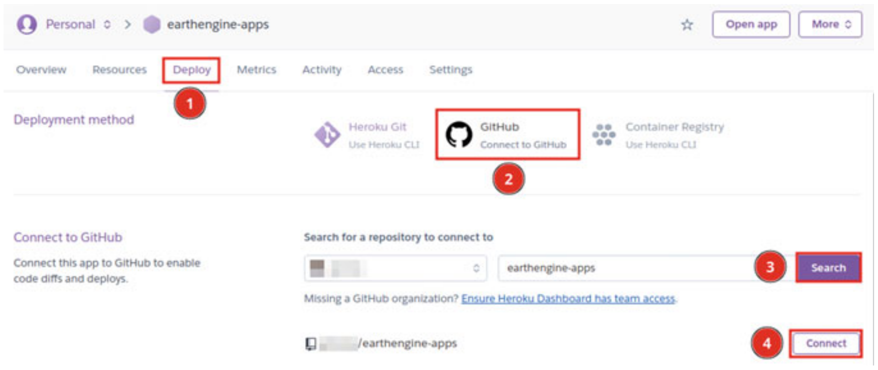
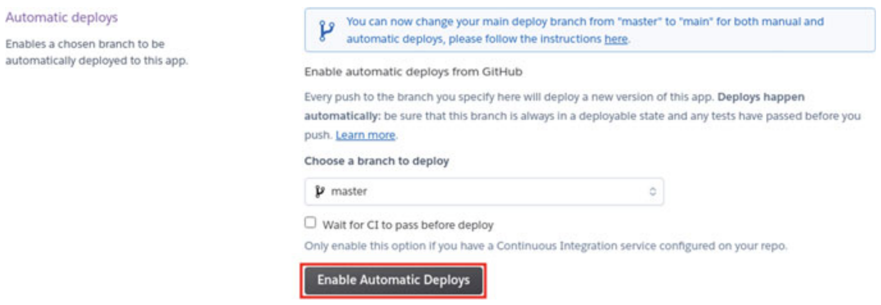


Fig. 30.13 Connecting a GitHub account to Heroku

Under the same **Deploy** tab, scroll down and click **Enable Automatic Deploys** (Fig. 30.14). This will enable Heroku to deploy a new version of the App whenever the GitHub repository gets updated.

Since using Earth Engine requires authentication, we need to set the Earth Engine token as an environment variable so that the web App can pass the Earth Engine authentication. If you have completed Sect. 30.2.3, you have successfully authenticated Earth Engine on your computer and the token can be found in the following file path, depending on the operating system you are using. Note that you might need to show the hidden directories on your computer in order to see the `.config` folder under the home directory.



Automatic deploys

Enables a chosen branch to be automatically deployed to this app.

You can now change your main deploy branch from "master" to "main" for both manual and automatic deploys, please follow the instructions here.

Enable automatic deploys from GitHub

Every push to the branch you specify here will deploy a new version of this app. **Deploys happen automatically:** be sure that this branch is always in a deployable state and any tests have passed before you push. Learn more.

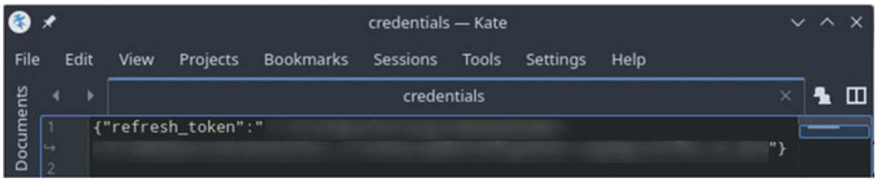
Choose a branch to deploy

master

☐ Wait for CI to pass before deploy

Only enable this option if you have a Continuous Integration service configured on your repo.

Fig. 30.14 Enabling Automatic Deploys on Heroku



```
{"refresh_token":"[redacted]"}
```

Fig. 30.15 Earth Engine authentication token

```
Windows: C:\Users\USERNAME\.config\earthengine\credentials
Linux: /home/USERNAME/.config/earthengine/credentials
MacOS: /Users/USERNAME/.config/earthengine/credentials
```

Open the **credentials** file using a text editor. Select and copy the long string wrapped by the double quotes (Fig. 30.15). Do not include the double quotes in the copied string.

Next, navigate to your web App on Heroku. Click the **Settings** tab, then click **Reveal Config Vars** on the page. Enter `EARTHENGINE_TOKEN` as the key and paste the string copied above as the value. Click the **Add** button to set `EARTHENGINE_TOKEN` as an environment variable (Fig. 30.16) that will be used by *geemap* to authenticate Earth Engine.

The last step is to commit some changes to your forked GitHub repository to trigger Heroku to build and deploy the App. If you are familiar with Git commands, you can push changes to the repository from your local computer to GitHub. If you have not used Git before, you can navigate to your repository on GitHub and make changes directly using the browser. For example, you can navigate to README.md and click the Edit icon on the page to start editing the file. Simply place the cursor at the end of the file and press Enter to add an empty line to the file, then click the **Commit changes** button at the bottom of the page to save changes. This should trigger Heroku to build the App. Check the build status under the latest activities of the **Overview** tab. Once the App has been

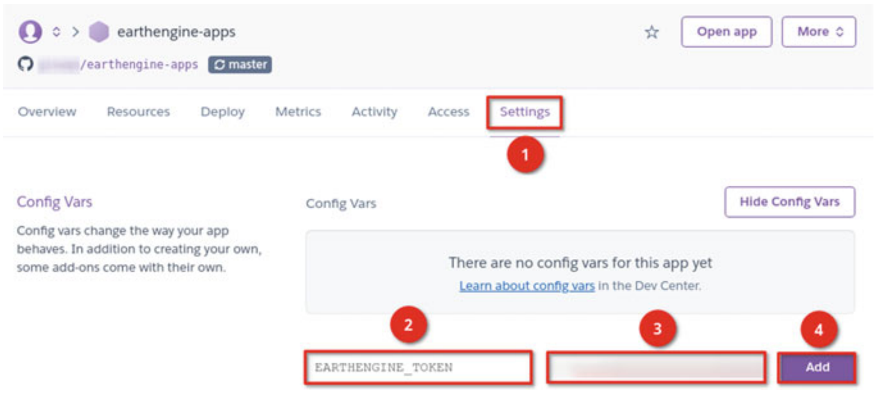

earthengine-apps
/earthengine-apps
master
Open app
More
Overview
Resources
Deploy
Metrics
Activity
Access
Settings
1
Config Vars
Config vars change the way your app behaves. In addition to creating your own, some add-ons come with their own.
Config Vars
Hide Config Vars
There are no config vars for this app yet
Learn about config vars in the Dev Center.
2
3
4
EARTHENGINE_TOKEN
Add

Fig. 30.16 Setting the Earth Engine token as an environment variable on Heroku

built and deployed successfully, you can click the **Open app** button in the top-right corner to launch the web App (Fig. 30.17). When the App is open in your browser, click `nlcd_app.ipynb` to launch the split-panel map for visualizing land cover change. This is the same notebook that we developed in Sect. 30.2.3. The App URL should look like this: https://APP-NAME.herokuapp.com/voila/render/nlcd_app.ipynb.

Congratulations! You have successfully deployed the Earth Engine App on Heroku.

Code Checkpoint F63d. The book's repository contains information about what your code should look like at this point.

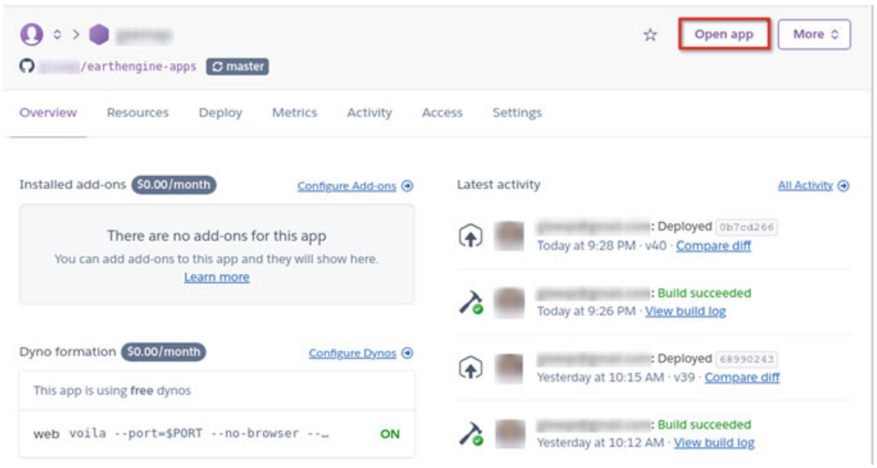

/earthengine-apps
master
Open app
More
Overview
Resources
Deploy
Metrics
Activity
Access
Settings
Installed add-ons $0.00/month
Configure Add-ons
There are no add-ons for this app
You can add add-ons to this app and they will show here.
Learn more
Dyno formation $0.00/month
Configure Dynos
This app is using free dynos
web voila --port=$PORT --no-browser --…
ON
Latest activity
All Activity
: Deployed 0b7cd266
Today at 9:28 PM · v40 · Compare diff
: Build succeeded
Today at 9:26 PM · View build log
: Deployed 68990243
Yesterday at 10:15 AM · v39 · Compare diff
: Build succeeded
Yesterday at 10:12 AM · View build log

Fig. 30.17 Clicking on the **Open app** button to launch the web app

Question 1. What are the pros and cons of designing Earth Engine Apps using *geemap* and *ipywidgets*, compared to the JavaScript Earth Engine User Interface API?

Question 2. What are the pros and cons of deploying Earth Engine Apps on Heroku, compared to a local web server and the Earth Engine Code Editor?

30.3 Synthesis

Assignment 1. Replace the NLCD datasets with other multitemporal land cover datasets (e.g., United States Department of Agriculture National Agricultural Statistics Service Cropland Data Layers, visible in the Earth Engine Data Catalog) and modify the web App for visualizing land cover change using the chosen land cover datasets. Deploy the web App using multiple platforms (i.e., JavaScript Code Editor, *ngrok*, and Heroku). More land cover datasets can be found at https://developers.google.com/earth-engine/datasets/tags/landcover.

30.4 Conclusion

In this chapter, you learned how to design Earth Engine Apps using both the Earth Engine User Interface API (JavaScript) and *geemap* (Python). You also learned how to deploy Earth Engine Apps on multiple platforms, such as the JavaScript Code Editor, a local web server, and Heroku. The skill of designing and deploying interactive Earth Engine Apps is essential for making your research and data products more accessible to the scientific community and the general public. Anyone with the link to your web App can analyze and visualize Earth Engine datasets without needing an Earth Engine account.

References

Earth Engine User Interface API: https://developers.google.com/earth-engine/guides/ui
Earth Engine Apps: https://developers.google.com/earth-engine/guides/apps
Voilà: https://voila.readthedocs.io
geemap: https://geemap.org
ngrok: https://ngrok.com
Heroku: https://heroku.com
Earthengine-apps: https://github.com/giswqs/earthengine-apps

Combining R and Earth Engine

31

Cesar Aybar, David Montero, Antony Barja, Fernando Herrera, Andrea Gonzales, and Wendy Espinoza

Overview

The purpose of this chapter is to introduce *rgee*, a non-official API for Earth Engine. You will explore the main features available in *rgee* and how to set up an environment that integrates *rgee* with third-party R and Python packages. After this chapter, you will be able to combine R, Python, and JavaScript in the same workflow.

C. Aybar (✉)
Z_GIS, University of Salzburg, Salzburg, Austria
e-mail: csaybar@gmail.com

D. Montero
RSC4Earth, University of Leipzig, Leipzig, Germany
e-mail: david.montero@uni-leipzig.de

A. Barja
Health Innovation Laboratory, Cayetano Heredia University, Lima, Peru
e-mail: antony.barja@upch.pe

F. Herrera · A. Gonzales · W. Espinoza
National University of San Marcos, Lima, Peru
e-mail: fernando.herrera4@unmsm.edu.pe

A. Gonzales
e-mail: karen.gonzales1@unmsm.edu.pe

W. Espinoza
e-mail: wendyjimena.espinoza@unmsm.edu.pe

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_31

Learning Outcomes

- Becoming familiar with *rgee*, the Earth Engine R API interface.
- Integrating *rgee* with other R packages.
- Displaying interactive maps.
- Integrating Python and R packages using *reticulate*.
- Combining Earth Engine JavaScript and Python APIs with R.

Assumes you know how to

- Install the Python environment (Chap. 30).
- Use the `require` function to load code from existing modules (Chap. 28).
- Use the basic functions and logic of Python.
- Configure an environment variable and use.Renviron files.
- Create Python virtual environments.

31.1 Introduction to Theory

R is a popular programming language established in statistical science with large support in reproducible research, geospatial analysis, data visualization, and much more. To get started with R, you will need to satisfy some extra software requirements. First, install an up-to-date R version (at least 4.0) in your work environment. The installation procedure will vary depending on your operating system (i.e., Windows, Mac, or Linux). *Hands-On Programming with R* (Garrett Grolemund 2014, Appendix A) explains step by step how to proceed. We strongly recommend that Windows users install *Rtools* to avoid compiling external static libraries.

If you are new to R, a good starting point is the book *Geocomputation with R* (Lovelace et al. 2019) or *Spatial Data Science with Application in R* (Pebesma and Bivand 2019). In addition, we recommend using an integrated development environment (e.g., Rstudio) or a Code Editor (e.g., Visual Studio Code) to create a suitable setting to display and interact with R objects.

The following R packages must be installed (find more information in the R manual) in order to go through the practicum section.

```
# Use install.packages to install R packages from the CRAN
repository.
install.packages('reticulate') # Connect Python with R.
install.packages('rayshader') # 2D and 3D data
visualizations in R.
install.packages('remotes') # Install R packages from
remote repositories.
remotes::install_github('r-earthengine/rgeeExtra') # rgee
extended.
install.packages('rgee') # GEE from within R.
install.packages('sf') # Simple features in R.
install.packages('stars') # Spatiotemporal Arrays and
Vector Data Cubes.
install.packages('geojsonio') # Convert data to 'GeoJSON'
from various R classes.
install.packages('raster') # Reading, writing,
manipulating, analyzing and modeling of spatial data.
install.packages('magick') # Advanced Graphics and Image-
Processing in R
install.packages('leaflet.extras2') # Extra Functionality
for leaflet
install.packages('cptcity') # colour gradients from the
'cpt-city' web archive
```

Earth Engine officially supports client libraries only for the JavaScript and Python programming languages. While the Earth Engine Code Editor offers a convenient environment for rapid prototyping in JavaScript, the lack of a mechanism for integration with local environments makes the development of complex scripts tricky. On the other hand, the Python client library offers much versatility, enabling support for third-party packages. However, not all Earth and environmental scientists code in Python. Hence, a significant number of professionals are not members of the Earth Engine community. In the R ecosystem, *rgee* (Aybar et al. 2020) tries to fill this gap by wrapping the Earth Engine Python API via *reticulate*. The *rgee* package extends and supports all the Earth Engine classes, modules, and functions, working as fast as the other APIs.

Figure 31.1 illustrates how *rgee* bridges the Earth Engine platform with the R ecosystem. When an Earth Engine request is created in R, *rgee* transforms this piece of code into Python. Next, the Earth Engine Python API converts the Python code into JSON. Finally, the JSON file request is received by the server through a Web REST API. Users could get the response using the `getInfo` method by following the same path in reverse.

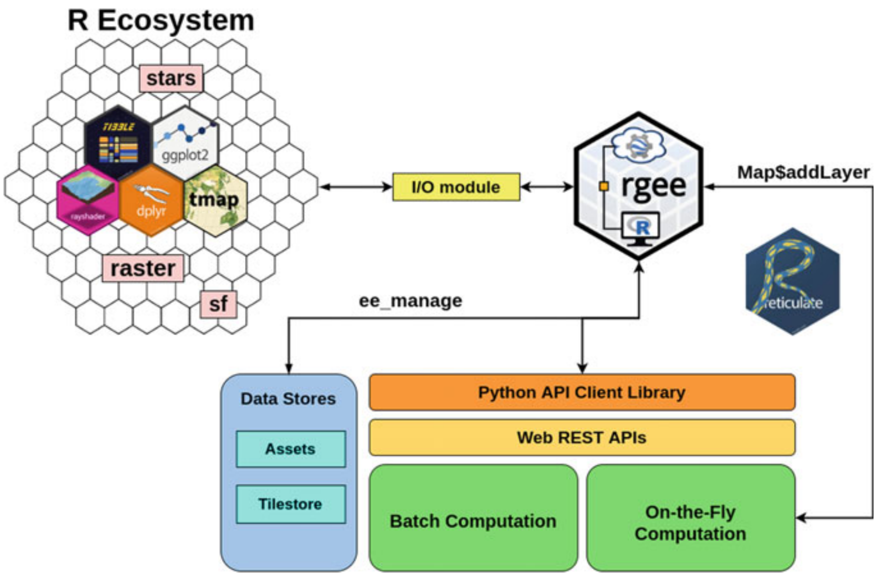


Fig. 31.1 Simplified diagram of *rgee* functionality

31.2 Practicum

31.2.1 Section 1. Installing *rgee*

To run, *rgee* needs a Python environment with two packages: *NumPy* and *earthengine-api*. Because instructions change frequently, installation is explained at the following checkpoint:

Code Checkpoint F64a. The book's repository contains information about setting up the *rgee* environment.

After installing both the R and Python requirements, you can now initialize your Earth Engine account from within R. Consider that R, in contrast to JavaScript and Python, supports three distinct Google APIs: Earth Engine, Google Drive, and Google Cloud Storage (GCS).

The Google Drive and GCS APIs will allow you to transfer your Earth Engine completed task exports to a local environment automatically. In these practice sessions, we will use only the Earth Engine and Google Drive APIs. Users that are interested in GCS can look up and explore the *GCS vignette*. To initialize your Earth Engine account alongside Google Drive, use the following commands.

```
# Initialize just Earth Engine
ee_Initialize()

# Initialize Earth Engine and GD
ee_Initialize(drive = TRUE)
```

If the Google account is verified and the permission is granted, you will be directed to an authentication token. Copy and paste this token into your R console. Consider that the GCS API requires setting up credentials manually; look up and explore the *rgee* vignette for more information. The verification step is only required once; after that, *rgee* saves the credentials in your system.

Code Checkpoint F64b. The book's repository contains information about what your code should look like at this point.

31.2.2 Section 2. Creating a 3D Population Density Map with *rgee* and *rayshader*

First, import the *rgee*, *rayshader*, and *raster* packages.

```
library(rayshader)
library(raster)
library(rgee)
```

Initialize the Earth Engine and Google Drive APIs using `ee_Initialize`. Both credentials must come from the same Google account.

```
ee_Initialize(drive = TRUE)
```

Then, we will access the WorldPop Global Project Population Data dataset. In *rgee*, the Earth Engine spatial classes (`ee$Image`, `ee$ImageCollection`, and `ee$FeatureCollection`) have a special attribute called `Dataset`. Users can use it along with autocompletion to quickly find the desired dataset.

```
collections <- ee$ImageCollection$Dataset
population_data <-
collections$CIESIN_GPWv411_GPW_Population_Density
population_data_max <- population_data$max()
```

If you need more information about the Dataset, use ee_utils_dataset_display to go to the official documentation in the Earth Engine Data Catalog.

```
population_data %>% ee_utils_dataset_display()
```

The *rgee* package provides various built-in functions to retrieve data from Earth Engine (Aybar et al. 2020). In this example, we use ee_as_raster, which automatically converts an ee$Image (server object) into a RasterLayer (local object).

```
sa_extent <- ee$Geometry$Rectangle(
  coords = c(-100, -50, -20, 12),
  geodesic = TRUE,
  proj = "EPSG:4326"
)

population_data_ly_local <- ee_as_raster(
  image = population_data_max,
  region = sa_extent,
  dsn = "/home/pc-user01/population.tif", # change for your
own path.
  scale = 5000
)
```

Now, turn a RasterLayer into a matrix suitable for *rayshader*.

```
pop_matrix <- raster_to_matrix(population_data_ly_local)
```

Next, modify the matrix population density values, adding:

- Texture, based on five colors (lightcolor, shadowcolor, leftcolor, rightcolor, and centercolor; see rayshader::create_texture documentation)
- Color and shadows (rayshader::sphere_shade)

```
pop_matrix %>%
  sphere_shade(
    texture = create_texture("#FFFFFF", "#0800F0",
"#FFFFFF", "#FFFFFF", "#FFFFFF")
  ) %>%
  plot_3d(
    pop_matrix,
    zoom = 0.55, theta = 0, zscale = 100, soliddepth = -24,
    solidcolor = "#525252", shadowdepth = -40, shadowcolor
= "black",
    shadowwidth = 25, windowsize = c(800, 720)
  )
```

Lastly, define a title and subtitle for the plot. Use `rayshader::render_snapshot` to export the final results (Fig. 31.2).

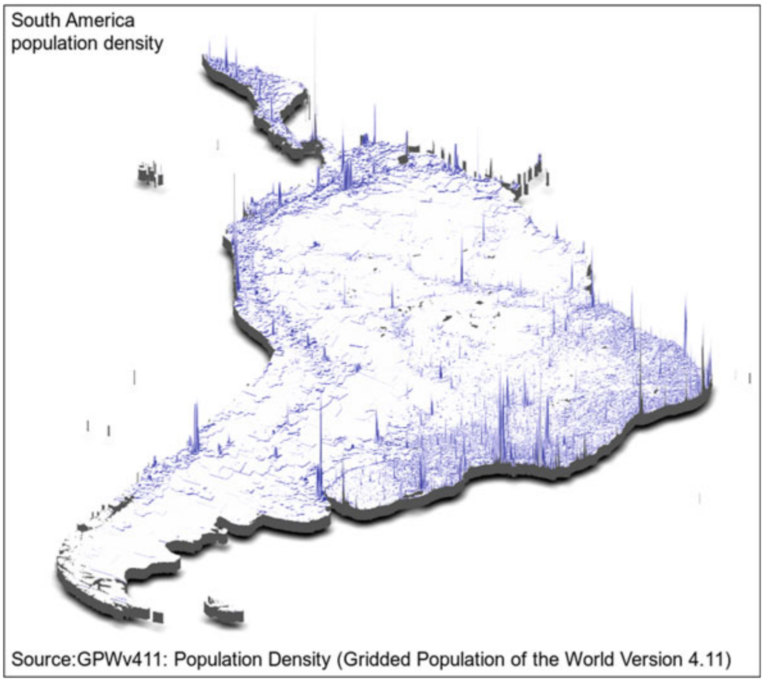

South America
population density
Source:GPWv411: Population Density (Gridded Population of the World Version 4.11)

Fig. 31.2 3D population density map of South America

```
text <- paste0(
  "South America\npopulation density",
  strrep("\n", 27),
  "Source:GPWv411: Population Density (Gridded Population
of the World Version 4.11)"
)

render_snapshot(
  filename = "30_poblacionsudamerica.png",
  title_text = text,
  title_size = 20,
  title_color = "black",
  title_font = "Roboto bold",
  clear = TRUE
)
```

Code Checkpoint F64c. The book's repository contains information about what your code should look like at this point.

31.2.3 Section 3. Displaying Maps Interactively

Similar to the Code Editor, *rgee* supports the interactive visualization of spatial Earth Engine objects by `Map$addLayer`. First, let's import the *rgee* and *cptcity* packages. The *cptcity* R package is a wrapper to the *cpt-city* color gradients web archive.

```
library(rgee)
library(cptcity)
ee_Initialize()
```

We will select an `ee$Image`; in this case, the Shuttle Radar Topography Mission 90m (SRTM-90) Version 4.

```
dem <- ee$Image$Dataset$CGIAR_SRTM90_V4
```

Then, we will set the visualization parameters as a list with the following elements.

- `min`: value(s) to map to 0
- `Max`: value(s) to map to 1

- `palette`: a list of CSS-style color strings.

```
viz <- list(
  min = 600,
  max = 6000,
  palette = cpt(pal = 'grass_elevation', rev = TRUE)
)
```

Then, we will create a simple display using `Map$addLayer`.

```
m1 <- Map$addLayer(dem, visParams = viz, name = "SRTM",
shown = TRUE)
```

Optionally, you could add a custom legend using `Map$addLayer` (Fig. 31.3).

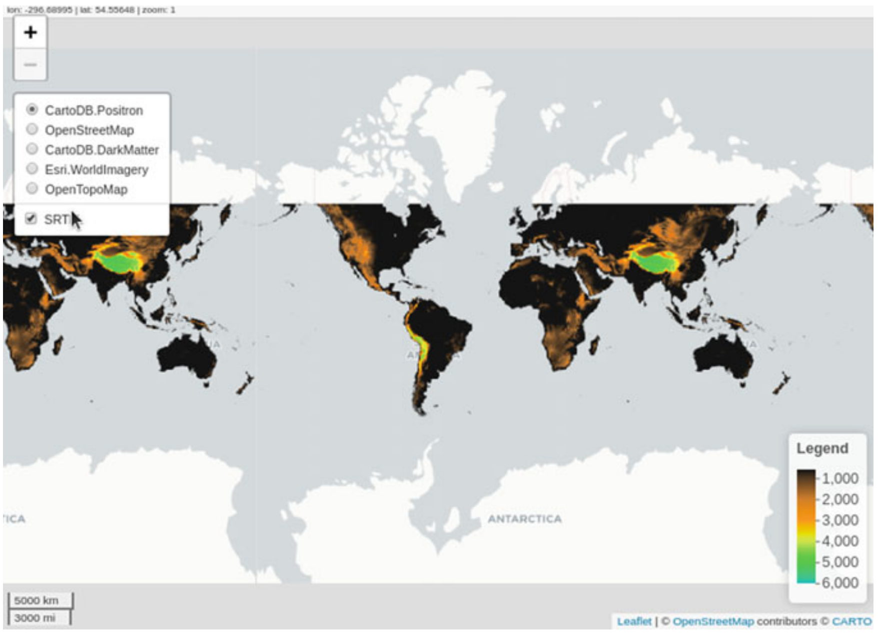


Fig. 31.3 Interactive visualization of SRTM-90 Version 4 elevation values

```
pal <- Map$addLegend(viz)
m1 + pal
```

The procedure to display `ee$Geometry`, `ee$Feature`, and `ee$FeatureCollections` objects is similar to the previous example effected on an `ee$Image`. Users just need to change the arguments: `eeObject` and `visParams`.

First, Earth Engine geometries (Fig. 31.4).

```
vector <- ee$Geometry$Point(-77.011,-11.98) %>%
  ee$Feature$buffer(50*1000)
Map$centerObject(vector)
Map$addLayer(vector) # eeObject is a ee$Geometry$Polygon.
```

Next, Earth Engine feature collections (Fig. 31.5).

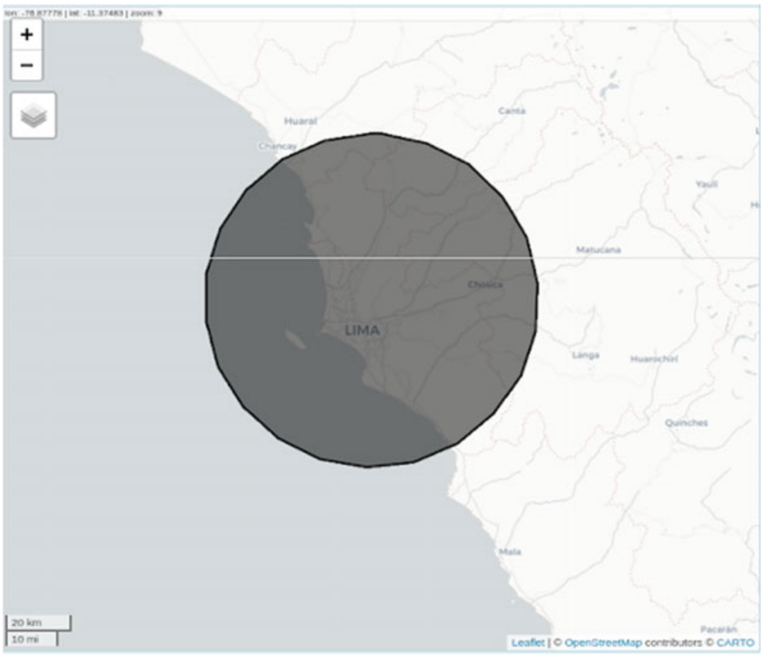


Fig. 31.4 Polygon buffer surrounding the city of Lima, Peru

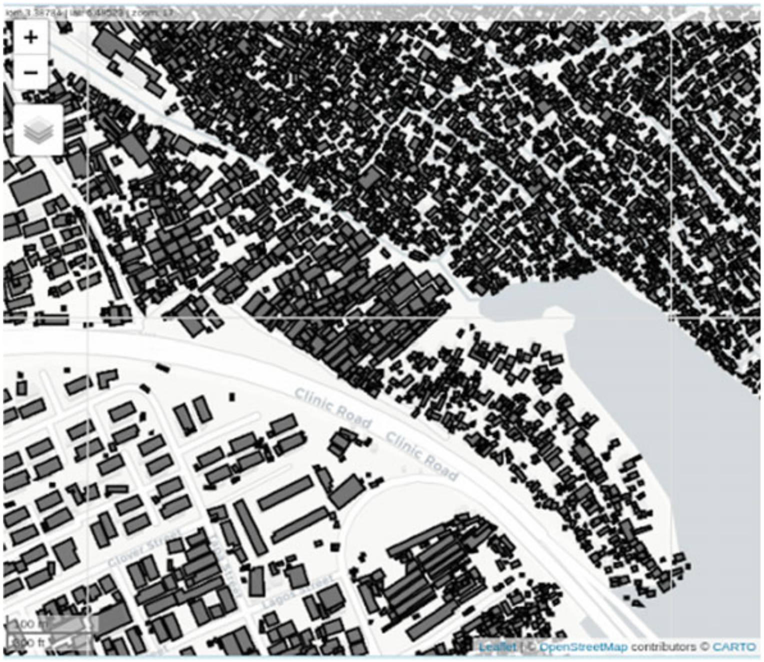


Fig. 31.5 Building footprints in Lagos, Nigeria

```
building <- ee$FeatureCollection$Dataset$
  `GOOGLE_Research_open-buildings_v1_polygon`
Map$setCenter(3.389, 6.492, 17)
Map$addLayer(building) # eeObject is a ee$FeatureCollection
```

The *rgee* functionality also supports the display of `ee$ImageCollection` via `Map$addLayers` (note the extra "s" at the end). `Map$addLayers` will use the same visualization parameters for all the images (Fig. 31.6). If you need different visualization parameters per image, use a `Map$addLayer` within a *for* loop.

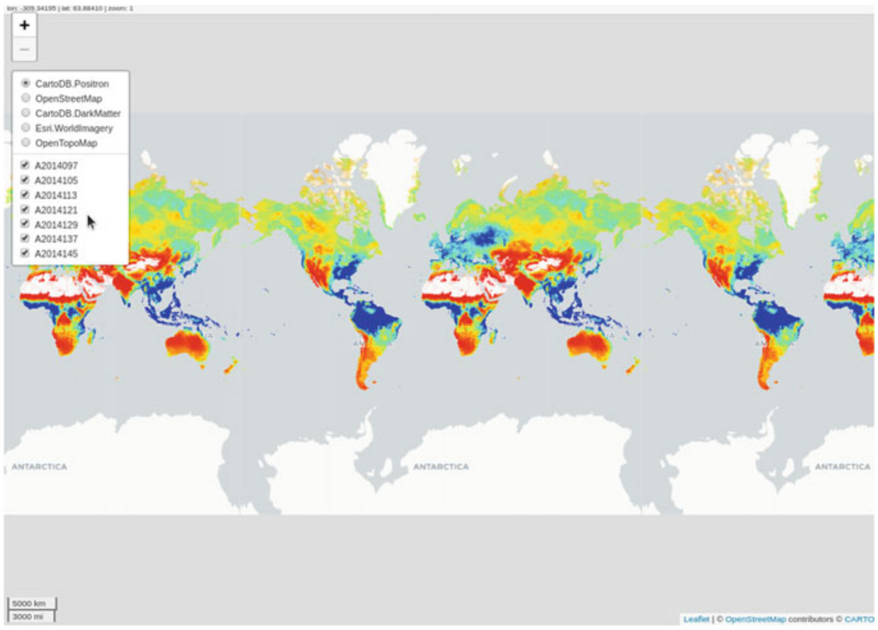


Fig. 31.6 MOD16A2 total evapotranspiration values (kg/m^2/8 day)

```
# Define a ImageCollection
etp <- ee$ImageCollection$Dataset$MODIS_NTSG_MOD16A2_105
%>%
  ee$ImageCollection$select("ET") %>%
  ee$ImageCollection$filterDate('2014-04-01', '2014-06-01')

# Set viz params
viz <- list(
  min = 0,  max = 300,
  palette = cpt(pal = "grass_bcyr", rev = TRUE)
)

# Print map results interactively
Map$setCenter(0, 0, 1)
etpmap <- Map$addLayers(etp, visParams = viz)
etpmap
```

Another useful *rgee* feature is the comparison operator (|), which creates a slider in the middle of the canvas, permitting quick comparison of two maps. For instance, load a Landsat 4 image:

```
landsat <- ee$Image('LANDSAT/LT04/C01/T1/LT04_008067_19890917')
```

Calculate the Normalized Difference Snow Index.

```
ndsi <- landsat$normalizedDifference(c('B3', 'B5'))
```

Define a constant value and use `ee$Image$gte` to return a binary image where pixels greater than or equal to that value are set as 1 and the rest are set as 0. Next, filter 0 values using `ee$Image$updateMask`.

```
ndsiMasked <- ndsi$updateMask(ndsi$gte(0.4))
```

Define the visualization parameters.

```
vizParams <- list(
  bands <- c('B5', 'B4', 'B3'), # vector of three bands (R,
G, B).
  min = 40,
  max = 240,
  gamma = c(0.95, 1.1, 1) # Gamma correction factor.
)

ndsiViz <- list(
  min = 0.5,
  max = 1,
  palette = c('00FFFF', '0000FF')
)
```

Center the map on the Huayhuash mountain range in Peru.

```
Map$setCenter(lon = -77.20, lat = -9.85, zoom = 10)
```

Finally, visualize both maps using the | operator (Fig. 31.7).

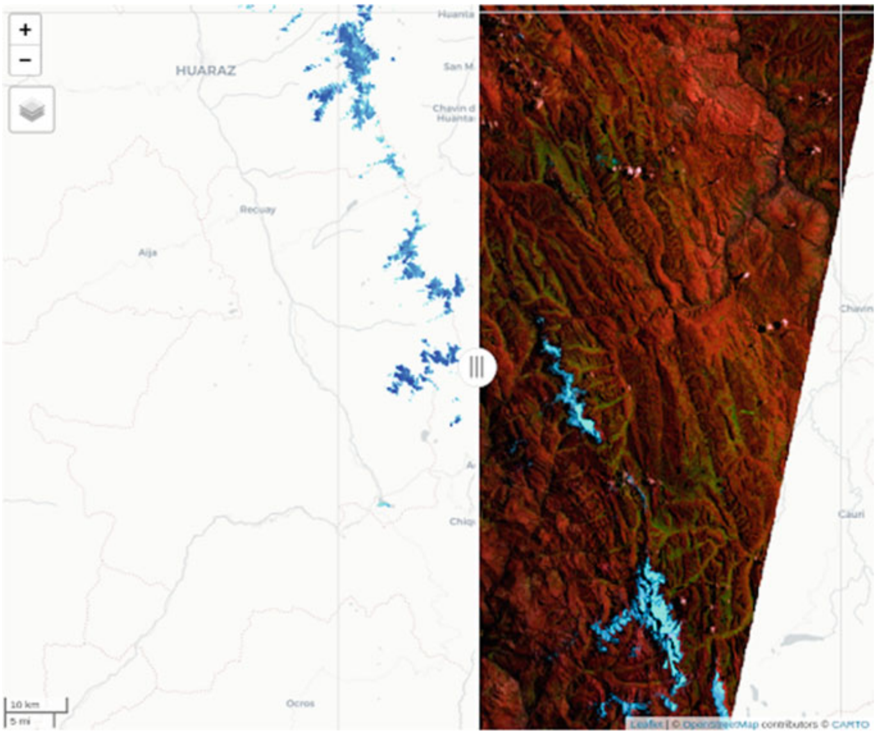


Fig. 31.7 False-color composite over the Huayhuash mountain range, Peru

```
m2 <- Map$addLayer(ndsiMasked, ndsiViz, 'NDSI masked')
m1 <- Map$addLayer(landsat, vizParams, 'false color
composite')
m2  | m1
```

Code Checkpoint F64d. The book's repository contains information about what your code should look like at this point.

31.2.4 Section 4. Integrating *rgee* with Other Python Packages

As noted in Sect. 31.2.1, *rgee* set up a Python environment with *NumPy* and *earthengine-api* in your system. However, there is no need to limit it to just two Python packages. In this section, you will learn how to use the Python package *ndvi2gif* to perform a Normalized Difference Vegetation Index (NDVI) multi-seasonal analysis in the Ocoña Valley without leaving R.

Whenever you want to install a Python package, you must run the following.

```
library(rgee)
library(reticulate)
ee_Initialize()
```

The `ee_Initialize` function not only authenticates your Earth Engine account but also helps *reticulate* to set up a Python environment compatible with *rgee*. After running `ee_Initialize`, use `reticulate::install_python` to install the desired Python package.

```
py_install("ndvi2gif")
```

The previous procedure is needed just once for each Python environment. Once installed, we simply load the package using `reticulate::import`.

```
ngif <- import("ndvi2gif")
```

Then, we define our study area using `ee$Geometry$Rectangle` (Fig. 31.8) and use the leaflet layers control to switch between basemaps.

```
colca <- c(-73.15315, -16.46289, -73.07465, -16.37857)
roi <- ee$Geometry$Rectangle(colca)
Map$centerObject(roi)
Map$addLayer(roi)
```

In *ndvi2gif*, there is just one class: `NdviSeasonality`. It has the following four public methods.

- `get_export`: Exports NDVI year composites in.GeoTIFF format to your local folder.
- `get_export_single`: Exports single composite as.GeoTIFF to your local folder.
- `get_year_composite`: Returns the NDVI composites for each year.
- `get_gif`: Exports NDVI year composites as a.gif to your local folder.

To run, the `NdviSeasonality` constructor needs to define the following arguments.

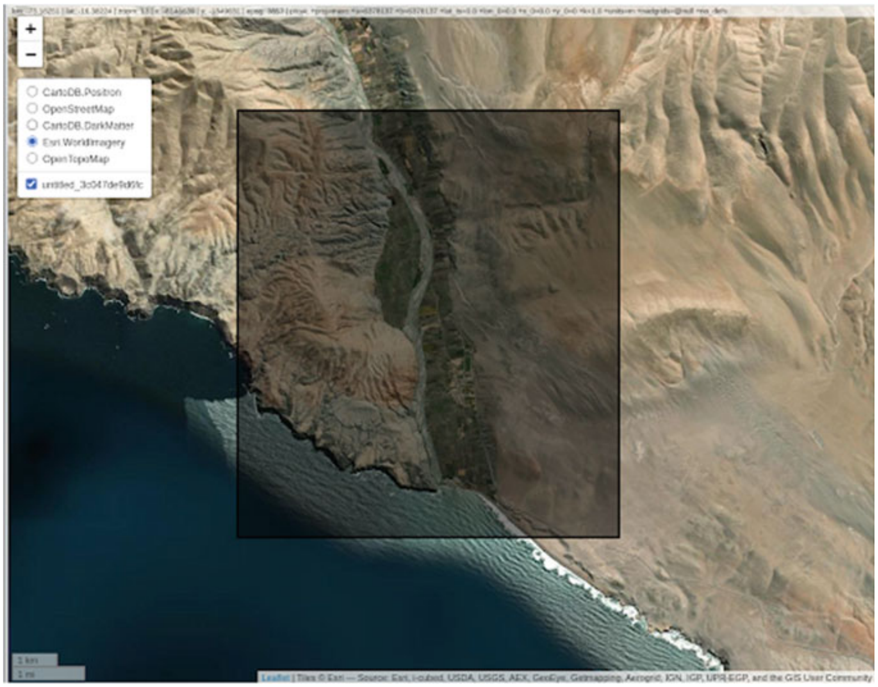


Fig. 31.8 Rectangle drawn over the Ocoña Valley, Peru

- `roi`: the region of interest
- `start_year`: the initial year to start to create yearly composites
- `end_year`: the end year to look for
- `sat`: the satellite sensor
- `key`: the aggregation rule that will be used to generate the yearly composite.

For each year, the `get_year_composite` method generates an NDVI `ee$Image` with four bands, one band per season. Color combination between images and bands will allow you to interpret the vegetation phenology over the seasons and years. In `ndvi2gif`, the seasons are defined as follows.

- winter = c('-01-01', '-03-31')
- spring = c('-04-01', '-06-30')
- summer = c('-07-01', '-09-30')
- autumn = c('-10-01', '-12-31').

```
myclass <- ngif$NdviSeasonality(
  roi = roi,
  start_year = 2016L,
  end_year = 2020L,
  sat = 'Sentinel', # 'Sentinel', 'Landsat', 'MODIS', 'sar'
  key = 'max' # 'max', 'median', 'perc_90'
)

# Estimate the median of the yearly composites from 2016 to
2020.
median <- myclass$get_year_composite()$median()

# Estimate the median of the winter season composites from
2016 to 2020.
wintermax <-
myclass$get_year_composite()$select('winter')$max()
```

We can display maps interactively using the `Map$addLayer` (Fig. 31.9) and use the leaflet layers control to switch between basemaps.

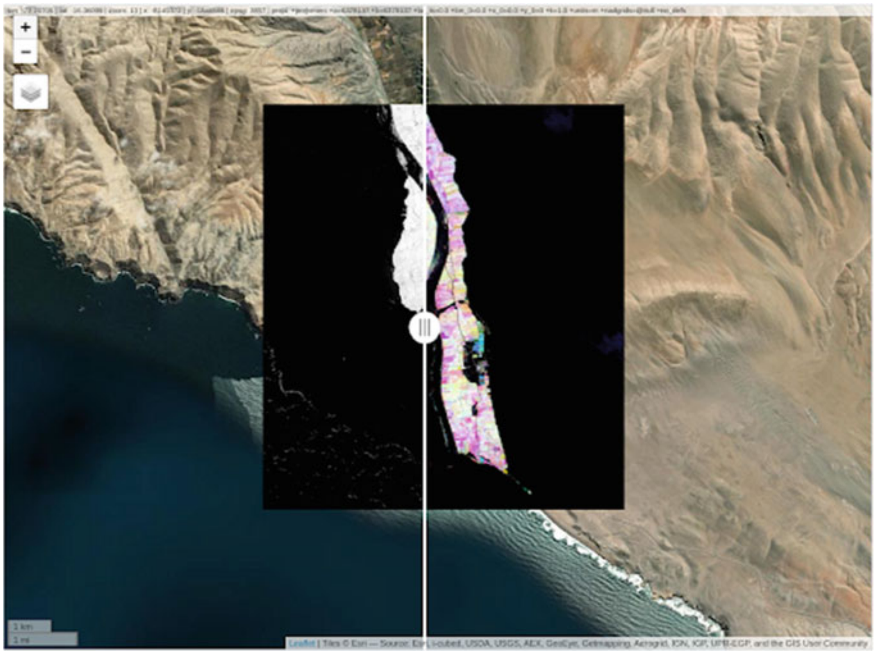


Fig. 31.9 Comparison between the maximum historic winter NDVI and the mean historic NDVI. Colors represent the season when the maximum value occurred

```
Map$addLayer(wintermax, list(min = 0.1, max = 0.8),
'winterMax') |
Map$addLayer(median, list(min = 0.1, max = 0.8), 'median')
```

And we can export the results to a GIF format.

```
myclass$get_gif()
```

To get more information about the *ndvi2gif* package, visit its GitHub repository.

Code Checkpoint F64e. The book's repository contains information about what your code should look like at this point.

31.2.5 Section 5. Converting JavaScript Modules to R

In recent years, the Earth Engine community has developed a lot of valuable third-party modules. Some incredible ones are *geeSharp*, *ee-palettes*, *spectral* (Montero 2021), and *LandsatLST* (Ermida et al. 2020). While some of these modules have been implemented in Python and JavaScript (e.g., *geeSharp* and *spectral*), most are available only for JavaScript. This is a critical drawback, because it divides the Earth Engine community by programming languages. For example, if an R user wants to use *tagee*, the user will have to first translate the entire module to R.

In order to close this breach, the *ee_extra* Python package has been developed to unify the Earth Engine community. The philosophy behind *ee_extra* is that all of its extended functions, classes, and methods must be functional for the JavaScript, Julia, R, and Python client libraries. Currently, *ee_extra* is the base of the *rgeeExtra* (Aybar et al. 2020) and *eemont* (Montero 2021) packages.

To demonstrate the potential of *ee_extra*, let's study an example from the Landsat Land Surface Temperature (LST) JavaScript module. The Landsat LST module computes the land surface temperature for Landsat products (Ermida et al. 2020). First, we will run it in the Earth Engine Code Editor; then, we will replicate those results in R.

First, JavaScript. In a new script in the Code Editor, we must `require` the Landsat LST module.

```
var LandsatLST = require(
'users/sofiaermida/landsat_smw_lst:modules/Landsat_LST.js')
;
```

The Landsat LST module contains a function named collection. This function receives the following parameters.

- The Landsat archive ID
- The starting date of the Landsat collection
- The ending date of the Landsat collection
- The region of interest as geometry
- A Boolean parameter specifying if we want to use the NDVI for computing a dynamic emissivity instead of using the emissivity from ASTER.

In the following code block, we are going to define all required parameters.

```
var geometry = ee.Geometry.Rectangle([-8.91, 40.0, -8.3,
40.4]);
var satellite = 'L8';
var date_start = '2018-05-15';
var date_end = '2018-05-31';
var use_ndvi = true;
```

Now, with all our parameters defined, we can compute the land surface temperature by using the collection method from Landsat LST.

```
var LandsatColl = LandsatLST.collection(satellite,
date_start, date_end, geometry, use_ndvi);
```

The result is stored as an `ImageCollection` in the `LandsatColl` variable. Now select the first element of the collection as an example by using the `first` method.

```
var exImage = LandsatColl.first();
```

This example image is now stored in a variable named ‘`exImage`’. Let’s display the LST result on the **Map**. For visualization purposes, we will define a color palette.

```
var cmap = ['blue', 'cyan', 'green', 'yellow', 'red'];
```

Then, we will center the map in the region of interest.

```
Map.centerObject(geometry);
```

Finally, let's display the LST with the cmap color palette by using the `Map.addLayer` method (Fig. 31.10). This method receives the image to visualize, the visualization parameters, the color palette, and the name of the layer to show in the layer control. The visualization parameters will be

- `min`: 290 (a minimum LST value of 290 K)
- `max`: 320 (a maximum LST value of 320 K)
- `palette`: `cmap` (the color palette that was created some steps before).

The name of the layer in the **Map** layer set will be `LST`.

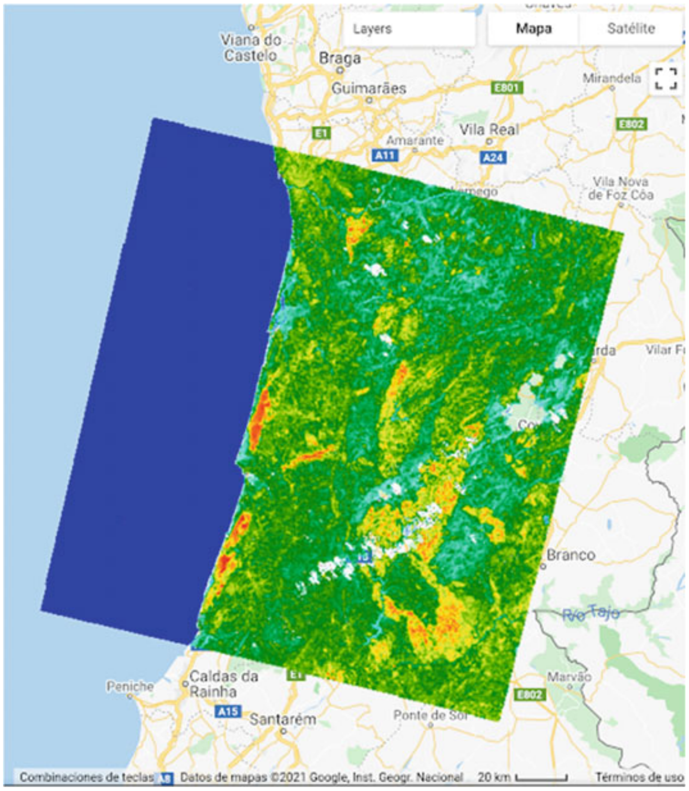


Fig. 31.10 Map illustrating LST, obtained by following the JavaScript example

```
Map.addLayer(exImage.select('LST'), {
    min: 290,
    max: 320,
    palette: cmap
}, 'LST')
```

Code Checkpoint F64f. The book's repository contains a script that shows what your code should look like at this point.

Now, let's use R to implement the same logic. As in the previous sections, import the R packages: *rgee* and *rgeeExtra*. Then, initialize your Earth Engine session.

```
library(rgee)
library(rgeeExtra)
library(reticulate)

ee_Initialize()
```

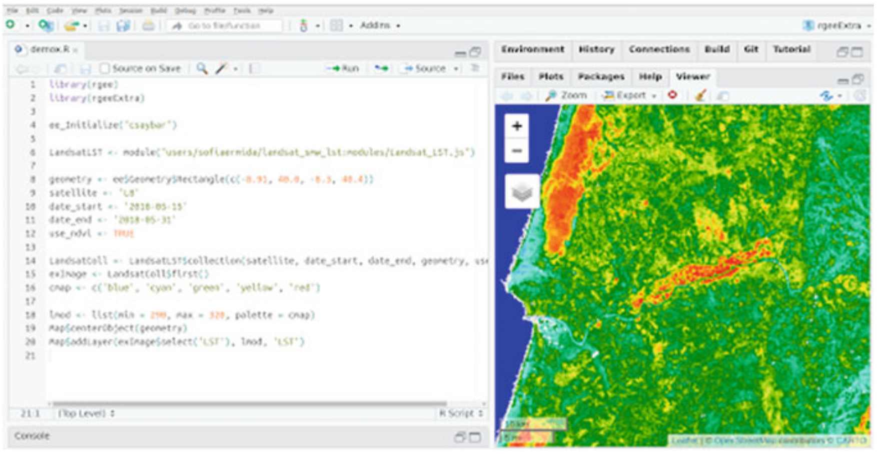
Install *rgeeExtra* Python dependencies.

```
py_install(packages = c("regex", "ee_extra",
"jsbeautifier"))
```

Using the function `rgeeExtra::module` loads the JavaScript module.

```
LandsatLST <-
module("users/sofiaermida/landsat_smw_lst:modules/Landsat_L
ST.js")
```

The rest of the code is exactly the same as in JavaScript (Fig. 31.11).



```
library(rgee)
library(rgeeExtra)

ee_Initialize("csaybar")

LandsatLST <- module("users/sofiaermida/landsat_smw_lst:modules/Landsat_LST.js")

geometry <- ee$Geometry$Rectangle(c(-8.91, 40.0, -8.3, 40.4))
satellite <- 'L8'
date_start <- '2018-05-15'
date_end <- '2018-05-31'
use_ndvi <- TRUE

LandsatColl <- LandsatLST$collection(satellite, date_start, date_end, geometry, use
exImage <- LandsatColl$first()
cmap <- c('blue', 'cyan', 'green', 'yellow', 'red')

lmod <- list(min = 290, max = 320, palette = cmap)
Map$centerObject(geometry)
Map$addLayer(exImage$select('LST'), lmod, 'LST')
```

File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
rgeeExtra
demox.R
Source on Save
Run
Source
21:1
(Top Level)
R Script
Console
Environment History Connections Build Git Tutorial
Files Plots Packages Help Viewer
Zoom
Export

Fig. 31.11 Map illustrating LST, obtained by following the R example

```
geometry <- ee$Geometry$Rectangle(c(-8.91, 40.0, -8.3,
40.4))
satellite <- 'L8'
date_start <- '2018-05-15'
date_end <- '2018-05-31'
use_ndvi <- TRUE

LandsatColl <- LandsatLST$collection(satellite, date_start,
date_end, geometry, use_ndvi)
exImage <- LandsatColl$first()
cmap <- c('blue', 'cyan', 'green', 'yellow', 'red')

lmod <- list(min = 290, max = 320, palette = cmap)
Map$centerObject(geometry)
Map$addLayer(exImage$select('LST'), lmod, 'LST')
```

Code Checkpoint F64g. The book's repository contains information about what your code should look like at this point.

Question 1. When and why might users prefer to use R instead of Python to connect to Earth Engine?

Question 2. What are the advantages and disadvantages of using *rgee* instead of the Earth Engine JavaScript Code Editor?

31.3 Synthesis

Assignment 1. Estimate the Gaussian curvature map from a digital elevation model using *rgee* and *rgeeExtra*. Hint**:** Use the module `'users/joselucassafanelli/TAGEE:TAGEE-functions'`.

31.4 Conclusion

In this chapter, you learned how to use Earth Engine and R in the same workflow. Since *rgee* uses *reticulate*, *rgee* also permits integration with third-party Earth Engine Python packages. You also learned how to use `Map$addLayer`, which works similarly to the Earth Engine User Interface API (Code Editor). Finally, we also introduced *rgeeExtra*, a new R package that extends the Earth Engine API and supports JavaScript module execution.

References

Aybar C, Wu Q, Bautista L, et al (2020) rgee: An R package for interacting with Google Earth Engine. J Open Source Softw 5:2272. https://doi.org/10.21105/joss.02272
Ermida SL, Soares P, Mantas V et al (2020) Google Earth Engine open-source code for land surface temperature estimation from the Landsat series. Remote Sens 12:1471. https://doi.org/10.3390/RS12091471
Grolemund G (2014) Hands-On Programming with R - Write Your Own Functions and Simulations. O'Reilly Media, Inc.
Lovelace R, Nowosad J, Muenchow J (2019) Geocomputation with R. Chapman and Hall/CRC
Montero D (2021) eemont: a Python package that extends Google Earth Engine. J Open Source Softw 6:3168. https://doi.org/10.21105/joss.03168
Pebesma E, Bivand R (2019) Spatial Data Science. https://r-spatial.org/book/

Part VII Human Applications

This part covers some of the many Human Applications of Earth Engine. It includes demonstrations of how Earth Engine can be used in agricultural and urban settings, including for sensing the built environment and the effects it has on air composition and temperature. This part also covers the complex topics of human health, illicit deforestation activity, and humanitarian actions.

Agricultural Environments 32

Sherrie Wang and George Azzari

Overview
The purpose of this chapter is to introduce how datasets and functions available in Earth Engine can be used to map agriculture at scale. We will walk through an example of mapping crop types in the US Midwest, which is one of the main breadbaskets of the world. The skills learned in this chapter will equip you to go on to map other agricultural characteristics, such as yields and management practices.

Learning Outcomes

- Classifying crop type in a county in the US Midwest using the Cropland Data Layer (CDL) as labels.
- Using `ee.Reducer.linearRegression` to fit a second-order harmonic regression to a crop time series and extract harmonic coefficients.
- Using the Green Chlorophyll Vegetation Index (GCVI) for crop type classification.
- Training a random forest to classify crop type from harmonic coefficients.
- Applying the trained random forest classifier to the study region and assessing its performance.

S. Wang (✉)
Massachusetts Institute of Technology, Cambridge, USA
e-mail: sherwang@mit.edu

G. Azzari
AtlasAI, Palo Alto, USA

Stanford University, Stanford, USA

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_32

Helps if you know how to

- Import images and image collections, filter, and visualize (Part 1).
- Create a graph using `ui.Chart` (Chap. 4).
- Perform basic image analysis: select bands, compute indices, create masks, classify images (Part 2).
- Obtain accuracy metrics from classifications (Chap. 7)
- Recognize similarities and differences among satellite spectral bands (Parts 1–3).
- Write a function and `map` it over an `ImageCollection` (Chap. 12).
- Mask cloud, cloud shadow, snow/ice, and other undesired pixels (Chap. 15).
- Fit linear and nonlinear functions with regression in an `ImageCollection` time series (Chap. 18).
- Filter a `FeatureCollection` to obtain a subset (Chaps. 22 and 23).

32.1 Introduction to Theory

Agriculture is one of the primary ways in which humans have altered the surface of our planet. Globally, about five billion hectares, or 38% of Earth's land surface, is devoted to agriculture (FAO 2020). About one-third of this is used to grow crops, while the other two-thirds are used to graze livestock.

In the face of a growing human population and changing climate, it is more important than ever to manage land resources effectively in order to grow enough food for all people while minimizing damage to the environment. Toward this end, remote sensing offers a critical source of data for monitoring agriculture. Since most agriculture occurs outdoors, sensors on satellites and airplanes can capture many crop characteristics. Research has shown that crops' spectral reflectance over time can be used to classify crop types (Wang et al. 2019), predict yield (Burke and Lobell 2017), detect crop stress (Ihuoma and Madramootoo 2017), identify irrigation (Deines et al. 2017), quantify species diversity (Duro et al. 2014), and pinpoint sowing and harvest dates (Jain et al. 2016). Remotely sensed imagery has also been key to the rise of precision agriculture, where management practices are varied at fine scales to respond to differences in crop needs within a field (Azzari et al. 2019; Jin et al. 2017; Seifert et al. 2018). Ultimately, the goal of measuring agricultural practices and outcomes is to improve yields and reduce environmental degradation.

32.2 Practicum

In this practicum, we will use Earth Engine to pull Landsat imagery and classify crop types in the US Midwest. The US is the world's largest producer of corn and second-largest producer of soybeans; therefore, maintaining high yields in the US is vital for global food security and price stability. Crop type mapping is an important prerequisite to using satellite imagery to estimate agricultural production. We will show how to obtain features from a time series by fitting a harmonic regression and extracting the coefficients. Then we will use a random forest to classify crop type, where the 'ground truth' labels will come from the Cropland Data Layer (CDL) from the US Department of Agriculture. CDL is itself a product of a classifier using satellite imagery as input and survey-based ground truth as training labels (USDA 2021). We demo crop type classification in the US Midwest because CDL allows us to validate our predictions. While using satellite imagery to map crop types is already operational in the US, mapping crop types remains an open challenge in much of the world.

32.2.1 Section 1. Pull All Landsat Imagery for the Study Area

In the Midwest, corn and soybeans dominate the landscape. We will map crop types in McLean County, Illinois. This is the county that produces the most corn and soybeans in the United States, at 62.9 and 19.3 million bushels, respectively. Let's start by defining the study area and visualizing it (Fig. 32.1). We obtain county boundaries from the United States Census Bureau's TIGER dataset, which can be found in the Earth Engine Data Catalog. We extract the McLean County feature from TIGER by name and by using Illinois' FIPS code of 17.

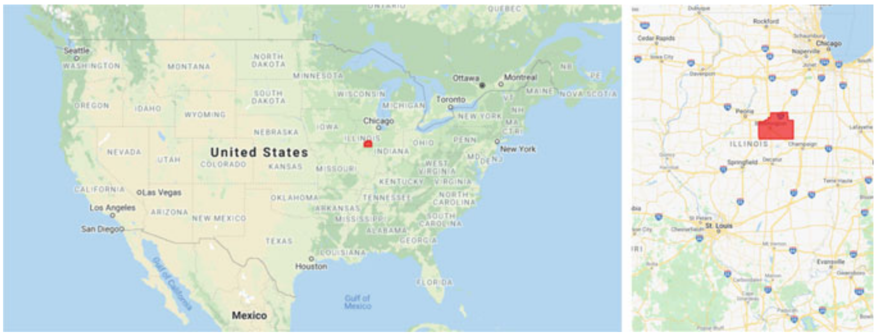


Fig. 32.1 Location and borders of McLean County, Illinois

```
// Define study area.
var TIGER = ee.FeatureCollection('TIGER/2018/Counties');
var region = ee.Feature(TIGER
    .filter(ee.Filter.eq('STATEFP', '17'))
    .filter(ee.Filter.eq('NAME', 'McLean'))
    .first());
var geometry = region.geometry();
Map.centerObject(region);
Map.addLayer(region, {
    'color': 'red'
}, 'McLean County');
```

Next, we import Landsat 7 and 8 imagery, as presented in Part 1. In particular, we use Level-2 data, which contains atmospherically corrected surface reflectance and land surface temperature. While top-of-atmosphere data will also yield good results for many agricultural applications, we prefer data that has been corrected for atmospheric conditions when available, since atmospheric variation is just one more source of noise in the inputs.

```
// Import Landsat imagery.
var landsat7 =
ee.ImageCollection('LANDSAT/LE07/C02/T1_L2');
var landsat8 =
ee.ImageCollection('LANDSAT/LC08/C02/T1_L2');
```

We define a few functions that will make it easier to work with Landsat data. The first two functions, `renameL7` and `renameL8`, rename the bands of either a Landsat 7 or Landsat 8 image to more intuitive names. For example, instead of calling the first band of a Landsat 7 image `'B1'`, we rename it to `'BLUE'`, since band 1 captures light in the blue portion of the visible spectrum. Renaming also makes it easier to work with Landsat 7 and 8 images together, since the band numbering of their sensors (ETM+ and OLI) is different.

```
// Functions to rename Landsat 7 and 8 images.
function renameL7(img) {
    return img.rename(['BLUE', 'GREEN', 'RED', 'NIR',
'SWIR1',
        'SWIR2', 'TEMP1', 'ATMOS_OPACITY', 'QA_CLOUD',
        'ATRAN', 'CDIST',
        'DRAD', 'EMIS', 'EMSD', 'QA', 'TRAD', 'URAD',
        'QA_PIXEL',
        'QA_RADSAT'
    ]);
}
```

```
function renameL8(img) {
    return img.rename(['AEROS', 'BLUE', 'GREEN', 'RED',
'NIR',
        'SWIR1',
        'SWIR2', 'TEMP1', 'QA_AEROSOL', 'ATRAN', 'CDIST',
        'DRAD', 'EMIS',
        'EMSD', 'QA', 'TRAD', 'URAD', 'QA_PIXEL',
'QA_RADSAT'
    ]);
}
```

The Landsat 7 Level-2 product has seven surface reflectance and temperature bands, while the Landsat 8 Level-2 product has eight. Both have a number of other bands for image quality, atmospheric conditions, etc. For this chapter, we will mostly be concerned with the near-infrared (NIR), short-wave infrared (SWIR1 and SWIR2), and pixel quality (QA_PIXEL) bands. Many differences among vegetation types can be seen in the NIR, SWIR1, and SWIR2 bands of the electromagnetic spectrum. The pixel quality band is important for masking out clouds when working with optical imagery (Chap. 15). Below, we define two functions: the `addMask` function to turn the `QA_PIXEL` bitmask into multiple masking layers, and the `maskQAClear` function to remove all non-clear pixels from each image.

```
// Functions to mask out clouds, shadows, and other
unwanted features.
function addMask(img) {
    // Bit 0: Fill
    // Bit 1: Dilated Cloud
    // Bit 2: Cirrus (high confidence) (L8) or unused (L7)
    // Bit 3: Cloud
    // Bit 4: Cloud Shadow
    // Bit 5: Snow
    // Bit 6: Clear
    //        0: Cloud or Dilated Cloud bits are set
    //        1: Cloud and Dilated Cloud bits are not set
    // Bit 7: Water
    var clear =
img.select('QA_PIXEL').bitwiseAnd(64).neq(0);
    clear = clear.updateMask(clear).rename(['pxqa_clear']);

    var water =
img.select('QA_PIXEL').bitwiseAnd(128).neq(0);
    water = water.updateMask(water).rename(['pxqa_water']);
    var cloud_shadow =
img.select('QA_PIXEL').bitwiseAnd(16).neq(0);
    cloud_shadow =
cloud_shadow.updateMask(cloud_shadow).rename([
        'pxqa_cloudshadow'
    ]);

    var snow =
img.select('QA_PIXEL').bitwiseAnd(32).neq(0);
    snow = snow.updateMask(snow).rename(['pxqa_snow']);

    var masks = ee.Image.cat([
        clear, water, cloud_shadow, snow
    ]);

    return img.addBands(masks);
}

function maskQAClear(img) {
    return img.updateMask(img.select('pxqa_clear'));
}
```

In addition to the raw bands sensed by ETM+ and OLI, vegetation indices (VI) can also help distinguish different vegetation types. Prior work has found the Green Chlorophyll Vegetation Index (GCVI) particularly useful for distinguishing corn from soybeans in the Midwest (Wang et al. 2019). We will add it as a band to each Landsat image.

```
// Function to add GCVI as a band.
function addVIs(img){
  var gcvi = img.expression('(nir / green) - 1', {
      nir: img.select('NIR'),
      green: img.select('GREEN')
  }).select([0], ['GCVI']);

  return ee.Image.cat([img, gcvi]);
}
```

Now, we are ready to pull Landsat 7 and 8 imagery for our study area and apply the above functions. We will access all images from January 1 to December 31, 2020, that intersect with McLean County. Because different crops have different growth characteristics, it is valuable to obtain a time series of images to distinguish when crops grow, senesce, and are harvested. Differences in spectral reflectance at individual points in time can also be informative.

```
// Define study time period.
var start_date = '2020-01-01';
var end_date = '2020-12-31';

// Pull Landsat 7 and 8 imagery over the study area between
start and end dates.
var landsat7coll = landsat7
    .filterBounds(geometry)
    .filterDate(start_date, end_date)
    .map(renameL7);

var landsat8coll = landsat8
    .filterDate(start_date, end_date)
    .filterBounds(geometry)
    .map(renameL8);
```

Next, we merge the Landsat 7 and Landsat 8 collections, mask out clouds, and add GCVI as a band.

```
// Merge Landsat 7 and 8 collections.
var landsat = landsat7coll.merge(landsat8coll)
    .sort('system:time_start');

// Mask out non-clear pixels, add VIs and time variables.
landsat = landsat.map(addMask)
    .map(maskQAClear)
    .map(addVIs);
```

We can use a `ui.Chart` object to visualize the combined Landsat GCVI time series at a particular point, which in this case falls in a corn field, according to CDL.

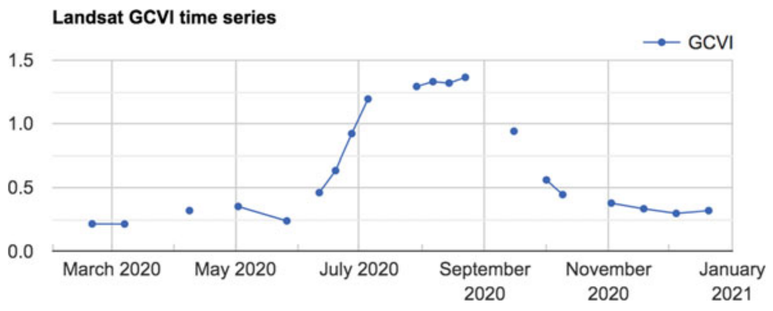


Fig. 32.2 Landsat GCVI time series visualized at a corn point in McLean County, Illinois. A higher GCVI corresponds to greater leaf chlorophyll content; the crop can be seen greening up in June and senescing in September

```
// Visualize GCVI time series at one location.
var point = ee.Geometry.Point([-88.81417685576481,
    40.579804398254005
]);
var landsatChart =
ui.Chart.image.series(landsat.select('GCVI'),
        point)
    .setChartType('ScatterChart')
    .setOptions({
        title: 'Landsat GCVI time series',
        lineWidth: 1,
        pointSize: 3,
    });
print(landsatChart);
```

You should see the chart shown in Fig. 32.2 in the Earth Engine **Console**.

Finally, let's also take a look at the crop type dataset that we will be using to train and evaluate our classifier. We load the CDL image for 2020 and select the layer that contains crop type information.

```
// Get crop type dataset.
var cdl = ee.Image('USDA/NASS/CDL/2020').select(['cropland']);
Map.addLayer(cdl.clip(geometry), {}, 'CDL 2020');
```

Corn fields are visualized in yellow and soybean fields in green in CDL (Fig. 32.3).

Code Checkpoint A11a. The book's repository contains a script that shows what your code should look like at this point.

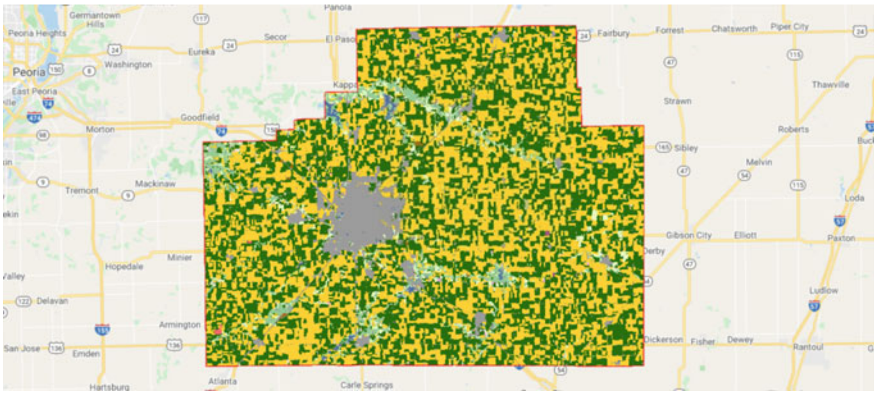


Fig. 32.3 CDL visualized over McLean County, Illinois. Corn fields are shown in yellow, while soybean fields are shown in dark green. Other land cover types include urban areas in gray, deciduous forest in teal, and water in blue

Question 1. How many images were taken by Landsat 7 and Landsat 8 over McLean County in 2020?

Question 2. By using `ui.Chart`, can you qualitatively spot any difference between the time series of corn fields and soybean fields?

32.2.2 Section 2. Add Bands to Landsat Images for Harmonic Regression

Now that we have a collection that combines all available Landsat imagery over McLean County in 2020, we can extract features from the time series for crop type classification. Harmonic regression is one way to extract features by fitting sine and cosine functions to a time series (Chap. 18). Also known as a Fourier transform, the harmonic regression is especially well suited for data patterns that reappear at regular intervals.

In particular, we fit a second-order harmonic regression, which takes the form:

$$f(t) = a_1 \cos(2\pi \omega t) + b_1 \sin(2\pi \omega t) + a_2 \cos(4\pi \omega t) + b_2 \sin(4\pi \omega t) + c \tag{32.1}$$

Here, t is the time variable; $f(t)$ is the observed variable to be fit; a, b, and c are coefficients found through regression; and ω is a hyperparameter that controls the periodicity of the harmonic basis. For this exercise, we will use $\omega = 1$.

To prepare the collection, we define two functions that add a harmonic basis and intercept to the `ImageCollection`. The first function adds the acquisition time of each image as a band to the image in units of years. The function takes an image and a reference date as arguments; the new time band is computed relative to this reference date.

```
// Function that adds time band to an image.
function addTimeUnit(image, refdate) {
    var date = image.date();

    var dyear = date.difference(refdate, 'year');
    var t =
image.select(0).multiply(0).add(dyear).select([0], ['t'])
        .float();

    var imageplus = image.addBands(t);

    return imageplus;
}
```

The second function calls `addTimeUnit`, then takes the sine and cosine of each image's acquisition time and adds them as new bands to each image. We are fitting a second-order harmonic regression, so we will add two sine and two cosine terms as bands, along with a constant term. The function also allows you to use any value of ω (omega). Note that, to create the `'constant'` band, we divide the time band by itself; this results in a band with value 1 and the same mask as the time band.

```
// Function that adds harmonic basis to an image.
function addHarmonics(image, omega, refdate) {
    image = addTimeUnit(image, refdate);
    var timeRadians = image.select('t').multiply(2 *
Math.PI * omega);
    var timeRadians2 = image.select('t').multiply(4 *
Math.PI *
    omega);

    return image
        .addBands(timeRadians.cos().rename('cos'))
        .addBands(timeRadians.sin().rename('sin'))
        .addBands(timeRadians2.cos().rename('cos2'))
        .addBands(timeRadians2.sin().rename('sin2'))
        .addBands(timeRadians.divide(timeRadians)
            .rename('constant'));
}
```

We have all the tools we need to add the sine and cosine terms to each image—we just use `map` to apply `addHarmonics` to each image in the collection. We use the `start_date` of January 1, 2020, as the reference date.

```
// Apply addHarmonics to Landsat image collection.
var omega = 1;
var landsatPlus = landsat.map(
    function(image) {
        return addHarmonics(image, omega, start_date);
    });
print('Landsat collection with harmonic basis: ',
landsatPlus);
```

When we print the new `ImageCollection`, the **Console** tab should display a collection of images with bands that now include `cos`, `sin`, `cos2`, `sin2`, and `constant`—the terms we just added.

Code Checkpoint A11b. The book's repository contains a script that shows what your code should look like at this point.

Question 3. What assumption are we making about crop spectral reflectance when we fit a harmonic regression to a variable like GCVI using $\omega = 1$? What types of time series might be better suited for a smaller value of ω? A larger ω?

Question 4. What happens to the harmonic fit if you change the reference date by a quarter of a year? By half of a year? By a full year?

32.2.3 Section 3. Fit a Harmonic Regression at Each Landsat Pixel

In the previous section, we added the harmonic basis (cosine, sine, and constant terms) to each image in our Landsat collection. Now, we can run a linear regression using this basis as the independent variables, and Landsat bands and GCVI as the dependent variables. This section defines several functions that will help us complete this regression.

The key step is linear regression performed by the `ee.Reducer.linearRegression` reducer. This reducer takes in two parameters: the number of independent variables and the number of dependent variables. When applied to an `ImageCollection`, it expects each image in the collection to be composed of its independent variable bands followed by a dependent variable band or bands. The reducer returns an array image whose first band is an array of regression coefficients computed through linear least squares and whose second band is the regression residual.

The first function we define is called `arrayimgHarmonicRegr` and helps us apply the linear regression reducer to each image.

```
// Function to run linear regression on an image.
function arrayimgHarmonicRegr(harmonicColl, dependent,
independents) {

    independents = ee.List(independents);
    dependent = ee.String(dependent);

    var regression = harmonicColl
        .select(independents.add(dependent))

.reduce(ee.Reducer.linearRegression(independents.length(),
        1));

    return regression;
}
```

Next, we want to transform the returned array image into an image with each coefficient as its own band (Chap. 9). Furthermore, since we will be running harmonic regressions for multiple Landsat bands, we want to rename the harmonic coefficients returned by our regression to match their corresponding band. The second function, `imageHarmonicRegr`, performs these operations. The array image is transformed into a multiband image using the functions `arrayProject` and `arrayFlatten`, and the coefficients for the cosine, sine, and constant terms are renamed by adding the band name as a prefix (e.g., the first cosine coefficient for the NIR band becomes `NIR_cos`).

```
// Function to extract and rename regression coefficients.
function imageHarmonicRegr(harmonicColl, dependent,
independents) {

    var hregr = arrayimgHarmonicRegr(harmonicColl,
dependent,
        independents);

    independents = ee.List(independents);
    dependent = ee.String(dependent);

    var newNames = independents.map(function(b) {
        return dependent.cat(ee.String('_')).cat(ee.String(
        b));
    });

    var imgCoeffs = hregr.select('coefficients')
        .arrayProject([0])
        .arrayFlatten([independents])
        .select(independents, newNames);

    return imgCoeffs;
}
```

Finally, the third function is called `getHarmonicCoeffs` and performs the harmonic regression for all dependent bands by mapping the `imageHarmonicRegr` function over each band. It then creates a multiband image comprised of the regression coefficients.

```
// Function to apply imageHarmonicRegr and create a multi-
band image.
function getHarmonicCoeffs(harmonicColl, bands,
independents) {
    var coefficients =
ee.ImageCollection.fromImages(bands.map(
        function(band) {
            return imageHarmonicRegr(harmonicColl, band,
                independents);
        }));
    return coefficients.toBands();
}
```

Now that we have defined all the helper functions, we are ready to apply them to our Landsat `ImageCollection` with the harmonic basis. We are using GCVI, NIR, SWIR1, and SWIR2 as the bands for crop type mapping. For each band, we have five coefficients from the regression corresponding to the `cos`, `sin`, `cos2`, `sin2`, and `constant` terms. In total, then, we will end up with $4 \times 5 = 20$ harmonic coefficients. After applying `getHarmonicCoeffs`, we clip the coefficients to the study area and apply a transformation to change the coefficients from a 64-bit double data type to a 32-bit integer. This last step allows us to save space on storage.

```
// Apply getHarmonicCoeffs to ImageCollection.
var bands = ['NIR', 'SWIR1', 'SWIR2', 'GCVI'];
var independents = ee.List(['constant', 'cos', 'sin',
'cos2',
'sin2']);
var harmonics = getHarmonicCoeffs(landsatPlus, bands,
independents);

harmonics = harmonics.clip(geometry);
harmonics = harmonics.multiply(10000).toInt32();
```

Let's take a look at the harmonics that we've fit. Following the same logic as in Chap. 18, we can compute the fitted values by multiplying the regression coefficients with the independent variables. We'll do this just for GCVI, as we did in Sect. 32.2.1.

```
// Compute fitted values.
var gcviHarmonicCoefficients = harmonics
    .select([
        '3_GCVI_constant', '3_GCVI_cos',
        '3_GCVI_sin', '3_GCVI_cos2', '3_GCVI_sin2'
    ])
    .divide(10000);

var fittedHarmonic = landsatPlus.map(function(image) {
    return image.addBands(
        image.select(independents)
        .multiply(gcviHarmonicCoefficients)
        .reduce('sum')
        .rename('fitted')
    );
});
```

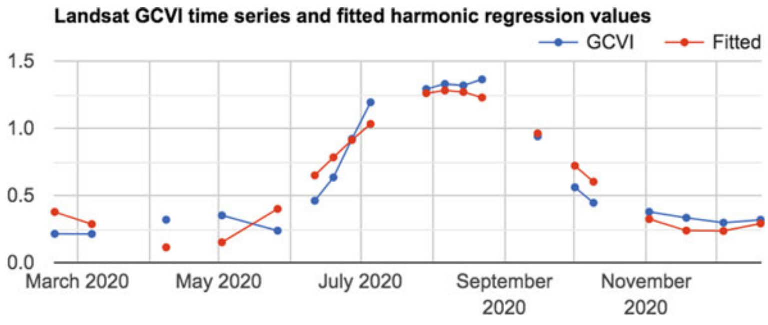


Fig. 32.4 Landsat GCVI time series (blue line) and fitted harmonic regression (red line) curve visualized at a corn point in McLean County, Illinois. While the fit is not perfect, the fitted harmonic captures the general characteristics of crop green-up and senescence

Now, we can plot the fitted values along with the original Landsat GCVI time series in one chart (Fig. 32.4).

```
// Visualize the fitted harmonics in a chart.
var harmonicsChart = ui.Chart.image.series(
        fittedHarmonic.select(
            ['fitted', 'GCVI']), point, ee.Reducer.mean(),
30)
    .setSeriesNames(['GCVI', 'Fitted'])
    .setOptions({
        title: 'Landsat GCVI time series and fitted
harmonic regression values',
        lineWidth: 1,
        pointSize: 3,
    });

print(harmonicsChart);
```

Before we move on to classifying crop type, we will add CDL as a band to the harmonics and export the final image as an asset. Recall that CDL will be the source of crop type labels for model training; adding this band now is not necessary but makes the labels more accessible at training time.

```
// Add CDL as a band to the harmonics.
var harmonicsPlus = ee.Image.cat([harmonics, cdl]);
```

We are ready to export the harmonics we've computed to an asset. Exporting to an asset is an optional step but can make it easier in the next step to train a model. For a larger study area, the operations may time out without the export step because computing harmonics is computationally intensive. You should replace `'your_asset_path_here/'` and `filename` below with the place you want to save your harmonics.

```
// Export image to asset.
var filename = 'McLean_County_harmonics';
Export.image.toAsset({
    image: harmonicsPlus,
    description: filename,
    assetId: 'your_asset_path_here/' + filename,
    dimensions: null,
    region: region,
    scale: 30,
    maxPixels: 1e12
});
```

We visualized the fitted harmonics at one point in the study area; let's also take a look at the coefficients over the entire study area using `Map.addLayer`. Since we can only visualize three bands at a time, we will have to pick three and visualize them in false color. Sometimes, it takes a bit of tweaking to visualize regression coefficients in false color; below, we transform three fitted GCVI bands via division and addition by constants (obtained through trial and error) to obtain a nice image for visualization.

```
// Visualize harmonic coefficients on map.
var visImage = ee.Image.cat([
    harmonics.select('3_GCVI_cos').divide(7000).add(0.6),
    harmonics.select('3_GCVI_sin').divide(7000).add(0.5),

harmonics.select('3_GCVI_constant').divide(7000).subtract(
        0.6)
]);

Map.addLayer(visImage, {
    min: -0.5,
    max: 0.5
}, 'Harmonic coefficient false color');
```

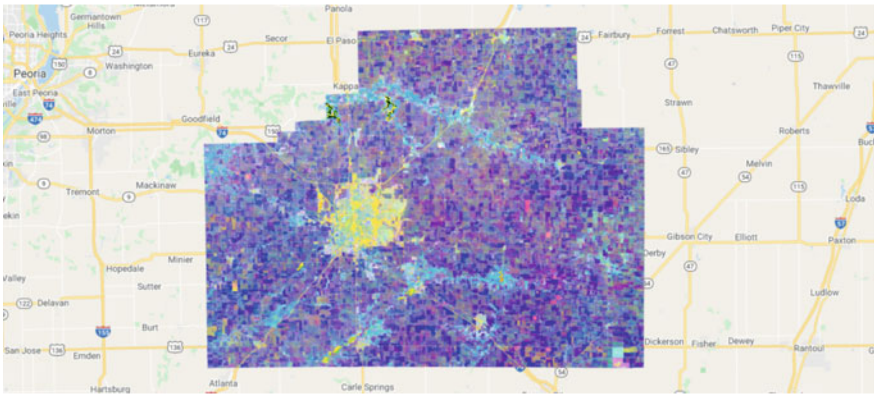


Fig. 32.5 False-color image of harmonic coefficients fitted to GCVI time series in McLean County, Illinois

These harmonic features allow us to see differences between urban land cover, water, and what we are currently interested in, crop types. Looking closely at the false-color image shown in Fig. 32.5, we can also see striping unrelated to land cover throughout the image. This striping, most visible in the bottom right of the image, is due to the failure of Landsat 7's Scan Line Corrector in 2003, which resulted in data gaps that occur in a striped pattern across Landsat 7 images. It is worth noting that these artifacts can affect classification performance, but exactly how much depends on the task.

Code Checkpoint A11c. The book's repository contains a script that shows what your code should look like at this point.

Question 5. What color do forested areas appear as in Fig. 32.5? Urban areas? Water?

Question 6. Comparing the map layers in Figs. 32.5 and 32.3, can you tell whether the GCVI cosine, sine, and constant terms capture differences between corn and soybean time series? What color do corn fields appear as in the false-color visualization? What about soybean fields?

32.2.4 Section 4. Train and Evaluate a Random Forest Classifier

We have our features, and we are ready to train a random forest in McLean County to classify crop types. As explained in Chap. 6, random forests are a classifier available in Earth Engine that can achieve high accuracies while being computationally efficient. We define a `ee.Classifier` object with 100 trees and a minimum leaf population of 10.

```
// Define a random forest classifier.
var rf = ee.Classifier.smileRandomForest({
    numberOfTrees: 50,
    minLeafPopulation: 10,
    seed: 0
});
```

The bands that we will use as features in the classification are the harmonic coefficients fitted to the NIR, SWIR1, SWIR2, and GCVI time series at each pixel. This may be different from previous classification examples you have seen, where band values at one time step are used as dependent variables. Using harmonic coefficients is one way to provide information from the temporal dimension to a classifier. Temporal information is very useful for distinguishing crop types.

We can obtain the band names by calling `bandNames` on the harmonics image and removing the CDL band and the `system:index` band.

```
// Get harmonic coefficient band names.
var bands = harmonicsPlus.bandNames();
bands = bands.remove('cropland').remove('system:index');
```

To prepare the CDL band to be the output of classification, we remap the crop type values in CDL (corn = 1, soybeans = 5, and over a hundred other crop types) to corn taking on a value of 1, soybeans a value of 2, and everything else a value of 0. We focus on classifying corn and soybeans for this exercise.

```
// Transform CDL into a 3-class band and add to harmonics.
var cornSoyOther = harmonicsPlus.select('cropland').eq(1)

.add(harmonicsPlus.select('cropland').eq(5).multiply(2));
var dataset = ee.Image.cat([harmonicsPlus.select(bands),
    cornSoyOther]);
```

Next, we sample 100 points from McLean County to serve as the training sample for our model. The classifier will learn to differentiate among corn, soybeans, and everything else using this set of points.

```
// Sample training points.
var train_points = dataset.sample(geometry, 30, null, null,
100, 0);
print('Training points', train_points);
```

Table 32.1 Confusion matrix obtained for the training set

		Predicted crop type		
		Other (class 0)	Corn (class 1)	Soy (class 2)
Actual crop type	Other (class 0)	27	1	1
	Corn (class 1)	2	34	0
	Soy (class 2)	3	4	28

Training the model is as simple as calling `train` on the classifier and feeding it the training points, the name of the label band, and the names of the input bands. To assess model performance on the training set, we can then compute the confusion matrix (Chap. 7) by calling `confusionMatrix` on the model object. Finally, we can compute the accuracy by calling `accuracy` on the confusion matrix object.

```
// Train the model!
var model = rf.train(train_points, 'cropland', bands);
var trainCM = model.confusionMatrix();
print('Training error matrix: ', trainCM);
print('Training overall accuracy: ', trainCM.accuracy());
```

You should see the confusion matrix shown in Table 32.1 on the training set.

Table 32.1 tells us that the classifier is achieving 89% accuracy on the training set. Given that the model is successful on the training set, we next want to explore how well it will generalize to the entire study region. To estimate generalization performance, we can sample a test set (also called a 'validation set') and apply the model to that feature collection using `classify(model)`. To keep the amount of computation manageable and avoid exceeding Earth Engine's memory quota, we sample 50 test points.

```
// Sample test points and apply the model to them.
var test_points = dataset.sample(geometry, 30, null, null,
50, 1);
var tested = test_points.classify(model);
```

Next, we can compute the confusion matrix and accuracy on the test set by calling `errorMatrix('cropland', 'classification')` followed by accuracy on the classified points. The `'classification'` argument refers to the property where the model predictions are stored.

```
// Compute the confusion matrix and accuracy on the test
set.
var testAccuracy = tested.errorMatrix('cropland',
'classification');
print('Test error matrix: ', testAccuracy);
print('Test overall accuracy: ', testAccuracy.accuracy());
```

You should see a test set confusion matrix matching (or perhaps almost matching) Table 32.2.

The test set accuracy of the model is lower than the training set accuracy, which is indicative of some overfitting and is common when the model is expressive. We also see from Table 32.2 that most of the error comes from confusing corn pixels with soybean pixels and vice versa. Understanding the errors in your classifier can help you build better models in the next iteration.

Beyond estimating the generalization error, we can actually apply the model to the entire study region and see how well the model performs across space. We do this by calling `classify(model)` again on the harmonic image for McLean County. We add it to the map to visualize our predictions (Fig. 32.6).

Table 32.2 Confusion matrix obtained for the test set

		Predicted crop type		
		Other (class 0)	Corn (class 1)	Soy (class 2)
Actual crop type	Other (class 0)	12	1	0
	Corn (class 1)	1	15	2
	Soy (class 2)	1	5	13

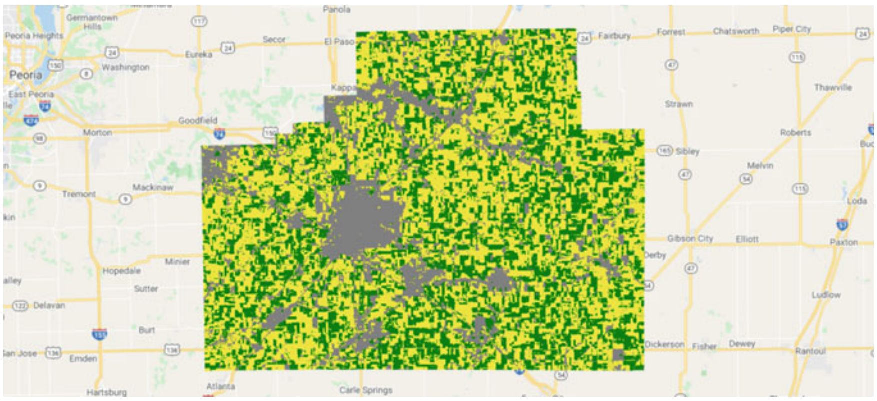


Fig. 32.6 Model predictions of crop type in McLean County, Illinois. Yellow is corn, green is soybeans, and gray is other land cover

```
// Apply the model to the entire study region.
var regionClassified =
harmonicsPlus.select(bands).classify(model);
var predPalette = ['gray', 'yellow', 'green'];
Map.addLayer(regionClassified, {
    min: 0,
    max: 2,
    palette: predPalette
}, 'Classifier prediction');
```

How do our model predictions compare to the CDL labels? We can see where the two are equal and where they differ by calling `eq` on the two images.

```
// Visualize agreements/disagreements between prediction
and CDL.
Map.addLayer(regionClassified.eq(cornSoyOther), {
    min: 0,
    max: 1
}, 'Classifier agreement');
```

Figure 32.7 shows agreement between our predictions and CDL labels. When visualized this way, some patterns become clear: Many errors occur along field boundaries, and sometimes, entire fields are misclassified.

Code Checkpoint A11d. The book's repository contains a script that shows what your code should look like at this point.

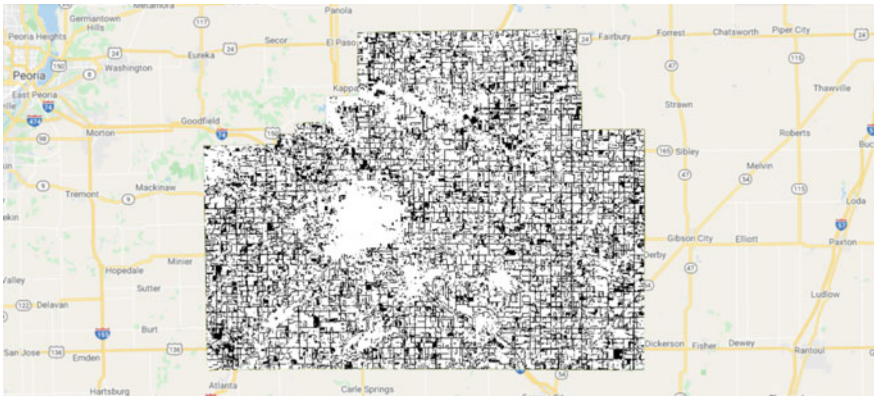


Fig. 32.7 Prediction agreement with CDL as ground truth labels. White pixels are where our predictions and CDL agree; black pixels are where they disagree. Errors happen on field boundaries and sometimes inside fields as well

Question 7. As important as accuracy is for summarizing a classifier's overall performance, we often want to know class-specific performance as well. What are the producer's and user's accuracies (Chap. 7) for the corn class? For the soybean class? For the 'other' class?

Question 8. Investigate the labels of field boundaries. What class is our model classifying them as, and what class does CDL classify them as? Now investigate the GCVI time series and fitted harmonic regressions at field boundaries (Fig. 32.4). Do they look different from time series inside fields? Why do you think our model is having a hard time classifying field boundary as the CDL class?

32.3 Synthesis

When we trained the above classifier to identify crop types, we made many choices, such as which type of classifier to use (random forest), which Landsat 7 bands to use (NIR, SWIR1, SWIR2, GCVI), how to extract features (harmonic regression), and how to sample training points (100 uniformly random points). In practice, any machine learning task will involve choices like these, and it is important to understand how these choices affect the performance of the classifier we end up with. To gain an understanding of how some of these choices affect crop classification accuracy, try the following:

Assignment 1. Instead of using only three Landsat bands plus GCVI, add the blue, green, and red bands as well. You should end up with 35 harmonic coefficients per pixel instead of 20. What happens to the out-of-sample classification accuracy when you add these three additional bands as features?

Assignment 2. Instead of a second-order harmonic regression, try a third-order harmonic regression. You should end up with 28 harmonic coefficients per pixel instead of 20. What happens to the out-of-sample classification accuracy?

Assignment 3. Using the original set of features (20 harmonic coefficients), train a random forest using 10, 20, 50, 200, 500, and 1000 training points. How does the out-of-sample classification accuracy change as the training set size increases?

Assignment 4. Extra challenge: Instead of using Landsat 7 and 8 as the input imagery, perform the same crop type classification using Sentinel-2 data and report the out-of-sample classification accuracy.

32.4 Conclusion

In this chapter, we showed how to use datasets and functions available in Earth Engine to classify crop types in the US Midwest. In particular, we used Landsat 7 and 8 time series as inputs, extracted features from the time series using harmonic regression, obtained labels from the CDL, and predicted crop type with a random forest classifier. Remote sensing data can be used to understand many more aspects of agriculture beyond crop type, such as crop yields, field boundaries, irrigation, tillage, cover cropping, soil moisture, biodiversity, and pest pressure. While the exact imagery source, classifier type, and classification task can all differ, the general workflow is often similar to what has been laid out in this chapter. Since farming sustainably while feeding 11 billion people by 2100 is one of the great challenges of this century, we encourage you to continue exploring how Earth Engine and remote sensing data can help us understand agricultural environments around the world.

References

Azzari G, Grassini P, Edreira JIR et al (2019) Satellite mapping of tillage practices in the North Central US region from 2005 to 2016. Remote Sens Environ 221:417–429. https://doi.org/10.1016/j.rse.2018.11.010

Burke M, Lobell DB (2017) Satellite-based assessment of yield variation and its determinants in smallholder African systems. Proc Natl Acad Sci USA 114:2189–2194. https://doi.org/10.1073/pnas.1616919114

Deines JM, Kendall AD, Hyndman DW (2017) Annual irrigation dynamics in the U.S. Northern High Plains derived from Landsat satellite data. Geophys Res Lett 44:9350–9360. https://doi.org/10.1002/2017GL074071

Duro DC, Girard J, King DJ et al (2014) Predicting species diversity in agricultural environments using Landsat TM imagery. Remote Sens Environ 144:214–225. https://doi.org/10.1016/j.rse.2014.01.001

Food and Agriculture Organization of the United Nations (2020) Land use in agriculture by the numbers. Sustain Food Agric. https://www.fao.org/sustainability/news/detail/en/c/1274219/. Accessed 11 Nov 2021

Ihuoma SO, Madramootoo CA (2017) Recent advances in crop water stress detection. Comput Electron Agric 141:267–275. https://doi.org/10.1016/j.compag.2017.07.026

Jain M, Srivastava AK, Balwinder-Singh et al (2016) Mapping smallholder wheat yields and sowing dates using micro-satellite data. Remote Sens 8:860. https://doi.org/10.3390/rs8100860

Jin Z, Prasad R, Shriver J, Zhuang Q (2017) Crop model- and satellite imagery-based recommendation tool for variable rate N fertilizer application for the US Corn system. Precis Agric 18:779–800. https://doi.org/10.1007/s11119-016-9488-z

Seifert CA, Azzari G, Lobell DB (2018) Satellite detection of cover crops and their effects on crop yield in the Midwestern United States. Environ Res Lett 13:64033. https://doi.org/10.1088/1748-9326/aaf933

USDA (2022) CropScape—Cropland Data Layer. In: Crop—NASS CDL Progr. https://nassgeodata.gmu.edu/CropScape/. Accessed 12 Nov 2021

Wang S, Azzari G, Lobell DB (2019) Crop type mapping without field-level labels: random forest transfer and unsupervised clustering techniques. Remote Sens Environ 222:303–317. https://doi.org/10.1016/j.rse.2018.12.026

Urban Environments

33

Michelle Stuhlmacher and Ran Goldblatt

Overview

Urbanization has dramatically changed Earth's surface. This chapter starts with a qualitative look at the impact urban expansion has on the landscape, covering three existing urban classification schemes that have been created by other remote sensing scientists. We look at how these classifications can be used to quantify urban areas, and close with instructions on how to perform per-pixel supervised image classification to map built-up land cover at any location on Earth and at any point in time using Landsat 7 imagery.

Learning Outcomes

- Creating an animated GIF.
- Implementing quantitative and qualitative analyses with pre-existing urban classifications.
- Running your own classification of built-up land cover.
- Quantifying and mapping the extent of urbanization across space and time.

M. Stuhlmacher (✉)
Department of Geography, DePaul University, 990 W Fullerton, Chicago, IL 60614, USA
e-mail: michelle.stuhlmacher@depaul.edu

R. Goldblatt
New Light Technologies, Inc. (NLT), 655 15th Street, NW Suite 800, Washington, DC 20005, USA
e-mail: ran.goldblatt@nltgis.com

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_33

Helps if you know how to

- Import images and image collections, filter, and visualize (Part 1).
- Use drawing tools to create points lines and polygons (Chap. 6).
- Perform a supervised image classification (Chap. 6).
- Use `reduceRegions` to summarize an image with zonal statistics in irregular shapes (Chaps. 22 and 24).

33.1 Introduction to Theory

Urbanization and its consequences have expanded rapidly over the last several decades. In 1950, only 30% of the world's population lived in urban areas, but that figure is expected to be 68% by 2050 (United Nations 2019). Already, over 50% of the world's population lives in urban areas (United Nations 2019). This shift to urban dwelling has helped lift millions of people out of poverty, but it is also creating complicated socio-environmental challenges, such as habitat loss, urban heat islands, flooding, and greater greenhouse gas emissions (Bazaz et al. 2018; United Nations 2019).

An important part of addressing these challenges is understanding urbanization trajectories and their consequences. Satellite imagery provides a rich source of historic and current information on urbanization all over the globe, and it is increasingly being leveraged to contribute to research on urban sustainability (Goldblatt et al. 2018; Prakash et al. 2020). In this chapter, we will cover several methods for visualizing and quantifying urbanization's impact on the landscape.

33.2 Practicum

33.2.1 Section 1. Time Series Animation

To start our examination of urban environments, we'll create a time series animation (GIF) for a qualitative depiction of the impact of urbanization. We'll use Landsat imagery to visualize how the airport northwest of Bengaluru, India, grew between 2010 and 2020.

First, search for "Gangamuthanahalli" in the Code Editor search bar and click on the city name to navigate there. Gangamuthanahalli is home to Kempegowda International Airport. Use the **Geometry Tools**, as shown in Chap. 6, to draw a rectangle around the airport.

Now search for "landsat 8 level 2" and import the "USGS Landsat 8 Level 2, Collection 2, Tier 1" `ImageCollection`. Name the import `L8`. Filter the collection by the geometry you created and the date range of interest, 2010–2020. Limit cloud cover on land to 3%.

```
// Filter collection.
var collection = L8
    .filterBounds(geometry)
    .filterDate('2010-01-01', '2020-12-31')
    .filter(ee.Filter.lte('CLOUD_COVER_LAND', 3));
```

Next, set up the parameters for creating a GIF. We want to visualize the three visible bands over the airport region and cycle through the images at 15 frames per second.

```
// Define GIF visualization arguments.
var gifParams = {
    bands: ['SR_B4', 'SR_B3', 'SR_B2'],
    min: 0.07 * 65536,
    max: 0.3 * 65536,
    region: geometry,
    framesPerSecond: 15,
    format: 'gif'
};
```

Last, render the GIF in the **Console**.

```
// Render the GIF animation in the console.
print(ui.Thumbnail(collection, gifParams));
```

Code Checkpoint A12a. The book's repository contains a script that shows what your code should look like at this point.

The GIF gives a qualitative sense of the way the airport has changed. To quantify urban environment change (i.e., summing the area of total new impervious surfaces), we often turn to classifications. Section 2 covers pre-existing classifications you can leverage to quantify urban extent.

33.2.2 Section 2. Pre-existing Urban Classifications

There are several publicly accessible land cover classifications that include urban areas as one of their classes. These classifications are a quick way to qualify urban extent. In this section, we'll cover three from the Earth Engine Data Catalog and show you how to visualize and quantify the urban classes: (1) the MODIS Land Cover Type Yearly Global; (2) the Copernicus CORINE Land Cover; and (3) the United States Geological Survey (USGS) National Land Cover Database (NLCD). These three classifications vary in their spatial and temporal resolution. For example, MODIS is a global dataset, while the NLCD covers only the United States, and CORINE covers only the European Union. We'll explore more of these differences later.

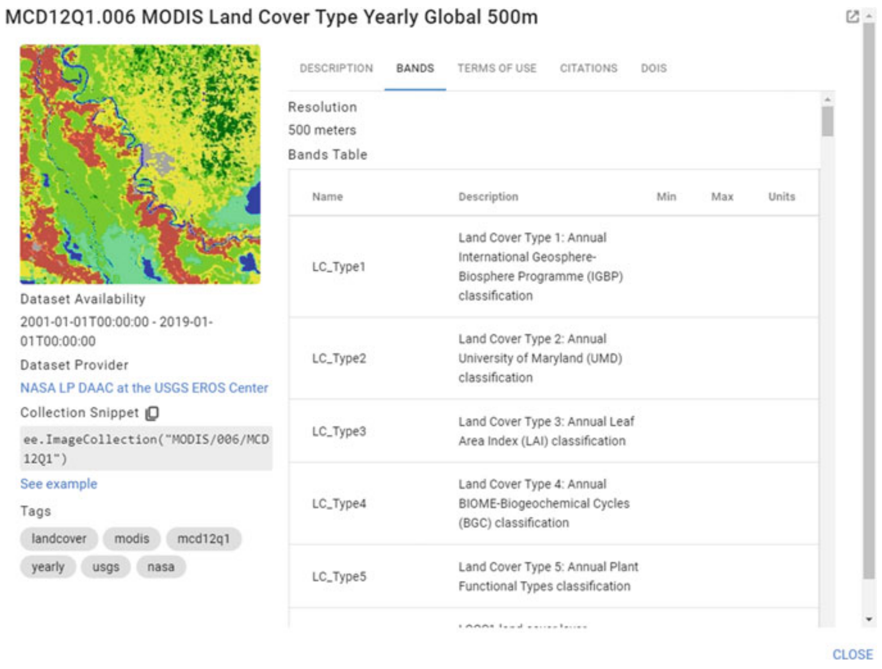
First, start a new code and search for "MODIS land cover." Find the `ImageCollection` titled "MCD12Q1.006 MODIS Land Cover Type Yearly Global 500 m," import it, and rename it to `MODIS`. We will first look at the MODIS land cover classification over Accra, Ghana:

```
// MODIS (Accra)
// Center over Accra.
Map.setCenter(-0.2264, 5.5801, 10);
```

The Land Cover Type Yearly Global dataset contains multiple classifications (Fig. 33.1). We will use `LC_Type1`, the annual classification using the International Geosphere-Biosphere Programme (IGBP) categories, in this example.

Select the `LC_Type1` band, copy and paste the visualization parameters for the classification, and visualize the full classification.

```
// Visualize the full classification.
var MODIS_lc = MODIS.select('LC_Type1');
var igbpLandCoverVis = {
    min: 1.0,
    max: 17.0,
    palette: ['05450a', '086a10', '54a708', '78d203',
'009900',
        'c6b044', 'dcd159', 'dade48', 'fbff13', 'b6ff05',
        '27ff87', 'c24f44', 'a5a5a5', 'ff6d4c', '69fff8',
        'f9ffa4', '1c0dff'
    ],
};
Map.addLayer(MODIS_lc, igbpLandCoverVis, 'IGBP Land
Cover');
```



MCD12Q1.006 MODIS Land Cover Type Yearly Global 500m

Dataset Availability
2001-01-01T00:00:00 - 2019-01-01T00:00:00
Dataset Provider
NASA LP DAAC at the USGS EROS Center
Collection Snippet
```
ee.ImageCollection("MODIS/006/MCD12Q1")
```
See example
Tags
landcover modis mcd12q1 yearly usgs nasa

Resolution
500 meters
Bands Table

Name	Description	Min	Max	Units
LC_Type1	Land Cover Type 1: Annual International Geosphere-Biosphere Programme (IGBP) classification			
LC_Type2	Land Cover Type 2: Annual University of Maryland (UMD) classification			
LC_Type3	Land Cover Type 3: Annual Leaf Area Index (LAI) classification			
LC_Type4	Land Cover Type 4: Annual BIOME-Biogeochemical Cycles (BGC) classification			
LC_Type5	Land Cover Type 5: Annual Plant Functional Types classification			

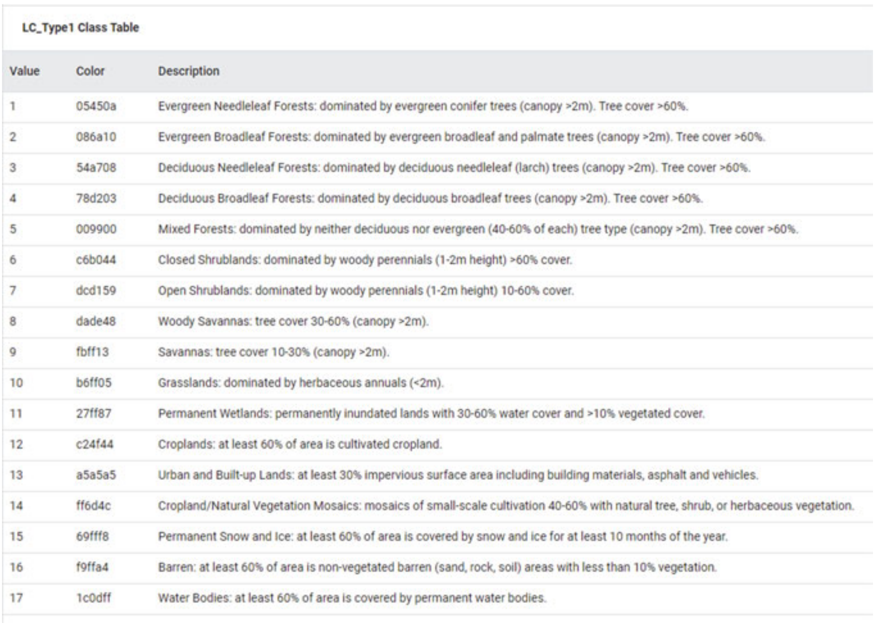
Fig. 33.1 Metadata of the MODIS land cover classification types

Press **Run** and view the output with the classification visualization. Use the **Inspector** tab to click on the different colors and determine the land cover classes they represent.

Next, we're going to visualize only the urban class at points almost two decades apart: from 2001 and 2019. We'll start by filtering for 2019 dates.

```
// Visualize the urban extent in 2001 and 2019.
// 2019
var MODIS_2019 = MODIS_lc.filterDate(ee.Date('2019-01-
01'));
```

Looking at the metadata shows us that the urban class has a value of 13 (Fig. 33.2), so we'll select only the pixels with this value and visualize them for 2019. To show only the urban pixels, mask the urban image with itself in the `Map.addLayer` command.



LC_Type1 Class Table

Value	Color	Description
1	05450a	Evergreen Needleleaf Forests: dominated by evergreen conifer trees (canopy >2m). Tree cover >60%.
2	086a10	Evergreen Broadleaf Forests: dominated by evergreen broadleaf and palmate trees (canopy >2m). Tree cover >60%.
3	54a708	Deciduous Needleleaf Forests: dominated by deciduous needleleaf (larch) trees (canopy >2m). Tree cover >60%.
4	78d203	Deciduous Broadleaf Forests: dominated by deciduous broadleaf trees (canopy >2m). Tree cover >60%.
5	009900	Mixed Forests: dominated by neither deciduous nor evergreen (40-60% of each) tree type (canopy >2m). Tree cover >60%.
6	c6b044	Closed Shrublands: dominated by woody perennials (1-2m height) >60% cover.
7	dcd159	Open Shrublands: dominated by woody perennials (1-2m height) 10-60% cover.
8	dade48	Woody Savannas: tree cover 30-60% (canopy >2m).
9	fbff13	Savannas: tree cover 10-30% (canopy >2m).
10	b6ff05	Grasslands: dominated by herbaceous annuals (<2m).
11	27ff87	Permanent Wetlands: permanently inundated lands with 30-60% water cover and >10% vegetated cover.
12	c24f44	Croplands: at least 60% of area is cultivated cropland.
13	a5a5a5	Urban and Built-up Lands: at least 30% impervious surface area including building materials, asphalt and vehicles.
14	ff6d4c	Cropland/Natural Vegetation Mosaics: mosaics of small-scale cultivation 40-60% with natural tree, shrub, or herbaceous vegetation.
15	69fff8	Permanent Snow and Ice: at least 60% of area is covered by snow and ice for at least 10 months of the year.
16	f9ffa4	Barren: at least 60% of area is non-vegetated barren (sand, rock, soil) areas with less than 10% vegetation.
17	1c0dff	Water Bodies: at least 60% of area is covered by permanent water bodies.

Fig. 33.2 Classes for the `LC_Type1` classification

```
var M_urb_2019 = MODIS_2019.mosaic().eq(13);
Map.addLayer(M_urb_2019.mask(M_urb_2019), {
    'palette': 'FF0000'
}, 'MODIS Urban 2019');
```

Repeat the same steps for 2001, but give it a gray palette to contrast with the red palette of 2019.

```
var MODIS_2001 = MODIS_lc.filterDate(ee.Date('2001-01-
01'));
var M_urb_2001 = MODIS_2001.mosaic().eq(13);
Map.addLayer(M_urb_2001.mask(M_urb_2001), {
    'palette': 'a5a5a5'
}, 'MODIS Urban 2001');
```

The result is a visualization showing the extent of Accra in 2002 in gray and the new urbanization between 2001 and 2019 in red (Fig. 33.3).

Code Checkpoint A12b. The book's repository contains a script that shows what your code should look like at this point.

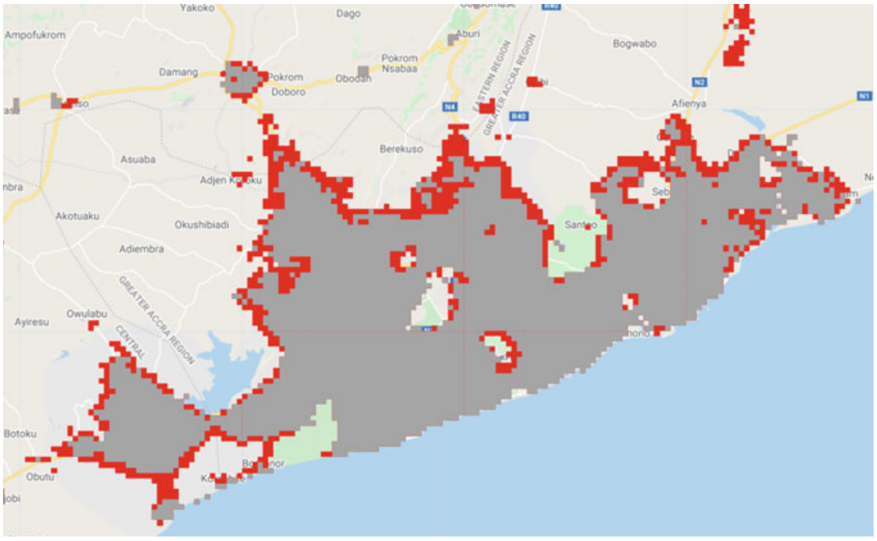


Fig. 33.3 Output of MODIS visualization

Next, we'll look at CORINE. Open a new script, search for "CORINE," select the Copernicus CORINE Land Cover dataset to import, rename it to `CORINE`, and center it over London, England.

```
// CORINE (London)
// Center over London
Map.setCenter(-0.1795, 51.4931, 10);
```

Conducting a similar process to the one we used with MODIS, first select the image for 2018.

```
// Visualize the urban extent in 2000 and 2018.
// 2018 (2017-2018)
var CORINE_2018 =
CORINE.select('landcover').filterDate(ee.Date(
    '2017-01-01'));
```

Why filter by 2017 and not 2018? The description section of CORINE's metadata shows that the time period covered by the 2018 asset includes both 2017 and 2018. To select the 2018 asset in the code above, we filter by the date of the first year. (If you're curious, you can test what happens when you filter using 2018 as the date.)

The metadata for CORINE shows that the built-up land cover classes range from 111 to 133, so we'll select all classes less than or equal to 133 and visualize them in red.

```
var C_urb_2018 = CORINE_2018.mosaic().lte(133); //Select
urban areas
Map.addLayer(C_urb_2018.mask(C_urb_2018), {
    'palette': 'FF0000'
}, 'CORINE Urban 2018');
```

Repeat the same steps for 2000, but visualize the pixels in gray.

```
// 2000 (1999-2001)
var CORINE_2000 =
CORINE.select('landcover').filterDate(ee.Date(
    '1999-01-01'));
var C_urb_2000 = CORINE_2000.mosaic().lte(133); //Select
urban areas
Map.addLayer(C_urb_2000.mask(C_urb_2000), {
    'palette': 'a5a5a5'
}, 'CORINE Urban 2000');
```

The output shows the extent of London's built-up land in 2000 in gray, and the larger extent in 2018 in red (Fig. 33.4).

Code Checkpoint A12c. The book's repository contains a script that shows what your code should look like at this point.

Last, we'll look at NLCD for 2016. Open a new script, search for "NLCD," import the NLCD: USGS National Land Cover Database, rename it to `NLCD`, and center it over Chicago, Illinois, USA.

```
// NLCD (Chicago)
// Center over Chicago.
Map.setCenter(-87.6324, 41.8799, 10);
```

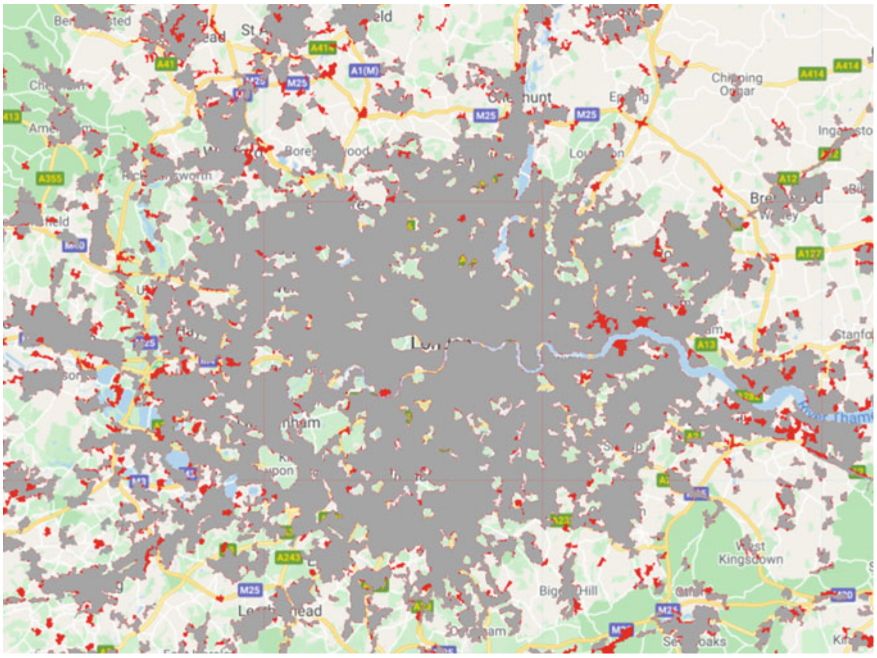


Fig. 33.4 Output of CORINE visualization

Select the `landcover` classification band and filter it to the 2016 classification. Previously, we have done so using the `filterDate` command, but you can also filter on the `system:index`.

```
// Select the land cover band.
var NLCD_lc = NLCD.select('landcover');

// Filter NLCD collection to 2016.
var NLCD_2016 = NLCD_lc.filter(ee.Filter.eq('system:index',
'2016'))
    .first();
Map.addLayer(NLCD_2016, {}, 'NLCD 2016');
```

Hit **Run** and view the classification visualization. Use the **Inspector** tab to explore the classification values. The shades of red represent different levels of development. Dark red is "Developed, High Intensity" and the lightest pink is "Developed, Open Space." You can look at the metadata description of this dataset to learn about the rest of the classes and their corresponding colors.

One of the benefits of classifications is that you can use them to calculate quantitative information such as the area of a specific land cover. Now we'll cover how to do so by summing the high-intensity urban development land cover in Chicago. Import a boundary of the city of Chicago and clip the NLCD classification to its extent.

```
// Calculate the total area of the 'Developed high
intensity' class (24) in Chicago.
var Chicago = ee.FeatureCollection(
    'projects/gee-book/assets/A1-2/Chicago');

// Clip classification to Chicago
var NLCD_2016_chi = NLCD_2016.clip(Chicago);
```

Select the "Developed, High Intensity" class (24), and mask the class with itself (to ensure that we calculate the area of only the highly developed pixels and no surrounding ones), then visualize the output.

```
// Set class 24 pixels to 1 and mask the rest.
var NLCD_2016_chi_24 = NLCD_2016_chi.eq(24).selfMask();
Map.addLayer(NLCD_2016_chi_24, {},
    'Chicago developed high intensity');
```

Multiply the pixels by their area, and use `reduceRegions` to sum the area of the pixels.

```
// Area calculation.
var areaDev =
NLCD_2016_chi_24.multiply(ee.Image.pixelArea())
    .reduceRegion({
        reducer: ee.Reducer.sum(),
        geometry: Chicago.geometry(),
        scale: 30
    })
    .get('landcover');
print(areaDev);
```

Table 33.1 Spatial and temporal extents of three classifications

	Pixel size (m)	Years available	Coverage	Number of urban classes
MODIS	500 m			
CORINE			European Union	
NLCD				4

Code Checkpoint A12d. The book's repository contains a script that shows what your code should look like at this point.

If everything worked correctly, you should see "203722704.70588234" printed to the **Console**. This means that highly developed land covered about 204,000,000 m^2 (204 km^2) in the city of Chicago in 2016. Let's roughly check this answer by looking up the area of the city of Chicago. Chicago is about 234 square miles (~ 606 km^2), which means highly developed areas cover about a third of the city, which visually matches what we see in the **Map** viewer.

Question 1. Return to your MODIS classification visualization of Accra and look at the pixels that became urban between 2002 and 2019. What land cover classes were they before they became urbanized? Hint: you will need to change the date of your full classification visualization to 2002 in order to answer this question.

Question 2. Return to your NLCD area calculation. Write new code to calculate the area of the highly developed class in Chicago in 2001. How much did the highly developed class change between 2001 and 2016? Report your answer in square meters.

Question 3. Fill out Table 33.1 on the spatial and temporal extents of the three classifications that we've just experimented with (some entries are filled in as an example). Based on your table, what are the benefits and the limitations of each of the classifications for studying urban areas?

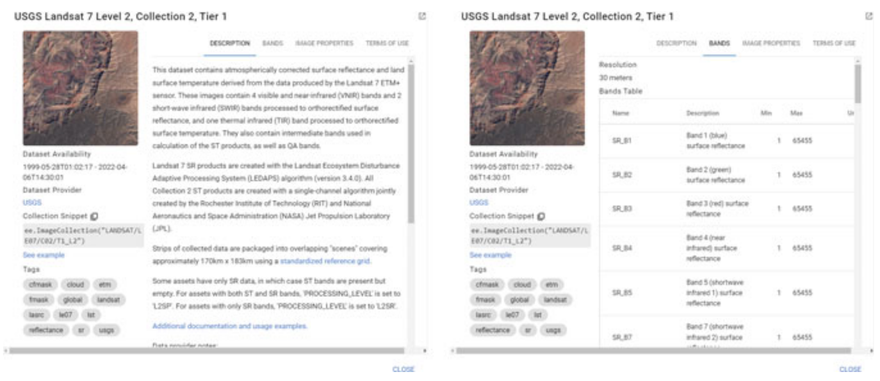
33.2.3 Section 3. Classifying Urban Areas

In the previous section, we relied on several LULC classifications to estimate the distribution of built-up land cover and its change over time. While for many applications these products would be sufficient, for others their temporal or spatial granularity may be a limiting factor. For example, your research question may require tracking monthly changes in the distribution of built-up land cover or depicting granular intra-city spatial patterns of built-up areas.

Luckily, with emerging and more accessible machine learning tools—like those available in Earth Engine—you can now create your own classification maps, including maps that depict the distribution of built-up land cover (Goldblatt et al. 2016). Machine learning deals with how machines learn rules from examples and generalize these rules to new examples. In this section, we will use one of the most common supervised machine learning classifiers: Random Forests (Breiman 2001).

You will use Landsat 7 imagery to classify built-up land cover in the city of Ahmedabad, Gujarat, India, at two points in time, 2020 and 2010. You will hand draw polygons (rectangles) representing areas of built-up and not-built-up land cover, and then build a Random Forest classifier and train it to predict whether a pixel is "built-up" or not.

First, start a new script. In the search bar, search for "USGS Landsat 7 Collection 2 Tier 1." Of the similarly named datasets that come up, select the one with the data set specifier "LANDSAT/LE07/C02/T1_L2." The metadata shows that this is an `ImageCollection` of images collected since 1999 (Fig. 33.5, left) and that each pixel is characterized by 11 spectral bands (Fig. 33.5, right). In this exercise, we will work only with bands 1–6. Import this `ImageCollection` and change the name of the variable to `L7`.


USGS Landsat 7 Level 2, Collection 2, Tier 1

Dataset Availability
1999-05-28T01:02:17 - 2022-04-06T14:30:01
Dataset Provider
USGS
Collection Snippet
```
ee.ImageCollection("LANDSAT/LE07/C02/T1_L2")
```
See example
Tags
cfmask cloud etm fmask global landsat lasrc le07 lst reflectance sr usgs

DESCRIPTION BANDS IMAGE PROPERTIES TERMS OF USE

This dataset contains atmospherically corrected surface reflectance and land surface temperature derived from the data produced by the Landsat 7 ETM+ sensor. These images contain 4 visible and near-infrared (VNIR) bands and 2 short-wave infrared (SWIR) bands processed to orthorectified surface reflectance, and one thermal infrared (TIR) band processed to orthorectified surface temperature. They also contain intermediate bands used in calculation of the ST products, as well as QA bands.

Landsat 7 SR products are created with the Landsat Ecosystem Disturbance Adaptive Processing System (LEDAPS) algorithm (version 3.4.0). All Collection 2 ST products are created with a single-channel algorithm jointly created by the Rochester Institute of Technology (RIT) and National Aeronautics and Space Administration (NASA) Jet Propulsion Laboratory (JPL).

Strips of collected data are packaged into overlapping "scenes" covering approximately 170km x 183km using a standardized reference grid.

Some assets have only SR data, in which case ST bands are present but empty. For assets with both ST and SR bands, 'PROCESSING_LEVEL' is set to 'L2SP'. For assets with only SR bands, 'PROCESSING_LEVEL' is set to 'L2SR'.

Additional documentation and usage examples.

CLOSE

USGS Landsat 7 Level 2, Collection 2, Tier 1

DESCRIPTION BANDS IMAGE PROPERTIES TERMS OF USE

Resolution
30 meters
Bands Table

Name	Description	Min	Max
SR_B1	Band 1 (blue) surface reflectance	1	65455
SR_B2	Band 2 (green) surface reflectance	1	65455
SR_B3	Band 3 (red) surface reflectance	1	65455
SR_B4	Band 4 (near infrared) surface reflectance	1	65455
SR_B5	Band 5 (shortwave infrared 1) surface reflectance	1	65455
SR_B7	Band 7 (shortwave infrared 2) surface	1	65455

CLOSE

Fig. 33.5 Metadata for the "USGS Landsat 7 Collection 2 Tier 1" dataset

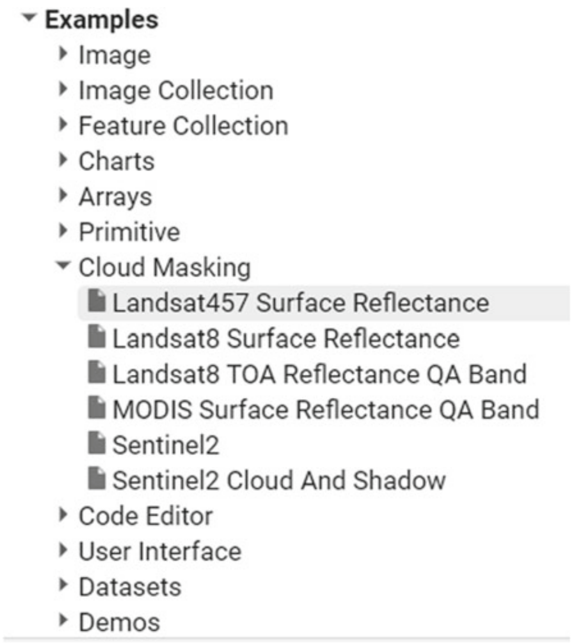

Examples
Image
Image Collection
Feature Collection
Charts
Arrays
Primitive
Cloud Masking
Landsat457 Surface Reflectance
Landsat8 Surface Reflectance
Landsat8 TOA Reflectance QA Band
MODIS Surface Reflectance QA Band
Sentinel2
Sentinel2 Cloud And Shadow
Code Editor
User Interface
Datasets
Demos

Fig. 33.6 Location of the surface reflectance code in the **Examples** repository

Next, you will create an annual composite for the year 2020 using the Google example **Landsat457 Surface Reflectance** script, located in the **Cloud Masking Examples** folder at the bottom of your Scripts section (Fig. 33.6).

This function computes surface reflectance. To create a composite for the year 2020, use the `ee.FilterDate` method to identify the images captured between January 1, 2020, and December 31, 2020, then map the function over that collection.

```
// Surface reflectance function from example:
function maskL457sr(image) {
    var qaMask =
image.select('QA_PIXEL').bitwiseAnd(parseInt('11111',
        2)).eq(0);
    var saturationMask = image.select('QA_RADSAT').eq(0);

    // Apply the scaling factors to the appropriate bands.
    var opticalBands =
image.select('SR_B.').multiply(0.0000275).add(-
        0.2);
    var thermalBand =
image.select('ST_B6').multiply(0.00341802).add(
        149.0);

    // Replace the original bands with the scaled ones and
apply the masks.
    return image.addBands(opticalBands, null, true)
        .addBands(thermalBand, null, true)
        .updateMask(qaMask)
        .updateMask(saturationMask);
}

// Map the function over one year of data.
var collection = L7.filterDate('2020-01-01', '2021-01-
01').map(
    maskL457sr);
var landsat7_2020 = collection.median();
```

Now, in the search bar, search for and navigate to Ahmedabad, India. Add the `landsat7_2020` composite to the map. Specify the red, green, and blue bands to achieve a natural color visualization, noting that in Landsat 7, band 3 (B3) is red, band 2 (B2) is green, and band 1 (B1) is blue. Also specify the colors' stretch parameters. In this case, we stretch the values between 0 and 0.3, but feel free to experiment with other stretches.

```
Map.addLayer(landsat7_2020, {
    bands: ['SR_B3', 'SR_B2', 'SR_B1'],
    min: 0,
    max: 0.3
}, 'landsat 7, 2020');
```

Code Checkpoint A12e. The book's repository contains a script that shows what your code should look like at this point.

The next step is probably the most time consuming in this exercise. In supervised classification, a classifier is trained with labeled examples to predict the labels of new examples. In our case, each training example will be a pixel, labeled as either built-up or not-built-up. Rather than selecting pixels for the training set one at a time as was done in Chap. 7, you will hand-draw polygons (rectangles) and label them as built-up or not-built-up, and assign to each pixel with the label of the spatially overlapping polygon. You will create two feature collections: one with 50 rectangles over built-up areas (labeled as "1") and one with 50 examples of rectangles over not-built-up areas (labeled as "0"). To do this, follow these steps:

Start by hand-drawing the not-built-up examples. Click on the rectangle icon in the upper-left corner of the screen (Fig. 33.7). Clicking on this icon will create a new variable (called `geometry`). Click on the gear-shaped icon of this imported variable. Change the name of the variable to `nbu` (stands for "not-built-up") (Fig. 8a). Change the **Import as** parameter from **Geometry** to **FeatureCollection** (Fig. 8b). Click on the + **Property** button (Fig. 8c), to create a new property called "class," and assign it with a value of 0 (Fig. 8d). You have just created a new empty `FeatureCollection`, where each feature will have a property called "class", with a value of 0 assigned to it.

The `nbu` `FeatureCollection` is now initialized, and empty. You now need to populate it with features: In this exercise, you will draw 50 rectangles drawn over areas representing not-built-up locations. To identify these areas, you can use either the 2020 Landsat 7 scene (`landsat7_2020`) as the background reference or use the high-resolution satellite basemap (provided in Earth Engine) as reference. Note, however, that you can use the basemap as reference only if you classify the current land cover, since Earth Engine's satellite background is continually updated.

For this exercise, we define "not-built-up" land cover as any type of land cover that is not-built-up (this includes vegetation, bodies of water, bare land, etc.). Click on the `nbu` layer you created under **Geometry Imports** (Fig. 9a) to select it (the rectangle button will be highlighted; see Fig. 9b) and draw 50 rectangles representing examples of areas that are not-built-up (Fig. 9c). It is important to select diverse and representative examples. To avoid exceeding the user memory limit, draw relatively small rectangles (see examples in Fig. 33.9).

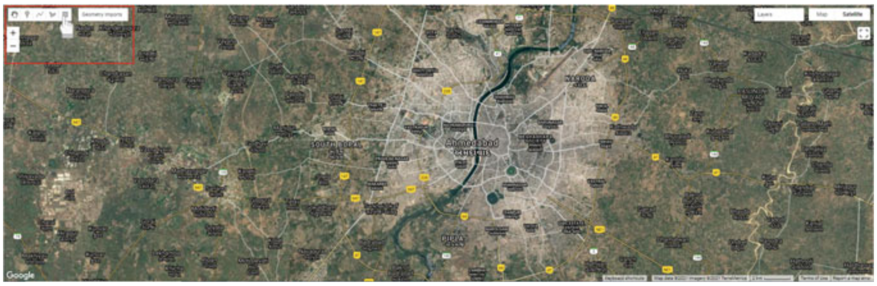
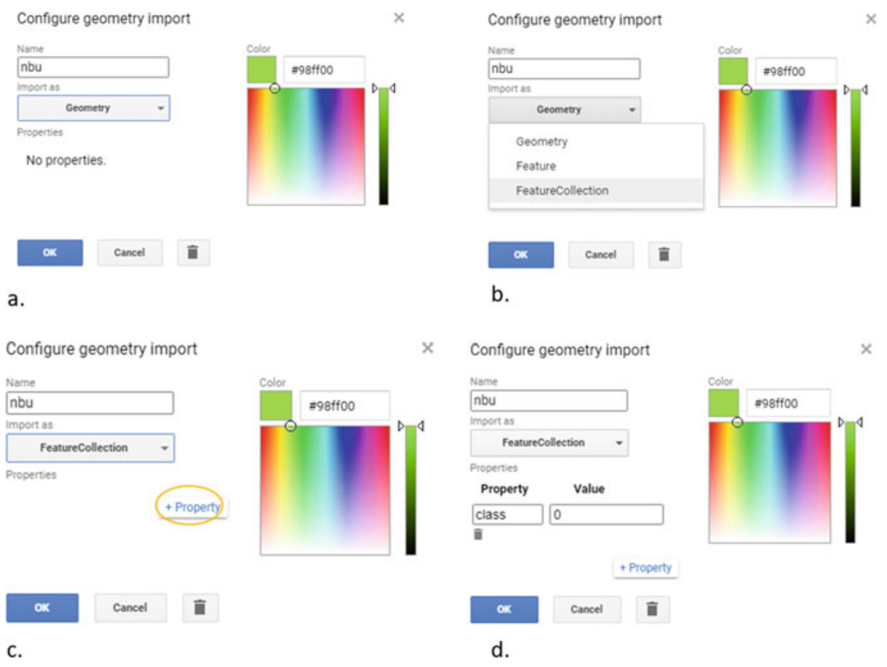


Fig. 33.7 To hand-draw examples of not-built-up rectangles, click on the rectangle icon in the upper-left corner of the screen


Configure geometry import

Name: nbu

Import as: Geometry

Properties: No properties.

Color: #98ff00

Configure geometry import

Name: nbu

Import as: Geometry

Configure geometry import

Name: nbu

Import as: FeatureCollection

Properties

Color: #98ff00

Configure geometry import

Name: nbu

Import as: FeatureCollection

Properties

Property	Value
class	0

Color: #98ff00

Fig. 33.8 Creating a new `FeatureCollection` for examples of not-built-up areas: **a** change the name of the variable to `nbu`; **b** change the parameter **Import as** parameter from **Geometry** to **FeatureCollection**; **c** click on the + **Property** button; **d** to create a new property called “class,” and assign to it the value 0

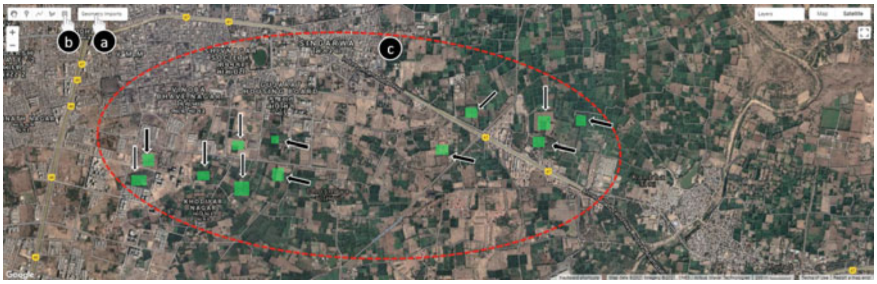


Fig. 33.9 Rectangles representing example locations of not-built-up areas

Tip: To delete a feature, simply click on **Esc**, select the feature you want to delete, and click **Delete**. You can also click on **Esc** if you want to navigate (zoom in, zoom out, pan). If you’d like to re-draw a rectangle, click again on the name **nbu** layer under **Geometry Imports** and modify the geometry. When done, click on the **Exit** button or **Esc**.

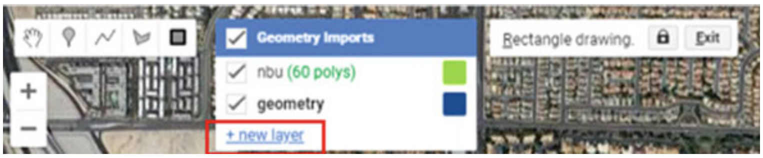


Fig. 33.10 Creating a new layer for the built-up FeatureCollection To create a new empty feature collection of built-up examples, click on the + **new layer** (at the bottom of the **Geometry Imports**)

Note that while adding more examples may improve the accuracy of your classification, creating too many examples could potentially improve the accuracy of the classification of your specific area of interest—but it may not fit well in other geographical areas. That is an example of a type of "overfitting," in which samples used to train a model are overly specific to one area and not generalizable to other locations.

Now that you created 50 examples of not-built-up areas, it is time to create a second additional `FeatureCollection` of 50 examples of built-up areas. As with the not-built-up examples, you will first create an empty `FeatureCollection` and then populate it with example features.

Important: you will need to create a new layer. Make sure that you do not add these new examples to the not-built-up `FeatureCollection`. To create a new layer, click on + **new layer** (Fig. 33.10). This will create a new variable called `geometry`.

Click on the gear-shaped icon next to the new variable you just created. Change the name of the variable from `geometry` to `bu` (stands for built-up) (Fig. 11a). Change the **Import as** parameter from **geometry** to **FeatureCollection** (Fig. 11b). Click on + **Property** (Fig. 11c), and create a property called `class` (Fig. 11d), and assign to it a value of 1 (Fig. 11e). Recall that this is the same property name you used for the `nbu` examples. Assign the features with a value "1" (Fig. 11e). Also change the color of the `bu` examples to red (Fig. 11f), representing built-up areas. Click **OK**.

The `bu` `FeatureCollection` is now initialized, and empty. To populate it with examples of built-up areas, hover over the **bu** `FeatureCollection` (Fig. 12a), select the rectangle tool (Fig. 12b), and draw 50 rectangles over built-up areas. Keep (keep the size of the rectangles similar to the not-built-up examples) (Fig. 33.12, location 'c').

You've now created two feature collections: 50 features representing built-up areas (the variable is called `bu`), and 50 features representing not-built-up areas (the variable is called `nbu`). Next, you will merge these two feature collections to create one collection of 100 features. Remember that in both collections, you added a property called `class`; features in the `FeatureCollection` `nbu` were assigned a value of 0 and features in the `FeatureCollection` `nbu` were assigned a value of 1. The merged `FeatureCollection` will also include the property `class`: 50 features with a value of 1 and 50 with a value

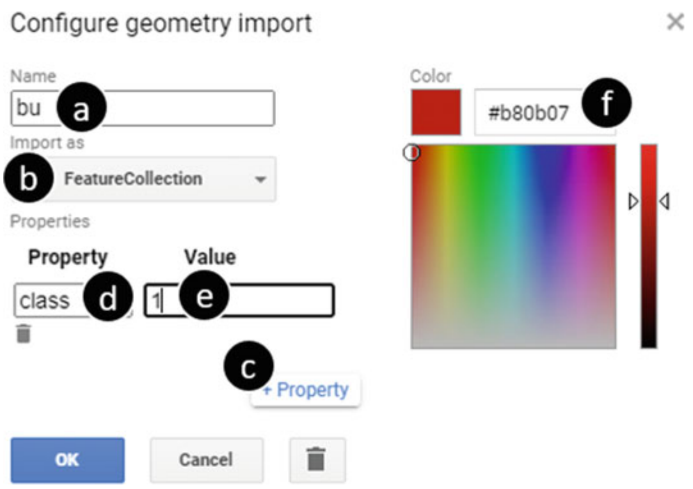


Fig. 33.11 Creating a new `FeatureCollection` for examples of built-up areas: **a** change the name of the variable to "bu"; **b** change the parameter **Import as** parameter from **Geometry** to **FeatureCollection**; **c** click on the + **Property** button; **d** to create a new property called "class," and assign to it the value 1

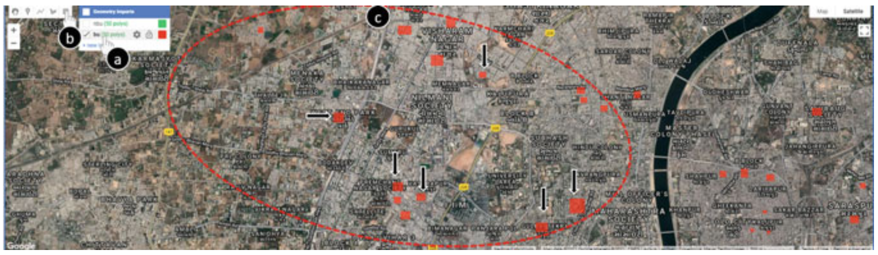


Fig. 33.12 Rectangles representing examples of built-up areas. To create these examples, hover over the new "bu" feature collection you just created (**a**), select the rectangle tool (**b**), and draw 50 examples of rectangles over built-up areas (**c**)

of 0. To merge the feature collections, use the method merge. Call the merged `FeatureCollection` with the variable `lc`.

```
var lc = nbu.merge(bu);
```

Code Checkpoint A12f. The book's repository contains a script that shows what your code should look like at this point.

List the properties that the classifier will use to determine whether a pixel is built-up or not-built-up. Recall that Landsat 7 has 11 bands; two of them are QA Bitmask bands, which likely do not vary by land cover. Create a new variable called bands with a list of the bands to be used by the classifier.

```
var bands = ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5',
'ST_B6', 'SR_B7'];
```

Next, you will create the labeled examples (pixels) that will serve as the training set of the classifier. Remember that, at this point, the pixels are not yet labeled; you labeled only the rectangles (stored in the `lc` variable), which are not associated with the bands of the Landsat pixels. To associate the Landsat pixels with a label, you will sample the pixels that overlap with these rectangles and assign them with the class of the overlapping rectangle.

This can be done with the method `sampleRegions`. Define a new variable and call it `training`. Under this variable, call the 2020 Landsat composite (`landsat7_2020`), select the relevant bands (the variable `bands`), and sample the overlapping pixels. You will sample the `landsat7_2020` pixels that overlap with the `FeatureCollection` `lc` and copy the property `class` from each feature to the overlapping pixel (each pixel will now include the 9 properties and a class, either 1 or 0). Set the argument `scale` to `30` (30 m—Landsat 7's spatial resolution—is the scale of the projection to sample in). These steps are shown in the code block below and were described in Chap. 6.

```
var training = landsat7_2020.select(bands).sampleRegions({
    collection: lc,
    properties: ['class'],
    scale: 30
});
```

Now it is time to create the classifier. Create an empty Random Forest classifier using the method `ee.Classifier.smileRandomForest`. You can keep most of the arguments' values as default and only set the number of decision trees in the forest (the argument `numberOfTrees`) to 20. Use the method `train` to train the classifier. This method trains a classifier on a collection of features, using the specified numeric properties of each feature as training data. The collection to train on is called `training` and the name of the property containing the class value is called `'class'`. The list of properties (`inputProperties`) to be considered by the classifier is stored in the band's variable. Note that, by default, the classifier predicts the class of the pixel (in this case, "1," built-up, or "0," not-built-up). You could also set the mode of the classifier to `PROBABILITY`, which will result in the probability that a pixel has the value "1." In this example, we used the default `CLASSIFIER` mode).

```
// Create a random forest classifier with 20 trees.
var classifier = ee.Classifier.smileRandomForest({
    numberOfTrees: 20
}).train({ // Train the classifier.
    // Use the examples we got when we sampled the pixels.
    features: training,
    // This is the class that we want to predict.
    classProperty: 'class',
    // The bands the classifier will use for training and
classification.
    inputProperties: bands
});
```

In the previous step, you trained a classifier. Next, you will use the trained model to predict that class on new pixels, using the method `classify`. You'll define a new variable, `classified20`, which will take the Landsat composite; select the relevant bands (the variable `bands`); and classify this image using the trained classifier (called `'classifier'`).

```
 // Apply the classifier on the 2020 image.
 var classified20 =
landsat7_2020.select(bands).classify(classifier);
```

You can visualize the classification by adding the classified image to the map. In our case, we have only two classes (built-up and not-built-up); thus, the classifier's prediction is binary (either "1" for built-up, or "0" for not-built-up). To show only the pixels with a value of 1, use the `mask` method to mask out the pixels with a value of 0. Set the visualization parameters to visualize the remaining pixels as red, with a transparency of 60%.

```
Map.addLayer(classified20.mask(classified20), {
    palette: ['#ff4218'],
    opacity: 0.6
}, 'built-up, 2020');
```

Code Checkpoint A12g. The book's repository contains a script that shows what your code should look like at this point.

Lastly, many applications require mapping the extent of built-up land cover at more than one point in time (e.g., to understand urbanization processes). As you recall, Landsat 7 has been collecting imagery from every location on Earth since 1999. We can use our trained classifier—which was trained based on Landsat 7 2020 imagery—to predict the extent of built-up land cover in any year collected (with the assumption that the characteristics of a built-up pixel did not change over time). In this exercise, you will use the trained classifier to map the built-up land cover in 2010.

First, create a composite for 2010 using the `simpleComposite` method:

```
var landsat7_2010 = L7.filterDate('2010-01-01', '2010-12-
31')
    .map(maskL457sr).median();
```

Now, you can use the classifier to classify the 2010 image using the `classify` method. Classify the 2010 image (`landsat7_2010`) and add it to the map, this time in a yellow color.

```
// Apply the classifier to the 2010 image.
var classified10 = landsat7_2010.select(bands).classify(
    classifier);
Map.addLayer(classified10.mask(classified10), {
    palette: ['#f1ff21'],
    opacity: 0.6
}, 'built-up, 2010');
```

The result should be something like what is presented in Fig. 33.13. Built-up land cover in 2020 is presented in red, and in 2010 in yellow.

Want to see which areas were developed in the period between 2010 and 2020? Create a new variable and call it `difference`. In this variable, subtract the values of 2010 from 2020. Any pixel that changed from “0” in 2010 to “1” in 2020 will be assigned with a value of 1. You can then visualize this difference on the map in blue.

```
var difference = classified20.subtract(classified10);

Map.addLayer(difference.mask(difference), {
    palette: ['#315dff'],
    opacity: 0.6
}, 'difference');
```

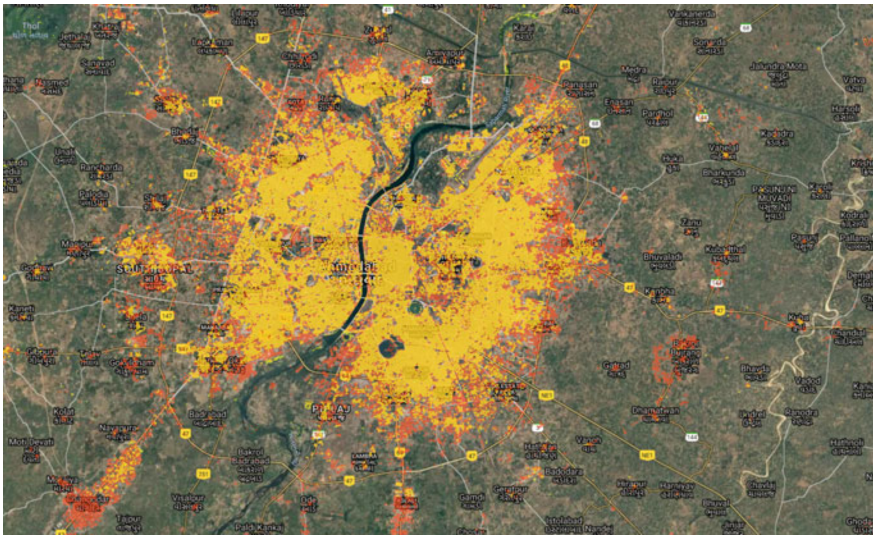


Fig. 33.13 The classified built-up land cover in 2020 (red) and in 2010 (yellow)

Code Checkpoint A12h. The book's repository contains a script that shows what your code should look like at this point.

If your code worked as expected, you should be able to see the three layers you classified: the built-up land cover in 2010, in 2020, and the difference between the two.

Question 4. Calculate the total built-up land cover in Ahmedabad in 2010 and in 2020. How much built-up area was added to the city? What is the percent increase of this growth?

Note: Use this `FeatureCollection`, filtered to Ahmedabad, to bound your area calculation:

```
var indiaAdmin3 = ee.FeatureCollection(
    'projects/gee-book/assets/A1-2/IndiaAdmin3');
var ahmedabad = indiaAdmin3.filterMetadata('VARNAME_3',
'equals',
    'Ahmadabad city');
Map.addLayer(ahmedabad, {}, 'Ahmedabad city');
```

Question 5. Add two lines to your code to visualize the built-up area that was added between 2010 and 2020.

33.3 Synthesis

Assignment 1. In this exercise, we mapped built-up land cover with Landsat 7 imagery. Now try to repeat this exercise using Sentinel-2 as an input to the classifier. Calculate the total area of built-up land cover in Ahmedabad in 2020, and answer the following questions.

Note, that you can create an annual Sentinel-2 composite by calculating the median value of all cloud-free scenes captured in a given year (see an example of how to filter by cloud values in Sect. 33.2.1 of this chapter). Note that the property that stores the cloudy pixel percentage in the Sentinel-2 `ImageCollection` is `'CLOUDY_PIXEL_PERCENTAGE'`.

Question 6. How do the area totals differ from each other? (Use the Ahmedabad `FeatureCollection` boundary from Question 4)?

Question 7. Why do you think that is the case? Think about spatial and spectral resolutions.

Question 8. What is the area of the MODIS urban classification in 2019?

Question 9. Is it larger or smaller than Landsat/Sentinel? Why?

1. How do the area totals differ from each other? (Use the Ahmedabad FeatureCollectionfeature collection boundary from Question 4 above.)
2. Why do you think that is the case? Think about spatial and spectral resolutions.
3. What is the area of the MODIS urban classification in 2019?
4. Is it larger or smaller than Landsat/Sentinel? Why?

33.4 Conclusion

In this chapter, you have learned how to visualize and quantify the magnitude and pace of urbanization anywhere in the world. Combined with your knowledge from other chapters, you're now equipped to look at urban heat islands, food and water system stresses, and many other socio-environmental impacts of urbanization (Bazaz et al. 2018). As more and higher-resolution remote sensing data become available, the scope of questions we can answer about urban areas will widen and the accuracy of our predictions will improve. This knowledge can be used for policy and decision-making, in particular in the context of the United Nations Sustainable Development Goals (United Nations 2020; Prakash et al. 2020).

References

Bazaz A, Bertoldi P, Buckeridge M et al (2018) Summary for urban policymakers—what the IPCC special report on 1.5°C means for cities. IIHS Indian Institute for Human Settlements

Breiman L (2001) Random forests. Mach Learn 45:5–32. https://doi.org/10.1023/A:1010933404324

Goldblatt R, Stuhlmacher MF, Tellman B et al (2018) Using Landsat and nighttime lights for supervised pixel-based image classification of urban land cover. Remote Sens Environ 205:253–275. https://doi.org/10.1016/j.rse.2017.11.026

Goldblatt R, You W, Hanson G, Khandelwal AK (2016) Detecting the boundaries of urban areas in India: a dataset for pixel-based image classification in Google Earth Engine. Remote Sens 8:634. https://doi.org/10.3390/rs8080634

Prakash M, Ramage S, Kavvada A, Goodman S (2020) Open Earth observations for sustainable urban development. Remote Sens 12:1646. https://doi.org/10.3390/rs12101646

United Nations Department of Economic and Social Affairs (2019) World urbanization prospects: the 2018 revision

United Nations Department of Economic and Social Affairs (2020) The sustainable development goals report 2020

Built Environments

34

Erin Trochim

Overview
The built environment consists of the human-made physical parts of the environment, including homes, buildings, streets, open spaces, and infrastructure. This chapter will focus on analyzing global infrastructure datasets.

Learning Outcomes

- Quantifying road characteristics.
- Comparing road and transmission line distributions.
- Contrasting changes in impervious surfaces with flooding.
- Understanding vector-based versus raster-based approaches.

Helps if you know how to

- Create a function for code reuse (Chap. 1).
- Summarize an `ImageCollection` with reducers (Chaps. 12 and 12).
- Filter a `FeatureCollection` to obtain a subset (Chaps. 22 and 23).
- Convert between raster and vector data (Chap. 23).
- Join elements of vector datasets together (Chap. 25).

E. Trochim (✉)
Alaska Center for Energy and Power, University of Alaska Fairbanks, PO Box 755910, Fairbanks, AK 99775-5910, USA
e-mail: edtrochim@alaska.edu

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_34

34.1 Introduction to Theory

Until the recent past, data about the built environment were mostly produced and managed at local and regional levels, and global datasets were very rare. Municipal, state, and national governments require easy access to data in order to plan, build, and maintain assets, which can be either privately or publicly owned. In the last decade, there has been a push to produce globally consistent data to support economic assessment and environmental transitions. The existence of these data makes it possible to explore the impact of factors including weather, climate, and growth over time.

Built environment datasets currently available in the Earth Engine Data Catalog include the Global Power Plant Database, Open Buildings V1 Polygons (which currently includes over half of Africa), and Global Artificial Impervious Area. Complementary environmental information includes the Global Flood Database. In addition, the Awesome GEE Community Catalog hosts the Global Roads Inventory Project, Global Power System, and Global Fixed Broadband and Mobile (Cellular) Network Performance collections.

As will be seen in the rest of this chapter, information on the built environment is most often stored as vector data: that is, as points, lines, and polygons. This format readily lends itself to representing, for example, bridges as points, roads as lines, and buildings as polygons. Many of the built environment datasets listed above now have both raster and vector components. It is useful to understand the utility of each of these formats and how to extract information between them and ancillary datasets.

34.2 Practicum

34.2.1 Section 1. Road Characteristics

We will start by calculating road length using the Global Roads Inventory Project (GRIP) dataset (Meijer et al. 2018). This dataset was created to provide a consistent global roadways' dataset for environmental and biodiversity assessments. For this exercise, we will focus on examining the largest countries in Africa, North America, and Europe.

Start by importing the `grip4` datasets into Earth Engine using the following code, and examine how they look in the Code Editor.

```
// Import roads data.
var grip4_africa = ee.FeatureCollection(
        'projects/sat-io/open-datasets/GRIP4/Africa'),
    grip4_north_america = ee.FeatureCollection(
        'projects/sat-io/open-datasets/GRIP4/North-
America'),
    grip4_europe = ee.FeatureCollection(
        'projects/sat-io/open-datasets/GRIP4/Europe');

// Check the roads data sizes.
print('Grip4 Africa size', grip4_africa.size());
print('Grip4 North America size',
grip4_north_america.size());
print('Grip4 Europe size', grip4_europe.size());

// Display the roads data.
Map.addLayer(ee.FeatureCollection(grip4_africa).style({
    color: '413B3A',
    width: 1
}), {}, 'Grip4 Africa');
Map.addLayer(ee.FeatureCollection(grip4_north_america).styl
e({
    color: '413B3A',
    width: 1
}), {}, 'Grip4 North America');
Map.addLayer(ee.FeatureCollection(grip4_europe).style({
    color: '413B3A',
    width: 1
}), {}, 'Grip4 Europe');
```

Using the Large Scale International Boundary dataset from the US Office of the Geographer, below we will import the simplified country boundaries. After calculating the area enclosed by the boundary of each country, we will select Algeria, a quite large country with many roads.

```
// Import simplified countries.
var countries =
ee.FeatureCollection('USDOS/LSIB_SIMPLE/2017');

// Add a function to calculate the feature's geometry area.
// Add the function as a property.
var addArea = function(feature) {
    return feature.set({
        areaKm: feature.geometry().area().divide(1000 *
            1000)
    }); // km2 squared
};

// Map the area getting function over the
FeatureCollection.
var countriesArea = countries.map(addArea);

// Filter to the largest country in Africa.
var Algeria =
countriesArea.filter(ee.Filter.inList('country_na', [
    'Algeria'
]));

// Display the selected countries.
Map.addLayer(Algeria.style({
    fillColor: 'b5ffb4',
    color: '00909F',
    width: 1.0
}), {}, 'Algeria');
```

Next, we will calculate the road density for Algeria. For this example, we will implement a function to join the roads to each of the countries. The approach below joins the roads to the countries if there is spatial overlap between the features. Then, for each joined road in the specified country, an intersection is performed to keep only the portion of the road within the country. The length of the road is calculated and added as a property per feature, as is the road density per country.

```
// This function calculates the road length per country for
the associated GRIP dataset.
var roadLength4Country = function(country, grip4) {

    // Join roads to countries.
    var intersectsFilter = ee.Filter.intersects({
        leftField: '.geo',
        rightField: '.geo',
        maxError: 10
    });

    var grip4Selected = grip4.filterBounds(country);

    var countriesWithRds = ee.Join.saveAll('roads').apply({
        primary: country,
        secondary: grip4Selected,
        condition: intersectsFilter
    }).filter(ee.Filter.neq('roads', null));

    // Return country with calculation of roadLength and
roadsPerArea.
    return countriesWithRds.map(function(country) {
        var roadsList = ee.List(country.get('roads'));
        var roadLengths = roadsList.map(function(road) {
            return ee.Feature(road).intersection(
                country, 10).length(10);
        });
        var roadLength = ee.Number(roadLengths.reduce(ee
            .Reducer.sum()));
        return country.set({
            roadLength: roadLength.divide(
                1000), // Convert to km.
            roadsPerArea: roadLength.divide(ee
                .Number(country.get('areaKm'))
            )
        });
    }).select(['country_na', 'areaKm', 'roadLength',
        'roadsPerArea'
    ]);
};

// Apply the road length function to Algeria.
var roadLengthAlgeria = roadLength4Country(Algeria,
grip4_africa);
```

Print the roads per area in Algeria.

```
// Print the road statistics for Algeria.
print('Roads statistics in Algeria', roadLengthAlgeria);
```

Question 1. How many roads are there in Africa? Hint: Check the size of the `grip4` Africa dataset.

Question 2. Which continent has the most roads: Africa, Europe, or North America?

Question 3. How many roads per square kilometer are there in Algeria?

Question 4. What is the total road length in kilometers in Algeria?

Code Checkpoint A13a. The book's repository contains a script that shows what your code should look like at this point.

Earth Engine is powerful, but its processing power is not infinite. If you try applying the same code to countries such as Canada and France, it does not run very well interactively due to the enormous size of the request. To successfully execute long-running, complex tasks, you can export results. Exporting forces Earth Engine to continue to run for a very long time; in that case, you cannot see the results interactively, but you will be able to get an answer for even very large problems, and then load the exported data. An example of this is shown below.

```
// Export feature collection to drive.
Export.table.toDrive({
    collection: roadLengthAlgeria,
    description: 'RoadStatisticsforAlgeria',
    selectors: ['country_na', 'roadLength', 'roadsPerArea']
});
```

Let us think about how to simplify this analysis, to see if we can optimize the computation in an interactive environment. Print the first feature of the grip4 Africa feature collection and display it using the following commands.

```
// Print the first feature of the grip4 Africa feature
collection.
print(grip4_africa.limit(1));

Map.setCenter(-0.759, 9.235, 6);
Map.addLayer(grip4_africa.limit(1),
    {},
    'Road length comparison');
```

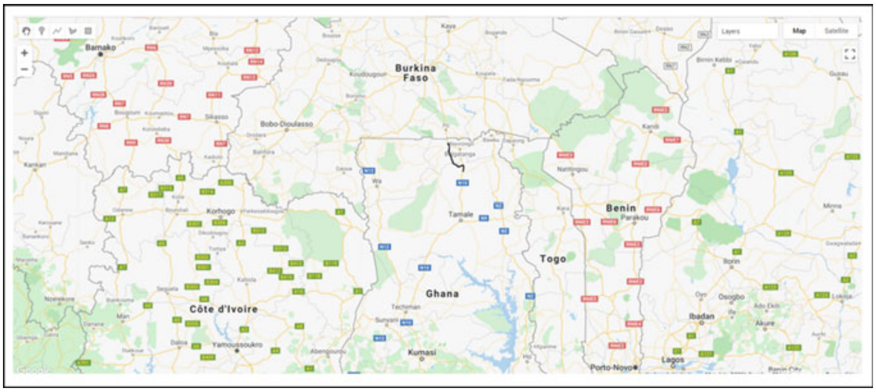


Fig. 34.1 Exploring the GRIP datasets by visualizing the first feature in the Africa region. According to the feature properties, the road is 0.76, but the unit is not specified. Compare this to the scale at the bottom of the **Map** panel in the Code Editor

You can find the road in northern Ghana as shown in Fig. 34.1.

In the **Console**, examine the properties of the road. Notice that there is an existing column called Shape_Leng. The length of each road appears to have already been precalculated and included as a value. The length of this road is given as 0.76, but the unit is not specified. This looks suspicious in comparison to the scale—the road is obviously longer than 0.76 m, kilometers, or miles. It is easy to recalculate the length of the lines: use the following code to do so and compare the results.

```
// This function adds line length in km.
var addLength = function(feature) {
    return feature.set({
        lengthKm: feature.length().divide(1000)
    });
};

// Calculate the line lengths for all roads in Africa.
var grip4_africaLength = grip4_africa.map(addLength);

// Compare with other values.
print('Calculated road length property',
grip4_africaLength.limit(1));
```

The road in Ghana has a new calculated length of about 84 km, which looks accurate.

Repeat the road calculation analysis, but examine each step. First, filter the road data in Africa for Algeria. Visualize the result, taking note of whether any roads are included that originate in Algeria but extend into other countries. Then, reduce the lengthKm column and calculate the sum in the filtered Algeria road data.

```
// Repeat the analysis to calculate the length of all
roads.
// Filter the table geographically: only keep roads in
Algeria.
var grip4_Algeria =
grip4_africaLength.filterBounds(Algeria);

// Visualize the output.
Map.addLayer(grip4_Algeria.style({
    color: 'green',
    width: 2.0
}), {}, 'Algeria roads');

// Sum the lengths for roads in Algeria.
var sumLengthKmAlgeria = ee.Number(
    // Reduce to get the sum.
    grip4_Algeria.reduceColumns(ee.Reducer.sum(),
['lengthKm'])
    .get('sum')
);

// Print the result.
print('Length of all roads in Algeria',
sumLengthKmAlgeria);
```

Question 5. What is the total recalculated road length in kilometers in Algeria?

Question 6. Is this higher or lower than the first value?

Code Checkpoint A13b. The book's repository contains a script that shows what your code should look like at this point.

The difference in values when recalculating is due to extra roads being included in the second calculation. This was due to skipping the joining and intersection of the Algeria feature collection. Using only filterBounds to approximate the road limits was not as accurate.

Let us try another method to make this more computationally efficient. Rasterize the roads by interpolating the current feature collection into an image, and use reduceRegions to calculate the area of the road pixels. For this exercise, set the scale as 100 m. In order to convert the area into the appropriate dimension and units, divide the results by 1000 and apply a square root to transform the length into kilometers.

```
// Repeat the analysis again to calculate length of all
roads using rasters.
// Convert to raster.
var empty = ee.Image().float();

var grip4_africaRaster = empty.paint({
    featureCollection: grip4_africaLength,
    color: 'lengthKm'
}).gt(0);

Map.addLayer(grip4_africaRaster, {
    palette: ['orange'],
    max: 1
}, 'Rasterized roads');

// Add reducer output to the features in the collection.
var AlgeriaRoadLength = ee.Image.pixelArea()
    .addBands(grip4_africaRaster)
    .reduceRegions({
        collection: Algeria,
        reducer: ee.Reducer.sum(),
        scale: 100,
    }).map(function(feature) {
        var num = ee.Number.parse(feature.get('area'));
        return feature.set('length',
num.divide(1000).sqrt()
            .round());
    });

// Print the first feature to illustrate the result.
print('Length of all roads in Algeria calculated via
rasters', ee
    .Number(AlgeriaRoadLength.first().get('length')));
```

Notice that this value is about 13%, or 7000 km, lower than the first estimate.

Take a look at the rasterized roads' visualization. With no scale set, it looks very similar to the vectors. Zoom in to an area like the example in Fig. 34.2 and examine the structure of the vectors, rasters, and roads themselves. Note that, in some areas, divided roads are represented by two separate features (vectors). The initial rasters match the scale of the vector. One 100 m pixel, though, would cover both roads and represent only a single unit length. Understanding how the data represent the

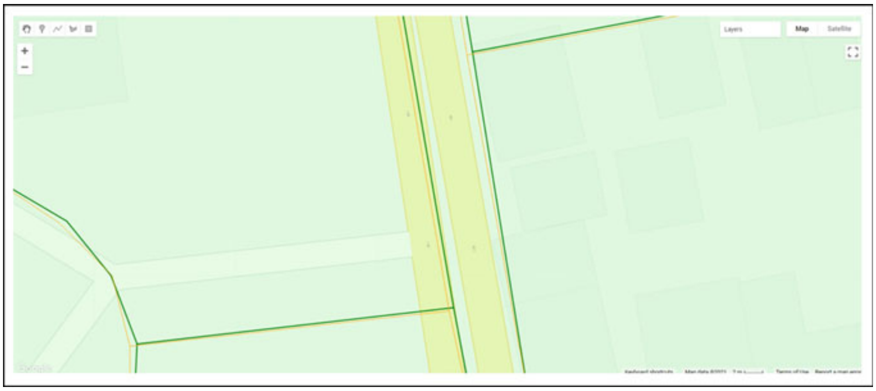


Fig. 34.2 Examining roads from the GRIP dataset as features (in green) versus rasters (in orange) overlaid with transparency on the **Map** view. There can be offset between datasets and different spatial shapes (lines, polygons, and rasters)

actual built environment is critical for accuracy and precision estimates. There is also a tradeoff in computation time and whether the calculations can be performed on the fly.

The advantage to this approach is that the analysis can be performed interactively across much larger areas. We will test this by calculating total road length for the largest countries in Africa, North America, and Europe, which are Algeria, Canada, and France.

```
// Calculate line lengths for all roads in North America
and Europe.
var grip4_north_americaLength =
grip4_north_america.map(addLength);
var grip4_europeLength = grip4_europe.map(addLength);

// Merge all vectors.
var roadLengthMerge = grip4_africaLength.merge(
    grip4_north_americaLength).merge(grip4_europeLength);

// Convert to raster.
var empty = ee.Image().float();

var roadLengthMergeRaster = empty.paint({
    featureCollection: roadLengthMerge,
    color: 'roadsPerArea'
}).gt(0);
```

```
// Filter to largest countries in Africa, North America and
Europe.
var countriesSelected = countries.filter(ee.Filter.inList(
    'country_na', ['Algeria', 'Canada', 'France']));

// Clip image to only countries of analysis.
var roadLengthMergeRasterClipped = roadLengthMergeRaster
    .clipToCollection(countriesSelected);

// Add reducer output to the features in the collection.
var countriesRoadLength = ee.Image.pixelArea()
    .addBands(roadLengthMergeRasterClipped)
    .reduceRegions({
        collection: countriesSelected,
        reducer: ee.Reducer.sum(),
        scale: 100,
    }).map(function(feature) {
        var num = ee.Number.parse(feature.get('area'));
        return feature.set('length',
num.divide(1000).sqrt()
            .round());
    });

// Compute totaled road lengths in km, grouped by country.
print('Length of all roads in Canada',
countriesRoadLength.filter(ee
    .Filter.equals('country_na', 'Canada')).aggregate_sum(
    'length'));
print('Length of all roads in France',
countriesRoadLength.filter(ee
    .Filter.equals('country_na', 'France')).aggregate_sum(
    'length'));
```

Question 7. Which country has the highest total length of roads?

Question 8. Explore the effect of the scale value by reprojecting the rasterized roads (for example, `Map.addLayer(grip4_africaRaster.reproject({crs: 'EPSG:4326', scale: 100}), {palette: ['orange'], max: 1}, 'Rasterized roads 100 m'))`. Does increasing the scale result in an over- or underestimation of roads? What is an optimal scale to still allow the large calculations to run in real time?

Code Checkpoint A13c. The book's repository contains a script that shows what your code should look like at this point.

34.2.2 Section 2. Road and Transmission Line Comparison

Next, let us compare the overlap between roads and transmission lines, which often are found in the same vicinity. We will test this concept by examining the Global Power System data (Arderne et al. 2020).

Let us start by creating a new script and importing data into the Code Editor. As in the first exercise, import the GRIP road datasets. We will reuse the same `addLength` function from the previous exercise and apply it to roads in Africa. Then, convert the roads to a raster using length in kilometers as the pixel value.

```
// Import roads data.
var grip4_africa = ee.FeatureCollection(
        'projects/sat-io/open-datasets/GRIP4/Africa'),
    grip4_europe = ee.FeatureCollection(
        'projects/sat-io/open-datasets/GRIP4/Europe'),
    grip4_north_america = ee.FeatureCollection(
        'projects/sat-io/open-datasets/GRIP4/North-
America');

// Add a function to add line length in km.
var addLength = function(feature) {
    return feature.set({
        lengthKm: feature.length().divide(1000)
    }); // km;
};

// Calculate line lengths for all roads in Africa.
var grip4_africaLength = grip4_africa.map(addLength);

// Convert the roads to raster.
var empty = ee.Image().float();

var grip4_africaRaster = empty.paint({
    featureCollection: grip4_africaLength,
    color: 'lengthKm'
});
```

Next, import the simplified countries again and filter the feature collection, this time to all countries on the continent of Africa.

Import the global power system data that represent transmission lines. For this exercise, we will use both the OpenStreetMap and predicted values in this dataset. Validation of this dataset included several parts of Africa, so it tends to have higher accuracy here than in other parts of the world like the Arctic.

Apply a filter to the transmission lines' data to limit the analysis to Africa. Calculate the length of the lines and convert it to rasters, as with the roads' data.

```
// Import simplified countries.
var countries =
ee.FeatureCollection('USDOS/LSIB_SIMPLE/2017');

// Filter to Africa.
var Africa = countries.filter(ee.Filter.eq('wld_rgn',
'Africa'));

// Import global power transmission lines.
var transmission = ee.FeatureCollection(
    'projects/sat-io/open-datasets/predictive-global-power-
system/distribution-transmission-lines'
);

// Filter transmission lines to Africa.
var transmissionAfrica = transmission.filterBounds(Africa);

// Calculate line lengths for all transmission lines in
Africa.
var transmissionAfricaLength =
transmissionAfrica.map(addLength);

// Convert the transmission lines to raster.
var transmissionAfricaRaster = empty.paint({
    featureCollection: transmissionAfricaLength,
    color: 'lengthKm'
});
```

Question 9. In this exercise, we are rasterizing our feature collections to calculate spatial association indicators. If the data were left as a feature collection, what would be the simplest way of comparing the two datasets in a similar way?

Code Checkpoint A13d. The book's repository contains a script that shows what your code should look like at this point.

In order to compare the road and transmission line rasters, they need to be stacked together into a single image. It is a good idea to rename the bands to keep track of the input values. Clip this dataset to the Africa feature collection in order to minimize computation in the following steps.

```
// Add roads and transmission lines together into one
image.
// Clip to Africa feature collection.
var stack = grip4_africaRaster
    .addBands(transmissionAfricaRaster)
    .rename(['roads', 'transmission'])
    .clipToCollection(Africa);
```

Next, use the following code to calculate the spatial statistics' local Geary's C (Anselin 1995). This will indicate clustering due to spatial autocorrelation. In this case, we are comparing whether roads and transmission lines are located together. Geary's C values of − 1 indicate dispersion, while those of 1 indicate clustering.

```
// Calculate spatial statistics: local Geary's C.
// Create a list of weights for a 9x9 kernel.
var list = [1, 1, 1, 1, 1, 1, 1, 1, 1];

// The center of the kernel is zero.
var centerList = [1, 1, 1, 1, 0, 1, 1, 1, 1];

// Assemble a list of lists: the 9x9 kernel weights as a 2-
D matrix.
var lists = [list, list, list, list, centerList, list,
list, list, list
];

// Create the kernel from the weights.
// Non-zero weights represent the spatial neighborhood.
var kernel = ee.Kernel.fixed(9, 9, lists, -4, -4, false);

// Use the max among bands as the input.
var maxBands = stack.reduce(ee.Reducer.max());

// Convert the neighborhood into multiple bands.
var neighs = maxBands.neighborhoodToBands(kernel);
```

```
// Compute local Geary's C, a measure of spatial
association
// - 0 indicates perfect positive autocorrelation/clustered
// - 1 indicates no autocorrelation/random
// - 2 indicates perfect negative autocorrelation/dispersed
var gearys =
maxBands.subtract(neighs).pow(2).reduce(ee.Reducer.sum())
    .divide(Math.pow(9, 2));

// Convert to a -/+1 scale by: calculating C* = 1 - C
// - 1 indicates perfect positive autocorrelation/clustered
// - 0 indicates no autocorrelation/random
// - -1 indicates perfect negative
autocorrelation/dispersed
var gearysStar = ee.Image(1).subtract(gearys);
```

Examine the output by creating a custom palette and using a high-contrast basemap. The example illustrated in Fig. 34.3 shows how to add a focal max to the final image in order to better visualize the results as a raster. The color blue indicates values toward − 1 and therefore dispersion, while red represents values toward 1 and therefore spatially clustered.

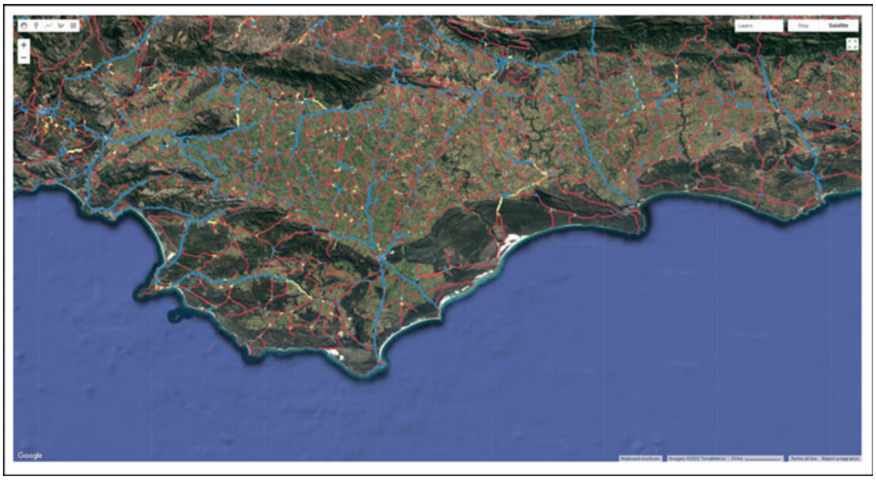


Fig. 34.3 Results of Geary's C* spatial autocorrelation between roads and transmission lines in South Africa. Red indicates clustering, while blue indicates dispersion. Shorter stretches in this example appear to have more co-located linear infrastructure

```
// Import palettes.
var palettes = require('users/gena/packages:palettes');

// Create custom palette, blue is negative while red is
positive autocorrelation/clustered.
var palette = palettes.colorbrewer.Spectral[7].reverse();

// Normalize the image and add it to the map.
var visParams = {
    min: -1,
    max: 1,
    palette: palette
};

// Display the image.
Map.setCenter(19.8638, -34.5705, 10);
Map.addLayer(gearysStar.focalMax(1), visParams, 'local
Gearys C*');
```

Question 10. What are the characteristics of the roads and transmission lines that are clustered?

Question 11. What are some other local indicators of spatial association?

Code Checkpoint A13e. The book's repository contains a script that shows what your code should look like at this point.

Save your script for your own future use, as outlined in Chap. 1. Then, refresh the Code Editor to begin with a new script for the next section.

34.2.3 Section 3. Impervious Surfaces and Flooding

For our final exercise, we will compare impervious surfaces and flooding over time in a new script. The Global Artificial Impervious Area dataset captures annual change information at a 30 m resolution of impervious surface area from 1985 to 2018 (Gong et al. 2020). Impervious surfaces can include a variety of built environment surfaces, including anything with pavement or water-resistant materials—roads, sidewalks, airports, parking lots, ports, distribution centers, rooftops, etc. Artificial impervious areas are important because they are a clear representation of human settlements (Gong et al. 2020).

We will compare these impervious areas with the Global Flood Database, which describes flood extent and population characteristics for 913 large flood events from 2000 to 2018 at 250 m resolution (Tellman et al. 2021).

In this example, let us compare the area of impervious surfaces in 2000 versus 2018 over the satellite-observed historical floodplain. Below, we will load the

Global Artificial Impervious Area dataset, to display the images and compare the data for the two dates.

```
// Import Tsinghua FROM-GLC Year of Change to Impervious
Surface
var impervious = ee.Image('Tsinghua/FROM-GLC/GAIA/v10');

// Use the change year values found in the band.
// The change year values is described here:
// https://developers.google.com/earth-
engine/datasets/catalog/Tsinghua_FROM-GLC_GAIA_v10#bands
// Select only those areas which were impervious by 2000.
var impervious2000 = impervious.gte(19);
```

```
// Select only those areas which were impervious by 2018.
var impervious2018 = impervious.gte(1);

Map.setCenter(-98.688, 39.134, 5);

// Display the images.
Map.addLayer(
    impervious2000.selfMask(),
    {
        min: 0,
        max: 1,
        palette: ['014352', '856F96']
    },
    'Impervious Surface 2000');

Map.addLayer(
    impervious2018.selfMask(),
    {
        min: 0,
        max: 1,
        palette: ['014352', '1A492C']
    },
    'Impervious Surface 2018');
```

Subtract the two images to find the change between 2018 and 2000, and again display the results:

```
// Calculate the difference between impervious areas in
2000 and 2018.
var imperviousDiff =
impervious2018.subtract(impervious2000);

Map.addLayer(
    imperviousDiff.selfMask(),
    {
        min: 0,
        max: 1,
        palette: ['014352', 'FFBF00']
    },
    'Impervious Surface Diff 2000-18');
```

Import the Global Flood Database. Select the `'flooded'` band, and sum all values to create the satellite-observed historical floodplain. Mask out the areas of permanent water in the floodplain using the JRC Global Surface Water dataset included in the Global Flood Database as the `'jrc_perm_water'` band. The JRC data use the dataset's original `'transition'` band. Display the final floodplain results:

```
// Import the Global Flood Database v1 (2000-2018).
var gfd =
ee.ImageCollection('GLOBAL_FLOOD_DB/MODIS_EVENTS/V1');

// Map all floods to generate the satellite-observed
historical flood plain.
var gfdFloodedSum = gfd.select('flooded').sum();

// Mask out areas of permanent water.
var gfdFloodedSumNoWater =
gfdFloodedSum.updateMask(gfd.select(
    'jrc_perm_water').sum().lt(1));

var durationPalette = ['C3EFFE', '1341E8', '051CB0',
'001133'];

Map.addLayer(
    gfdFloodedSumNoWater.selfMask(),
    {
        min: 0,
        max: 10,
        palette: durationPalette
    },
    'GFD Satellite Observed Flood Plain');
```

Now, we will calculate which states have been developing the greatest amounts of impervious surfaces in floodplain areas. Start by masking out areas in the `imperviousDiff` image that are not in the floodplains (`gfdFloodedSumNoWater` greater than or equal to 1). Then, we will import the first-order administrative Global Administrative Unit Layers from the UN Food and Agriculture Organization. Filter this feature collection to administrative names (`'ADM0_NAME'`) equal to `'United States of America'`. Create an area image by multiplying the impervious difference flood image by pixel area. Apply a `reduceRegions` reducer to the area image using the US feature collection and an initial scale of 100 m. Sort the output sum by descending order and get only the five highest states. Print the output.

```
// Mask areas in the impervious difference image that are
not in flood plains.
var imperviousDiffFloods = imperviousDiff
    .updateMask(gfdFloodedSumNoWater.gte(1));

// Which state has built the most area in the flood plains?
// Import FAO countries with first level administrative
units.
var countries =
ee.FeatureCollection('FAO/GAUL/2015/level1');

// Filter to the United States.
var unitedStates =
countries.filter(ee.Filter.eq('ADM0_NAME',
    'United States of America'));

// Calculate the image area.
var areaImage =
imperviousDiffFloods.multiply(ee.Image.pixelArea());

// Sum the area image for each state.
var unitedStatesImperviousDiffFlood =
areaImage.reduceRegions({
        collection: unitedStates,
        reducer: ee.Reducer.sum(),
        scale: 100,
    }) // Sort descending.
    .sort('sum', false)
    // Get only the 5 highest states.
    .limit(5);
```

```
// Print state statistics for change in impervious area in
flood plain.
print('Impervious-flood change statistics for states in
US',
    unitedStatesImperviousDiffFlood);
```

Question 12. Which state built the highest amount of impervious surfaces on satellite-derived floodplains between 2000 and 2018?

Question 13. Which state built the lowest amount of impervious surfaces on satellite-derived floodplains between 2000 and 2018?

Code Checkpoint A13f. The book's repository contains a script that shows what your code should look like at this point.

34.3 Synthesis

Assignment 1. Rerun the first exercise using the first-order administrative Global Administrative Unit Layers. Try selecting different countries to explore which ones have the highest road density. You will likely need to save the output in order to run this analysis.

Assignment 2. Compute local Geary's C for roads and transmission lines on other continents. Which continent appears to have the strongest clustering?

Assignment 3. Examine the Global Flood Database in relation to roads in an area of your choice. Try country-level analysis. Which country in your area has the highest proportion of roads potentially affected by flooding?

34.4 Conclusion

In this chapter, we have examined data from the built environment. We started by looking at characteristics of roads, then examined the interactions between roads and transmission lines, and finished by examining where built environments were most commonly developed in floodplains. Data in the built environment can be found in both vector and raster forms. We examined analysis using both forms, including rasterizing vector data. Our analysis also focused on larger-scale spatial analysis at the state, country, and continental levels. Understanding how to do this type of analysis allows us to develop a better understanding of these systems as more seamless data across temporal and spatial dimensions continue to become available.

References

Anselin L (1995) Local indicators of spatial association—LISA. Geogr Anal 27:93–115. https://doi.org/10.1111/j.1538-4632.1995.tb00338.x

Arderne C, Zorn C, Nicolas C, Koks EE (2020) Predictive mapping of the global power system using open data. Sci Data 7:1–12. https://doi.org/10.1038/s41597-019-0347-4

Gong P, Li X, Wang J et al (2020) Annual maps of global artificial impervious area (GAIA) between 1985 and 2018. Remote Sens Environ 236:111510. https://doi.org/10.1016/j.rse.2019.111510

Meijer JR, Huijbregts MAJ, Schotten KCGJ, Schipper AM (2018) Global patterns of current and future road infrastructure. Environ Res Lett 13:64006. https://doi.org/10.1088/1748-9326/aabd42

Tellman B, Sullivan JA, Kuhn C et al (2021) Satellite imaging reveals increased proportion of population exposed to floods. Nature 596:80–86. https://doi.org/10.1038/s41586-021-03695-w

35 Air Pollution and Population Exposure

Zander S Venter and Sourangsu Chowdhury

Overview

After high blood pressure and smoking, air pollution is the third-largest risk factor for death globally (Murray et al. 2020). Air pollution can therefore be described as a global "pandemic" that should arguably be monitored and addressed with the same intensity with which the COVID-19 pandemic has been. Remote sensing and cloud computing technologies allow us to do so.

The purpose of this chapter is to explore and analyze gridded air pollution data from Sentinel-5P in the context of changes brought about by COVID-19 lockdowns. Practical components will include analyzing changes in nitrogen dioxide (NO_2) concentrations over time and quantifying population-weighted NO_2 concentrations for selected administrative units.

Learning Outcomes

- Understanding Sentinel-5P data.
- Quantifying changes in air pollutant concentrations over time.
- Generating a split-panel map to compare two time epochs.
- Calculating population-weighted air pollutant concentrations.

Z. S. Venter (✉)
Norwegian Institute for Nature Research, Trondheim, Norway
e-mail: zander.venter@nina.no

S. Chowdhury
Center for International Climate and Environmental Research (CICERO), University of Oslo, 0349 Oslo, Norway
e-mail: sourangsu.chowdhury@cicero.oslo.no

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_35

Helps if you know how to

- Import images and image collections, filter, and visualize (Part I).
- Create a graph using `ui.Chart` (Chap. 4).
- Perform basic image analysis: select bands, compute indices, create masks (Part II).
- Use `ee.Reducer` functions to summarize pixels over an area (Chaps. 8 and 9).
- Write a function and `map` it over an `ImageCollection` (Chap. 12).
- Mask cloud, cloud shadow, snow/ice, and other undesired pixels (Chap. 15).
- Design user interfaces for an Earth Engine App (Chap. 30).

35.1 Introduction to Theory

Air pollution can be generally defined as any chemical, physical, or biological agent that alters the natural composition of the atmosphere. Pollutants that are of primary concern for public health include particulate matter with diameter less than 2.5 μm ($PM_{2.5}$), carbon monoxide (CO), ozone (O_3), NO_2, and sulfur dioxide (SO_2). Globally, chronic exposure to air pollution results in greater loss of life than HIV/AIDS, malaria, and tuberculosis combined, and more than an order of magnitude more deaths than all forms of violence (Lelieveld et al. 2020). Exposure to $PM_{2.5}$ and O_3 is estimated to result in ~ 4.7 million excess deaths annually across the globe (Murray et al. 2020), although these estimates range between 3 and 10 million excess deaths per year, based on the disease categories considered and the exposure–response function used (Burnett et al. 2018; Chowdhury et al. 2022). Exposure to NO_2 may result in 4 million new pediatric asthma cases annually (Achakulwisut et al. 2019).

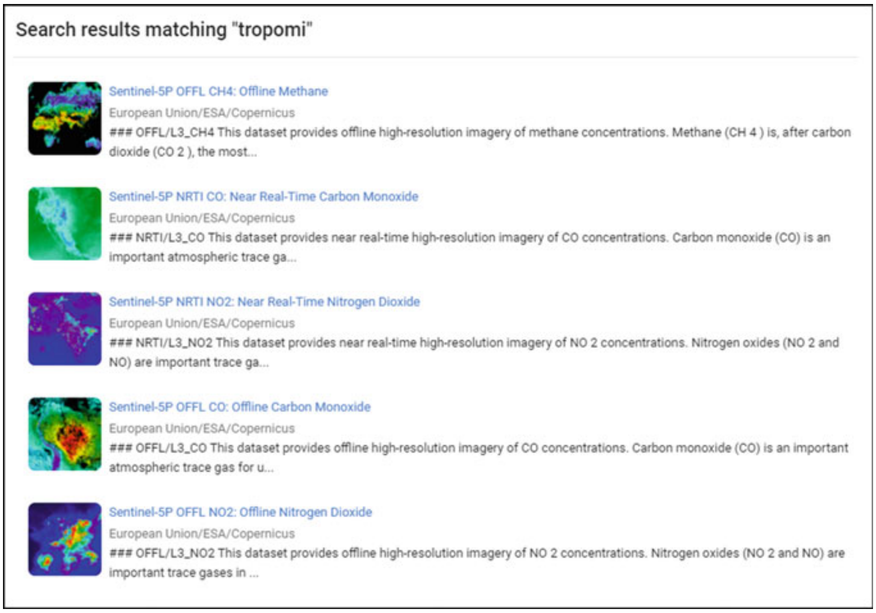
Knowledge about the global distribution of these air pollutants and their sources has improved over the last decade, with the expansion of networks of ground-based monitors in many countries, the evolution of satellite products, and the advancement of complex atmospheric chemistry models. Studies have found that more than 70% of the global health burden from air pollution is attributable to anthropogenic emissions (Chowdhury et al. 2022; Lelieveld et al. 2019). The main anthropogenic sources of air pollution are industries, motor vehicles, power generation, agricultural activities, and household combustion, while non-anthropogenic sources include desert dust, biogenic emissions, forest fires, and even volcanoes. The reduction in transport and industrial activity during the COVID-19 lockdowns significantly reduced global air pollution levels, thereby highlighting the significance of anthropogenic emissions (Venter et al. 2020). In fact, it is estimated that the decline in air pollution during the first five months of 2020 resulted in 49,900 avoided deaths and 89,000 fewer pediatric asthma emergency room visits (Venter et al. 2021).

Despite the recent growth in monitoring networks, the air in most regions of Earth is insufficiently monitored, limiting air quality management. Given the paucity of ground-based monitoring, alternative monitoring approaches such as satellite remote sensing are gaining popularity and becoming more accurate (e.g., Griffin et al. 2019). Over the past few decades, we have had increasing access to a range of satellite sensors that monitor the contents of Earth's atmosphere. However, it is important to note that satellites measure pollutant concentrations in the troposphere and stratosphere, which extend for many kilometers above the Earth's surface. As a result, satellite measurements are not necessarily representative of the concentrations humans are exposed to on the ground, and consequently, relying on satellite data alone for human health applications is not advised. However, more sophisticated methods combine information from satellite remote sensing data, complex atmospheric chemistry models, and ground-based monitors to provide ground-level concentrations of pollutants with high confidence (Dey et al. 2020, Donkelar et al. 2021).

35.2 Practicum

35.2.1 Section 1: Data Importing and Cleaning

There is a range of satellite-based datasets on air pollution to choose from in the Earth Engine Data Catalog. The main datasets relevant to air pollution include the Moderate Resolution Imaging Spectroradiometer and Advanced Very-High-Resolution Radiometer for monitoring aerosol optical depth (a proxy for $PM_{2.5}$); the Total Ozone Mapping Spectrometer Ozone Monitoring Instrument for monitoring O_3; and more recently the TROPOspheric Monitoring Instrument (TROPOMI) on board the Sentinel-5 Precursor (Sentinel-5P), which monitors a range of air pollutants. We will use Sentinel-5P in this practicum, but the methods covered here are easily transferable to the datasets mentioned above.


Search results matching "tropomi"

Sentinel-5P OFFL CH4: Offline Methane
European Union/ESA/Copernicus
OFFL/L3_CH4 This dataset provides offline high-resolution imagery of methane concentrations. Methane (CH 4) is, after carbon dioxide (CO 2), the most...

Sentinel-5P NRTI CO: Near Real-Time Carbon Monoxide
European Union/ESA/Copernicus
NRTI/L3_CO This dataset provides near real-time high-resolution imagery of CO concentrations. Carbon monoxide (CO) is an important atmospheric trace ga...

Sentinel-5P NRTI NO2: Near Real-Time Nitrogen Dioxide
European Union/ESA/Copernicus
NRTI/L3_NO2 This dataset provides near real-time high-resolution imagery of NO 2 concentrations. Nitrogen oxides (NO 2 and NO) are important trace ga...

Sentinel-5P OFFL CO: Offline Carbon Monoxide
European Union/ESA/Copernicus
OFFL/L3_CO This dataset provides offline high-resolution imagery of CO concentrations. Carbon monoxide (CO) is an important atmospheric trace gas for u...

Sentinel-5P OFFL NO2: Offline Nitrogen Dioxide
European Union/ESA/Copernicus
OFFL/L3_NO2 This dataset provides offline high-resolution imagery of NO 2 concentrations. Nitrogen oxides (NO 2 and NO) are important trace gases in ...

Fig. 35.1 Earth engine data catalog results for the search term "tropomi"

Now, let's load the satellite data for this practicum. If you search "tropomi" in the Earth Engine Data Catalog, you will see a range of datasets from Sentinel-5P, which can all be of value in quantifying air quality (Fig. 35.1).

Although Sentinel-5 was launched in October 2017, the data available for analysis in Earth Engine are from July 2018 onward. TROPOMI, the sensor on board the satellite, is a spectrometer sensing ultraviolet, visible, near-infrared, and shortwave infrared wavelengths to monitor NO_2, O_3, aerosol, methane (CH_4), formaldehyde, CO, and SO_2 in the atmosphere. The swath width of TROPOMI is approximately 2600 km on the ground, resulting in a global daily coverage with a spatial resolution of 7×7 km. All of the Sentinel-5P datasets, except CH_4, have two versions: Near Real-Time (NRTI) and Offline (OFFL); CH_4 is available as OFFL only. The NRTI assets cover a smaller area than the OFFL assets but appear more quickly after acquisition. The OFFL assets have a delayed availability, but each asset contains data from an entire orbit and is arguably easier to work with for retrospective analyses. We will use the OFFL NO_2 product in this practicum.

First we need to define an area of interest. Wuhan is infamous for being the epicenter of the COVID-19 pandemic and witnessed severe lockdowns. In the next section of this practicum, we will test to see if we can detect a reduction in NO_2 during the early 2020 lockdowns in the surrounding province, Hubei. To start, in the code below, we import a global dataset of administrative boundaries and filter them for intersection with an `ee.Geometry.Point` object, which appears under the **Imports** section at the top of your script. This `geometry` has to be drawn with the drawing tool and can be moved to a new location to rerun the analysis for that administrative boundary.

After centering the **Map** on Hubei Province, we will import a population dataset, which is necessary for calculating population-weighted exposures in Sect. 3 of this practicum. We will use the Gridded Population of the World dataset for 2020, which includes a total population count per ~ 1 × 1 km grid (Fig. 35.2).

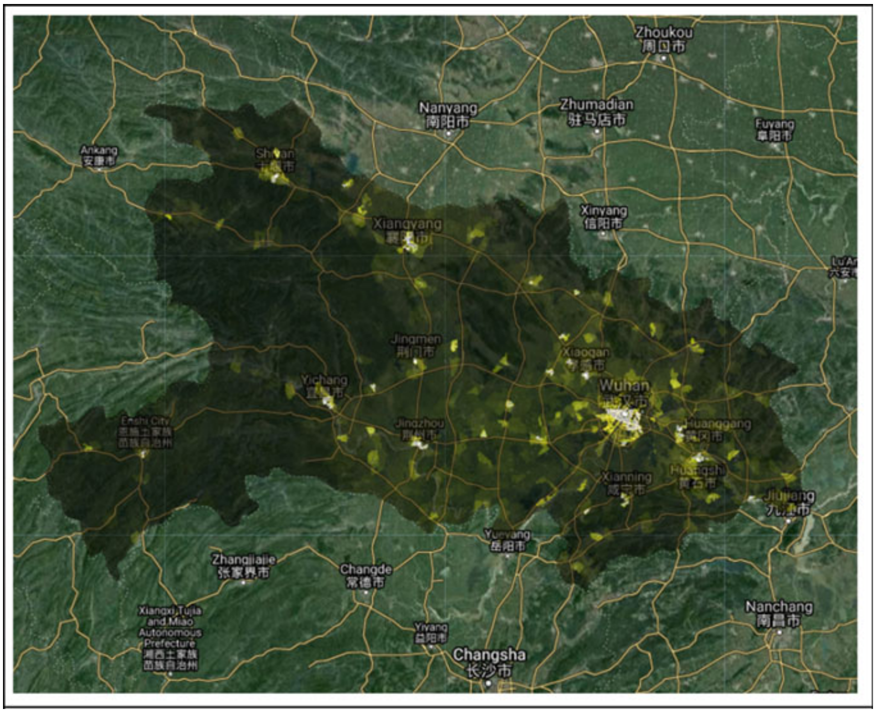


Fig. 35.2 Population density over Hubei Province. Brighter areas have higher population counts

```
// Import a global dataset of administrative units level 1.
var adminUnits = ee.FeatureCollection(
    'FAO/GAUL_SIMPLIFIED_500m/2015/level1');

// Filter for the administrative unit that intersects
// the geometry located at the top of this script.
var adminSelect = adminUnits.filterBounds(geometry);

// Center the map on this area.
Map.centerObject(adminSelect, 8);

// Make the base map HYBRID.
Map.setOptions('HYBRID');

// Add it to the map to make sure you have what you want.
Map.addLayer(adminSelect, {}, 'selected admin unit');

// Import the population count data from Gridded Population of
the World Version 4.
var population = ee.ImageCollection(
        'CIESIN/GPWv411/GPW_Population_Count')
    // Filter for 2020 using the calendar range function.
    .filter(ee.Filter.calendarRange(2020, 2020, 'year'))
    // There should be only 1 image, but convert to an image
using .mean().
    .mean();

// Clip it to your area of interest (only necessary for
visualization purposes).
var populationClipped =
population.clipToCollection(adminSelect);

// Add it to the map to see the population distribution.
var popVis = {
    min: 0,
    max: 4000,
    palette: ['black', 'yellow', 'white'],
    opacity: 0.55
};
Map.addLayer(populationClipped, popVis, 'population count');
```

Question 1. There are two other datasets of gridded population in the Earth Engine Data Catalog, namely WorldPop and Global Human Settlement Layers. Use the search bar to find them and add them to the map to compare them with the Gridded Population of the World dataset. Which one looks more realistic in your opinion, and why?

Now it is time to import the NO_2 data. As with most optical satellite data, there can be things in the atmosphere that contaminate the signal from the object or chemical you want to measure. Clouds are a common issue for land surface reflectance products (Chap. 15), and they are also an issue when trying to measure air pollutant concentrations. In the code below, we create a function to mask out pixels with a cloud fraction above 0.3 (i.e., 30% cloud cover). You can test different masking thresholds to see what suits your use case best. After masking out cloudy pixels, we create a median composite from images during March 2021. It is important to note that we are working with the band that gives measurements for the tropospheric vertical column of NO_2 and not the stratospheric or total vertical column. The troposphere is the closest we can get to ground-level measurements with Sentinel-5P. The median image for March 2021 should look like the map shown in Fig. 35.3.

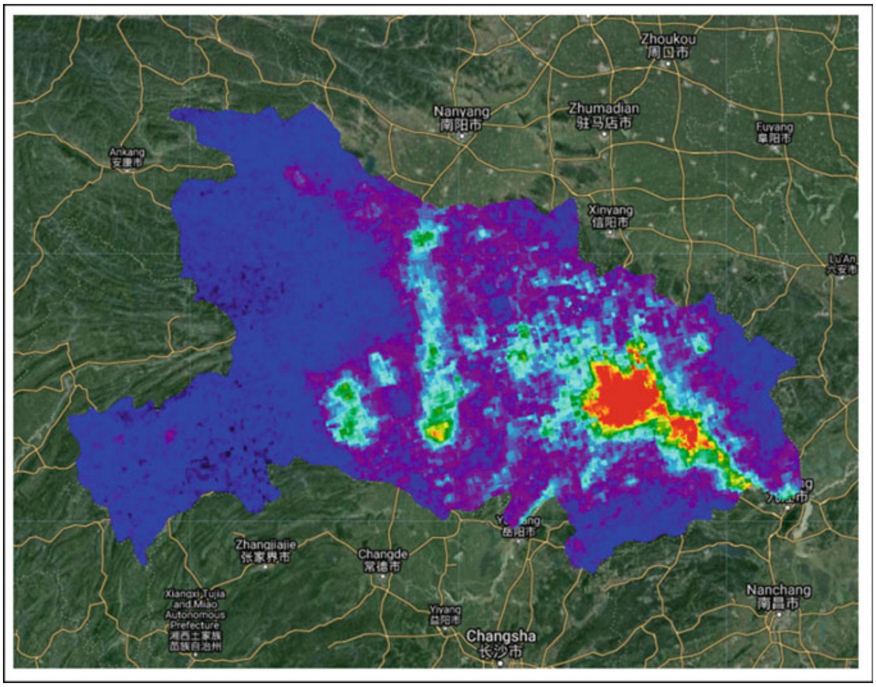


Fig. 35.3 Tropospheric NO_2 concentrations over Hubei Province. Hotter colors have higher concentrations, while cooler colors have lower concentrations

```
// Import the Sentinel-5P NO2 offline product.
var no2Raw = ee.ImageCollection('COPERNICUS/S5P/OFFL/L3_NO2');

// Define function to exclude cloudy pixels.
function maskClouds(image) {
    // Get the cloud fraction band of the image.
    var cf = image.select('cloud_fraction');
    // Create a mask using 0.3 threshold.
    var mask = cf.lte(0.3); // You can play around with this
value.
    // Return a masked image.
    return image.updateMask(mask).copyProperties(image);
}

// Clean and filter the Sentinel-5P NO2 offline product.
var no2 = no2Raw
    // Filter for images intersecting our area of interest.
    .filterBounds(adminSelect)
    // Map the cloud masking function over the image collection.
    .map(maskClouds)
    // Select the tropospheric vertical column of NO2 band.
    .select('tropospheric_NO2_column_number_density');

// Create a median composite for March 2021
var no2Median = no2.filterDate('2021-03-01', '2021-04-
01').median();

// Clip it to your area of interest (only necessary for
visualization purposes).
var no2MedianClipped = no2Median.clipToCollection(adminSelect);

// Visualize the median NO2.
var no2Viz = {
    min: 0,
    max: 0.00015,
    palette: ['black', 'blue', 'purple', 'cyan', 'green',
        'yellow', 'red'
    ]
};
Map.addLayer(no2MedianClipped, no2Viz, 'median no2 Mar 2021');
```

Code Checkpoint A14a. The book’s repository contains a script that shows what your code should look like at this point.

35.2.2 Section 2: Quantifying and Visualizing Changes

Next we will test to see if we can visualize a change in NO_2 concentrations during the 2020 COVID-19 lockdowns. We will compare the median NO_2 concentration during March 2020 (during which Hubei Province was in lockdown) with the median value during March 2019.

Weather can significantly affect air pollutant concentrations (e.g., wind causing long-range transport of smoke), and therefore differences between 2020 and 2019 could be an artifact of differences in weather. By comparing the same month in different years, we partly control for the effects of seasonal weather patterns, but not completely. If you would like to control for weather effects more thoroughly, see Venter et al. (2020) for details. In the code below, we calculate and visualize median composite images for March 2019 and March 2020. The visualization makes use of Earth Engine's comprehensive library of user-interface widgets (see Chap. 30 for more details). Specifically, we use the `ui.SplitPanel` widget to compare the two median composites side by side (Fig. 35.4). This widget can be set to have a wiping effect where maps are overlaid on top of one another, or a side-by-side comparison.

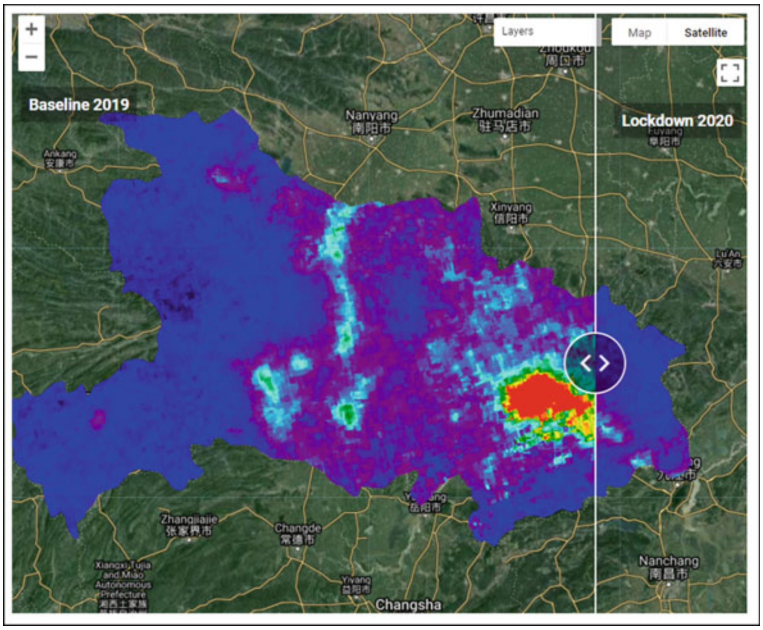


Fig. 35.4 Split-panel map showing tropospheric NO_2 concentrations over Hubei Province for March 2019 (left) and March 2020 (right). Hotter colors have higher concentrations, while cooler colors have lower concentrations

```
// Define a lockdown NO2 median composite.
var no2Lockdown = no2.filterDate('2020-03-01', '2020-04-
01')
    .median().clipToCollection(adminSelect);

// Define a baseline NO2 median using the same month in the
previous year.
var no2Baseline = no2.filterDate('2019-03-01', '2019-04-
01')
    .median().clipToCollection(adminSelect);

// Create a ui map widget to hold the baseline NO2 image.
var leftMap = ui.Map().centerObject(adminSelect,
8).setOptions(
    'HYBRID');

// Create ta ui map widget to hold the lockdown NO2 image.
var rightMap = ui.Map().setOptions('HYBRID');

// Create a split panel widget to hold the two maps.
var sliderPanel = ui.SplitPanel({
    firstPanel: leftMap,
    secondPanel: rightMap,

    orientation: 'horizontal',
    wipe: true,
    style: {
        stretch: 'both'
    }
});
var linker = ui.Map.Linker([leftMap, rightMap]);

// Make a function to add a label with fancy styling.
function makeMapLab(lab, position) {
    var label = ui.Label({
        value: lab,
        style: {
            fontSize: '16px',
            color: '#ffffff',
            fontWeight: 'bold',
            backgroundColor: '#ffffff00',
            padding: '0px'
        }
    });
    var panel = ui.Panel({
        widgets: [label],
        layout: ui.Panel.Layout.flow('horizontal'),
        style: {
            position: position,
            backgroundColor: '#00000057',
            padding: '0px'
        }
    });
    return panel;
}

// Create baseline map layer, add it to the left map, and
add the label.
var no2BaselineLayer = ui.Map.Layer(no2Baseline, no2Viz);
leftMap.layers().reset([no2BaselineLayer]);
leftMap.add(makeMapLab('Baseline 2019', 'top-left'));


// Create lockdown map layer, add it to the right map, and
add the label.
var no2LockdownLayer = ui.Map.Layer(no2Lockdown, no2Viz);
rightMap.layers().reset([no2LockdownLayer]);
rightMap.add(makeMapLab('Lockdown 2020', 'top-right'));

// Reset the map interface (ui.root) with the split panel
widget.
// Note that the Map.addLayer() calls earlier on in Section
1
// will no longer be shown because we have replaced the Map
widget
// with the sliderPanel widget.
ui.root.widgets().reset([sliderPanel]);
```

Question 2. Comparing the two maps in the split-panel map, do you find a reduction in NO_2 concentrations during the lockdown? Where is the change in NO_2 concentrations most significant?

Question 3. How are changes in NO_2 concentrations related to population density? To help answer this question, you can (1) create a difference image by subtracting the `no2Lockdown` image from the `no2Baseline` image, (2) create a new `ui.Map.Layer` for the difference image and the population image created in Sect. 35.1, and (3) add these to the left or right map. Hint: You can change the opacity of the NO_2 layers to aid interpretability.

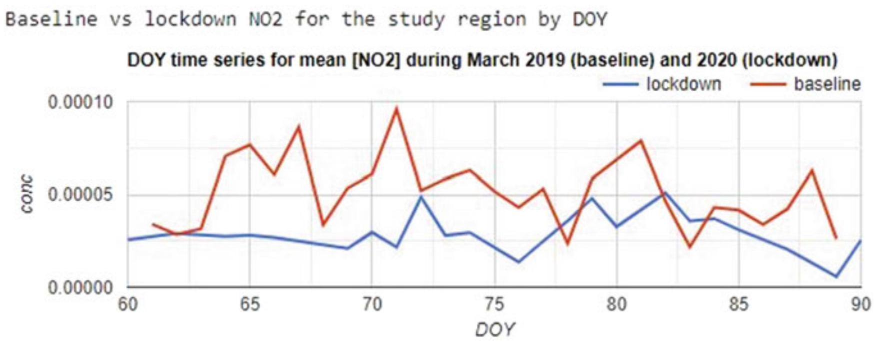


Fig. 35.5 Time-series graph showing average NO_2 concentrations for Hubei Province during March 2019 and March 2020

Exploring the differences in NO_2 concentrations as `ee.Image` objects can be visually informative, but quantifying the changes for specific regions requires further work. In the code below, we calculate the mean NO_2 concentrations for Hubei Province by applying a `reduceRegion` function to each image in the March 2019 and March 2020 collections. The resulting time series are visualized in the chart shown in Fig. 35.5.

```
// Create a function to get the mean NO2 for the study
region
// per image in the NO2 collection.
function getConc(collectionLabel, img) {
    return function(img) {
        // Calculate the mean NO2.
        var no2Mean = img.reduceRegion({
            reducer: ee.Reducer.mean(),
            geometry: adminSelect.geometry(),
            scale: 7000
        }).get('tropospheric_NO2_column_number_density');

        // Get the day-of-year of the image.
        var doy = img.date().getRelative('day', 'year');
        // Return a feature with NO2 concentration and day-
of-year properties.
        return ee.Feature(null, {
            'conc': no2Mean,
            'DOY': doy,
            'type': collectionLabel
        });
    };
}

// Get the concentrations for a baseline and lockdown
collection
// and merge for plotting.
var no2AggChange_forPlotting = no2
    .filterDate('2020-03-01', '2020-04-01')
    .map(getConc('lockdown'))
    .merge(no2.filterDate('2019-03-01', '2019-04-01')
        .map(getConc('baseline')));
no2AggChange_forPlotting = no2AggChange_forPlotting
    .filter(ee.Filter.notNull(['conc']));

// Make a chart.
var chart1 = ui.Chart.feature.groups(
        no2AggChange_forPlotting, 'DOY', 'conc', 'type')
    .setChartType('LineChart')
    .setOptions({
        title: 'DOY time series for mean [NO2] during ' +
            'March 2019 (baseline) and 2020 (lockdown)'
    });

// Print it to the console.
print('Baseline vs lockdown NO2 for the study region by
DOY', chart1);
```

Code Checkpoint A14b. The book's repository contains a script that shows what your code should look like at this point.

35.2.3 Section 3: Calculating Population-Weighted Concentrations

In Sect. 35.2, we used the `ee.Reducer.mean` reducer in the `reduceRegion` function to get the average NO_2 concentration over Hubei Province. However, when aggregating pollutant concentrations to define population exposure, we need a different approach. Imagine there was a large concentration of NO_2 in a rural area in the east of Hubei Province where very few people live. If we simply calculated the average of all pixels, this rural NO_2 anomaly would skew our representation of population exposure. Using the population number dataset imported in Sect. 35.1, we can calculate the population-weighted exposure (Exp) aggregated across n pixels in the area of interest (in this case, Hubei Province) using Eq. A1.4.1 below, where C_i is the NO_2 concentration and P_i is the subpopulation in pixel i.

$$\text{Exp} = \sum_{i}^{n} \frac{P_i}{\sum_i^n (P)} \cdot C_i \tag{35.1}$$

In the code below, we map a function to calculate population-weighted exposure over all the images in the NO_2 `ImageCollection`. Remember that in Sect. 35.1 we masked out pixels from images that had a cloud cover value greater than 30%. Therefore, an important step in this function is to calculate the percentage of available Sentinel-5P pixels within Hubei Province per image. We need to decide what percentage pixel coverage is enough to calculate a representative average for the province. Here we choose 25% for illustrative purposes, but depending on your research question, you may want to calculate averages only when you have 100% coverage by/from Sentinel-5P that is free of clouds. The contrast between the simple average and population-weighted average is shown in Fig. 35.6. The difference may appear small in this case, but when aggregating over larger areas with greater variation in population density, population-weighted averages can be very different from simple averages.

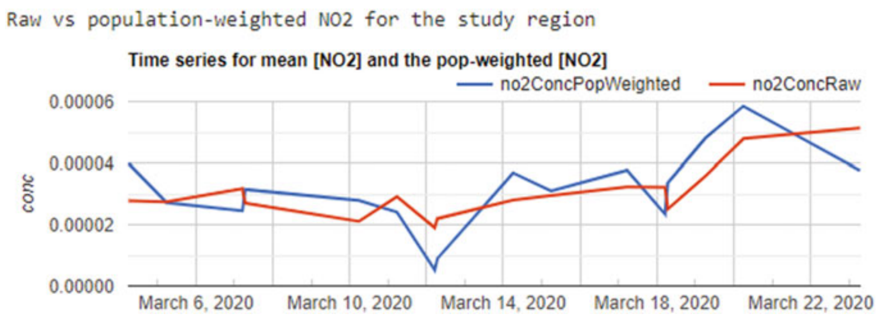


Fig. 35.6 Time-series graph showing average (no2ConcRaw) and population-weighted average (no2ConcPopWeighted) NO_2 concentrations for Hubei Province in March 2020

```
// Define the spatial resolution of the population data.
var scalePop = 927.67; // See details in GEE Catalogue.

// Now we define a function that will map over the NO2
collection
// and calculate population-weighted concentrations.
// We will use the formula Exp = SUM {(Pi/P)*Ci}.
// We can calculate P outside of the function
// so that it is not computed multiple times for each NO2 image.
var P = population.reduceRegion({
    reducer: ee.Reducer.sum(),
    geometry: adminSelect.geometry(),
    scale: scalePop
}).get('population_count');

// And here is the function.
function getPopWeightedConc(P, region, regionName, img) {
    return function(img) {
        var Ci = img;
        var Pi = population;
        // Calculate the percentage of valid pixels in the
region.
        // (masked pixels will not be counted).
        var pixelCoverPerc = Ci.gte(0).unmask(0).multiply(100)
            .reduceRegion({
                reducer: ee.Reducer.mean(),
                geometry: region.geometry(),
                scale: scalePop // Add in the scale of the
population raster.
            }).get('tropospheric_NO2_column_number_density');

        // Calculate the per-pixel EXP (see formula above).
        var exp =
Pi.divide(ee.Image(ee.Number(P))).multiply(Ci);

        // Sum the exp over the region.
        var expSum = exp.reduceRegion({
            reducer: ee.Reducer.sum(),
            geometry: region.geometry(),
            scale: scalePop
        }).get('population_count');
        // Calculate the mean NO2 - the approach that would
usually
        // be taken without population weighting.
        var no2Mean = Ci.reduceRegion({
            reducer: ee.Reducer.mean(),
            geometry: region.geometry(),
            scale: scalePop
        }).get('tropospheric_NO2_column_number_density');

        // Return a feature with properties
        var featOut = ee.Feature(null, {
            'system:time_start': img.get(
                'system:time_start'),
            'dateString': img.date().format('YYYY-MM-DD'),
            'regionName': regionName,
            'no2ConcPopWeighted': expSum,
            'no2ConcRaw': no2Mean,
            'pixelCoverPerc': pixelCoverPerc
        });

        return featOut;
    };

}

// Filter the NO2 collection for March 2020 and map the function
over it.
var no2Agg_popWeighted = no2.filterDate('2020-03-01', '2020-04-
01')
    .map(getPopWeightedConc(P, adminSelect, 'Wuhan'));
no2Agg_popWeighted = ee.FeatureCollection(no2Agg_popWeighted);

// Define the percentage of valid pixels you want in your region
per time point.
// Here we choose 25; i.e. only images with at least 25% valid
NO2 pixels.
var validPixelPerc = 25; // you can play around with this value

// Filter the feature collection based on your pixel criteria.
no2Agg_popWeighted = no2Agg_popWeighted
    .filter(ee.Filter.greaterThanOrEquals('pixelCoverPerc',
        validPixelPerc));
print('Population weighted no2 feature collection:',
 no2Agg_popWeighted);
```

```
// Create a feature collection for plotting the mean [NO2]
// and the mean pop-weighted [NO2] on the same graph.
var no2Agg_forPlotting = no2Agg_popWeighted.map(function(ft) {
    return ft.set('conc', ft.get('no2ConcPopWeighted'),
        'type', 'no2ConcPopWeighted');
}).merge(no2Agg_popWeighted.map(function(ft) {
    return ft.set('conc', ft.get('no2ConcRaw'), 'type',
        'no2ConcRaw');
}));

// Make a chart
var chart2 = ui.Chart.feature.groups(
        no2Agg_forPlotting, 'system:time_start', 'conc', 'type')
    .setChartType('LineChart')
    .setOptions({
        title: 'Time series for mean [NO2] and the pop-weighted
[NO2]'
    });

// Print it to the console
print('Raw vs population-weighted NO2 for the study region',
chart2);
```

Finally, although we can plot this data in Earth Engine, it is often easier to process with other statistical software, such as R or Python. So, to conclude, let us code for exporting time series of population-weighted averages for more than one area of interest (in this case, administrative units). In the code below, we map the function over two regions and then export the resulting table as a CSV file to Google Drive.

```
// Export population-weighted data for multiple regions.
// First select the regions. This can also be done with
// .filterBounds() as in Line 9 above.
var regions = adminUnits
    .filter(ee.Filter.inList('ADM1_NAME', ['Chongqing Shi',
        'Hubei Sheng'
    ]));
```

```
// Map a function over the regions that calculates population-
weighted [NO2].
var No2AggMulti_popWeighted = regions.map(function(region) {
    var P = population.reduceRegion({
        reducer: ee.Reducer.sum(),
        geometry: region.geometry(),
        scale: scalePop
    }).get('population_count');
    var innerTable = no2.filterDate('2020-03-01',
            '2020-04-01')
        .map(getPopWeightedConc(P, region, region.get(
            'ADM1_NAME')));
    return innerTable;
}).flatten();
// Remember to filter out readings that have pixel percentage
cover
// below your threshold
No2AggMulti_popWeighted = No2AggMulti_popWeighted
    .filter(ee.Filter.greaterThanOrEquals('pixelCoverPerc',
        validPixelPerc));

// Run the export under the 'Tasks' tab on the right
// and find your CSV file in Google Drive later on.
Export.table.toDrive({
    collection: No2AggMulti_popWeighted,
    description: 'no2_popWeighted',
    fileFormat: 'CSV'
});
```

Code Checkpoint A14c. The book's repository contains a script that shows what your code should look like at this point.

35.3 Synthesis

In this practicum, we focused on a particular pollutant (NO_2), region (Hubei), and time period (March 2019 and March 2020). To reinforce your comprehension and understanding, consider the following assignments.

Assignment 1. How would you run this analysis for a different pollutant? Try substituting the NO_2 collection with the Sentinel-5P NRTI SO_2 collection. Hint: The main emission source for SO_2 is electricity generation, for which coal is the most significant fuel. Use this information to inform your selection of a location and time period so that you can detect interesting changes.

Assignment 2. How would you run this analysis for a different geographic area? Try deleting the `ee.Geometry.Point` at the top of your script and using the **Geometry Tools** to digitize your own point on which to focus the analysis. If you are running the latter part of the script, you can also change the list of named administrative units. Hint: Add the `adminUnits` object from Sect. 35.1 of the code to the map. You can use the **Inspector** tab to click on polygons and get the name of the administrative unit under the 'ADM1_NAME' property.

Assignment 3. Finally, try changing the dates in the script so that you are comparing two different time periods. Remember that the Sentinel-5P data are available from July 2018 onward; defining dates before this will cause the script to throw an error.

35.4 Conclusion

In this chapter, we covered the basics of importing Sentinel-5P air pollution data, comparing changes over time, and calculating population-weighted averages for spatial units. Satellite detection of air pollutants is an important tool for monitoring air quality from local to global scales, but ground-station measurements and atmospheric modeling are often necessary to draw conclusions about human health risk. The fusion of ground-level and satellite data with advanced machine learning models to map and forecast air pollution is a growing research field with important societal applications (e.g., https://www.iqair.com/). Earth Engine is a well-suited and currently underutilized resource to advance this field.

References

Achakulwisut P, Brauer M, Hystad P, Anenberg SC (2019) Global, national, and urban burdens of paediatric asthma incidence attributable to ambient NO_2 pollution: estimates from global datasets. Lancet Planet Heal 3:e166–e178. https://doi.org/10.1016/S2542-5196(19)30046-4

Benedetti A, Morcrette J-J, Boucher O, et al (2009) Aerosol analysis and forecast in the European centre for medium-range weather forecasts integrated forecast system: 2. Data assimilation. J Geophys Res Atmos 114https://doi.org/10.1029/2008JD011235

Burnett R, Chen H, Szyszkowicz M et al (2018) Global estimates of mortality associated with long-term exposure to outdoor fine particulate matter. Proc Natl Acad Sci USA 115:9592–9597. https://doi.org/10.1073/pnas.1803222115

Chowdhury S, Pozzer A, Haines A et al (2022) Global health burden of ambient $PM_{2.5}$ and the contribution of anthropogenic black carbon and organic aerosols. Environ Int 159:107020. https://doi.org/10.1016/j.envint.2021.107020

Dey S, Purohit B, Balyan P et al (2020) A satellite-based high-resolution (1-km) ambient $PM_{2.5}$ database for India over two decades (2000–2019): applications for air quality management. Remote Sens 12:1–22. https://doi.org/10.3390/rs12233872

Griffin D, Zhao X, McLinden CA et al (2019) High-resolution mapping of nitrogen dioxide with TROPOMI: first results and validation over the Canadian oil sands. Geophys Res Lett 46:1049–1060. https://doi.org/10.1029/2018GL081095

Lelieveld J, Klingmüller K, Pozzer A et al (2019) Effects of fossil fuel and total anthropogenic emission removal on public health and climate. Proc Natl Acad Sci USA 116:7192–7197. https://doi.org/10.1073/pnas.1819989116

Lelieveld J, Pozzer A, Pöschl U et al (2020) Loss of life expectancy from air pollution compared to other risk factors: a worldwide perspective. Cardiovasc Res 116:1910–1917. https://doi.org/10.1093/cvr/cvaa025

Murray CJL, Aravkin AY, Zheng P et al (2020) Global burden of 87 risk factors in 204 countries and territories, 1990–2019: a systematic analysis for the global burden of disease study 2019. Lancet 396:1223–1249.https://doi.org/10.1016/S0140-6736(20)30752-2

Van Donkelaar A, Hammer MS, Bindle L et al (2021) Monthly global estimates of fine particulate matter and their uncertainty. Environ Sci Technol 55:15287–15300. https://doi.org/10.1021/acs.est.1c05309

Venter ZS, Aunan K, Chowdhury S, Lelieveld J (2020) COVID-19 lockdowns cause global air pollution declines. Proc Natl Acad Sci USA 117:18984–18990. https://doi.org/10.1073/pnas.2006853117

Venter ZS, Aunan K, Chowdhury S, Lelieveld J (2021) Air pollution declines during COVID-19 lockdowns mitigate the global health burden. Environ Res 192:110403. https://doi.org/10.1016/j.envres.2020.110403

Heat Islands

36

TC Chakraborty

Overview

In this chapter, you will learn about urban heat islands and how they can be calculated from satellite measurements of thermal radiation from the Earth's surface.

Learning Outcomes

- Understanding how to derive land surface temperature.
- Understanding how to generate urban and rural references.
- Knowing how to calculate the surface urban heat island intensity.

Helps if you know how to

- Import, filter, and visualize images (Part I).
- Perform basic image analysis: select bands, compute indices, and create masks (Part II).
- Use expressions to perform calculations on image bands (Chap. 9).
- Write a function and map it over an `ImageCollection` (Chap.12).
- Mask cloud, cloud shadow, snow/ice, and other undesired pixels (Chap. 15).
- Conduct basic vector analyses: vectorizing and buffering (Part 5).
- Write a function and map it over a `FeatureCollection` (Chap. 23 and 24).

T C Chakraborty (✉)
Pacific Northwest National Laboratory, Yale School of the Environment, Yale University, New Haven, CT, USA
e-mail: tc.chakraborty@pnnl.gov

Pacific Northwest National Laboratory, Richland, WA, USA

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_36

36.1 Introduction to Theory

Urbanization involves the replacement of natural landscapes with built-up structures such as buildings, roads, and parking lots. This land cover modification also changes the properties of the land surface. These changes can range from how much radiation is reflected and absorbed by the surface to how the heat is dissipated from the surface (e.g., removal of vegetation for urban development reduces evaporative cooling). These changes in surface properties can modify local weather and climate (Kalnay and Cai 2003). The most-studied local climate modification due to urbanization is the urban heat island (UHI) effect (Arnfield 2003; Qian et al. 2022). The UHI is the phenomenon in which a city is warmer than either its surroundings or an equivalent surface that is not urbanized. We have known about the UHI effect for almost 200 years (Howard 1833).

Traditionally, the UHI was defined as the difference in air temperature, measured by weather stations, between a city and some rural reference outside the city (Oke 1982). One issue with this method is that different parts of the city can have different air temperatures, making it difficult to capture the UHI for the entire city. Using satellite observations in the thermal bands allows us to get another measure of temperature: the radiometric skin temperature, often known as the land surface temperature (LST). We can use LST to calculate a surface UHI (SUHI) intensity, including how it varies within cities at the pixel scale (Ngie et al. 2014). It is important to stress here that the UHI values observed by satellites and those calculated using air temperature measurements can be very different (Chakraborty et al. 2017; Hu et al. 2019; Venter et al. 2021).

36.2 Practicum

36.2.1 Deriving Land Surface Temperature

36.2.2 Deriving Land Surface Temperature from MODIS

Land surface temperature can either be extracted from derived products, such as the MODIS Terra and Aqua satellite products (Wan 2006), or estimated directly from satellite measurements in the thermal band. We will explore both options using the city of New Haven, Connecticut, USA, as the region of interest (Fig. 36.1). We will start with the MODIS LST.

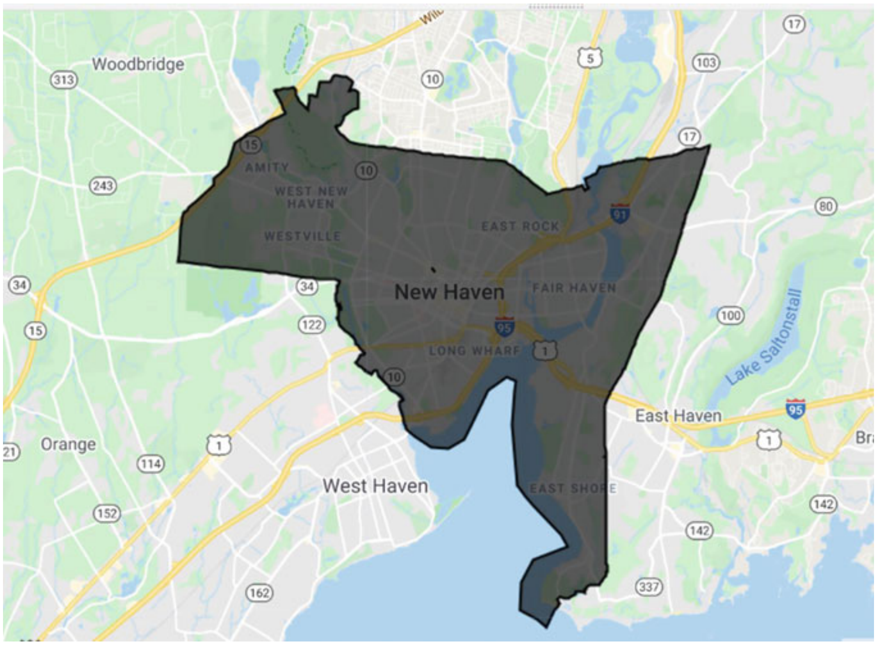


Fig. 36.1 Boundary of New Haven, Connecticut

We start by loading the feature collection, which, being a census tract-level aggregation, we dissolve to get the overall boundary using the `union` operation. The `FeatureCollection` is added to the map for demonstration:

```
// Load feature collection of New Haven's census tracts
from user assets.
var regionInt = ee.FeatureCollection(
    'projects/gee-book/assets/A1-5/TC_NewHaven');

// Get dissolved feature collection using an error margin
of 50 meters.
var regionInt = regionInt.union(50);

// Set map center and zoom level (Zoom level varies from 1
to 20).
Map.setCenter(-72.9, 41.3, 12);

// Add layer to map.
Map.addLayer(regionInt, {}, 'New Haven boundary');
```

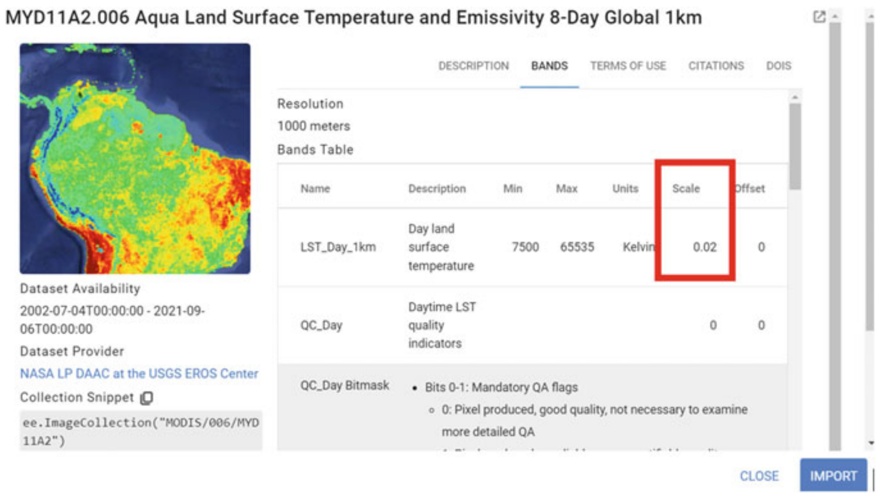
Next we load in the MODIS MYD11A2 version 6 product, which provides eight-day composites of LST from the Aqua satellite. This corresponds to an equatorial crossing time of roughly 1:30 p.m. during daytime and 1:30 a.m. at night. In contrast, the MODIS sensor onboard the Terra platform (MOD11A2 version 6) has an overpass of roughly 10:30 a.m. and 10:30 p.m local time.

```
// Load MODIS image collection from the Earth Engine data
catalog.
var modisLst = ee.ImageCollection('MODIS/006/MYD11A2');

// Select the band of interest (in this case: Daytime LST).
var landSurfTemperature = modisLst.select('LST_Day_1km');
```

We want to focus on only summertime SUHI, so we will create a five-year summer composite of LST using a day-of-year filter assembling images from June 1 (day 152) to August 31 (day 243) in each year:

```
// Create a summer filter.
var sumFilter = ee.Filter.dayOfYear(152, 243);

// Filter the date range of interest using a date filter.
var lstDateInt = landSurfTemperature
    .filterDate('2014-01-01', '2019-01-01')
    .filter(sumFilter);

// Take pixel-wise mean of all the images in the
collection.
var lstMean = lstDateInt.mean();
```

We now convert this image into LST in degrees Celsius and mask out all the water pixels (the high specific heat capacity of water would affect LST, and we are focused on land pixels). For the water mask, we use the Global Surface Water dataset (Pekel et al. 2016); to convert the pixel values, we use the scaling factor for the band from the data provider and then subtract by 273.15 to convert from Kelvin to degrees Celsius. The scaling factor can be found in the Earth Engine data summary page (Fig. 36.2).


MYD11A2.006 Aqua Land Surface Temperature and Emissivity 8-Day Global 1km

Resolution

1000 meters

Bands Table

Name	Description	Min	Max	Units	Scale	Offset
LST_Day_1km	Day land surface temperature	7500	65535	Kelvin	0.02	0
QC_Day	Daytime LST quality indicators				0	0
QC_Day Bitmask	• Bits 0-1: Mandatory QA flags ◦ 0: Pixel produced, good quality, not necessary to examine more detailed QA					

Dataset Availability

2002-07-04T00:00:00 - 2021-09-06T00:00:00

Dataset Provider

NASA LP DAAC at the USGS EROS Center

Collection Snippet

```
ee.ImageCollection("MODIS/006/MYD
11A2")
```

Fig. 36.2 Scaling factor in the data summary

Finally, we clip the image using the city boundary and add the layer to the map.

```
// Multiply each pixel by scaling factor to get the LST
values.
var lstFinal = lstMean.multiply(0.02);

// Generate a water mask.
var water =
ee.Image('JRC/GSW1_0/GlobalSurfaceWater').select(
    'occurrence');
var notWater = water.mask().not();
// Clip data to region of interest, convert to degree
Celsius, and mask water pixels.
var lstNewHaven = lstFinal.clip(regionInt).subtract(273.15)
    .updateMask(notWater);

// Add layer to map.
Map.addLayer(lstNewHaven, {
        palette: ['blue', 'white', 'red'],
        min: 25,
        max: 38
    },
    'LST_MODIS');
```

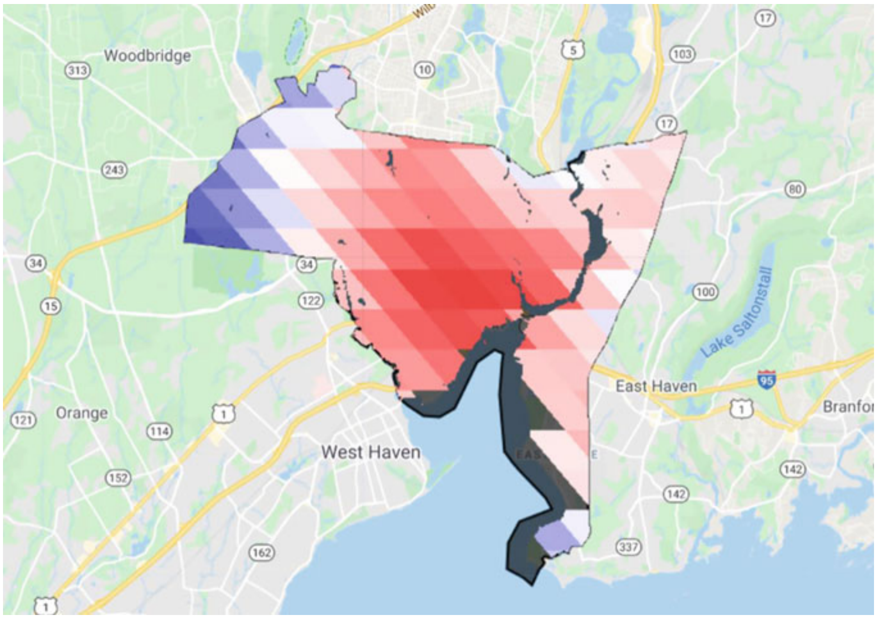


Fig. 36.3 Five-year summer composite of daytime MODIS Aqua LST over New Haven, Connecticut. Red pixels show higher LST values and blue pixels have lower values

Code Checkpoint A15a. The book's repository contains a script that shows what your code should look like at this point.

36.2.2.1 Section 1.2: Deriving Land Surface Temperature from Landsat

Working with MODIS LST is relatively simple because the data are already processed by the NASA team. You can also derive LST from Landsat, which has a much finer native resolution (between ~60 m and ~120 m depending on satellite) than the ~ 1 km MODIS pixels. However, you need to derive LST yourself from the measurements in the thermal bands, which also usually involves some estimate of surface emissivity (Li et al. 2013). The surface emissivity (ε) of a material is the effectiveness with which it can emit thermal radiation compared to a black body at the same temperature and can range from 0 (for a perfect reflector) to 1 (for a perfect absorber and emitter). Since the thermal radiation captured by satellites is a function of both LST and ε, you need to accurately prescribe or estimate ε to get to the correct LST. Let's consider one such simple method using Landsat 8 data.

We will start by loading in the Landsat data, cloud screening, and then filtering to a time and region of interest. Continuing in the same script, add the following code:

```
// Function to filter out cloudy pixels.
function cloudMask(cloudyScene) {
    // Add a cloud score band to the image.
    var scored =
ee.Algorithms.Landsat.simpleCloudScore(cloudyScene);

    // Create an image mask from the cloud score band and
specify threshold.
    var mask = scored.select(['cloud']).lte(10);

    // Apply the mask to the original image and return the
masked image.
    return cloudyScene.updateMask(mask);
}

// Load the collection, apply coud mask, and filter to date
and region of interest.
var col = ee.ImageCollection('LANDSAT/LC08/C02/T1_TOA')
    .filterBounds(regionInt)
    .filterDate('2014-01-01', '2019-01-01')
    .filter(sumFilter)
    .map(cloudMask);

print('Landsat collection', col);
```

After creating a median composite as a simple way to further reduce the influence of clouds, we mask out the water pixels and select the brightness temperature band.

```
// Generate median composite.
var image = col.median();

// Select thermal band 10 (with brightness temperature).
var thermal = image.select('B10')
    .clip(regionInt)
    .updateMask(notWater);

Map.addLayer(thermal, {
        min: 295,
        max: 310,
        palette: ['blue', 'white', 'red']
    },
    'Landsat_BT');
```

Brightness temperature (Fig. 36.4) is the temperature equivalent of the infrared radiation escaping the top of the atmosphere, assuming the Earth to be a black body. It is not the same as the LST, which requires accounting for atmospheric absorption and re-emission, as well as the emissivity of the land surface. One way to derive pixel-level emissivity is as a function of the vegetation fraction of the pixel (Malakar et al. 2018). For this, we start by calculating the Normalized Difference Vegetation Index (NDVI) from the Landsat surface reflectance data (see Fig. 36.5).

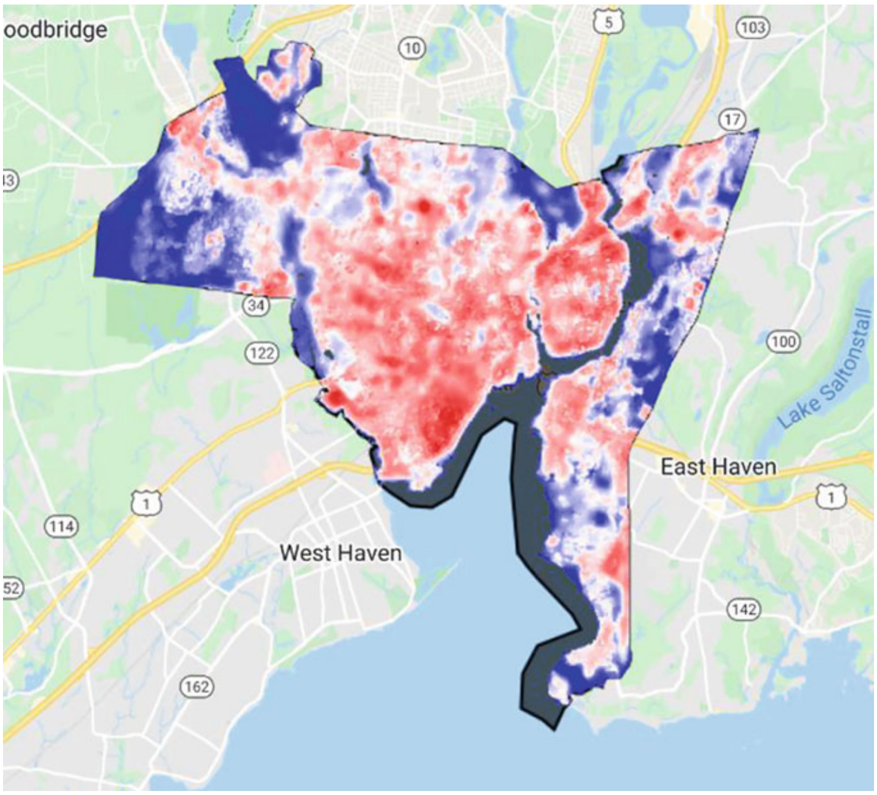


Fig. 36.4 Five-year summer median composite of Landsat brightness temperature over New Haven, Connecticut. Red pixels show higher values, and blue pixels have lower values

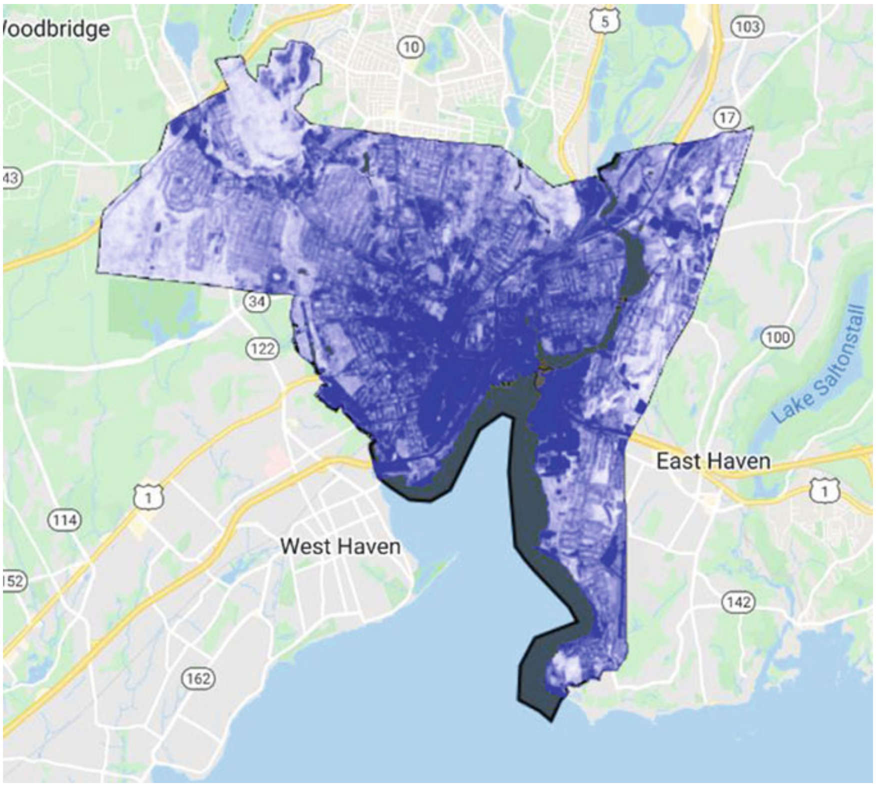


Fig. 36.5 Five-year summer median composite of Landsat-derived NDVI over New Haven, Connecticut. White pixels show higher NDVI values, and blue pixels have lower values

```
// Calculate Normalized Difference Vegetation Index (NDVI)
// from Landsat surface reflectance.
var ndvi = ee.ImageCollection('LANDSAT/LC08/C02/T1_L2')
    .filterBounds(regionInt)
    .filterDate('2014-01-01', '2019-01-01')
    .filter(sumFilter)
    .median()
    .normalizedDifference(['SR_B5',
'SR_B4']).rename('NDVI')
    .clip(regionInt)
    .updateMask(notWater);

Map.addLayer(ndvi, {
        min: 0,
        max: 1,
        palette: ['blue', 'white', 'green']
    },
    'ndvi');
```

To map NDVI for each pixel to the actual fraction of the pixel with vegetation (fractional vegetation cover), we next use a relationship based on the range of NDVI values for each pixel.

```
// Find the minimum and maximum of NDVI.  Combine the
reducers
// for efficiency (single pass over the data).
var minMax = ndvi.reduceRegion({
    reducer: ee.Reducer.min().combine({
        reducer2: ee.Reducer.max(),
        sharedInputs: true
    }),
    geometry: regionInt,
    scale: 30,
    maxPixels: 1e9
});
print('minMax', minMax);

var min = ee.Number(minMax.get('NDVI_min'));
var max = ee.Number(minMax.get('NDVI_max'));
```

```
// Calculate fractional vegetation.
var fv =
ndvi.subtract(min).divide(max.subtract(min)).rename('FV');
Map.addLayer(fv, {
    min: 0,
    max: 1,
    palette: ['blue', 'white', 'green']
}, 'fv');
```

Now we use an empirical model of emissivity based on this fractional vegetation cover (Sekertekin and Bonafoni 2020).

```
// Emissivity calculations.
var a = ee.Number(0.004);
var b = ee.Number(0.986);
var em =
fv.multiply(a).add(b).rename('EMM').updateMask(notWater);

Map.addLayer(em, {
        min: 0.98,
        max: 0.99,
        palette: ['blue', 'white', 'green']
    },
    'EMM');
```

As seen in Fig. 36.6, emissivity is lower over the built-up structures compared to over vegetation, which is expected. Note that different models of estimating emissivity would lead to some differences in LST values as well as the SUHI intensity (Sekertekin and Bonafoni 2020; Chakraborty et al. 2021a).

Then we combine this emissivity with the brightness temperature to calculate the LST for each pixel using a simple single-channel algorithm, which is a linearized approximation of the radiation transfer equation (Ermida et al. 2020).

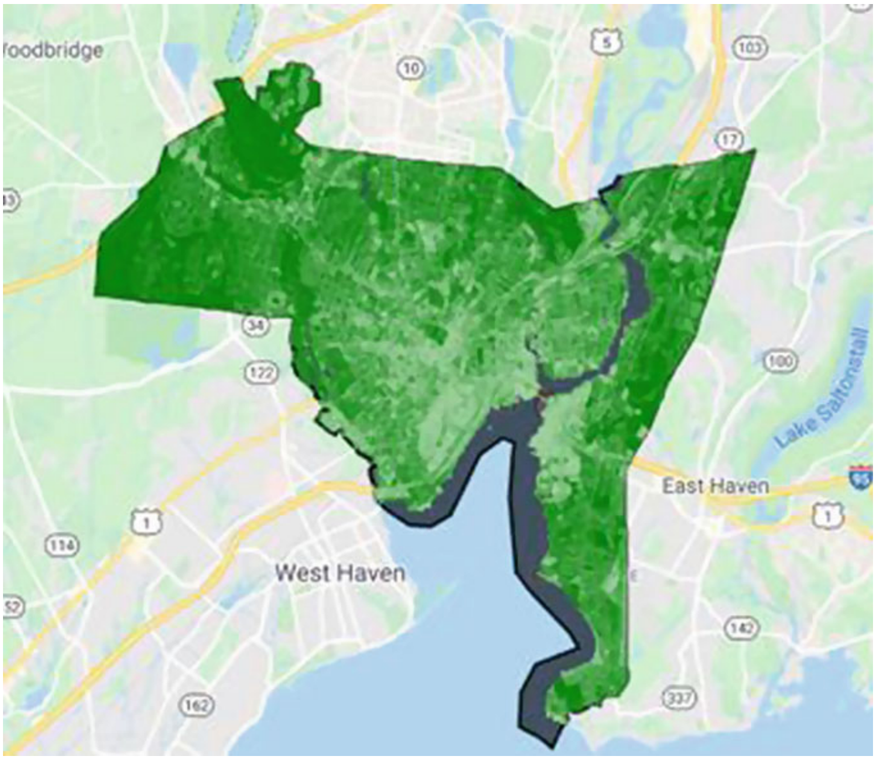


Fig. 36.6 Surface emissivity over New Haven, Connecticut, based on vegetation fraction. Green pixels show higher values, and white pixels have lower values

```
// Calculate LST from emissivity and brightness
temperature.
var lstLandsat = thermal.expression(
    '(Tb/(1 + (0.001145* (Tb / 1.438))*log(Ep)))-273.15', {
        'Tb': thermal.select('B10'),
        'Ep': em.select('EMM')
    }).updateMask(notWater);

Map.addLayer(lstLandsat, {
        min: 25,
        max: 35,
        palette: ['blue', 'white', 'red'],
    },
    'LST_Landsat');
```

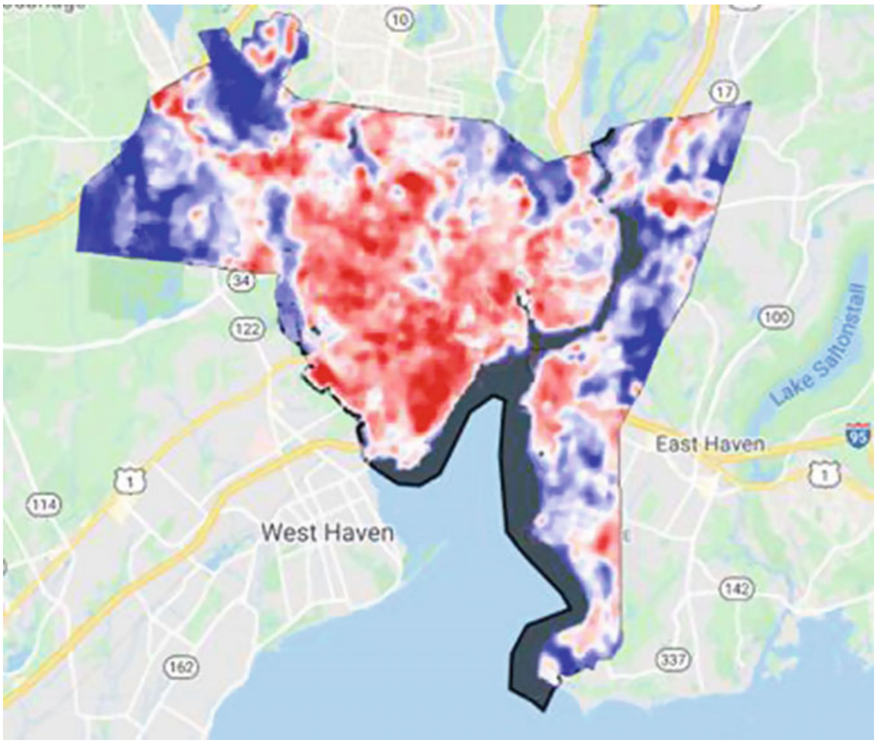


Fig. 36.7 Five-year summer median composite of Landsat-derived LST over New Haven, Connecticut. Red pixels show higher LST values, and blue pixels have lower values

The Landsat-derived values correspond to those of the MODIS Terra daytime overpass. Overall, you do see similar patterns in Figs. 36.3 and 36.7, but Landsat picks up a lot more heterogeneity than MODIS due to its finer resolution.

Code Checkpoint A15b. The book's repository contains a script that shows what your code should look like at this point.

36.2.2.2 Section 1.3: Deriving Land Surface Temperature Using the Earth Engine Landsat LST Toolbox

In the previous section, we explored an LST retrieval algorithm to give an example of the standard steps to get to LST from the satellite measurements in the thermal bands. In this section, we will use an Earth Engine module developed for this purpose to calculate LST (Fig. 36.8).

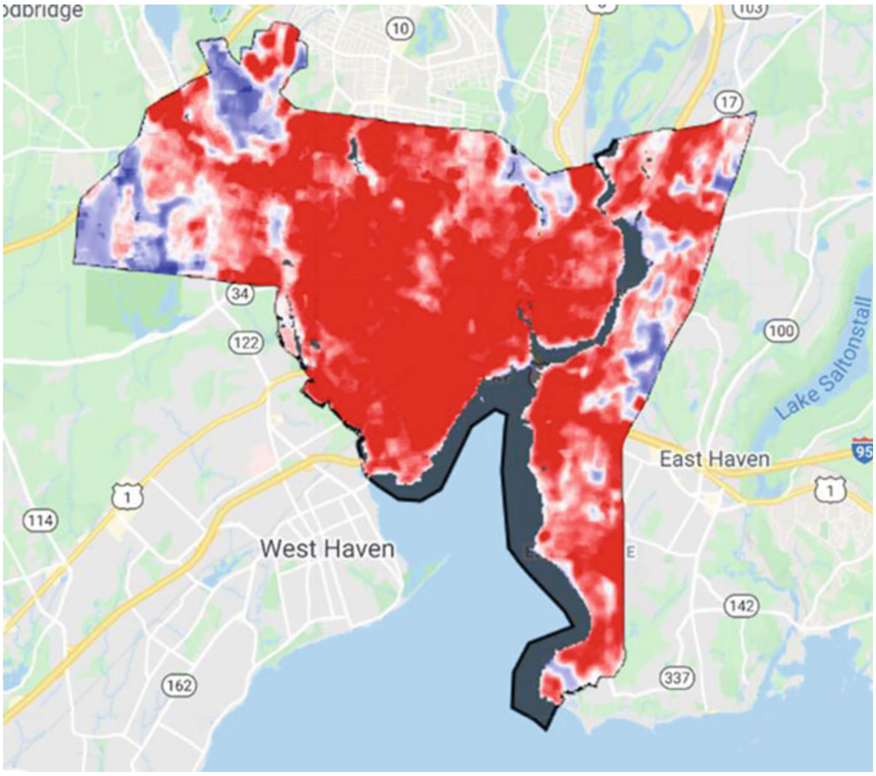


Fig. 36.8 Five-year summer median composite of Landsat-derived LST over New Haven, Connecticut, using the Statistical Mono-Window algorithm. Red pixels show higher LST values, and blue pixels have lower values

```
// Link to the module that computes the Landsat LST.
var landsatLST = require(
    'projects/gee-edu/book:Part A - Applications/A1 - Human
Applications/A1.5 Heat Islands/modules/Landsat_LST.js');

// Select region of interest, date range, and Landsat
satellite.
var geometry = regionInt.geometry();
var satellite = 'L8';
var dateStart = '2014-01-01';
var dateEnd = '2019-01-01';
var useNdvi = true;

 // Get Landsat collection with additional necessary
 variables.
 var landsatColl = landsatLST.collection(satellite,
 dateStart, dateEnd,
     geometry, useNdvi);

 // Create composite, clip, filter to summer, mask, and
 convert to degree Celsius.
 var landsatComp = landsatColl
     .select('LST')
     .filter(sumFilter)
     .median()
     .clip(regionInt)
     .updateMask(notWater)
     .subtract(273.15);

Map.addLayer(landsatComp, {
        min: 25,
        max: 38,
        palette: ['blue', 'white', 'red']
    },
    'LST_SMW');
```

As an aside, the Landsat Collection 2 products have recently incorporated LST bands, which can be processed similar to the MODIS data, but with the bands' own specific offsets and scaling factors.

Code Checkpoint A15c. The book's repository contains a script that shows what your code should look like at this point.

36.2.2.3 Section 2: Defining Urban and Rural References

Now that we have estimates of LST using various products and algorithms, we can calculate the rural LST and subtract from the urban LST to get the SUHI intensity.

There are many ways to estimate the rural reference temperature (Li et al. 2022), and we will explore a few of them in this section.

The simplest and probably the most commonly used method to get the rural reference when calculating the SUHI is to generate a buffered area around the urban boundary. The exact width of the buffer varies across studies, with buffers of 2–30 km in width being used in previous studies (Clinton and Gong 2013; Venter et al. 2021; Yao et al. 2019). In Earth Engine, generating such a buffer is simple:

```
// Function to subtract the original urban cluster from the
buffered cluster
// to generate rural references.
function bufferSubtract(feature) {
    return ee.Feature(feature.geometry()
        .buffer(2000)
        .difference(feature.geometry()));
}

var ruralRef = regionInt.map(bufferSubtract);

Map.addLayer(ruralRef, {
    color: 'green'
}, 'Buffer_ref');
```

In the script above, a buffered polygon with a 2 km width is generated around the urban boundary, and the original urban boundary is subtracted from the buffered polygon. The result is shown in Fig. 36.9.

The use of a constant buffer assumes that all urban areas, regardless of size, have a similar influence around the city. This may not be true for large cities. In fact, there is some evidence that there is a footprint of the SUHI that is dependent on the size of the city (Yang et al. 2019; Zhou et al. 2015). As such, another way to define the buffered region is to normalize its area by the area of the urban cluster it surrounds (Chakraborty et al. 2021b, Peng et al. 2012). One way to do so in Earth Engine is by using an iterative method. This method (see code block below) uses functions to first calculate buffers of different widths around a geometry and then select the buffered region that is closest in size to the original geometry.

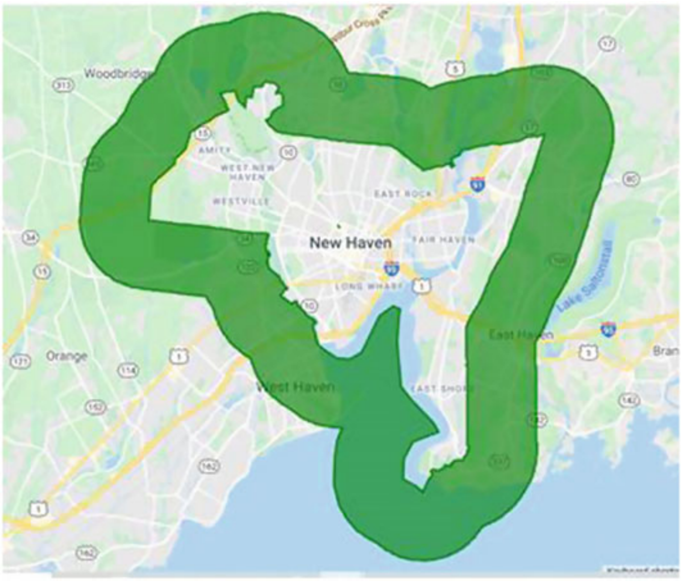


Fig. 36.9 A 2 km buffer around the original city boundary to serve as the rural reference

```
// Define sequence of buffer widths to be tested.
var buffWidths = ee.List.sequence(30, 3000, 30);

// Function to generate standardized buffers (approximately
comparable to area of urban cluster).
function bufferOptimize(feature) {
    function buff(buffLength) {
        var buffedPolygon = ee.Feature(feature.geometry()
                .buffer(ee.Number(buffLength)))
            .set({
                'Buffer_width': ee.Number(buffLength)
            });
        var area =
buffedPolygon.geometry().difference(feature
            .geometry()).area();
        var diffFeature = ee.Feature(
            buffedPolygon.geometry().difference(feature
                .geometry()));
        return diffFeature.set({
            'Buffer_diff': area.subtract(feature.geometry()
                .area()).abs(),
            'Buffer_area': area,
            'Buffer_width':
buffedPolygon.get('Buffer_width')
        });
    }

    var buffed =
ee.FeatureCollection(buffWidths.map(buff));
    var sortedByBuffer = buffed.sort({
        property: 'Buffer_diff'
    });
    var firstFeature = ee.Feature(sortedByBuffer.first());
    return firstFeature.set({
        'Urban_Area': feature.get('Area'),
        'Buffer_width': firstFeature.get('Buffer_width')
    });
}

// Map function over urban feature collection.
var ruralRefStd = regionInt.map(bufferOptimize);

Map.addLayer(ruralRefStd, {
    color: 'brown'
}, 'Buffer_ref_std');

print('ruralRefStd', ruralRefStd);
```

Note how mapping the `buff` function over a sequence of pre-defined values, as done here, does not require loops, which are best avoided when using Earth Engine. The same is true of mapping the `bufferOptimize` function: here it is mapped over a `FeatureCollection` with a single feature, but it would work even if `regionInt` contained multiple features. In this way, nested `map` functions in Earth Engine have the utility of nested loops in other languages.

Check the printed value on the **Console**. According to the result, within an uncertainty of 30 m, a buffer of 1170 m in width creates a polygon that is roughly equal to the area of the city. This function is best to run via export when working with large feature collections.

The final way to define a rural reference does not use a buffer at all, but relies on land cover classes to select pixels that are urban versus non-urban (the Simplified Urban Extent algorithm; Chakraborty et al. 2020; Chakraborty and Lee 2019). For this, we will rely on the NLCD 2016 land cover data (Wickham et al. 2021) and create masks for urban and non-urban pixels (Fig. 36.10).

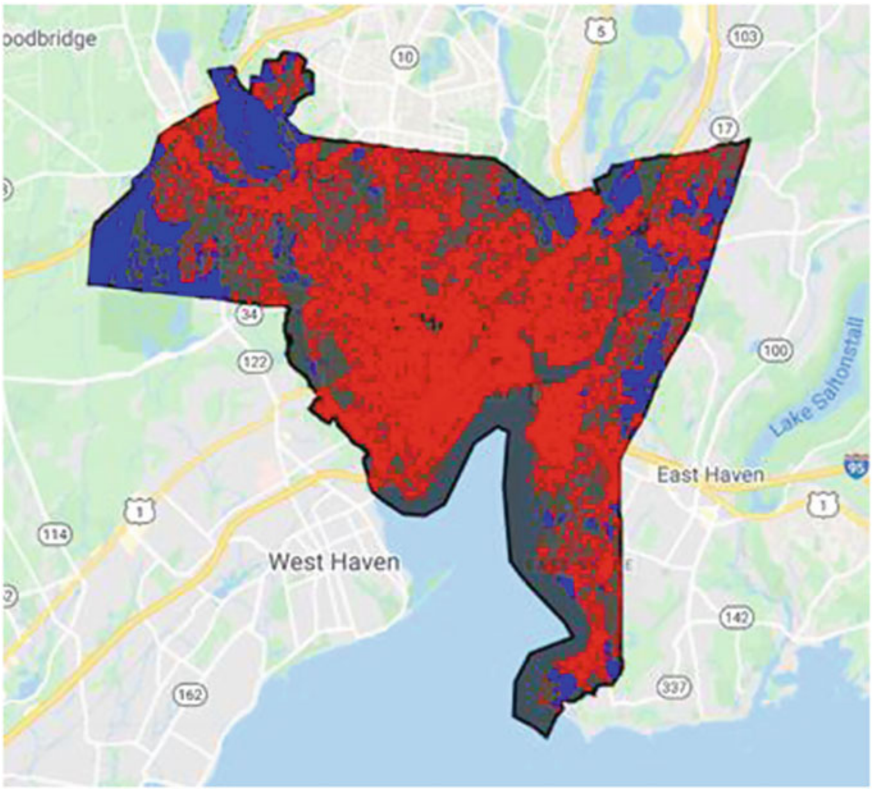


Fig. 36.10 Urban (red) and rural (blue) pixels in New Haven, Connecticut

```
// Select the NLCD land cover data.
var landCover =
ee.Image('USGS/NLCD/NLCD2016').select('landcover');
var urban = landCover;

// Select urban pixels in image.
var urbanUrban =
urban.updateMask(urban.eq(23).or(urban.eq(24)));

// Select background reference pixels in the image.
var nonUrbanVals = [41, 42, 43, 51, 52, 71, 72, 73, 74, 81,
82];
var nonUrbanPixels =
urban.eq(ee.Image(nonUrbanVals)).reduce('max');
var urbanNonUrban = urban.updateMask(nonUrbanPixels);

Map.addLayer(urbanUrban.clip(regionInt), {
    palette: 'red'
}, 'Urban pixels');
Map.addLayer(urbanNonUrban.clip(regionInt), {
    palette: 'blue'
}, 'Non-urban pixels');
```

We can then subsequently use these as masks to select urban versus rural LST pixels. You will find more about this in the next section.

Code Checkpoint A15d. The book's repository contains a script that shows what your code should look like at this point.

36.2.2.4 Section 3: Calculating the Surface Urban Heat Island Intensity

Since the SUHI is the temperature difference between the urban area and the rural reference, we will calculate summary temperature values for the urban boundary and the different versions of rural reference using the Landsat and MODIS LST.

```
// Define function to reduce regions and summarize pixel
values
// to get mean LST for different cases.
function polygonMean(feature) {

    // Calculate spatial mean value of LST for each case
    // making sure the pixel values are converted to °C
from Kelvin.
    var reducedLstUrb =
lstFinal.subtract(273.15).updateMask(notWater)
        .reduceRegion({
            reducer: ee.Reducer.mean(),
            geometry: feature.geometry(),
            scale: 30
        });
    var reducedLstUrbMask =
lstFinal.subtract(273.15).updateMask(
            notWater)
        .updateMask(urbanUrban)
        .reduceRegion({
            reducer: ee.Reducer.mean(),
            geometry: feature.geometry(),
            scale: 30
        });
    var reducedLstUrbPix =
lstFinal.subtract(273.15).updateMask(
            notWater)
        .updateMask(urbanUrban)
        .reduceRegion({
            reducer: ee.Reducer.mean(),
            geometry: feature.geometry(),
            scale: 500
        });
    var reducedLstLandsatUrbPix =
landsatComp.updateMask(notWater)
        .updateMask(urbanUrban)
        .reduceRegion({
            reducer: ee.Reducer.mean(),
            geometry: feature.geometry(),
            scale: 30
        });
    var reducedLstRurPix =
lstFinal.subtract(273.15).updateMask(
            notWater)
        .updateMask(urbanNonUrban)
        .reduceRegion({
            reducer: ee.Reducer.mean(),
            geometry: feature.geometry(),
            scale: 500
        });
    var reducedLstLandsatRurPix =
landsatComp.updateMask(notWater)
        .updateMask(urbanNonUrban)
        .reduceRegion({
            reducer: ee.Reducer.mean(),
            geometry: feature.geometry(),
            scale: 30
        });

    // Return each feature with the summarized LSY values
as properties.
    return feature.set({
        'MODIS_LST_urb': reducedLstUrb.get('LST_Day_1km'),
        'MODIS_LST_urb_mask': reducedLstUrbMask.get(
            'LST_Day_1km'),
        'MODIS_LST_urb_pix': reducedLstUrbPix.get(
            'LST_Day_1km'),
        'MODIS_LST_rur_pix': reducedLstRurPix.get(
            'LST_Day_1km'),
        'Landsat_LST_urb_pix': reducedLstLandsatUrbPix.get(
            'LST'),
        'Landsat_LST_rur_pix': reducedLstLandsatRurPix.get(
            'LST')
    });
}

// Map the function over the urban boundary to get mean
urban and rural LST
// for cases without any explicit buffer-based boundaries.
var reduced = regionInt.map(polygonMean);
```

As you know from the code above, we extract urban temperature from MODIS (’MODIS_LST_urb) and from Landsat and MODIS (’MODIS_LST_urb_pix’ and ’Landsat_LST_urb_pix’) after considering only the urban pixels within the boundary.

Corresponding values are also extracted from the rural reference (including using only rural reference pixels within the urban boundary). For the buffered regions (both using constant width and variable width), we define and call another function.

```
// Define a function to reduce region and summarize pixel
values
// to get mean LST for different cases.
function refMean(feature) {
    // Calculate spatial mean value of LST for each case
    // making sure the pixel values are converted to °C
from Kelvin.
    var reducedLstRur =
lstFinal.subtract(273.15).updateMask(notWater)
        .reduceRegion({
            reducer: ee.Reducer.mean(),
            geometry: feature.geometry(),
            scale: 30
        });
    var reducedLstRurMask =
lstFinal.subtract(273.15).updateMask(
            notWater)
        .updateMask(urbanNonUrban)
        .reduceRegion({
            reducer: ee.Reducer.mean(),
            geometry: feature.geometry(),
            scale: 30
        });
    return feature.set({
        'MODIS_LST_rur': reducedLstRur.get('LST_Day_1km'),
        'MODIS_LST_rur_mask': reducedLstRurMask.get(
            'LST_Day_1km'),
    });
}
```

```
// Map the function over the constant buffer rural
reference boundary one.
var reducedRural =
ee.FeatureCollection(ruralRef).map(refMean);

// Map the function over the standardized rural reference
boundary.
var reducedRuralStd = ruralRefStd.map(refMean);

print('reduced', reduced);
print('reducedRural', reducedRural);
print('reducedRuralStd', reducedRuralStd);
```

We can print the newly created feature collections to go through these values for the different cases. Even though absolute MODIS and Landsat urban LSTs are different (29 for Landsat and 34 for MODIS), the SUHI is similar (3.6 C for MODIS and 4.8 C from Landsat). As one might expect, when only urban pixels are considered within the boundary, the average LST is higher (and lower for rural LST).

The SUHI variability within the city (Fig. 36.11) can then be displayed by subtracting the rural LST from the total LST:

```
// Display SUHI variability within the city.
var suhi = landsatComp
    .updateMask(urbanUrban)
    .subtract(ee.Number(ee.Feature(reduced.first())
        .get('Landsat_LST_rur_pix')));

Map.addLayer(suhi, {
    palette: ['blue', 'white', 'red'],
    min: 2,
    max: 8
}, 'SUHI');
```

Code Checkpoint A15e. The book's repository contains a script that shows what your code should look like at this point.

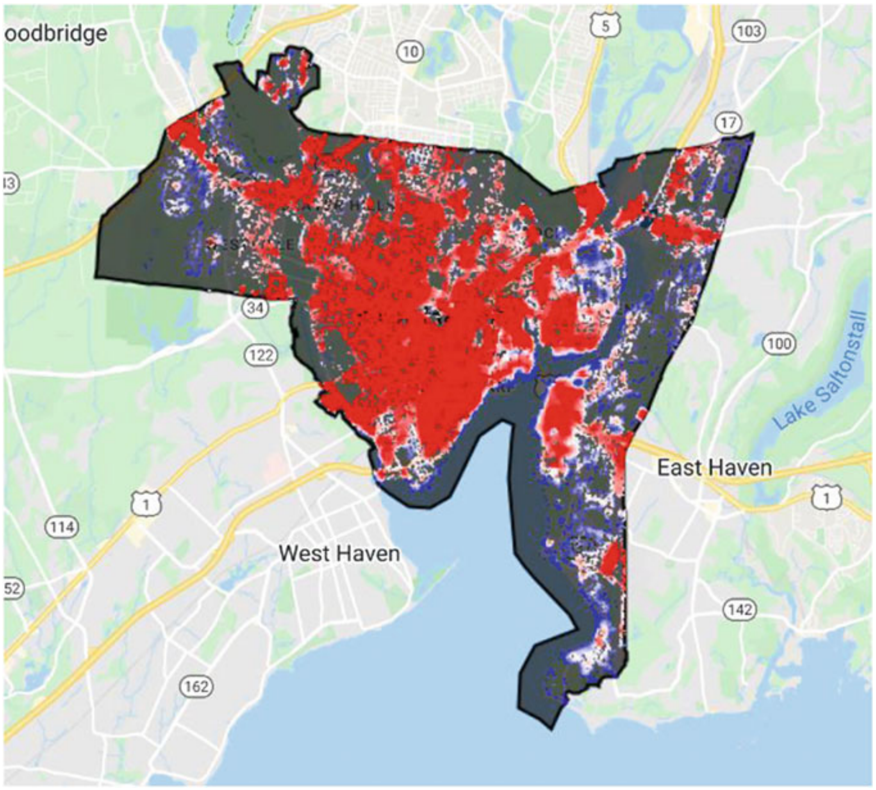


Fig. 36.11 Spatial variability in SUHI for the period of interest for New Haven, Connecticut. Red pixels show higher values and blue pixels show lower values

36.3 Synthesis

Now that you know the different ways to calculate the SUHI and estimate LST using Earth Engine, load in your own city's feature collection and compare the different methods.

Question 1. What are the SUHI values during summer and winter from the two products?

Question 2. How does a pixel-based urban–rural delineation method compare to a buffer-based method for SUHI estimation?

36.4 Conclusion

You should now have a good understanding of satellite measurements in the thermal bands, how they can be used to estimate LST, and how we can calculate the SUHI using these measurements.

References

Arnfield AJ (2003) Two decades of urban climate research: A review of turbulence, exchanges of energy and water, and the urban heat island. Int J Climatol 23:1–26. https://doi.org/10.1002/joc.859

Chakraborty TC, Lee X, Ermida S, Zhan W (2021a) On the land emissivity assumption and Landsat-derived surface urban heat islands: A global analysis. Remote Sens Environ 265:112682. https://doi.org/10.1016/j.rse.2021.112682

Chakraborty TC, Sarangi C, Lee X (2021b) Reduction in human activity can enhance the urban heat island: Insights from the COVID-19 lockdown. Environ Res Lett 16:54060. https://doi.org/10.1088/1748-9326/abef8e

Chakraborty T, Hsu A, Manya D, Sheriff G (2020) A spatially explicit surface urban heat island database for the United States: Characterization, uncertainties, and possible applications. ISPRS J Photogramm Remote Sens 168:74–88. https://doi.org/10.1016/j.isprsjprs.2020.07.021

Chakraborty T, Lee X (2019) A simplified urban-extent algorithm to characterize surface urban heat islands on a global scale and examine vegetation control on their spatiotemporal variability. Int J Appl Earth Obs Geoinf 74:269–280. https://doi.org/10.1016/j.jag.2018.09.015

Chakraborty T, Sarangi C, Tripathi SN (2017) Understanding diurnality and inter-seasonality of a sub-tropical urban heat island. Boundary-Layer Meteorol 163:287–309. https://doi.org/10.1007/s10546-016-0223-0

Clinton N, Gong P (2013) MODIS detected surface urban heat islands and sinks: Global locations and controls. Remote Sens Environ 134:294–304. https://doi.org/10.1016/j.rse.2013.03.008

Ermida SL, Soares P, Mantas V et al (2020) Google Earth Engine open-source code for land surface temperature estimation from the Landsat series. Remote Sens 12:1471. https://doi.org/10.3390/RS12091471

Howard L (1833) The Climate of London: Deduced from Meteorological Observations Made in the Metropolis and at Various Places Around it. Harvey and Darton, J. and A. Arch, Longman, Hatchard, S. Highley and R. Hunter

Hu Y, Hou M, Jia G et al (2019) Comparison of surface and canopy urban heat islands within megacities of eastern China. ISPRS J Photogramm Remote Sens 156:160–168. https://doi.org/10.1016/j.isprsjprs.2019.08.012

Kalnay E, Cai M (2003) Impact of urbanization and land-use change on climate. Nature 425:102. https://doi.org/10.1038/nature01952

Li K, Chen Y, Gao S (2022) Uncertainty of city-based urban heat island intensity across 1112 global cities: Background reference and cloud coverage. Remote Sens Environ 271:112898. https://doi.org/10.1016/j.rse.2022.112898

Li ZL, Wu H, Wang N et al (2013) Land surface emissivity retrieval from satellite data. Int J Remote Sens 34:3084–3127. https://doi.org/10.1080/01431161.2012.716540

Malakar NK, Hulley GC, Hook SJ et al (2018) An operational land surface temperature product for Landsat thermal data: Methodology and validation. IEEE Trans Geosci Remote Sens 56:5717–5735. https://doi.org/10.1109/TGRS.2018.2824828

Ngie A, Abutaleb K, Ahmed F et al (2014) Assessment of urban heat island using satellite remotely sensed imagery: A review. South African Geogr J 96:198–214. https://doi.org/10.1080/03736245.2014.924864

Oke TR (1982) The energetic basis of the urban heat island. Q J R Meteorol Soc 108:1–24. https://doi.org/10.1002/qj.49710845502

Pekel JF, Cottam A, Gorelick N, Belward AS (2016) High-resolution mapping of global surface water and its long-term changes. Nature 540:418–422. https://doi.org/10.1038/nature20584

Peng S, Piao S, Ciais P et al (2012) Surface urban heat island across 419 global big cities. Environ Sci Technol 46:6889–6890. https://doi.org/10.1021/es301811b

Qian Y, Chakraborty TC, Li J et al (2022) Urbanization impact on regional climate and extreme weather: Current understanding, uncertainties, and future research directions. Adv Atmos Sci 39:819–860. https://doi.org/10.1007/s00376-021-1371-9

Sekertekin A, Bonafoni S (2020) Sensitivity analysis and validation of daytime and nighttime land surface temperature retrievals from Landsat 8 using different algorithms and emissivity models. Remote Sens 12:2776. https://doi.org/10.3390/RS12172776

Venter ZS, Chakraborty T, Lee X (2021) Crowdsourced air temperatures contrast satellite measures of the urban heat island and its mechanisms. Sci Adv 7:eabb9569. https://doi.org/10.1126/sciadv.abb9569

Wan Z (2006) MODIS land surface temperature products users' guide. Inst Comput Earth Syst Sci Univ Calif St Barbar CA, USA, p 805

Wickham J, Stehman SV, Sorenson DG et al (2021) Thematic accuracy assessment of the NLCD 2016 land cover for the conterminous United States. Remote Sens Environ 257:112357. https://doi.org/10.1016/j.rse.2021.112357

Yang Q, Huang X, Tang Q (2019) The footprint of urban heat island effect in 302 Chinese cities: Temporal trends and associated factors. Sci Total Environ 655:652–662. https://doi.org/10.1016/j.scitotenv.2018.11.171

Yao R, Wang L, Huang X et al (2019) Greening in rural areas increases the surface urban heat island intensity. Geophys Res Lett 46:2204–2212. https://doi.org/10.1029/2018GL081816

Zhou D, Zhao S, Zhang L et al (2015) The footprint of urban heat island effect in China. Sci Rep 5:1–11. https://doi.org/10.1038/srep11160

37 Health Applications

Dawn Nekorchuk

Overview

The purpose of this chapter is to demonstrate how Google Earth Engine may be used to support modeling and forecasting of vector-borne infectious diseases such as malaria. In doing so, the chapter will also show how Earth Engine may be used to gather data for subsequent analyses outside of Earth Engine, the results of which can then also be brought back into Earth Engine.

We will be calculating and exporting data of remotely sensed environmental variables: precipitation, temperature, and a vegetation water index. These factors can impact mosquito life cycles, malaria parasites, and transmission dynamics. These data can then be used in R for modeling and forecasting malaria in the Amhara region of Ethiopia, using the Epidemic Prognosis Incorporating Disease and Environmental Monitoring for Integrated Assessment (EPIDEMIA) system, developed by the EcoGRAPH research group at the University of Oklahoma.

Learning Outcomes

- Extracting and calculating malaria-relevant variables from existing datasets: precipitation, temperature, and wetness.
- Importing satellite data and filtering for images over a region and time period.
- Joining two data products to get additional quality information.
- Computing zonal summaries of the calculated variables for elements in a `FeatureCollection`.

D. Nekorchuk (✉)
Ecological and Geospatial Research and Application in Planetary Health Lab, Department of Geography and Environmental Sustainability, University of Oklahoma, Norman, OK, USA

Spatial Informatics Group, Pleasanton, CA, USA
e-mail: dnekorchuk@sig-gis.com

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_37

Helps if you know how to

- Import images and image collections, filter, and visualize (Part 1).
- Perform basic image analysis: select bands, compute indices, create masks (Part 2).
- Use expressions to perform calculations on image bands (Chap. 9).
- Write a function and `map` it over an `ImageCollection` (Chap. 12).
- Mask cloud, cloud shadow, snow/ice, and other undesired pixels (Chap. 15).
- Flatten a table for export to CSV (Chap. 22).
- Use `reduceRegions` to summarize an image with zonal statistics in irregular shapes (Chaps. 22 and 24).
- Write a function and `map` it over a `FeatureCollection` (Chaps. 23 and 24).

37.1 Introduction to Theory

Vector-borne diseases cause more than 700,000 deaths per year, of which approximately 400,000 are due to malaria, a parasitic infection spread by *Anopheles* mosquitoes (World Health Organization 2018, 2020). The WHO estimates that there were around 229 million clinical cases of malaria worldwide in 2019 (WHO 2020). Environmental factors including temperature, humidity, and rainfall are known to be important determinants of malaria risk as these affect mosquito and parasite development and life cycles, including larval habitats, mosquito fecundity, growth rates, mortality, and *Plasmodium* parasite development rates within the mosquito vector (Franklinos et al. 2019; Jones et al. 2008; Wimberly et al. 2021).

Data from Earth-observing satellites can be used to monitor spatial and temporal changes in these environmental factors (Ford et al. 2009). These data can be incorporated into disease modeling, usually as lagged functions, to help develop early warning systems for forecasting outbreaks (Wimberly et al. 2021, 2022). Accurate forecasts would allow limited resources for prevention and control to be more efficiently and effectively targeted at appropriate locations and times (WHO 2018).

To implement near-real-time forecasting, meteorological and climatic data must be acquired, processed, and integrated on a regular and frequent basis. Over the past 10 years, the Epidemic Prognosis Incorporating Disease and Environmental Monitoring for Integrated Assessment (EPIDEMIA) project has developed and tested a malaria forecasting system that integrates public health surveillance with monitoring of environmental and climate conditions. Since 2018, the environmental data have been acquired using Earth Engine scripts and apps (Wimberly et al. 2022). In 2019, a local team at Bahir Dar University in Ethiopia had been using EPIDEMIA with near-real-time epidemiological data to generate weekly malaria early warning reports in the Amhara region of Ethiopia.

In this example, we are looking at near-real-time environmental conditions that affect disease vectors and human transmission dynamics. On longer time scales, issues such as climate change can alter vector-borne disease transmission cycles and the geographic distributions of various vector and host species (Franklinos et al. 2019). More broadly, health applications involving Earth Engine data likely align with a One Health approach to complex health issues. Under One Health, a core assumption is that environmental, animal, and human health are inextricably linked (Mackenzie and Jeggo 2019).

37.2 Practicum

The goal of the practicum is to create a download of three environmental variables:

1. Precipitation.
2. Mean land surface temperature (LST).
3. Normalized Difference Water Index (NDWI) spectral index.

These downloads will be zonal summaries based on our uploaded shapefile of *woredas* (districts) in the Amhara region of Ethiopia.

The practicum is an extract from the longer Retrieving Environmental Analytics for Climate and Health (REACH) Earth Engine script (developed by Dr. Michael C. Wimberly and Dr. Dawn Nekorchuk) used in the EPIDEMIA project (Dr. Michael C. Wimberly, PI). This script also has a more advanced user interface for the user to request date ranges for the download of data. Links to this script and related apps can be found in the "For Further Reading" section of this book.

37.2.1 Section 1: Data Import

To start, we need to import the data we will be working with. The first item is an external asset of our study area—these are *woredas* in the Amhara region of Ethiopia. The four that follow are remotely sensed data that we will be processing:

- The Integrated Multi-satellite Retrievals for GPM (IMERG) rainfall estimates from Global Precipitation Measurement (GPM) v6.
- Terra Land Surface Temperature and Emissivity 8-Day Global 1 km.
- MODIS Nadir Bidirectional Reflectance Distribution Function (BRDF) Adjusted Reflectance Daily 500 m.
- MODIS BRDF-Albedo Quality Daily 500 m.

```
// Section 1: Data Import
var woredas = ee.FeatureCollection(
    'projects/gee-book/assets/A1-
6/amhara_woreda_20170207');
// Create region outer boundary to filter products on.
var amhara = woredas.geometry().bounds();
var gpm = ee.ImageCollection('NASA/GPM_L3/IMERG_V06');
var LSTTerra8 = ee.ImageCollection('MODIS/061/MOD11A2')
    // Due to MCST outage, only use dates after this for
this script.
    .filterDate('2001-06-26', Date.now());
var brdfReflect = ee.ImageCollection('MODIS/006/MCD43A4');
var brdfQa = ee.ImageCollection('MODIS/006/MCD43A2');
```

We can take a look at the *woreda* boundaries by adding the following code to draw it onto the map (Fig. 37.1). See Chap. 25 for more information on visualizing feature collections.

```
// Visualize woredas with black borders and no fill.
// Create an empty image into which to paint the features,
cast to byte.
var empty = ee.Image().byte();
// Paint all the polygon edges with the same number and
width.
var outline = empty.paint({
    featureCollection: woredas,
    color: 1,
    width: 1
});
// Add woreda boundaries to the map.
Map.setCenter(38, 11.5, 7);
Map.addLayer(outline, {
    palette: '000000'
}, 'Woredas');
```

Code Checkpoint A16a. The book's repository contains a script that shows what your code should look like at this point.

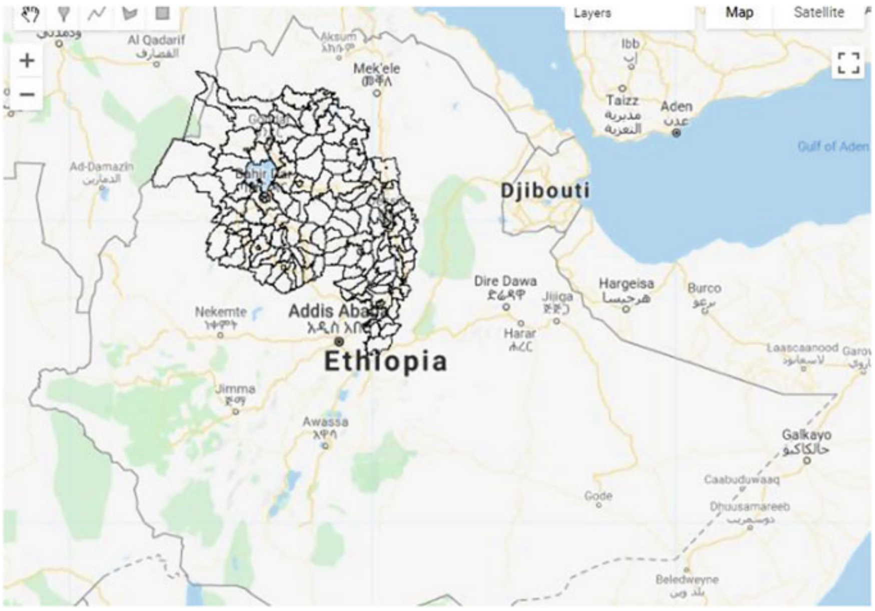


Fig. 37.1 *Woreda* (district) boundaries in the Amhara region of Ethiopia

37.2.2 Section 2: Date Preparation

The user will be requesting the date range for the summarized data, and it is expected that they will be looking for near-real-time data. Different data products that we are using have different data lags, and some data may not be available in the user-requested date range. We will want to get the last available data date, so we can properly create and name our export datasets.

We need daily data, but the LST data are in 8-day composites. For this, we will assign the 8-day composite value to each of the eight days in the range. This means we also need to acquire the 8-day composite value that covers the requested start date (i.e., the previous image).

```
// Section 2: Handling of dates

// 2.1 Requested start and end dates.
var reqStartDate = ee.Date('2021-10-01');
var reqEndDate = ee.Date('2021-11-30');

// 2.2 LST Dates
// LST MODIS is every 8 days, and a user-requested date
will likely not match.
// We want to get the latest previous image date,
//  i.e. the date the closest, but prior to, the requested
date.
// We will filter later.
// Get date of first image.
var LSTEarliestDate = LSTTerra8.first().date();
// Filter collection to dates from beginning to requested
start date.
var priorLstImgCol = LSTTerra8.filterDate(LSTEarliestDate,
    reqStartDate);
// Get the latest (max) date of this collection of earlier
images.
var LSTPrevMax = priorLstImgCol.reduceColumns({
    reducer: ee.Reducer.max(),
    selectors: ['system:time_start']
});
var LSTStartDate = ee.Date(LSTPrevMax.get('max'));
print('LSTStartDate', LSTStartDate);

// 2.3 Last available data dates
// Different variables have different data lags.
// Data may not be available in user range.
// To prevent errors from stopping script,
//  grab last available (if relevant) & filter at end.

// 2.3.1 Precipitation
// Calculate date of most recent measurement for gpm (of
all time).
var gpmAllMax = gpm.reduceColumns(ee.Reducer.max(), [
    'system:time_start'
]);
var gpmAllEndDateTime = ee.Date(gpmAllMax.get('max'));
// GPM every 30 minutes, so get just date part.
```

```
var gpmAllEndDate = ee.Date.fromYMD({
    year: gpmAllEndDateTime.get('year'),
    month: gpmAllEndDateTime.get('month'),
    day: gpmAllEndDateTime.get('day')
});

// If data ends before requested start, take last data
date,
// otherwise use requested date.
var precipStartDate = ee.Date(gpmAllEndDate.millis()
    .min(reqStartDate.millis()));
print('precipStartDate', precipStartDate);
// 2.3.2 BRDF
// Calculate date of most recent measurement for brdf (of
all time).
var brdfAllMax = brdfReflect.reduceColumns({
    reducer: ee.Reducer.max(),
    selectors: ['system:time_start']
});
var brdfAllEndDate = ee.Date(brdfAllMax.get('max'));
// If data ends before requested start, take last data
date,
// otherwise use the requested date.
var brdfStartDate = ee.Date(brdfAllEndDate.millis()
    .min(reqStartDate.millis()));
print('brdfStartDate', brdfStartDate);
print('brdfEndDate', brdfAllEndDate);
```

Code Checkpoint A16b. The book's repository contains a script that shows what your code should look like at this point.

Question 1. Explore the earliest date of LST images you get if you do not specifically acquire the previous image. The following code may be useful:

```
var naiveLstFilter = LSTTerra8.filterDate(reqStartDate,
reqEndDate);
var naiveLstStart = naiveLstFilter.reduceColumns({
    reducer: ee.Reducer.min(),
    selectors: ['system:time_start']
});
var naiveLstStartDate = ee.Date(naiveLstStart.get('min'));
print('naiveLstStartDate', naiveLstStartDate);
```

Question 2. Try changing the requested dates to closer to the current date to see how the dates for the different data products adjust. If you have a narrow window (1–2 weeks), you may find that some data products do not have any data available for the requested time period yet.

37.2.3 Section 3: Precipitation

Now, we will calculate our precipitation variable for the appropriate date range and then perform a zonal summary (see Chap. 24) of our *woredas*.

37.2.3.1 Section 3.1: Precipitation Filtering and Dates

Using the dates when data actually exist in the user-requested date range, we create a list of dates for which we will calculate our variable.

```
// Section 3: Precipitation

// Section 3.1: Precipitation filtering and dates

// Filter gpm by date, using modified start if necessary.
var gpmFiltered = gpm
    .filterDate(precipStartDate, reqEndDate.advance(1,
'day'))
    .filterBounds(amhara)
    .select('precipitationCal');

// Calculate date of most recent measurement for gpm
// (in the modified requested window).
var gpmMax = gpmFiltered.reduceColumns({
    reducer: ee.Reducer.max(),
    selectors: ['system:time_start']
});
var gpmEndDate = ee.Date(gpmMax.get('max'));
var precipEndDate = gpmEndDate;
print('precipEndDate ', precipEndDate);
```

```
// Create a list of dates for the precipitation time
series.
var precipDays = precipEndDate.difference(precipStartDate,
'day');
var precipDatesPrep = ee.List.sequence(0, precipDays, 1);

function makePrecipDates(n) {
    return precipStartDate.advance(n, 'day');
}
var precipDates = precipDatesPrep.map(makePrecipDates);
```

37.2.3.2 Section 3.2: Calculate Daily Precipitation

In this section, we will map a function over our filtered `FeatureCollection` (`gpmFiltered`) to calculate the total daily rainfall per day. In this product, precipitation in millimeters per hour is recorded every half hour, so we will sum the day and divide by two.

```
// Section 3.2: Calculate daily precipitation

// Function to calculate daily precipitation:
function calcDailyPrecip(curdate) {
    curdate = ee.Date(curdate);
    var curyear = curdate.get('year');
    var curdoy = curdate.getRelative('day', 'year').add(1);
    var totprec = gpmFiltered
        .filterDate(curdate, curdate.advance(1, 'day'))
        .select('precipitationCal')
        .sum()
        //every half-hour
        .multiply(0.5)
        .rename('totprec');

    return totprec
        .set('doy', curdoy)
        .set('year', curyear)
        .set('system:time_start', curdate);
}
```

```
// Map function over list of dates.
var dailyPrecipExtended =

ee.ImageCollection.fromImages(precipDates.map(calcDailyPrec
ip));

// Filter back to the original user requested start date.
var dailyPrecip = dailyPrecipExtended
    .filterDate(reqStartDate, precipEndDate.advance(1,
'day'));
```

37.2.3.3 Section 3.3: Summarize Daily Precipitation by *Woreda*

In the last section for precipitation, we will calculate a zonal summary, a mean, of the rainfall per *woreda* and flatten for export as a CSV. The exports (of all variables) will be all done in Sect. 37.2.7.

```
// Section 3.3: Summarize daily precipitation by woreda

// Filter precip data for zonal summaries.
var precipSummary = dailyPrecip
    .filterDate(reqStartDate, reqEndDate.advance(1,
'day'));

// Function to calculate zonal statistics for precipitation
by woreda.
function sumZonalPrecip(image) {
    // To get the doy and year,
    // convert the metadata to grids and then summarize.
    var image2 = image.addBands([
        image.metadata('doy').int(),
        image.metadata('year').int()
    ]);
    // Reduce by regions to get zonal means for each
county.
    var output = image2.select(['year', 'doy', 'totprec'])
        .reduceRegions({
            collection: woredas,
            reducer: ee.Reducer.mean(),
            scale: 1000
        });
    return output;
}
```

```
// Map the zonal statistics function over the filtered
precip data.
var precipWoreda = precipSummary.map(sumZonalPrecip);
// Flatten the results for export.
var precipFlat = precipWoreda.flatten();
```

Code Checkpoint A16c. The book's repository contains a script that shows what your code should look like at this point.

37.2.4 Section 4: Land Surface Temperature

We will follow a similar pattern of steps for land surface temperatures, though first we will calculate the variable (mean LST). Then, we will calculate the daily values and summarize them by *woreda.*

37.2.4.1 Section 4.1: Calculate LST Variables

We will use the daytime and nighttime observed values to calculate a mean value for the day. We will use the quality layers to mask out poor-quality pixels. Working with the bitmask below, we are taking advantage of the fact that bits 6 and 7 are at the end, so the `rightShift(6)` just returns these two. Then, we check if they are less than or equal to 2, meaning average LST error<= 3k (see MODIS documentation for the meaning of each element in the bit sequence). For more information on how to use bitmasks in other situations, see Chap. 15. To convert the pixel values, we will use the scaling factor in the data product (0.2) and convert from Kelvin to Celsius values (−273.15). See Chap. 36, about Heat Islands, for another example using LST data.

```
// Section 4: Land surface temperature

// Section 4.1: Calculate LST variables

// Filter Terra LST by altered LST start date.
// Rarely, but at the end of the year if the last image is
late in the year
//  with only a few days in its period, it will sometimes
not grab
//  the next image. Add extra padding to reqEndDate and
//  it will be trimmed at the end.
var LSTFiltered = LSTTerra8
    .filterDate(LSTStartDate, reqEndDate.advance(8, 'day'))
    .filterBounds(amhara)
    .select('LST_Day_1km', 'QC_Day', 'LST_Night_1km',
'QC_Night');
```

```
// Filter Terra LST by QA information.
function filterLstQa(image) {
    var qaday = image.select(['QC_Day']);
    var qanight = image.select(['QC_Night']);
    var dayshift = qaday.rightShift(6);
    var nightshift = qanight.rightShift(6);
    var daymask = dayshift.lte(2);
    var nightmask = nightshift.lte(2);
    var outimage = ee.Image(image.select(['LST_Day_1km',
        'LST_Night_1km'
    ]));
    var outmask = ee.Image([daymask, nightmask]);
    return outimage.updateMask(outmask);
}
var LSTFilteredQa = LSTFiltered.map(filterLstQa);

// Rescale temperature data and convert to degrees Celsius
(C).
function rescaleLst(image) {
    var LST_day = image.select('LST_Day_1km')
        .multiply(0.02)
        .subtract(273.15)
        .rename('LST_day');
    var LST_night = image.select('LST_Night_1km')
        .multiply(0.02)
        .subtract(273.15)
        .rename('LST_night');
    var LST_mean = image.expression(
        '(day + night) / 2', {
            'day': LST_day.select('LST_day'),
            'night': LST_night.select('LST_night')
        }
    ).rename('LST_mean');
    return image.addBands(LST_day)
        .addBands(LST_night)
        .addBands(LST_mean);
}
var LSTVars = LSTFilteredQa.map(rescaleLst);
```

37.2.4.2 Section 4.2: Calculate Daily LST

Now, using a mapped function over our filtered collection, we will calculate a daily value from the 8-day composite value by assigning each of the eight days the value of the composite. We will also filter to our user-requested dates, as data exist in that range.

```
// Section 4.2: Calculate daily LST

// Create list of dates for time series.
var LSTRange = LSTVars.reduceColumns({
    reducer: ee.Reducer.max(),
    selectors: ['system:time_start']
});
var LSTEndDate = ee.Date(LSTRange.get('max')).advance(7,
'day');
var LSTDays = LSTEndDate.difference(LSTStartDate, 'day');
var LSTDatesPrep = ee.List.sequence(0, LSTDays, 1);

function makeLstDates(n) {
    return LSTStartDate.advance(n, 'day');
}
var LSTDates = LSTDatesPrep.map(makeLstDates);
// Function to calculate daily LST by assigning the 8-day
composite summary
// to each day in the composite period:
function calcDailyLst(curdate) {
    var curyear = ee.Date(curdate).get('year');
    var curdoy = ee.Date(curdate).getRelative('day',
'year').add(1);
    var moddoy =
curdoy.divide(8).ceil().subtract(1).multiply(8).add(
        1);
    var basedate = ee.Date.fromYMD(curyear, 1, 1);
    var moddate = basedate.advance(moddoy.subtract(1),
'day');
```

```
    var LST_day = LSTVars
        .select('LST_day')
        .filterDate(moddate, moddate.advance(1, 'day'))
        .first()
        .rename('LST_day');
    var LST_night = LSTVars
        .select('LST_day')
        .filterDate(moddate, moddate.advance(1, 'day'))
        .first()
        .rename('LST_night');
    var LST_mean = LSTVars
        .select('LST_mean')
        .filterDate(moddate, moddate.advance(1, 'day'))
        .first()
        .rename('LST_mean');
    return LST_day
        .addBands(LST_night)
        .addBands(LST_mean)
        .set('doy', curdoy)
        .set('year', curyear)
        .set('system:time_start', curdate);
}
// Map the function over the image collection
var dailyLstExtended =

ee.ImageCollection.fromImages(LSTDates.map(calcDailyLst));

// Filter back to original user requested start date
var dailyLst = dailyLstExtended
    .filterDate(reqStartDate, LSTEndDate.advance(1,
'day'));
```

37.2.4.3 Section 4.3: Summarize Daily LST by *Woreda*

In the final section for LST, we will perform a zonal mean of the temperature to our *woredas* and flatten in preparation for export as CSV. The exports (of all variables) will be all done in Sect. 37.2.7.

```
// Section 4.3: Summarize daily LST by woreda

// Filter LST data for zonal summaries.
var LSTSummary = dailyLst
    .filterDate(reqStartDate, reqEndDate.advance(1,
'day'));
// Function to calculate zonal statistics for LST by
woreda:
function sumZonalLst(image) {
    // To get the doy and year, we convert the metadata to
grids
    //  and then summarize.
    var image2 = image.addBands([
        image.metadata('doy').int(),
        image.metadata('year').int()
    ]);
     // Reduce by regions to get zonal means for each
 county.
     var output = image2
         .select(['doy', 'year', 'LST_day', 'LST_night',
 'LST_mean'])
         .reduceRegions({
             collection: woredas,
             reducer: ee.Reducer.mean(),
             scale: 1000
         });
     return output;
 }
 // Map the zonal statistics function over the filtered LST
 data.
 var LSTWoreda = LSTSummary.map(sumZonalLst);
 // Flatten the results for export.
 var LSTFlat = LSTWoreda.flatten();
```

Code Checkpoint A16d. The book’s repository contains a script that shows what your code should look like at this point.

37.2.5 Section 5: Spectral Index: NDWI

We will follow a similar pattern of steps for our spectral index, NDWI, as we did for precipitation and land surface temperatures: first, calculate the variable(s), then calculate the daily values, and finally summarize by *woreda*.

37.2.5.1 Section 5.1: Calculate NDWI

Here, we will focus on NDWI, which we actively used in forecasting malaria. For examples on other indices, see Chap. 9.

The MODIS MCD43A4 product contains simplified band quality information, and it is recommended to use the additional quality information in the MCD43A2 product for your particular application. We will join these two products to apply our selected quality information. (Note that we do not have to worry about snow in our study area.) For more information on joining image collections, see Chap. 21.

```
// Section 5: Spectral index NDWI

// Section 5.1: Calculate NDWI

// Filter BRDF-Adjusted Reflectance by date.
var brdfReflectVars = brdfReflect
    .filterDate(brdfStartDate, reqEndDate.advance(1,
'day'))
    .filterBounds(amhara)
    .select([
            'Nadir_Reflectance_Band1',
'Nadir_Reflectance_Band2',
            'Nadir_Reflectance_Band3',
'Nadir_Reflectance_Band4',
            'Nadir_Reflectance_Band5',
'Nadir_Reflectance_Band6',
            'Nadir_Reflectance_Band7'
        ],
        ['red', 'nir', 'blue', 'green', 'swir1', 'swir2',
'swir3']);
```

```
// Filter BRDF QA by date.
var brdfReflectQa = brdfQa
    .filterDate(brdfStartDate, reqEndDate.advance(1,
'day'))
    .filterBounds(amhara)
    .select([
            'BRDF_Albedo_Band_Quality_Band1',
            'BRDF_Albedo_Band_Quality_Band2',
            'BRDF_Albedo_Band_Quality_Band3',
            'BRDF_Albedo_Band_Quality_Band4',
            'BRDF_Albedo_Band_Quality_Band5',
            'BRDF_Albedo_Band_Quality_Band6',
            'BRDF_Albedo_Band_Quality_Band7',
            'BRDF_Albedo_LandWaterType'
        ],
        ['qa1', 'qa2', 'qa3', 'qa4', 'qa5', 'qa6', 'qa7',
'water']);

// Join the 2 collections.
var idJoin = ee.Filter.equals({
    leftField: 'system:time_end',
    rightField: 'system:time_end'
});
// Define the join.
var innerJoin = ee.Join.inner('NBAR', 'QA');
// Apply the join.
var brdfJoined = innerJoin.apply(brdfReflectVars,
brdfReflectQa,
    idJoin);

// Add QA bands to the NBAR collection.
function addQaBands(image) {
    var nbar = ee.Image(image.get('NBAR'));
    var qa = ee.Image(image.get('QA')).select(['qa2']);
    var water =
ee.Image(image.get('QA')).select(['water']);
    return nbar.addBands([qa, water]);
}
var brdfMerged =
ee.ImageCollection(brdfJoined.map(addQaBands));
```

```
// Function to mask out pixels based on QA and water/land
flags.
function filterBrdf(image) {
    // Using QA info for the NIR band.
    var qaband = image.select(['qa2']);
    var wband = image.select(['water']);
    var qamask = qaband.lte(2).and(wband.eq(1));
    var nir_r =
image.select('nir').multiply(0.0001).rename('nir_r');
    var swir2_r =
image.select('swir2').multiply(0.0001).rename(
        'swir2_r');
    return image.addBands(nir_r)
        .addBands(swir2_r)
        .updateMask(qamask);
}
var brdfFilteredVars = brdfMerged.map(filterBrdf);

// Function to calculate spectral indices:
function calcBrdfIndices(image) {
    var curyear =
ee.Date(image.get('system:time_start')).get('year');
    var curdoy = ee.Date(image.get('system:time_start'))
        .getRelative('day', 'year').add(1);
    var ndwi6 = image.normalizedDifference(['nir_r',
'swir2_r'])
        .rename('ndwi6');
    return image.addBands(ndwi6)
        .set('doy', curdoy)
        .set('year', curyear);
}
// Map function over image collection.
brdfFilteredVars = brdfFilteredVars.map(calcBrdfIndices);
```

37.2.5.2 Section 5.2: Calculate Daily NDWI

Similar to the other variables, we will calculate a daily value and filter to our user-requested dates, as data exist in that range.

```
// Section 5.2: Calculate daily NDWI

// Create list of dates for full time series.
var brdfRange = brdfFilteredVars.reduceColumns({
    reducer: ee.Reducer.max(),
    selectors: ['system:time_start']
});
var brdfEndDate = ee.Date(brdfRange.get('max'));
var brdfDays = brdfEndDate.difference(brdfStartDate,
'day');
var brdfDatesPrep = ee.List.sequence(0, brdfDays, 1);

function makeBrdfDates(n) {
    return brdfStartDate.advance(n, 'day');
}
var brdfDates = brdfDatesPrep.map(makeBrdfDates);

// List of dates that exist in BRDF data.
var brdfDatesExist = brdfFilteredVars
    .aggregate_array('system:time_start');

// Get daily brdf values.
function calcDailyBrdfExists(curdate) {
    curdate = ee.Date(curdate);
    var curyear = curdate.get('year');
    var curdoy = curdate.getRelative('day', 'year').add(1);
    var brdfTemp = brdfFilteredVars
        .filterDate(curdate, curdate.advance(1, 'day'));
    var outImg = brdfTemp.first();
    return outImg;
}
var dailyBrdfExtExists =
    ee.ImageCollection.fromImages(brdfDatesExist.map(
        calcDailyBrdfExists));
```

```
 // Create empty results, to fill in dates when BRDF data
 does not exist.
 function calcDailyBrdfFiller(curdate) {
     curdate = ee.Date(curdate);
     var curyear = curdate.get('year');
     var curdoy = curdate.getRelative('day', 'year').add(1);
     var brdfTemp = brdfFilteredVars
        .filterDate(curdate, curdate.advance(1, 'day'));
    var brdfSize = brdfTemp.size();
    var outImg = ee.Image.constant(0).selfMask()
        .addBands(ee.Image.constant(0).selfMask())
        .addBands(ee.Image.constant(0).selfMask())
        .addBands(ee.Image.constant(0).selfMask())
        .addBands(ee.Image.constant(0).selfMask())
        .rename(['ndvi', 'evi', 'savi', 'ndwi5', 'ndwi6'])
        .set('doy', curdoy)
        .set('year', curyear)
        .set('system:time_start', curdate)
        .set('brdfSize', brdfSize);
    return outImg;
}
// Create filler for all dates.
var dailyBrdfExtendedFiller =

ee.ImageCollection.fromImages(brdfDates.map(calcDailyBrdfF
iller));
// But only used if and when size was 0.
var dailyBrdfExtFillFilt = dailyBrdfExtendedFiller
    .filter(ee.Filter.eq('brdfSize', 0));
// Merge the two collections.
var dailyBrdfExtended = dailyBrdfExtExists
    .merge(dailyBrdfExtFillFilt);

// Filter back to original user requested start date.
var dailyBrdf = dailyBrdfExtended
    .filterDate(reqStartDate, brdfEndDate.advance(1,
'day'));
```

37.2.5.3 Section 5.3: Summarize Daily Spectral Indices by *Woreda*

Lastly, in our NDWI section, we will use the mean to summarize the values for each of the *woredas* and prepare for export by flattening the dataset. The exports (of all variables) will be all done in Sect. 37.2.7.

```
// Section 5.3: Summarize daily spectral indices by woreda

// Filter spectral indices for zonal summaries.
var brdfSummary = dailyBrdf
    .filterDate(reqStartDate, reqEndDate.advance(1,
'day'));

// Function to calculate zonal statistics for spectral
indices by woreda:

function sumZonalBrdf(image) {
    // To get the doy and year, we convert the metadata to
grids
    //  and then summarize.
    var image2 = image.addBands([
        image.metadata('doy').int(),
        image.metadata('year').int()
    ]);
    // Reduce by regions to get zonal means for each
woreda.
    var output = image2.select(['doy', 'year', 'ndwi6'])
        .reduceRegions({
            collection: woredas,
            reducer: ee.Reducer.mean(),
            scale: 1000
        });
    return output;
}

// Map the zonal statistics function over the filtered
spectral index data.
var brdfWoreda = brdfSummary.map(sumZonalBrdf);
// Flatten the results for export.
var brdfFlat = brdfWoreda.flatten();
```

Code Checkpoint A16e. The book's repository contains a script that shows what your code should look like at this point.

Question 3. Here, we are only calculating NDWI, which is calculated from the near-infrared (NIR) and shortwave infrared 2 (SWIR2) bands. If we wanted to calculate a vegetation index like the Normalized Difference Vegetation Index (NDVI), which bands would we need to add? Where in Sects. 37.2.5.1 through 37.2.5.3, would we need to add or select the raw bands and/or our new calculated band? Note: Fully implementing this is one of the synthesis challenges, so this is a good head start!

37.2.6 Section 6: Map Display

Here, we will take a look at our calculated variables but prior to zonal summary (Fig. 37.2). The full user interface restricts the date to display within the requested range, so be mindful in the code below which date you choose to view (we set our time range here in code Sect. 2.1, Sect. 37.2.2).

```
// Section 6: Map display of calculated environmental
variables
var displayDate = ee.Date('2021-10-01');

var precipDisp = dailyPrecip
    .filterDate(displayDate, displayDate.advance(1,
'day'));
var brdfDisp = dailyBrdf
    .filterDate(displayDate, displayDate.advance(1,
'day'));
var LSTDisp = dailyLst
    .filterDate(displayDate, displayDate.advance(1,
'day'));

// Select the image (should be only one) from each
collection.
var precipImage = precipDisp.first().select('totprec');
var LSTmImage = LSTDisp.first().select('LST_mean');
var ndwi6Image = brdfDisp.first().select('ndwi6');

// Palettes for environmental variable maps:
var palettePrecip = ['f7fbff', '08306b'];
var paletteLst = ['fff5f0', '67000d'];
var paletteSpectral = ['ffffe5', '004529'];
```

```
// Add layers to the map.
// Show precipitation by default,
//  others hidden until users picks them from layers drop
down.
Map.addLayer({
    eeObject: precipImage,
    visParams: {
        min: 0,
        max: 20,
        palette: palettePrecip
    },
    name: 'Precipitation',
    shown: true,
    opacity: 0.75
});
Map.addLayer({
    eeObject: LSTmImage,
    visParams: {
        min: 0,
        max: 40,
        palette: paletteLst
    },
    name: 'LST Mean',
    shown: false,
    opacity: 0.75
});
  Map.addLayer({
      eeObject: ndwi6Image,
      visParams: {
          min: 0,
          max: 1,
          palette: paletteSpectral
      },
      name: 'NDWI6',
      shown: false,
      opacity: 0.75
  });
```

Code Checkpoint A16f. The book's repository contains a script that shows what your code should look like at this point.

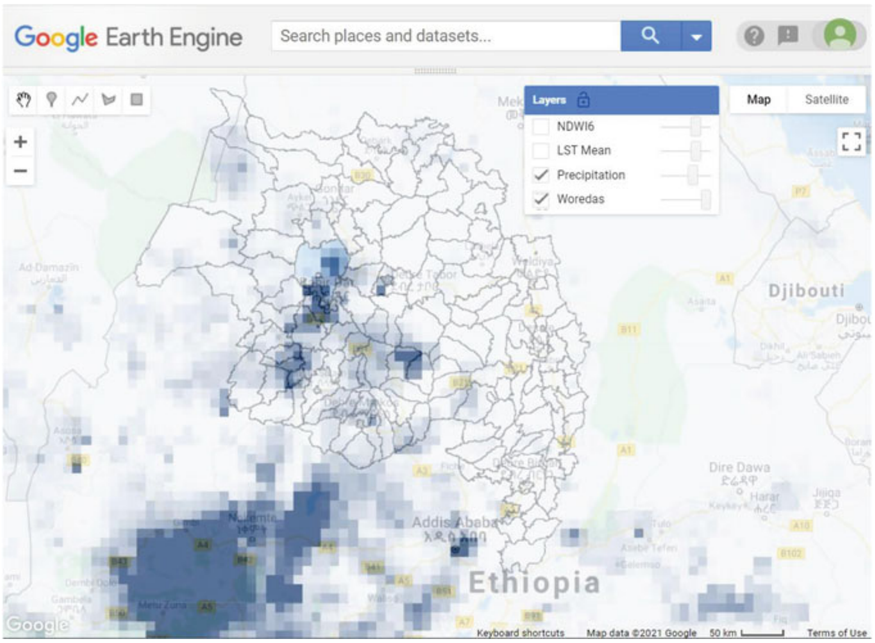
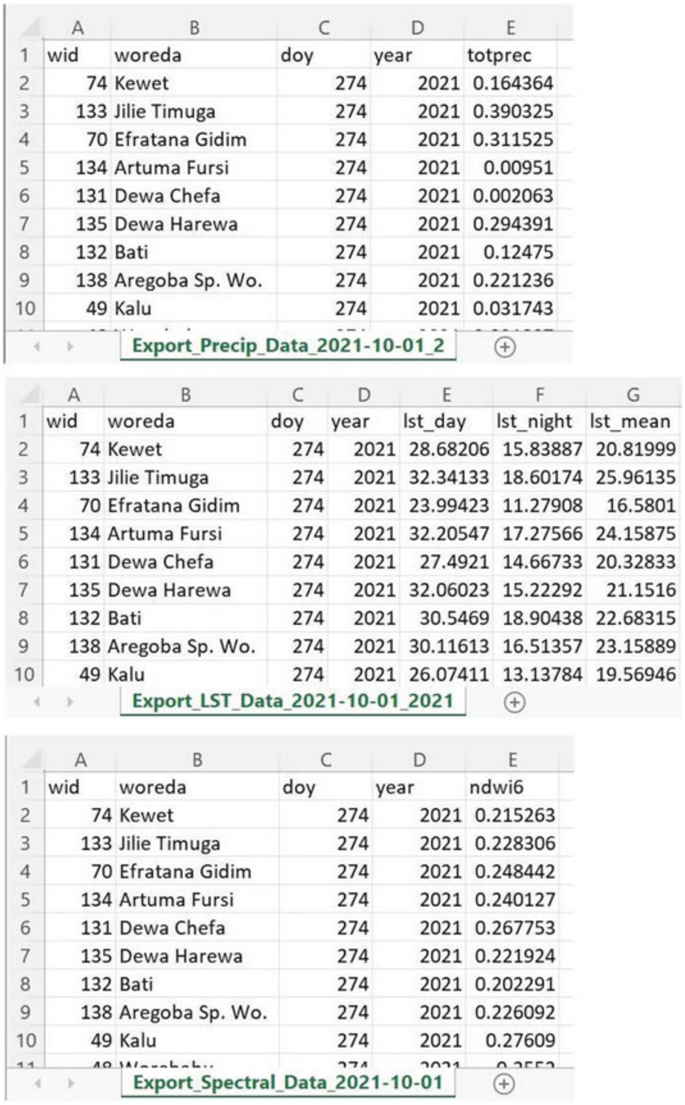


Fig. 37.2 Calculated total daily precipitation overlaid on *woreda* boundaries in the Amhara region of Ethiopia

37.2.7 Section 7: Exporting

Two important strengths of Google Earth Engine are the ability to gather and process the remotely sensed data all in the cloud and to have the only download be a small text file ready to use in the forecasting software. Most of our partners on this project were experts in public health and did not have a remote sensing or programming background. We also had partners in areas of limited or unreliable internet connectivity. We needed something that could be easily usable by our users in these types of situations.

In this section, we will create small text CSV downloads for each of our three environmental factors prepared earlier. Each factor may have different data availabilities within the user's requested range, and these dates will be added to the file name to indicate the actual date range of the downloaded data (Fig. 37.3).



wid	woreda	doy	year	totprec
74	Kewet	274	2021	0.164364
133	Jilie Timuga	274	2021	0.390325
70	Efratana Gidim	274	2021	0.311525
134	Artuma Fursi	274	2021	0.00951
131	Dewa Chefa	274	2021	0.002063
135	Dewa Harewa	274	2021	0.294391
132	Bati	274	2021	0.12475
138	Aregoba Sp. Wo.	274	2021	0.221236
49	Kalu	274	2021	0.031743

Export_Precip_Data_2021-10-01_2

wid	woreda	doy	year	lst_day	lst_night	lst_mean
74	Kewet	274	2021	28.68206	15.83887	20.81999
133	Jilie Timuga	274	2021	32.34133	18.60174	25.96135
70	Efratana Gidim	274	2021	23.99423	11.27908	16.5801
134	Artuma Fursi	274	2021	32.20547	17.27566	24.15875
131	Dewa Chefa	274	2021	27.4921	14.66733	20.32833
135	Dewa Harewa	274	2021	32.06023	15.22292	21.1516
132	Bati	274	2021	30.5469	18.90438	22.68315
138	Aregoba Sp. Wo.	274	2021	30.11613	16.51357	23.15889
49	Kalu	274	2021	26.07411	13.13784	19.56946

Export_LST_Data_2021-10-01_2021

wid	woreda	doy	year	ndwi6
74	Kewet	274	2021	0.215263
133	Jilie Timuga	274	2021	0.228306
70	Efratana Gidim	274	2021	0.248442
134	Artuma Fursi	274	2021	0.240127
131	Dewa Chefa	274	2021	0.267753
135	Dewa Harewa	274	2021	0.221924
132	Bati	274	2021	0.202291
138	Aregoba Sp. Wo.	274	2021	0.226092
49	Kalu	274	2021	0.27609

Export_Spectral_Data_2021-10-01

Fig. 37.3 Examples of the three CSV files returned from the script

```
 // Section 7: Exporting

 // 7.1 Export naming
 var reqStartDateText = reqStartDate.format('yyyy-MM-
 dd').getInfo();

 // Precipitation
 var precipPrefix = 'Export_Precip_Data';
 var precipLastDate = ee.Date(reqEndDate.millis()
     .min(precipEndDate.millis()));
 var precipSummaryEndDate = precipLastDate
     .format('yyyy-MM-dd').getInfo();
 var precipFilename = precipPrefix
     .concat('_', reqStartDateText,
         '_', precipSummaryEndDate);
 // LST
 var LSTPrefix = 'Export_LST_Data';
 var LSTLastDate = ee.Date(reqEndDate.millis()
     .min(LSTEndDate.millis()));
 var LSTSummaryEndDate = LSTLastDate
     .format('yyyy-MM-dd').getInfo();
 var LSTFilename = LSTPrefix
     .concat('_', reqStartDateText,
         '_', LSTSummaryEndDate);
// BRDF
var brdfPrefix = 'Export_Spectral_Data';
var brdfLastDate = ee.Date(reqEndDate.millis()
    .min(brdfEndDate.millis()));
var brdfSummaryEndDate = brdfLastDate
    .format('yyyy-MM-dd').getInfo();
var brdfFilename = brdfPrefix
    .concat('_', reqStartDateText,
        '_', brdfSummaryEndDate);

// 7.2 Export flattened tables to Google Drive
// Need to click 'RUN in the Tasks tab to configure and
start each export.
Export.table.toDrive({
    collection: precipFlat,
    description: precipFilename,
    selectors: ['wid', 'woreda', 'doy', 'year', 'totprec']
    });
Export.table.toDrive({
    collection: LSTFlat,
    description: LSTFilename,
    selectors: ['wid', 'woreda', 'doy', 'year',
        'LST_day', 'LST_night', 'LST_mean'
    ]
});
Export.table.toDrive({
    collection: brdfFlat,
    description: brdfFilename,
    selectors: ['wid', 'woreda', 'doy', 'year', 'ndwi6']
});
```

Code Checkpoint A16g. The book's repository contains a script that shows what your code should look like at this point.

In the Earth Engine **Tasks** tab, click **Run** to configure and start each export to Google Drive.

37.2.8 Section 8: Importing and Viewing External Analysis Results

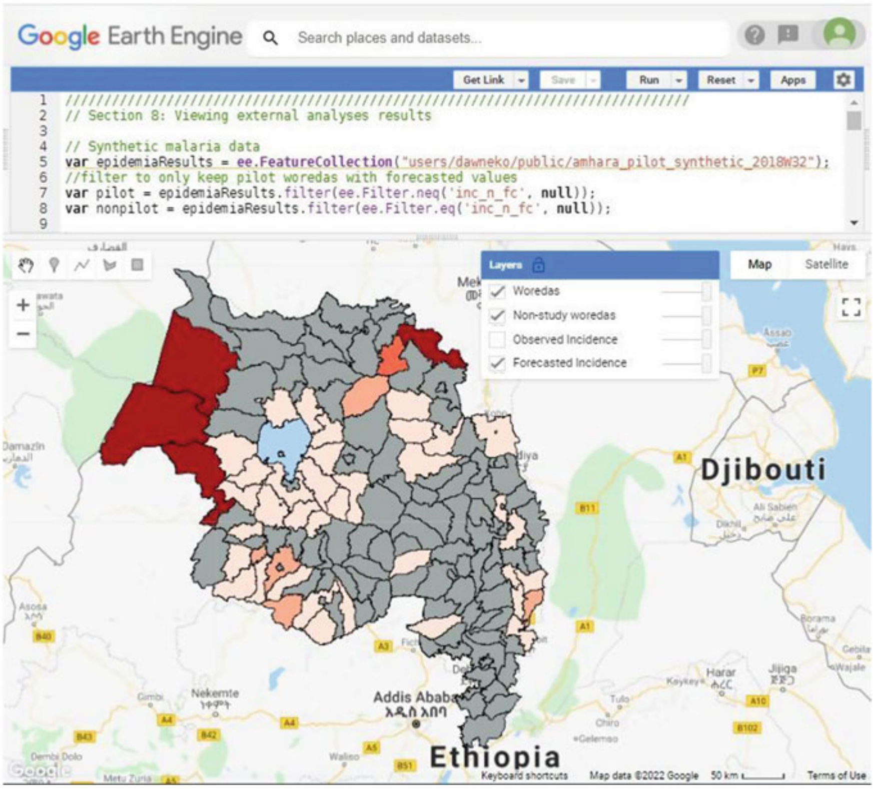
As mentioned at the start of the chapter, the environmental data obtained from Earth Engine can be used for infectious disease modeling and forecasting. The above Earth Engine code was written in support of EPIDEMIA, a software system based in the R language and computing environment for forecasting malaria, and was actively used in certain study pilot *woredas* in the Amhara region of Ethiopia. The R system consists of an R package—*epidemiar*—for generic functions and a companion R project for handling all the location-specific data and settings.

One of the main outputs of EPIDEMIA is the forecasted incidence of malaria in each *woreda* by week from one to eight (or more) weeks in advance. Using our publicly available demo project that uses synthetic data (not for use in epidemiological study), we created forecasts for week 32 of 2018 made eight weeks prior ("knowing" data up to week 24) and also added the observed incidence for comparison. (Note: dates and weeks follow International Organization for Standardization [ISO] standard 8601). These new data can be re-uploaded to Earth Engine for further analyses or exploration.

Starting a new script, you can use Sect. 37.2.8 code that follows to visualize the pre-generated demo 2018W32 results (Fig. 37.4).

```
// Section 8: Viewing external analyses results

// This is using *synthetic* malaria data.
// For demonstration only, not to be used for
epidemiological purposes.
var epidemiaResults = ee.FeatureCollection(
    'projects/gee-book/assets/A1-
6/amhara_pilot_synthetic_2018W32'
);
// Filter to only keep pilot woredas with forecasted
values.
var pilot = epidemiaResults
    .filter(ee.Filter.neq('inc_n_fc', null));
var nonpilot = epidemiaResults
    .filter(ee.Filter.eq('inc_n_fc', null));

Map.setCenter(38, 11.5, 7);

// Paint the pilot woredas with different colors for
forecasted* incidence
// fc_n_inc here is the forecasted incidence (cut into
factors)
// made on (historical) 2018W24 (i.e. 8 weeks in advance).
// * based on synthetic data for demonstration only.
// Incidence per 1000
// 1 : [0 - 0.25)
// 2 : [0.25 - 0.5)
// 3 : [0.5 - 0.75)
// 4 : [0.75 - 1)
// 5 : > 1

var empty = ee.Image().byte();
var fill_fc = empty.paint({
    featureCollection: pilot,
    color: 'inc_n_fc',
});
var palette = ['fee5d9', 'fcae91', 'fb6a4a', 'de2d26',
'a50f15'];
Map.addLayer(
    fill_fc, {
        palette: palette,
        min: 1,
        max: 5
    },
        'Forecasted Incidence'
 );

 // Paint the woredas with different colors for the
 observed* incidence.
 // * based on synthetic data for demonstration only

var fill_obs = empty.paint({
    featureCollection: pilot,
    color: 'inc_n_obs',
});
var palette = ['fee5d9', 'fcae91', 'fb6a4a', 'de2d26',
'a50f15'];
// Layer is off by default, users change between the two
in the map viewer.
Map.addLayer(
    fill_obs, {
        palette: palette,

        min: 1,
        max: 5
    },
    'Observed Incidence',
    false
);

// Add gray fill for nonpilot woredas (not included in
study).
var fill_na = empty.paint({
    featureCollection: nonpilot
});
Map.addLayer(
    fill_na, {
        palette: 'a1a9a8'
    },
    'Non-study woredas'
);

// Draw borders for ALL Amhara region woredas.
var outline = empty.paint({
    featureCollection: epidemiaResults,
    color: 1,
    width: 1
});
// Add woreda boundaries to map.
Map.addLayer(
    outline, {
        palette: '000000'
    },
    'Woredas'
);
```

Code Checkpoint A16h. The book's repository contains a script that shows what your code should look like at this point.



```
////////////////////////////////////////////////////////////////////////////////
// Section 8: Viewing external analyses results

// Synthetic malaria data
var epidemiaResults = ee.FeatureCollection("users/dawneko/public/amhara_pilot_synthetic_2018W32");
//filter to only keep pilot woredas with forecasted values
var pilot = epidemiaResults.filter(ee.Filter.neq('inc_n_fc', null));
var nonpilot = epidemiaResults.filter(ee.Filter.eq('inc_n_fc', null));

```

Fig. 37.4 Visualization of forecasted malaria incidence for week 32 of 2018 made during week 24 (an eight-week lead time). Malaria data are synthetic, for demonstration purposes only. The incidence has been categorized into five categories (from lighter to dark red): 0–0.25, 0.25–0.5, 0.5–0.75, 0.75–1, and greater than 1. Only *woredas* in the pilot project have values; the rest of the Amhara region is marked in gray fill. Another layer available to view is the observed (synthetic) incidence rate for 2018W32

37.3 Synthesis

Assignment 1. Calculate other spectral indices: In this chapter, we only calculate and export the NDWI from the spectral data. Calculate another index, such as a vegetation index like NDVI, Soil Adjusted Vegetation Index (SAVI), or Enhanced Vegetation Index (EVI) to the calculations. Think about what bands you will need, how to calculate the index, and how to propagate the band through all the remaining processing steps (including exporting).

Assignment 2. Change location: In this chapter, we obtained data for *woredas* in the Amhara region of Ethiopia. Upload or import a new shapefile of different locations and acquire environmental data for there instead. Remember that you will need to adjust any references to asset-specific fields (as we did here for "*woreda*"). See Chap. 22 for help with uploading assets, if needed.

37.4 Conclusion

In this chapter, we saw how Earth Engine can be used to acquire environmental data to support external analyses, such as forecasting of malaria, a vector-borne disease. An understanding of the biology of the vector (e.g., mosquito, tick) and how different environmental conditions can affect the disease system and transmission risk will help identify environmental variables to investigate for use in mathematical modeling.

In this chapter, we obtained data from three different satellite-based datasets: rainfall from IMERG/GPM, land surface temperature 8-day composite values from MODIS, and the calculation of spectral indices from MODIS bands. We saw how to perform zonal summaries to our location of interest and download CSV files that are suitable for import into other programs for additional analyses.

This chapter shows the value of cloud computation and generates small downloads for use by professionals who may not have expertise in remote sensing or the computing resources that would otherwise be needed. Finally, we saw that the results of intermediate processing and work outside of Earth Engine can be re-imported for additional analyses within Earth Engine.

References

Ford TE, Colwell RR, Rose JB et al (2009) Using satellite images of environmental changes to predict infectious disease outbreaks. Emerg Infect Dis 15:1341–1346. https://doi.org/10.3201/eid/1509.081334

Franklinos LHV, Jones KE, Redding DW, Abubakar I (2019) The effect of global change on mosquito-borne disease. Lancet Infect Dis 19:e302–e312. https://doi.org/10.1016/S1473-3099(19)30161-6

Jones KE, Patel NG, Levy MA et al (2008) Global trends in emerging infectious diseases. Nature 451:990–993. https://doi.org/10.1038/nature06536

Mackenzie JS, Jeggo M (2019) The one health approach—why is it so important? Trop Med Infect Dis 4:88. https://doi.org/10.3390/tropicalmed4020088

Wimberly MC, de Beurs KM, Loboda TV, Pan WK (2021) Satellite observations and malaria: new opportunities for research and applications. Trends Parasitol 37:525–537. https://doi.org/10.1016/j.pt.2021.03.003

Wimberly MC, Nekorchuk DM, Kankanala RR (2022) Cloud-based applications for accessing satellite Earth observations to support malaria early warning. Sci Data 9:1–11. https://doi.org/10.1038/s41597-022-01337-y

World Health Organization (2018) Malaria surveillance, monitoring and evaluation: a reference manual. World Health Organization

World Health Organization (2020) World Malaria report 2020: 20 years of global progress and challenges. World Health Organization

Humanitarian Applications

38

Jamon Van Den Hoek and Hannah K. Friedrich

Overview

The global refugee population has never been as large as it is today, with at least 26 million refugees living in more than 100 countries. Refugees are international migrants who have been forcibly displaced from their home countries due to violence or persecution and who cross an international border and settle elsewhere, most often in a neighboring country. Remote sensing can help refugee leaders, humanitarian agencies, and refugee-hosting countries gain new insights into refugee settlement, population, and land cover change dynamics (Maystadt et al. 2020; Van Den Hoek et al. 2021). In this chapter, we will examine the value of using satellite imagery and satellite-derived data to map a refugee settlement in Uganda, estimate its population, and gauge land cover changes in and around the settlement.

Learning Outcomes

- Using a range of techniques—maps, videos, and charts—to visualize and measure land cover changes before and after the establishment of a refugee settlement.
- Understanding the considerations and limitations involved in automated detection of refugee settlement boundaries using unsupervised classification.
- Becoming familiar with satellite-derived human settlement and population datasets and their application in a refugee settlement context.

J. Van Den Hoek (✉)
Geography and Geospatial Science, College of Earth, Ocean, and Atmospheric Sciences, Oregon State University, Strand Agriculture Hall 347, 170 SW Waldo Place, Corvallis, OR 97331, USA
e-mail: vandenhj@oregonstate.edu

H. K. Friedrich
School of Geography, Development and Environment, University of Arizona, ENR2 Building, South 4th Floor, 1064 E. Lowell Street, Tucson, AZ 85721, USA

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_38

Helps if you know how to

- Import images and image collections, filter, and visualize (Part 1).
- Perform basic image analysis: select bands, compute indices, create masks, classify images (Part 2).
- Create a graph using `ui.Chart` (Chap. 4).
- Use `normalizedDifference` to calculate vegetation indices (Chap. 5).
- Perform pixel-based supervised or unsupervised classification (Chap. 6).
- Use `ee.Reducer` functions to summarize pixels over an area (Chaps. 8 and 9).
- Perform image morphological operations (Chap. 10).
- Write a function and `map` it over an `ImageCollection` (Chap. 12).
- Use `reduceRegions` to summarize an image with zonal statistics in irregular shapes (Chaps. 22 and 24).
- Convert from a vector to a raster representation with `reduceToImage` (Chap. 23).
- Write a function and `map` it over a `FeatureCollection` (Chaps. 23 and 24).

38.1 Introduction to Theory

In a humanitarian context, remote sensing data and analysis have become essential tools for monitoring refugee settlement dynamics both immediately after refugee arrival and over the long term. Nonetheless, there remain important challenges to characterizing refugee settlement conditions. First, dwellings, roadways, and agricultural plots tend to be small in size within refugee settlements and generally difficult to detect without the use of very high resolution satellite imagery. Second, dwellings and other structures within refugee settlements may be diffusely distributed and intermixed with vegetation and bare earth. Third, data on settlement features, boundaries, and refugee populations are often out of date or otherwise inappropriate for detailed geospatial analysis. In this chapter, we will examine these challenges and do our best to document refugee settlement dynamics through analysis of multi-date Landsat imagery.

38.2 Practicum

The study area for this chapter is Pagirinya Refugee Settlement in northwestern Uganda (Fig. 38.1a). As of 2020, Uganda was home to 1.4 million refugees, the fourth-largest refugee population in the world and the largest in Africa (UNHCR 2020). Refugees living in Uganda primarily fled violence in South Sudan and the Democratic Republic of the Congo, and most live in rural refugee settlements. Pagirinya in particular is home to 36,000 South Sudanese refugees and was established in mid-2016.

In this practicum, we will visualize and document the land cover changes that have taken place in Pagirinya (Fig. 38.1b, c), use satellite data to estimate the settlement's boundary and compare it to the official boundary laid out by the United Nations High Commissioner for Refugees (UNHCR), and use satellite-derived demographic products to estimate the refugee population within Pagirinya.

38.2.1 Section 1: Seeing Refugee Settlements from Above

In preparation for the arrival of refugees, humanitarian actors and refugee settlement planners are often interested in analyzing local land cover conditions before a refugee settlement is established. The goal of this first section is to use Landsat satellite imagery to characterize initial land cover conditions and land cover changes at Pagirinya Refugee Settlement in the years before and following the settlement's establishment in 2016.

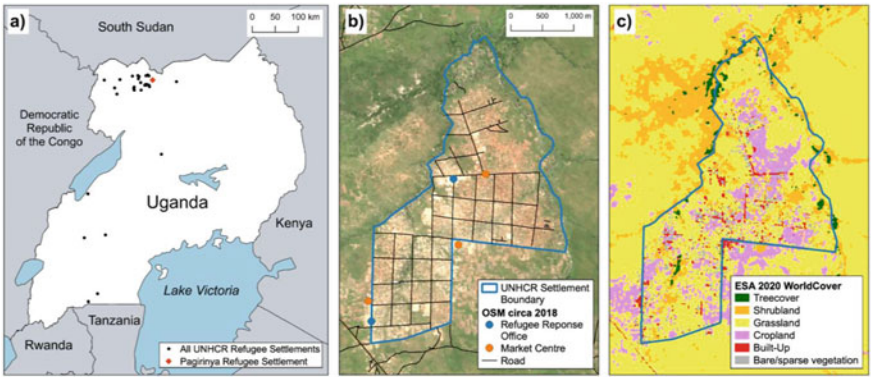


Fig. 38.1 Maps of **a** UNHCR refugee settlements in Uganda, **b** OpenStreetMap features, roadways, and the UNHCR settlement boundary for Pagirinya, and **c** European Space Agency 2020 WorldCover land cover at Pagirinya

Let's begin by adding the refugee settlement's boundary to the **Map** by loading the `FeatureCollection` of refugee settlement boundaries in Uganda and filtering to Pagirinya Refugee Settlement. We will also initialize the **Map** to center on Pagirinya and default to showing the satellite basemap for visual reference.

```
Map.setOptions('SATELLITE');

// Load UNHCR settlement boundary for Pagirinya Refugee
Settlement.
var pagirinya = ee.Feature(ee.FeatureCollection(
    'projects/gee-book/assets/A1-
7/pagirinya_settlement_boundary'
).first());

Map.addLayer(pagirinya, {}, 'Pagirinya Refugee
Settlement');
Map.centerObject(pagirinya, 14);
```

Next, let's create annual Landsat composites using the Landsat 8 surface reflectance `ImageCollection`. We will spatially filter the `ImageCollection` to a buffered settlement boundary and temporally filter to 2015–2020, which includes the full year before the settlement was established and the four years that followed. We will also apply a cloud filter of less than or equal to 40% to help ensure that our annual composites are cloud free.

For better legibility, we will rename the Landsat bands and add three new spectral index bands to each image in the `ImageCollection` using the `addIndices` function, which calculates the Normalized Difference Vegetation Index (NDVI), Normalized Difference Building Index (NDBI), and Normalized Burn Ratio (NBR) using `normalizedDifference`. Each of these metrics offers a different approach to characterizing land cover conditions and change over time. NDVI is commonly used for monitoring vegetation health; NDBI helps to characterize impervious and built-up surfaces; and NBR helps to identify land that has been cleared with fire, a common practice in our study region. Note that other spectral metrics or remote sensing platforms may be better suited for identifying refugee settlements in other regions.

```
// Create buffered settlement boundary geometry.
// 500 meter buffer size is arbitrary but large enough
// to capture area outside of the study settlement.
var bufferSize = 500; // (in meters)

// Buffer and convert to Geometry for spatial filtering and
clipping.
var bufferedBounds = pagirinya.buffer(bufferSize)
     .bounds().geometry();

function addIndices(img) {
     var ndvi = img.normalizedDifference(['nir', 'red'])
          .rename('NDVI'); // NDVI = (nir-red)/(nir+red)
     var ndbi = img.normalizedDifference(['swir1', 'nir'])
          .rename(['NDBI']); // NDBI = (swir1-
nir)/(swir1+nir)
     var nbr = img.normalizedDifference(['nir', 'swir2'])
          .rename(['NBR']); // NBR = (nir-swir2)/(nir+swir2)
     var imgIndices =
img.addBands(ndvi).addBands(ndbi).addBands(nbr);
     return imgIndices;
}

// Create L8 SR Collection 2 band names and new names.
var landsat8BandNames = ['SR_B2', 'SR_B3', 'SR_B4',
'SR_B5', 'SR_B6', 'SR_B7'];

var landsat8BandRename = ['blue', 'green', 'red', 'nir',
'swir1', 'swir2'];

// Create image collection.
var landsat8Sr =
ee.ImageCollection('LANDSAT/LC08/C02/T1_L2');
var ic = ee.ImageCollection(landsat8Sr.filterDate('2015-01-
01', '2020-12-31')
     .filterBounds(bufferedBounds)
     .filter(ee.Filter.lt('CLOUD_COVER', 40))
     .select(landsat8BandNames, landsat8BandRename)
     .map(addIndices));
```

To build annual composites from before and after Pagirinya's establishment in 2016, let's create two temporal subsets of the `ImageCollection`—one from 2015 and one from 2017—and use the `median` function to composite images for each time frame (Fig. 38.2). We will also clip our image collections to the buffered region around Pagirinya. To visualize the NDVI composites, we will use true-color

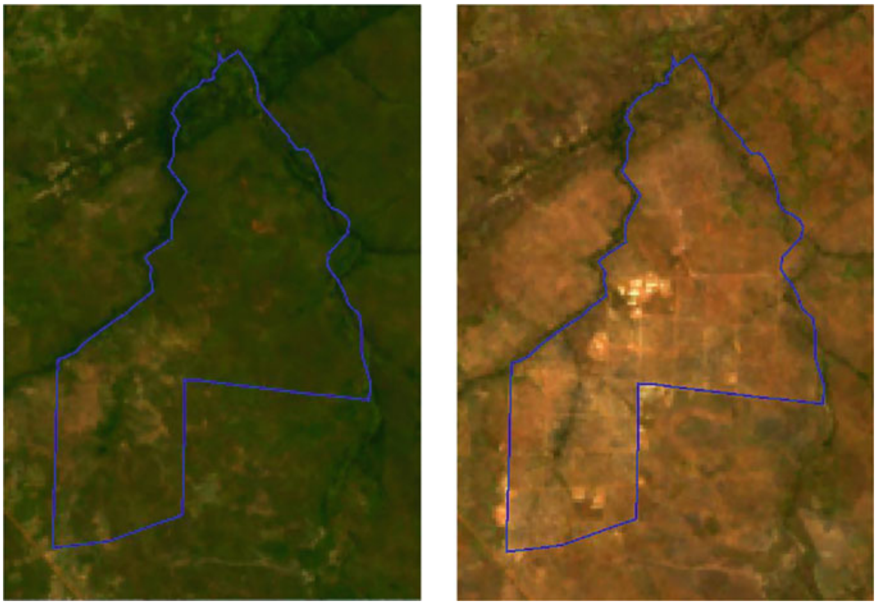


Fig. 38.2 Pre-establishment (left) and post-establishment (right) true-color composites with Pagirinya Refugee Settlement boundary overlaid in blue

and false-color visualizations and color palettes, which should help us identify and interpret features within and surrounding the settlement boundary.

```
// Make annual pre- and post-establishment composites.
var preMedian = ic.filterDate('2015-01-01', '2015-12-
31').median()
    .clip(bufferedBounds);
var postMedian = ic.filterDate('2017-01-01', '2017-12-
31').median()
    .clip(bufferedBounds);

// Import visualization palettes https://github.com/gee-
community/ee-palettes.
var palettes = require('users/gena/packages:palettes');
var greenPalette = palettes.colorbrewer.Greens[9];
var prGreenPalette = palettes.colorbrewer.PRGn[9];
```

```
// Set-up "true color" visualization parameters.
var TCImageVisParam = {
    bands: ['red', 'green', 'blue'],
    gamma: 1,
    max: 13600,
    min: 8400,
    opacity: 1
};

// Set-up "false color" visualization parameters.
var FCImageVisParam = {
    bands: ['nir', 'red', 'green'],
    gamma: 1,
    min: 9000,
    max: 20000,
    opacity: 1
};

// Display true-color composites.
Map.addLayer(preMedian, TCImageVisParam,
    'Pre-Establishment Median TC');
Map.addLayer(postMedian, TCImageVisParam,
    'Post-Establishment Median TC');

// Display false-color composites.
Map.addLayer(preMedian, FCImageVisParam,
    'Pre-Establishment Median FC');
Map.addLayer(postMedian, FCImageVisParam,
    'Post-Establishment Median FC');

// Display median NDVI composites.
Map.addLayer(preMedian, {
    min: 0,
    max: 0.7,
    bands: ['NDVI'],
    palette: greenPalette
}, 'Pre-Establishment Median NDVI');
```

```
Map.addLayer(postMedian, {
    min: 0,
    max: 0.7,
    bands: ['NDVI'],
    palette: greenPalette
}, 'Post-Establishment Median NDVI');

// Create an empty byte Image into which we'll paint the
settlement boundary.
var empty = ee.Image().byte();

// Convert settlement boundary's geometry to an Image for
overlay.
var pagirinyaOutline = empty.paint({
    featureCollection: pagirinya,
    color: 1,
    width: 2
});

// Display Pagirinya boundary in blue.
Map.addLayer(pagirinyaOutline,
    {
        palette: '0000FF'
    },
    'Pagirinya Refugee Settlement boundary');
```

Now that we have pre- and post-establishment composites to support a visual qualitative assessment, let’s make a complementary quantitative assessment by measuring pre- and post-establishment differences in median NDVI and plotting the distribution of NDVI from both periods.

```
// Compare pre- and post-establishment differences in NDVI.
var diffMedian = postMedian.subtract(preMedian);
Map.addLayer(diffMedian,
    {
        min: -0.1,
        max: 0.1,
        bands: ['NDVI'],
        palette: prGreenPalette
    },
    'Difference Median NDVI');
```

```
// Chart the NDVI distributions for pre- and post-
establishment.
var combinedNDVI = preMedian.select(['NDVI'], ['pre-NDVI'])
    .addBands(postMedian.select(['NDVI'], ['post-NDVI']));

var prePostNDVIFrequencyChart =
    ui.Chart.image.histogram({
        image: combinedNDVI,
        region: bufferedBounds,
        scale: 30

    }).setSeriesNames(['Pre-Establishment', 'Post-
Establishment'])
    .setOptions({
        title: 'NDVI Frequency Histogram',
        hAxis: {
            title: 'NDVI',
            titleTextStyle: {
                italic: false,
                bold: true
            },
        },
        vAxis:
        {
            title: 'Count',
            titleTextStyle: {
                italic: false,
                bold: true
            }
        },
        colors: ['cf513e', '1d6b99']
    });
print(prePostNDVIFrequencyChart);
```

In addition to the pre- and post-establishment annual composites, let's create an annotated video time series of the full 2015–2020 Landsat 8 surface reflectance `ImageCollection`. We will be able to use this video to view changes at our study refugee settlement image by image.

```
// Import package to support text annotation.
var text = require('users/gena/packages:text');
var rgbVisParam = {
    bands: ['red', 'green', 'blue'],
    gamma: 1,
    max: 12011,
    min: 8114,
    opacity: 1
};

// Define arguments for animation function parameters.
var videoArgs = {
    region: bufferedBounds,
    framesPerSecond: 3,
    scale: 10
};
var annotations = [{
    position: 'left',
    offset: '5%',
    margin: '5%',
    property: 'label',
    scale: 30
}];

function addText(image) {
    var date =
ee.String(ee.Date(image.get('system:time_start'))
        .format('YYYY-MM-dd'));
    // Set a property called label for each image.
    var image =
image.clip(bufferedBounds).visualize(rgbVisParam)
        .set({
            'label': date
        });
    // Create a new image with the label overlaid using
gena's package.
    var annotated = text.annotateImage(image, {},
bufferedBounds, annotations);
    return annotated;
}
```

```
// Add timestamp annotation to all images in video.
var tempCol = ic.map(addText);

// Click the URL to watch the time series video.
print('L8 Time Series Video',
tempCol.getVideoThumbURL(videoArgs));
```

Code Checkpoint A17a. The book's repository contains a script that shows what your code should look like at this point.

Question 1. How would you describe the land cover type in the area in 2015, before the establishment of the refugee settlement? Is the land cover consistent within the settlement's boundary in the pre-establishment period? Does the settlement boundary conform to land cover type or condition in any meaningful way?

Question 2. What features (dwellings, roadways, agricultural plots, etc.) present the greatest visual difference between the pre- and post-establishment periods? Comparing the visual differences in true color, false color, and NDVI with the satellite image basemap may be helpful here.

Question 3. Which of the annual composite visualizations (true color, false color, or NDVI) do you prefer for distinguishing the refugee settlement in the post-establishment period, and why?

Question 4. How do the range and mode of NDVI values change from pre- to post-establishment? How might the changes in NDVI distribution correlate to overall changes in land cover type in the post-establishment period?

Question 5. Beyond the rapid establishment of the settlement's dwellings and roads, what changes do you observe in the time series video? Do these changes occur within or outside the settlement boundary? What kinds of changes do you see in the imagery from 2019 or 2020, well after the settlement was established in 2016?

38.2.2 Section 2: Mapping Features Within the Refugee Settlement

In Sect. 38.2.1, we used Landsat data to gauge changes in land cover conditions and types, but we can also draw upon data products derived from satellite imagery. For instance, satellite-derived building footprints, which represent geometries of individual structures and dwellings, are often used to estimate human populations and population density in humanitarian contexts and to support the planning and delivery of food and other kinds of aid. In this section, we will identify different features within Pagirinya, which we will use to create a satellite image-based settlement boundary map in Sect. 38.2.3.

Let's add to our script from Sect. 38.2.1 by first loading the Open Buildings V1 Polygons dataset from the Earth Engine Data Catalog. This dataset includes satellite-derived building footprints based on very high resolution (0.5 m) satellite imagery, and each footprint has a confidence score. Let's visualize building footprints with a confidence score above 75% as orange and building footprints with a 75% or lower confidence score as purple.

```
// Visualize Open Buildings dataset.
var footprints = ee.FeatureCollection(
    'GOOGLE/Research/open-buildings/v1/polygons');
var footprintsHigh = footprints.filter('confidence > 0.75');
var footprintsLow = footprints.filter('confidence <= 0.75');

Map.addLayer(footprintsHigh, {
    color: 'FFA500'
}, 'Buildings high confidence');
Map.addLayer(footprintsLow, {
    color: '800080'
}, 'Buildings low confidence');
```

With a map of building footprints in place, let's turn to examining other features of interest that we identified in Sect. 38.2.1. Let's load a `FeatureCollection` of sample locations of infrastructure, forest, and agriculture visible on the satellite basemap as well as a sample of building footprint locations. Note in the `print` output that each feature has a `value`, which represents the feature type. Let's write a function to use this `value` property to automatically assign a unique color to each feature as part of a style (Fig. 38.3).

```
// Load land cover samples.
var lcPts = ee.FeatureCollection(
    'projects/gee-book/assets/A1-7/lcPts');
print('lcPts', lcPts);

// Create a function to set Feature properties based on
value.
var setColor = function(f) {
    var value = f.get('class');
    var mapDisplayColors = ee.List(['#13a1ed', '#7d02bf',
        '#f0940a', '#d60909'
    ]);
    // Use the class as an index to lookup the
corresponding display color.
    return f.set({
        style: {
            color: mapDisplayColors.get(value)
        }
    });
};

// Apply the function and view the results.
var styled = lcPts.map(setColor);
Map.addLayer(styled.style({
    styleProperty: 'style'
}), {}, 'Land cover samples');
```

Since we want to use these sample land cover locations to help delineate the refugee settlement boundary, these different land cover types should be spectrally distinguishable from each other. To see how the spectral values vary among different features, let's create spectral signature plots for the post-establishment period. We first need to add the land cover class to the post-establishment composites that we made in Sect. 38.2.1 so that the class and spectral value information can be referenced together in our spectral signature plots. To do that, let's use `reduceToImage` to convert our `lcPts` `FeatureCollection` to an image, `lcBand`, and then add that image to the post-establishment composite.

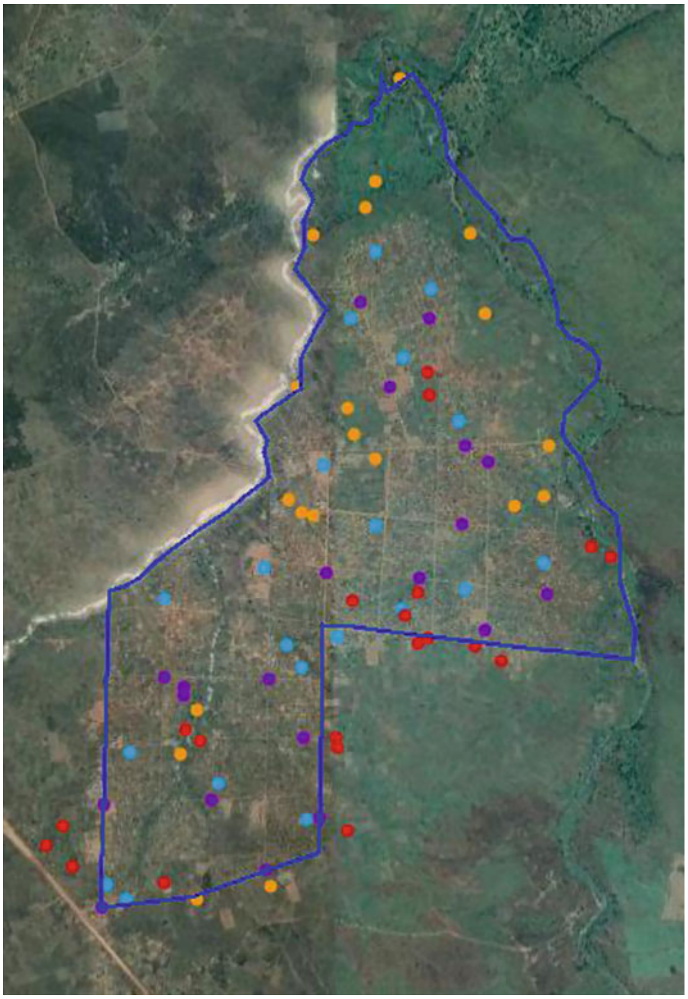


Fig. 38.3 Feature samples across Pagirinya Refugee Settlement (boundary shown in blue)

```
// Convert land cover sample FeatureCollection to an Image.
var lcBand = lcPts.reduceToImage({
    properties: ['class'],
    reducer: ee.Reducer.first()
}).rename('class');

// Add lcBand to the post-establishment composite.
postMedian = postMedian.addBands(lcBand);
```

Now we have a postMedian image that we can sample at specific sample locations and identify not only the spectral values but also the class type. Let's plot the spectral values by class type. Note that since band names are sorted alphabetically on the x-axis, nir values are plotted in between green and red and are therefore out of order with respect to band wavelengths.

```
// Define bands that will be visualized in chart.
var chartBands = ['blue', 'green', 'red', 'nir', 'swir1',
'swir2', 'class'
];

print(postMedian, 'postMedian');

// Plot median band value for each land cover type.
var postBandsChart = ui.Chart.image
    .byClass({
        image: postMedian.select(chartBands),
        classBand: 'class',
        region: lcPts,
        reducer: ee.Reducer.median(),
        scale: 30,
        classLabels: ['Settlement', 'Road', 'Forest',
            'Agriculture'
        ],
        xLabels: chartBands
    })
 .setChartType('ScatterChart')
 .setOptions({
    title: 'Band Values',
    hAxis: {
        title: 'Band Name',
        titleTextStyle: {
            italic: false,
            bold: true
        },
    },
    vAxis: {
        title: 'Reflectance (x1e4)',
        titleTextStyle: {
            italic: false,
            bold: true
        }
    },
        colors: ['#13a1ed', '#7d02 bf', '#f0940a', '#d60909'],
        pointSize: 0,
        lineSize: 5,
        curveType: 'function'
    });
print(postBandsChart);
```

Remember that we also calculated NDVI, NDBI, and NBR spectral indices in Sect. 38.2.1. Since these bands range from − 1 to 1, we have to plot their values separately from the Landsat band spectral signature plots above, which use scaled reflectance values.

```
// Define spectral indices that will be visualized in the
chart.
var indexBands = ['NDVI', 'NDBI', 'NBR', 'class'];

// Plot median index value for each land cover type.
var postIndicesChart = ui.Chart.image
    .byClass({
        image: postMedian.select(indexBands),
        classBand: 'class',
        region: lcPts,
        reducer: ee.Reducer.median(),
        scale: 30,
        classLabels: ['Settlement', 'Road', 'Forest',
            'Agriculture'
        ],
        xLabels: indexBands
    })
   .setChartType('ScatterChart')
   .setOptions({
       title: 'Index Values',
       hAxis: {
           title: 'Index Name',
           titleTextStyle: {
               italic: false,
               bold: true
           },
           //viewWindow: {min: wavelengths[0], max:
wavelengths[2]}
           scaleType: 'string'
       },
        vAxis: {
            title: 'Value',
            titleTextStyle: {
                italic: false,
                bold: true
            }
        },
        colors: ['#13a1ed', '#7d02bf', '#f0940a',
'#d60909'],
        pointSize: 5
    });
print(postIndicesChart);

// Create an empty image into which to paint the features,
cast to byte.
var empty = ee.Image().byte();

// Paint all the polygon edges with the same number and
width, display.
var pagirinyaOutline = empty.paint({
    featureCollection: pagirinya,
    color: 1,
    width: 2
});

// Map outline of Pagirinya in blue.
Map.addLayer(pagirinyaOutline,
    {
        palette: '0000FF'
    },
    'Pagirinya Refugee Settlement boundary');
```

Code Checkpoint A17b. The book's repository contains a script that shows what your code should look like at this point.

Question 6. How would you describe the coverage of the footprints within the settlement? Are there sections of the settlement visible in the basemap or the post-establishment composite that are missing footprints?

Question 7. How do NDVI, NDBI, and NBR change from the pre- to post-establishment period at building footprint locations?

Question 8. Are the spectral profiles of the four feature types distinct from each other? Which profiles are the most similar overall?

Question 9. Which bands or indices provide the greatest separation between the four feature types?

38.2.3 Section 3: Delineating Refugee Settlement Boundaries

Now that we have become familiar with the different land cover types and the changes that can occur once a refugee settlement is established, let's turn to formally delineating the refugee settlement from its surroundings by mapping a settlement boundary. Having information on refugee settlement boundaries is helpful for the basic accounting of refugee settlement extent and for confidently attributing land cover or land use changes to a specific refugee settlement (Friedrich and Van Den Hoek 2020; Van Den Hoek and Friedrich 2021). In this section, we will use a *k*-means unsupervised classifier to generate a settlement/non-settlement map that represents land that has been transformed by the refugee settlement's establishment or subsequent use. Note that the settlement boundary that we used in Sect. 38.2.1 is a settlement planning boundary established by the UNHCR and so represents the land within the formal boundary that *potentially* could be accessed or used by refugees.

To start making a binary classification that separates settlement from non-settlement, let's create a random sample of 500 NDVI values from across the post-establishment composite. Remember that the `postMedian` composite was clipped to the 500-m-buffered extent of the UNHCR settlement boundary geometry, so these sample sites should be dispersed inside and outside of the UNHCR boundary's geometry. For parameterization, we only need two values output from the classifier (`numClusters` = 2) and can set the maximum number of iterations to a low value of 5 (`maxIter` = 5) and the seed value to an arbitrary value of 21. Now let's apply the classifier to the post-establishment composite, view the coverage of settlement (pixel value of 1) and non-settlement (pixel value of 0), and visually compare the result with the UNHCR settlement boundary.

```
// Create samples to input to a K-means classifier.
var numPx = 500;
var samples = postMedian.select('NDVI').sample({
    scale: 30, // Landsat resolution
    numPixels: numPx,
    geometries: true
});

Map.addLayer(samples, {}, 'K-means samples');

// Set-up the parameters for K-means.
var numClusters = 2;
var maxIter = 5;
var seedValue = 21;
```

```
// Seed the classifier using land cover samples.
var clusterer = ee.Clusterer.wekaKMeans({
    nClusters: numClusters,
    maxIterations: maxIter,
    seed: seedValue
}).train(samples);

// Apply the K-means classifier.
var kmeansResult = postMedian.cluster(clusterer);
Map.addLayer(kmeansResult, {
    bands: ['cluster'],
    max: 1,
    min: 0
}, 'K-means output');
```

The resulting *k*-means classification looks promising for separating settlement from non-settlement pixels, but it has many gaps in settlement coverage as well as isolated settlement patches and pixels. To produce a single contiguous settlement coverage, let's apply spatial morphological operations of dilation and erosion on the *k*-means output. Dilation incrementally expands the boundary of a raster dataset, filling gaps and connecting patches along the way. Conversely, erosion chips away at the outermost pixels, thereby removing the surplus pixels that were added during the dilation step but still maintaining the filled-in gaps.

We will apply these in sequence, first dilation and then erosion, using `focal_max` and `focal_min`, respectively; `focal_max` works as a dilation since it outputs the maximum value detected within the kernel, which will always be a settlement pixel because the settlement pixel value of 1 is always greater than the non-settlement pixel value of 0. Since we just need to do some fine-tuning on the boundary of the settlement coverage, we can use a kernel with a small radius of 3. Finally, let's convert the output of the dilation and erosion to a polygon `FeatureCollection` where each contiguous patch of pixels becomes its own polygon (Fig. 38.4). Feel free to map the outline of Pagirinya in blue, as above, for a helpful visual reference.

```
// Define the kernel used for morphological operations.
var kernel = ee.Kernel.square({
    radius: 3
});

// Perform a dilation followed by an erosion.
var kMeansCleaned = kmeansResult
    .focal_max({
        kernel: kernel,
        iterations: 1
    }) // Dilation
    .focal_min({
        kernel: kernel,
        iterations: 1
    }); // Erosion
Map.addLayer(kMeansCleaned, {
    bands: ['cluster'],
    max: 1,
    min: 0
}, 'K-means cleaned');

// Convert cleaned K-means settlement and non-settlement
coverages to polygons.
var kMeansCleanedPolygon = kMeansCleaned.reduceToVectors({
    scale: 30,
    eightConnected: true
});

Map.addLayer(kMeansCleanedPolygon, {}, 'K-Means cleaned
polygon');

// Map outline of Pagirinya in blue.
Map.addLayer(pagirinyaOutline,
    {
        palette: '0000FF'
    },
    'Pagirinya Refugee Settlement boundary');
```

We have created a usable vector map of settlement and non-settlement polygons, but we are aiming for a single polygon that represents the settlement boundary. To filter these polygons to a single polygon that represents the refugee settlement's boundary, let's use a simple logic rule and select the polygon that has the largest overlap (i.e., intersected area) with the UNHCR boundary (Fig. 38.5).

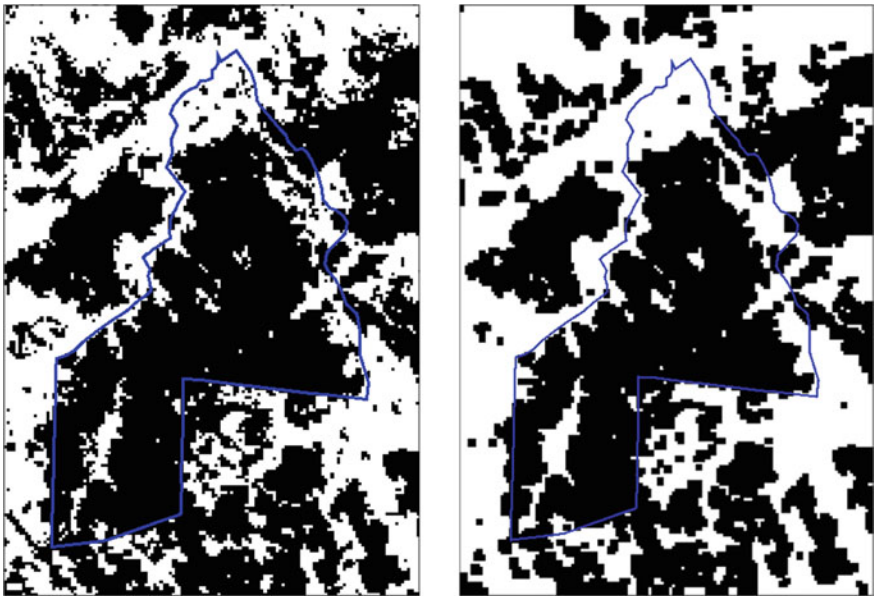


Fig. 38.4 *K*-means output before (left) and after (right) dilation and erosion, with Pagirinya Refugee Settlement boundary overlaid in blue

```
// Intersect K-means polygons with UNHCR settlement
boundary and
// return intersection area as a feature property.
var kMeansIntersect =
kMeansCleanedPolygon.map(function(feat) {
    var boundaryIsect = pagirinya.intersection(feat, ee
        .ErrorMargin(1));
    return ee.Feature(feat).set({
        'isectArea': boundaryIsect.area()
    });
});

// Sort to select the polygon with largest overlap with the
UNHCR settlement boundary.
var kMeansBoundary =
ee.Feature(kMeansIntersect.sort('isectArea',
    false).first());
Map.addLayer(kMeansBoundary, {}, 'K-Means Settlement
Boundary');
```

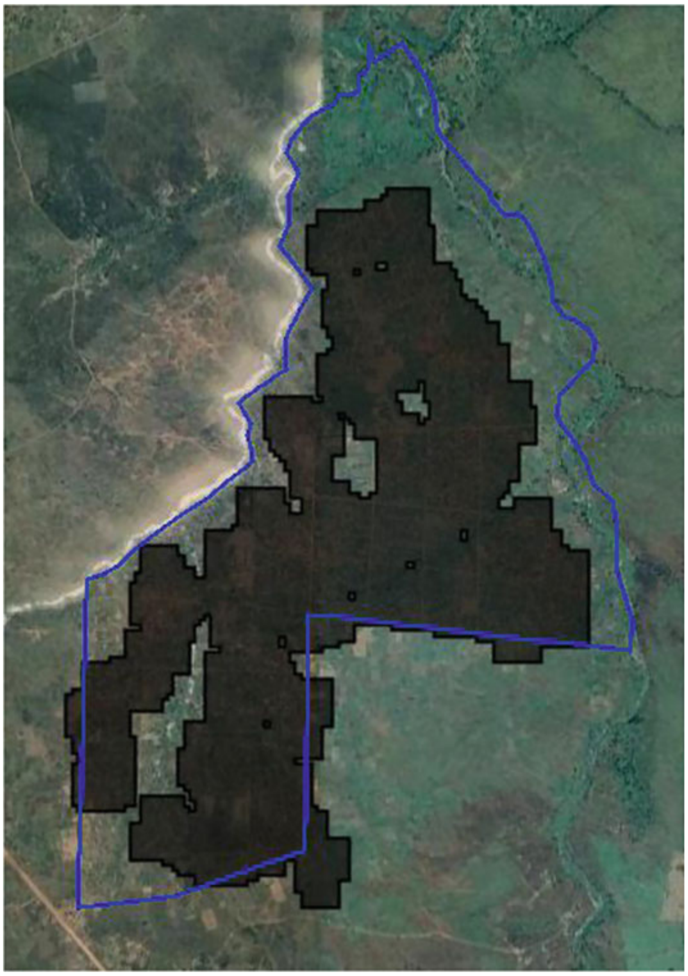


Fig. 38.5 *K*-means settlement boundary (black) overlaid by UNHCR settlement boundary (blue)

Code Checkpoint A17c. The book's repository contains a script that shows what your code should look like at this point.

Question 10. In your opinion, does the *k*-means boundary accurately separate the settlement from its surroundings? Considering differences between the UNHCR boundary and the *k*-means boundary, comment on potential errors of commission (areas that are inaccurately included in the *k*-means boundary) and omission (areas that are inaccurately excluded).

Question 11. Rather than collecting samples for input to *k*-means based only on NDVI in the `postMedian` image, adjust the script above to sample from all bands in `postMedian`. How does the resulting settlement polygon differ? Does increasing the amount of spectral information available to the classifier improve the result?

Question 12. Rerun the *k*-means classifier based on the `diffMedian` image from Sect. 38.2.1 rather than the `postMedian` image while keeping the other parameters the same. How does the resulting settlement boundary polygon differ?

38.2.4 Section 4: Estimating Refugee Population Within the Settlement

Thus far, we have looked at land cover conditions and land cover changes at Pagirinya and used that information to help map the extent of the settlement. Let's turn toward using satellite-derived data to estimate the size of the refugee population at Pagirinya. Knowing how many refugees are at a settlement is essential for gauging the need for food aid and for guiding sustainable development and disaster risk reduction efforts. Satellite-informed population estimates can be useful for these purposes, especially if no other data are available.

In this final section, we will work with several datasets designed to estimate the geographic distribution of human populations, each of which is based in part on remote sensing detection of buildings. We will analyze population estimates at Pagirinya Refugee Settlement from the Global Human Settlement Layer (GHSL), High Resolution Settlement Layer (HRSL), and WorldPop data products. To gauge the accuracy of these products, we will compare the population estimates with UNHCR-recorded refugee population data from September 2020.

These versions of HRSL and WorldPop are from 2020, and this version of GHSL has data for multiple years, most recently 2015. Let's filter the GHSL `ImageCollection` to only the 2015 dataset. We will also rename all relevant bands to `'population'` for consistency and visualize all population maps using the same approach to support a direct comparison. Use the **Inspector** tool to identify the different pixel-level values for each population dataset within and around Pagirinya. These values represent the human population estimated to be present at each pixel.

```
Map.centerObject(pagirinya, 14);

var ghslPop =
ee.ImageCollection('JRC/GHSL/P2016/POP_GPW_GLOBE_V1')
    .filter(ee.Filter.date('2015-01-01', '2016-01-
01')).first()
    .select(['population_count'], ['population']);
var hrslPop = ee.Image('projects/gee-book/assets/A1-
7/HRSL')
    .select(['b1'], ['population']);
var worldPop = ee.ImageCollection(
        'WorldPop/GP/100m/pop_age_sex_cons_unadj')
    .filterMetadata('country', 'equals', 'UGA')
    .first()
    .select(['population']);

// Set-up visualization to be shared by all population
datasets.
var visualization = {
    bands: ['population'],
    min: 0.0,
    max: 50.0,
    palette: ['24126c', '1fff4f', 'd4ff50']
};

// Map population datasets.
Map.addLayer(ghslPop, visualization, 'GHSL Pop');
Map.addLayer(hrslPop, visualization, 'HRSL Pop');
Map.addLayer(worldPop, visualization, 'WorldPop');
```

You will notice that each dataset has a different spatial resolution (also commonly referred to as the "scale"). We will need to know these different spatial resolutions when we summarize each dataset's population estimate across Pagirinya using `reduceRegion`. Once we have a population estimate, we will add it to the Pagirinya feature as a new property.

```
// Collect spatial resolution of each dataset.
var ghslPopProjection = ghslPop.projection();
var ghslPopScale = ghslPopProjection.nominalScale();
print(ghslPopScale, 'GHSL scale');

var hrslPopProjection = hrslPop.projection();
var hrslPopScale = hrslPopProjection.nominalScale();
print(hrslPopScale, 'HRSL scale');

var worldPopProjection = worldPop.projection();
var worldPopScale = worldPopProjection.nominalScale();
print(worldPopScale, 'WorldPop scale');

// Summarize population totals for each population product
at each settlement and
// assign as new properties to the UNHCR boundary Feature.
pagirinya = pagirinya.set(ghslPop.select(['population'],
['GHSL'])
    .reduceRegion({
        reducer: 'sum',
        scale: ghslPopScale,
        geometry: pagirinya.geometry(),
        maxPixels: 1e9,
    }));

pagirinya = pagirinya.set(hrslPop.select(['population'],
['HRSL'])
    .reduceRegion({
        reducer: 'sum',
        scale: hrslPopScale,
        geometry: pagirinya.geometry(),
        maxPixels: 1e9,
    }));

pagirinya = pagirinya.set(worldPop.select(['population'], [
    'WorldPop'])
    .reduceRegion({
        reducer: 'sum',
        scale: worldPopScale,
        geometry: pagirinya.geometry(),
        maxPixels: 1e9,
    }));

print(pagirinya, 'Pagirinya with population estimates');
```

Now we have three very different population estimates for Pagirinya based on the three population datasets. Let's see how they compare to the population recorded in 2020 by UNHCR, which is also stored as a property of the Pagirinya feature.

To do so, we will simply subtract the UNHCR population total from each dataset's estimated population total and store each difference as a new property. A negative difference indicates an underestimation of the UNHCR-recorded population, and a positive difference indicates an overestimation.

```
// Measure difference between settlement product and UNHCR-
recorded population values.
var unhcrPopulation =
ee.Number(pagirinya.get('UNHCR_Pop'));
var ghslDiff = ee.Number(pagirinya.get('GHSL')).subtract(
    unhcrPopulation);
var hrslDiff = ee.Number(pagirinya.get('HRSL')).subtract(
    unhcrPopulation);
var worldPopDiff =
ee.Number(pagirinya.get('WorldPop')).subtract(
    unhcrPopulation);

// Update UNHCR boundary Feature with population difference
properties.
pagirinya =
pagirinya.set(ee.Dictionary.fromLists(['GHSL_diff',
        'HRSL_diff', 'WorldPop_diff'],
    [ghslDiff, hrslDiff, worldPopDiff]));

print('Pagirinya Population Estimations', pagirinya);
```

Code Checkpoint A17d. The book's repository contains a script that shows what your code should look like at this point.

Question 13. Visually interpret the coverage of each population dataset alongside the building footprint data from Sect. 38.2.2. Which population dataset seems to better capture population density at hot spots of building footprints?

Question 14. Many buildings in Pagirinya are not household dwellings but rather administrative offices, shops, food market buildings, etc., and such differences in building use are not necessarily considered in generating the population estimates. How would the inclusion of non-dwellings in population datasets bias settlement-level population estimates?

Question 15. Note that the coverage of the WorldPop population data at Pagirinya is not wholly contained within the UNHCR settlement boundary. Is this "spillover" better captured by the *k*-means boundary from Sect. 38.2.3?

38.3 Synthesis

You may have noticed that we showed a 2020 land cover map from the European Space Agency (ESA) based on Sentinel-1 and Sentinel-2 data in Fig. 38.1c but did not make use of those land cover data in the practicum. How would your settlement boundary detection approach and results change if you used Sentinel-2 instead of Landsat data and sampled land cover sites from this ESA dataset? As a homework challenge, please complete the following assignment.

Assignment 1. Use Sentinel-2 surface reflectance data collected in 2020. Collect 20 samples of each land cover class in the ESA land cover product within Pagirinya using `ee.Image.stratifiedSample`. Assess the spectral separability between land cover classes. Then, run a modified *k*-means classifier that makes use of Sentinel-2 NDVI values collected across the ESA land cover map.

38.4 Conclusion

This chapter introduced approaches for characterizing land cover dynamics within and surrounding Pagirinya Refugee Settlement using a range of open-access satellite data and geospatial products. We saw that satellite remote sensing approaches are effective for characterizing land cover changes before and following the establishment of Pagirinya in 2016, and for delineating a refugee settlement boundary that represents land directly affected by the settlement's establishment and use. We also noted wide disagreement and pronounced inaccuracies in Pagirinya refugee population estimates based on satellite-informed human population datasets. This chapter shows the value of remote sensing for long-term monitoring of refugee settlements as well as the need for deeper integration of humanitarian data and scenarios in remote sensing applications.

References

Friedrich HK, Van Den Hoek J (2020) Breaking ground: automated disturbance detection with Landsat time series captures rapid refugee settlement establishment and growth in North Uganda. Comput Environ Urban Syst 82:101499. https://doi.org/10.1016/j.compenvurbsys.2020.101499

Maystadt JF, Mueller V, Van Den Hoek J, Van Weezel S (2020) Vegetation changes attributable to refugees in Africa coincide with agricultural deforestation. Environ Res Lett 15:44008. https://doi.org/10.1088/1748-9326/ab6d7c

UNHCR (2020) Global trends: forced displacement in 2020

Van Den Hoek J, Friedrich HK (2021) Satellite-based human settlement datasets inadequately detect refugee settlements: a critical assessment at thirty refugee settlements in Uganda. Remote Sens 13:3574. https://doi.org/10.3390/rs13183574

Van Den Hoek J, Friedrich HK, Wrathall D (2021) A primer on refugee-environment relationships. In: PERN cyberseminar on refugee and internally displaced populations, environmental impacts and climate risks

39 Monitoring Gold Mining Activity Using SAR

Lucio Villa, Sidney Novoa, Milagros Becerra, Andréa Puzzi Nicolau, Karen Dyson, Karis Tenneson, and John Dilger

Overview
The expansion of gold mining has had a large impact on the rainforests of the Amazon over the last decades. To take just one example, it has affected both the biodiversity and the lives of local people in the Madre de Dios region of southeastern Peru. In this chapter, we will review a methodology developed to generate early warnings of deforestation based on the use of synthetic aperture radar (SAR) images. First, we will identify the Sentinel-1 images suitable for the construction of a time series of preprocessed datasets. Second, we will run a change detection analysis based on a statistical analysis of the Sentinel-1 images. Finally, we will show the steps to follow in the post-processing stage by filtering information with forest/non-forest and bodies of water datasets.

L. Villa (✉)
Universidad Nacional Agraria La Molina (UNALM), Av. La Molina s/n. La Molina, 15024 Lima, Peru
e-mail: luciovilla@lamolina.edu.pe

S. Novoa · M. Becerra
Conservación Amazónica - ACCA, 627 Calle General Vargas Machuca, Miraflores, 15047 Lima, Peru

S. Novoa · M. Becerra · A. P. Nicolau · K. Dyson · K. Tenneson · J. Dilger
SERVIR-Amazonia, Cali, Colombia

A. P. Nicolau · K. Dyson · K. Tenneson · J. Dilger
Spatial Informatics Group, Pleasanton, CA, USA

K. Dyson
Dendrolytics, Seattle, WA, USA

J. Dilger
Astraea, Charlottesville, VA, USA

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_39

Learning Outcomes

- Selecting and creating a multi-temporal SAR mosaic.
- Generating SAR change detection based on a statistical analysis of Sentinel-1 images.
- Post-processing of alerts generated by filtering maximum area patches and information of forest/non-forest and water bodies.

Helps if you know how to

- Import, filter, and visualize images (Part 1).
- Work with time series data in Earth Engine (Part 4).
- Write a function and `map` it over an ImageCollection (Chap. 12).
- Use the `require` function to load code from existing modules (Chap. 28).
- Export results to assets (Chap. 29).

39.1 Introduction to Theory

Accelerated demand for natural resources has transformed the Amazon rainforest into a new economic frontier that generates commodities such as agricultural products, livestock, and more recently, minerals, especially gold (RAISG 2020). In Peru, illegal gold mining is a serious problem that affects local populations in the southeastern region of Madre de Dios (Yard et al. 2012; Asner and Tupayachi 2017, Alvarez-Berrios et al. (2021). According to Caballero et al. (2018), it has led to the deforestation of about 1000 km^2 of rainforest, affecting protected areas, Indigenous communities, and sustainable management areas. Gold mining is carried out throughout the year, even during the rainy season.

Optical Earth observation satellites have played a very important role in monitoring gold mining deforestation in recent years. Madre de Dios is one of the few tropical regions in the world for which there is well-documented information on annual forest loss due to this deforestation driver (Asner et al. 2013; Asner and Tupayachi 2017; Caballero et al. 2018; Nicolau et al. 2019; Csillick and Asner 2020; Aguirre et al. 2021). However, these resources do not reveal the progress of illegal mining during the rainy season and at other times when the optical images are obscured by clouds. This reduces the window of opportunity to know the dynamics of the activity and guide actions to control the advance of deforestation.

Satellite-based SAR sensors obtain information throughout the entire year thanks to the ability of microwaves to penetrate cloud cover (Ballère et al. 2021; Nicolau et al. 2021). Therefore, we are able to capture changes due to gold mining expansion with SAR data from the European Space Agency (ESA) Sentinel-1 C-band satellite. In this chapter, we will look at a successful example of how to process radar images to generate early warnings of gold mining deforestation.

39.2 Practicum

SAR sensors transmit microwave signals at an oblique angle and measure the backscattered portion of the signal in order to analyze features on the surface. Unlike optical sensors, which are passive, SAR is an active instrument with its own source of illumination, and it is one of the few sensing instruments that allows full control of the signal polarization on both the transmit and receive paths. The majority of today's SAR sensors are linearly polarized, and transmit horizontally and/or vertically polarized wave forms (i.e., SAR bands can be VV, VH, HV, or HH). While interpreting optical imagery is similar to interpreting a photograph, interpreting SAR data requires a different way of thinking, in that the signal is instead responsive in complex ways to surface characteristics such as structure and moisture. More information about the theory and concepts behind SAR is available in the SAR Handbook (Flores-Anderson et al. 2019).

The area of study for this chapter is the region surrounding a mining corridor located in Madre de Dios in the southern Peruvian Amazon (Fig. 39.1).

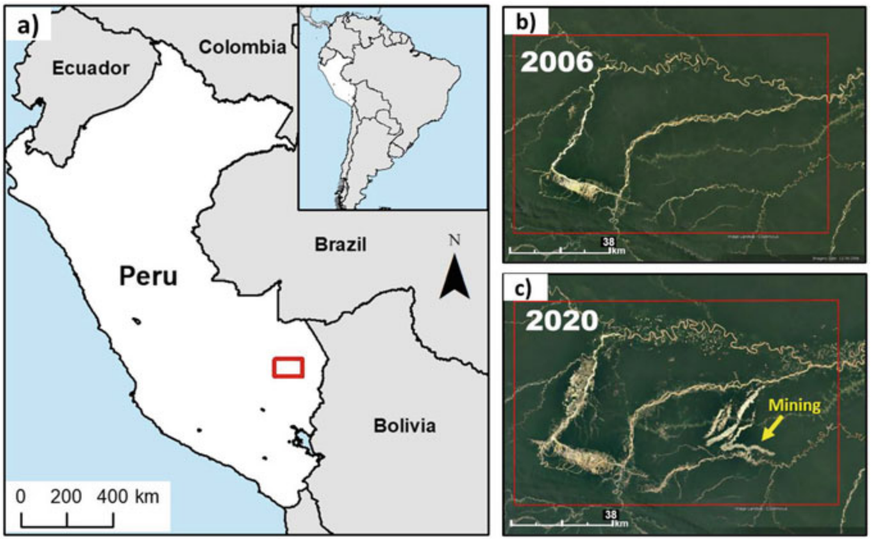


Fig. 39.1 Maps of (**a**) the area of study in the southern Peruvian Amazon and the expansion of alluvial mining activity between (**b**) 2006 and (**c**) 2020

Alluvial gold mining exploitation generates changes in the radar backscattering mechanism over forested areas, since it involves deforestation, removal of topsoil, excavation, and the use of water for the extraction of gold from the loose sediment. Figure 39.2 shows the changes in SAR Sentinel-1 images and a radar backscattering mechanism before (Fig. 39.2a) and after (Fig. 39.2b) the impact of alluvial gold mining activity.

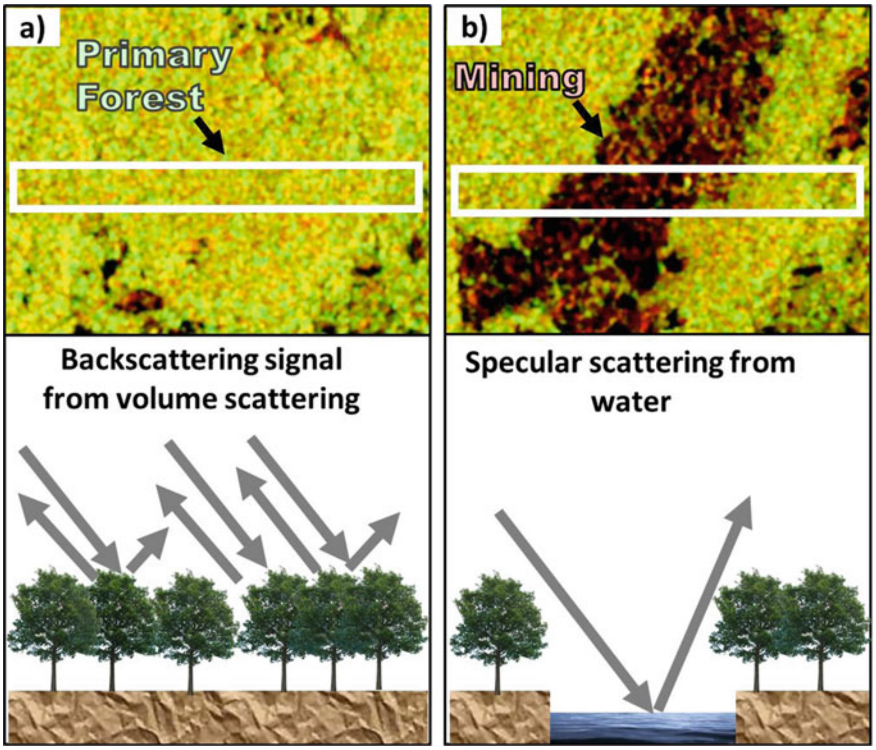

a)
Primary Forest
Backscattering signal from volume scattering
b)
Mining
Specular scattering from water

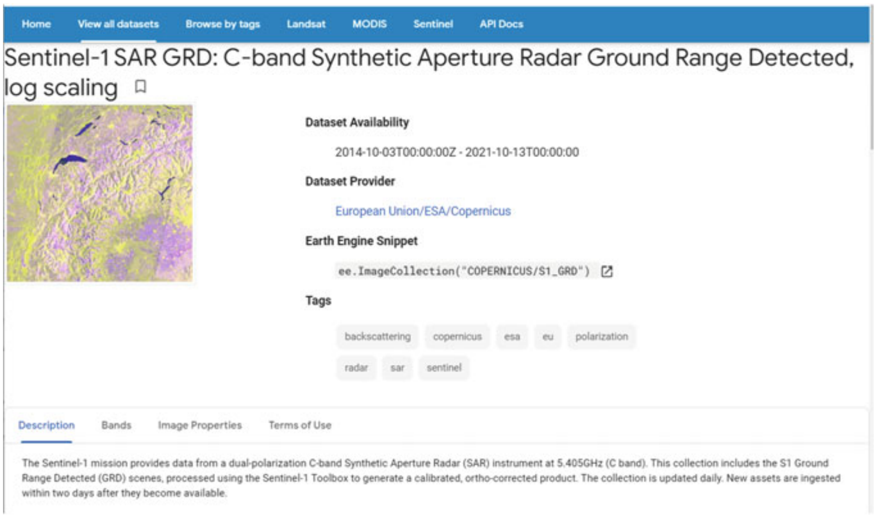
Fig. 39.2 Radar backscattering mechanism from an area affected by alluvial mining activity (**a**) before: backscattering signal from primary forest (volume scattering); (**b**) after: specular scattering from bearing soil and water as a result of clearing forest cover, removing topsoil, digging pits, and using water in the extraction process of gold from the loose sediment

Therefore, you will learn how to use SAR Sentinel-1 time series to detect changes generated by alluvial gold mining activities in forested areas.

The first step is to create a SAR mosaic for a given period of time and orbit pass. Then, we will apply the omnibus Q test algorithm to generate change alerts from the SAR Sentinel-1 time series. Finally, we will filter and eliminate potential false positive alerts coming from other activities with the same temporal pattern as the mining activity (e.g., natural forest loss by river expansion or water over bare soil during the rainy season).

39.2.1 Section 1: Creating a Single SAR Mosaic

We will use ESA's Sentinel-1 dual-polarized (VV + VH) in descending orbit pass to create a single SAR mosaic over the area of study. Sentinel-1 SAR Ground Range Detected (GRD) data (Fig. 39.3) are stored in Earth Engine in two formats: logarithmic scale (dB) and the original values (power scale named as `FLOAT`). We will use the latter since mathematical operations should not be applied into data on a logarithmic scale. The Sentinel-1 dataset is composed of Level-1 SAR amplitude multi-look images preprocessed according to the following steps: orbit metadata update, removal of border and thermal noise, radiometric calibration, and terrain correction.


Home View all datasets Browse by tags Landsat MODIS Sentinel API Docs

Sentinel-1 SAR GRD: C-band Synthetic Aperture Radar Ground Range Detected, log scaling

Dataset Availability

2014-10-03T00:00:00Z - 2021-10-13T00:00:00

Dataset Provider

European Union/ESA/Copernicus

Earth Engine Snippet

```
ee.ImageCollection("COPERNICUS/S1_GRD")
```

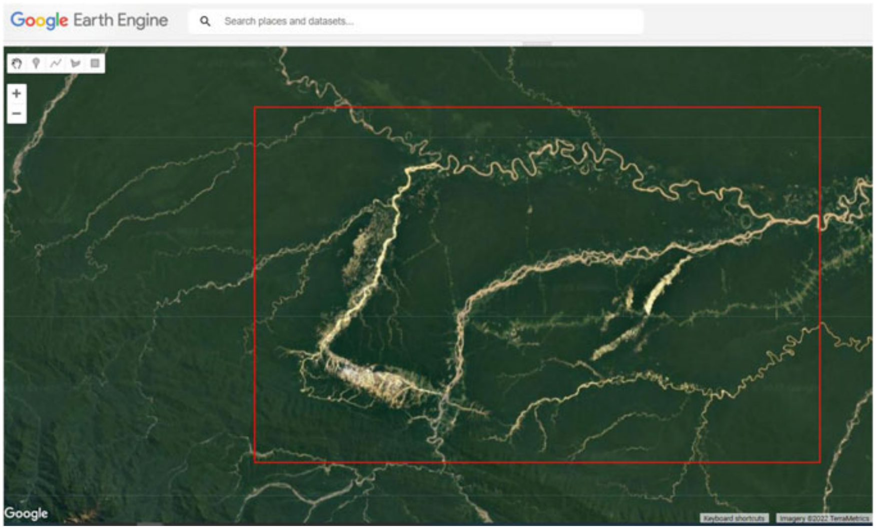
Tags

backscattering copernicus esa eu polarization radar sar sentinel

Description Bands Image Properties Terms of Use

The Sentinel-1 mission provides data from a dual-polarization C-band Synthetic Aperture Radar (SAR) instrument at 5.405GHz (C band). This collection includes the S1 Ground Range Detected (GRD) scenes, processed using the Sentinel-1 Toolbox to generate a calibrated, ortho-corrected product. The collection is updated daily. New assets are ingested within two days after they become available.

Sentinel-1 imagery available in the Earth Engine data catalog



Area of study around the mining corridor located in the Madre de Dios region

Copy and paste the code below to define the area of study (Fig. 39.1), convert this vector to a boundary image, and add it to the map (Fig. 39.4).

```
//////////////////////////////////////////////////////
/// Section One
//////////////////////////////////////////////////////

// Define the area of study.
var aoi = ee.FeatureCollection('projects/gee-
book/assets/A1-8/mdd');

// Center the map at the aoi.
Map.centerObject(aoi, 9);

// Create an empty image.
var empty = ee.Image().byte();

// Convert the area of study to an EE image object
// so we can visualize only the boundary.
var aoiOutline = empty.paint({
    featureCollection: aoi,
    color: 1,
    width: 2
});

// Select the satellite basemap view.
Map.setOptions('SATELLITE');

// Add the area of study boundary to the map.
Map.addLayer(aoiOutline, {
    palette: 'red'
}, 'Area of Study');
```

Next, copy and paste the code below to define two functions, `maskAngle` and `getCollection`. The first function masks sections of SAR images acquired at an incidence angle less than 31° or greater than 45°. The second function filters the Sentinel-1 imagery to a specific period of time, region of interest, and orbit pass. Note that the Sentinel-1 GRD dataset is imported inside the second function.

```
// Function to mask the SAR images acquired with an
incidence angle
// lower or equal to  31 and greater or equal to 45
degrees.
function maskAngle(image) {
    var angleMask = image.select('angle');
    return
image.updateMask(angleMask.gte(31).and(angleMask.lte(45)))
;
}

// Function to get the SAR Collection.
function getCollection(dates, roi, orbitPass0) {
    var sarCollFloat =
ee.ImageCollection('COPERNICUS/S1_GRD_FLOAT')
        .filterBounds(roi)
        .filterDate(dates[0], dates[1])
        .filter(ee.Filter.eq('orbitProperties_pass',
orbitPass0));
    return sarCollFloat.map(maskAngle).select(['VV',
'VH']);
}
```

Copy and paste the code below to import the Sentinel-1 collection, define time variables (a list of dates) and the orbit pass variable, apply the functions, create a mosaic by using the `mosaic` function, and clip the mosaic to the study area.

```
// Define variables: the period of time and the orbitpass.
var listOfDates = ['2021-08-01', '2021-08-12'];
var orbitPass = 'DESCENDING';

// Apply the function to get the SAR mosaic.
var sarImageColl = getCollection(listOfDates, aoi,
orbitPass)
    .mosaic()
    .clip(aoi);
print('SAR Image Mosaic', sarImageColl);
```

Before adding the mosaic to the map, it’s important to scale the values to a logarithmic scale (`log10().multiply(10.0)`). The parameters of visualization (`sarVis`) should be taken between 3 and −23 dB (decibels). Copy and paste the code below to do so and to add the mosaic to the map (Fig. 39.5). The code creates a scaled image and draws it using visualization parameters set through trial and error.

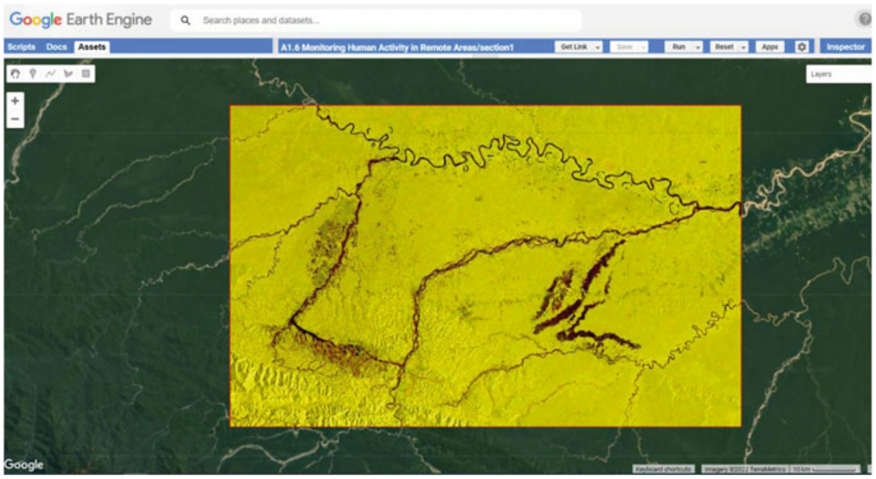

Google Earth Engine
Search places and datasets...
Scripts
Docs
Assets
A1.6 Monitoring Human Activity in Remote Areas/section1
Get Link
Save
Run
Reset
Apps
Inspector
Layers
Google

Fig. 39.5 SAR Sentinel-1 mosaic generated in previous section

```
// Apply logarithmic scale.
var sarImageScaled = sarImageColl.log10().multiply(10.0);

// Visualize results.
var sarVis = {
    bands: ['VV', 'VH', 'VV'],
    min: [-18, -23, 3],
    max: [-4, -11, 15]
};
Map.addLayer(sarImageScaled, sarVis, 'Sentinel-1 / SAR
Mosaic');
```

Code Checkpoint A18a. The book's repository contains a script that shows what your code should look like at this point.

Question 1. How many bands (polarizations) does Sentinel-1 have over the area of study?

Question 2. Using the **Inspector** tool, explore the values from the VV and VH bands over different land covers. Which one do you think is better able to detect forested areas?

39.2.2 Section 2: Creating a SAR Mosaic Time Series

We will reuse most of the code from Sect. 39.2.1, only changing the period of time and not applying the mosaic function just yet. Start this section by opening the following code checkpoint.

Code Checkpoint A18b. The book's repository contains a script to use to begin this section. You will need to start with that script and paste code below into it.

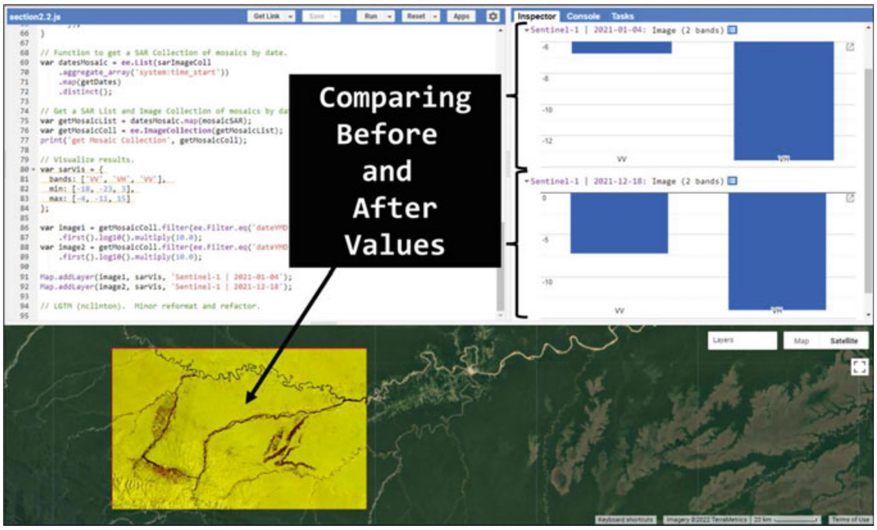
Expand the SAR `ImageCollection` in the **Console** and note that it is composed of 30 elements. To create a time series of mosaics, we need to define two additional functions (`getDates` and `mosaicSAR`). The first function converts the format of the date from milliseconds to format `'YYYY-MM-dd'`. The second function filters the SAR `ImageCollection` using the list of dates and generates a mosaic. The result is an ImageCollection of mosaics per date. Copy and paste the code below to define these two functions.

```
// Function to get dates in 'YYYY-MM-dd' format.
function getDates(dd) {
    return ee.Date(dd).format('YYYY-MM-dd');
}

// Function to get a SAR Mosaic clipped to the study area.
function mosaicSAR(dates1) {
    dates1 = ee.Date(dates1);
    var imageFilt = sarImageColl
        .filterDate(dates1, dates1.advance(1, 'day'));
    return imageFilt.mosaic()
        .clip(aoi)
        .set({
            'system:time_start': dates1.millis(),
            'dateYMD': dates1.format('YYYY-MM-dd')
        });
}
```

Now, copy and paste the code below to generate a list of dates without duplicate elements (i.e., where there are images from the same dates in the collection, we only keep one). We avoid duplicates by using the `ee.List.distinct` and the `getDates` functions and output an `ImageCollection` of mosaics per date.

```
// Function to get a SAR Collection of mosaics by date.
var datesMosaic = ee.List(sarImageColl
        .aggregate_array('system:time_start'))
    .map(getDates)
    .distinct();

// Get a SAR List and Image Collection of mosaics by date.
var getMosaicList = datesMosaic.map(mosaicSAR);
var getMosaicColl = ee.ImageCollection(getMosaicList);
print('get Mosaic Collection', getMosaicColl);
```

Finally, copy and paste the code below to set the visualization parameter (`sarVis`) and add two SAR mosaics filtered by the date of acquisition as an example (one from 2021-01-04 and the other from 2021-12-18; Fig. 39.6).



```
}

// Function to get a SAR Collection of mosaics by date.
var datesMosaic = ee.List(sarImageColl
    .aggregate_array('system:time_start'))
    .map(getDates)
    .distinct();

// Get a SAR List and Image Collection of mosaics by da
var getMosaicList = datesMosaic.map(mosaicSAR);
var getMosaicColl = ee.ImageCollection(getMosaicList);
print('get Mosaic Collection', getMosaicColl);

// Visualize results.
var sarVis = {
  bands: ['VV', 'VH', 'VV'],
  min: [-18, -23, 3],
  max: [-4, -11, 15]
};

var image1 = getMosaicColl.filter(ee.Filter.eq('dateYMD
    .first().log10().multiply(10.0);
var image2 = getMosaicColl.filter(ee.Filter.eq('dateYMD
    .first().log10().multiply(10.0);

Map.addLayer(image1, sarVis, 'Sentinel-1 | 2021-01-04');
Map.addLayer(image2, sarVis, 'Sentinel-1 | 2021-12-18');

// LGTM (nclinton).  Minor reformat and refactor.
```

```
Sentinel-1 | 2021-01-04: Image (2 bands)
Sentinel-1 | 2021-12-18: Image (2 bands)
```

Fig. 39.6 SAR Sentinel-1 mosaics generated in Sect. 39.2.2. The **Inspector** tool shows a comparison between before and after values of VV and VH SAR bands

```
// Visualize results.
var sarVis = {
    bands: ['VV', 'VH', 'VV'],
    min: [-18, -23, 3],
    max: [-4, -11, 15]
};

var image1 = getMosaicColl
    .filter(ee.Filter.eq('dateYMD', '2021-01-04'))
    .first().log10().multiply(10.0);
var image2 = getMosaicColl
    .filter(ee.Filter.eq('dateYMD', '2021-12-18'))
    .first().log10().multiply(10.0);

Map.addLayer(image1, sarVis, 'Sentinel-1 | 2021-01-04');
Map.addLayer(image2, sarVis, 'Sentinel-1 | 2021-12-18');
```

Note that we applied the logarithmic scale for visualization purposes. Zoom in and switch between the layers to note the differences between the images.

Code Checkpoint A18c. The book's repository contains a script that shows what your code should look like at this point.

Question 3. How many images were taken by Sentinel-1 over the area of study between 2018-01-01 and 2020-01-01?

Question 4. Using the **Inspector** tool, explore the temporal changes of the values from VV and VH bands over new mining areas.

39.2.3 Section 3: Generate SAR Change Detection

There are different methods for detecting changes using SAR data. In this case, we will use a SAR change detection method based on Canty et al. (2020).

This methodology allows us to identify changes for a series of 'k' uncorrelated SAR images using a pixel-based omnibus likelihood ratio test statistic Q for covariance matrices ($\sum_i$, $i = 1 \ldots k$), based on a Wishart distribution. The Q test is defined by

$$\ln Q = n\left(pk \ln k + \sum_{i=1}^{k} \ln |X_i| - k \ln\left|\sum_{i=1}^{k} X_i\right|\right)$$

where $X_i = n\Sigma^*$ (with Σ^* being the maximum likelihood estimate of the covariance matrices Σ_i), 'n' the number of looks, and 'p' the dimensionality of the covariance matrices (with $p = 2$ for dual polarización SAR data). The $|\cdot|$ denotes the determinant.

The Q test is an omnibus test statistic because it evaluates the equality of several covariance matrices simultaneously. Thus, this test statistic tests the null hypothesis (no change, H_0) against alternative hypothesis (change, H_1) using SAR time series pixel-based data and the level of significance (probability that the null hypothesis H_0 is true, also known as *p*-value) estimated in each iteration.

In the first iteration, the first two first images in the time series are tested (null hypothesis of no change against the alternative of change):

(1) $H_0 : \Sigma_1 = \Sigma_2$ against $H_1 : \Sigma_1 \neq \Sigma_2 \rightarrow$ Null hypothesis rejected?

If the null hypothesis (H_0) is not rejected, then the test continues including the values of the next image in the series:

(B) $H_0 : \Sigma_1 = \Sigma_2 = \Sigma_3$ against $H_1 : \Sigma_1 = \Sigma_2 \neq \Sigma_3 \rightarrow$ Null hypothesis rejected?

However, if the null hypothesis is rejected in this iteration, then the interval of time for this change is labeled and the test is restarted from there.

The explanation of the Omnibus likelihood ratio test statistic is beyond the scope of this chapter but more details can be found in Canty et al. (2020).

The next steps correspond to obtaining a single change detection output image using the time series of mosaics generated in Sect. 39.2.2.

To do so, we will use modules adapted from the original JavaScript libraries by Canty et al. (2020). Beginning from the last code checkpoint from Sect. 39.2.2, copy and paste the code below to import the adapted modules and define a variable that stores the number of images in our collection. This number will be used later on for visualization purposes.

```
// Libraries of SAR Change Detection (version modified).
// The original version can be found in:
// users/mortcanty/changedetection
var omb = require(
    'projects/gee-edu/book:Part A - Applications/A1 -
Human Applications/A1.8 Monitoring Gold Mining Activity
Using SAR/modules/omnibusTest_v1.1'
);
var util = require(
    'projects/gee-edu/book:Part A - Applications/A1 -
Human Applications/A1.8 Monitoring Gold Mining Activity
Using SAR/modules/utilities_v1.1'
);

// Count the length of the list of dates of the time-
series.
var countDates = datesMosaic.size().getInfo();
```

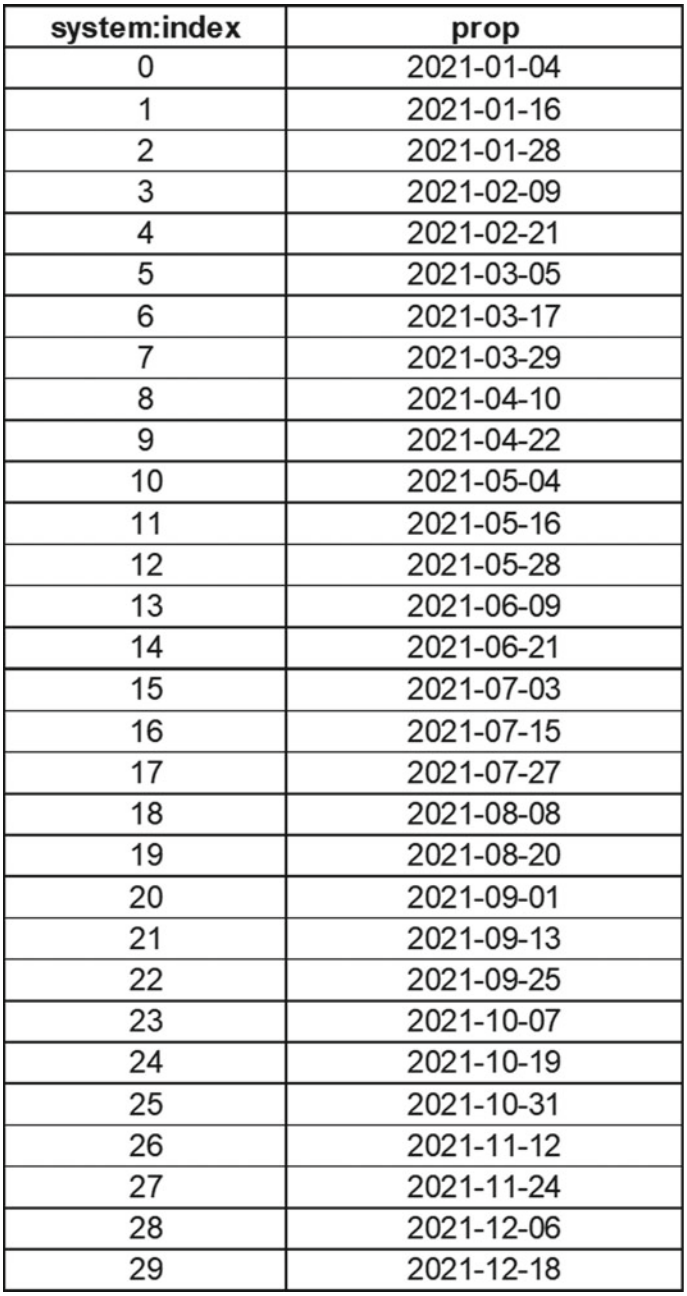
Before applying the SAR change algorithm, we need to define the input parameters such as the significance and the reducer to be applied (median in this case). The result is an ee.Dictionary that contains several images: among these are cmap, smap, fmap, bmap. The cmap image shows the occurrence of the most recent significant change, smap shows the first significant change, fmap shows the frequency of significant changes, and bmap shows the interval in which each significant change occurred.

Copy and paste the code below to define such variables, apply the algorithm to the list of SAR mosaics (getMosaicList), and extract the results.

```
// Run the algorithm and print the results.
var significance = 0.0001;
var median = true;
var result = ee.Dictionary(omb.omnibus(getMosaicList,
significance,
    median));
print('result', result);

// Change maps generated (cmap, smap, fmap and bmap)
// are detailed in the next commented lines.

// cmap: the interval in which the most recent significant
change occurred (single-band).
// smap: the interval in which the first significant
change occurred (single-band).
// fmap: the frequency of significant changes (single-
band).
// bmap: the interval in which each significant change
occurred ((k − 1)-band).

// Extract and print the images result
// (cmap, smap, fmap and bmap) from the ee.Dictionary.
var cmap = ee.Image(result.get('cmap')).byte();
var smap = ee.Image(result.get('smap')).byte();
var fmap = ee.Image(result.get('fmap')).byte();
var bmap = ee.Image(result.get('bmap')).byte();
```

The values for cmap, smap, and bmap are numbers that correspond to dates. These are the dates that are stored in the datesMosaic list. For example, the pixel value of 0 corresponds to the first date (2021-01-04), the pixel value of 1 corresponds to the second date (2021-01-16), and so on (Fig. 39.7).

If we want to export the resulting images, we need to also export the list of dates. To do so, we need to create a FeatureCollection since we can't currently export lists directly in Earth Engine. Copy and paste the code below to



system:index	prop
0	2021-01-04
1	2021-01-16
2	2021-01-28
3	2021-02-09
4	2021-02-21
5	2021-03-05
6	2021-03-17
7	2021-03-29
8	2021-04-10
9	2021-04-22
10	2021-05-04
11	2021-05-16
12	2021-05-28
13	2021-06-09
14	2021-06-21
15	2021-07-03
16	2021-07-15
17	2021-07-27
18	2021-08-08
19	2021-08-20
20	2021-09-01
21	2021-09-13
22	2021-09-25
23	2021-10-07
24	2021-10-19
25	2021-10-31
26	2021-11-12
27	2021-11-24
28	2021-12-06
29	2021-12-18

Fig. 39.7 List of dates to be exported as a CSV file. Each value (index) is associated with a specific date of change of the raster file (smap)

create a `FeatureCollection` where each feature contains the date information as a property. We are also printing the dates in order to visualize the pixel-date association (expand the list on the **Console** to see it.)

```
// Build a Feature Collection from Dates.
var fCollectionDates = ee.FeatureCollection(datesMosaic
    .map(function(element) {
        return ee.Feature(null, {
            prop: element
        });
    }));
print('Dates', datesMosaic);
```

Now, we can add the results to the map. Copy and paste the code below to define visualizations parameters, make a legend that associates date numbers with colors, and add the resulting images (Fig. 39.8).To load results faster, change the `Map.centerObject` function at the top of the script to `Map.setCenter (−70.003, −12.849, 12)`—we are zooming in to a specific area—and leave only the smap layer checked under the **Layers** panel.

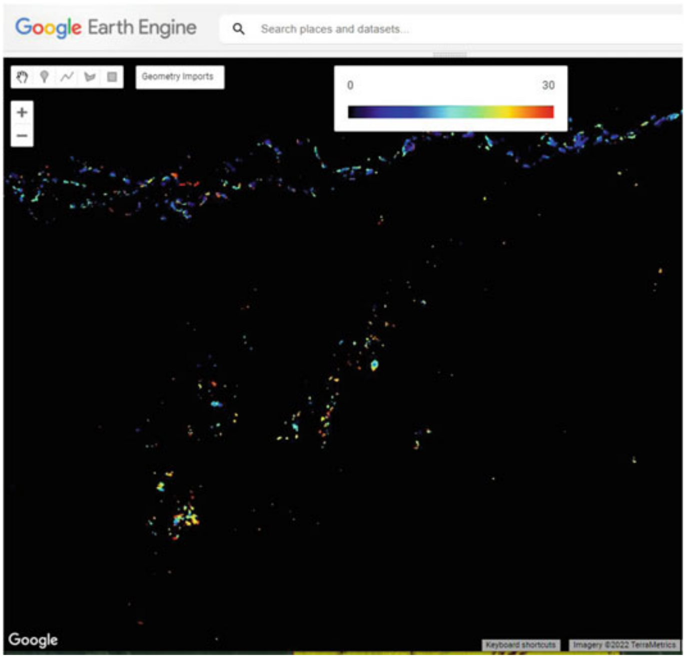


Fig. 39.8 SAR change detection results from applying the Q test algorithm

```
// Visualization parameters.
var jet = ['black', 'blue', 'cyan', 'yellow', 'red'];
var vis = {
    min: 0,
    max: countDates,
    palette: jet
};

// Add resulting images and legend to the map.
Map.add(util.makeLegend(vis));
Map.addLayer(cmap, vis, 'cmap - recent change
(unfiltered)');
Map.addLayer(smap, vis, 'smap - first change
(unfiltered)');
Map.addLayer(fmap.multiply(2), vis, 'fmap*2 - frequency of
changes');
```

Now, copy and paste the code below to export two items to Google Drive: `fCollectionDates`, the dates of SAR images processed (Fig. 39.7); and `smap`, the image of the first significant change (Fig. 39.8).

```
//  Export the Feature Collection with the dates of
change.
var exportDates = Export.table.toDrive({
    collection: fCollectionDates,
    folder: 'datesChangesDN',
    description: 'dates',
    fileFormat: 'CSV'
});
// Export the image of the first significant changes.
var exportImgChanges = Export.image.toAsset({
    image: smap,
    description: 'smap',
    assetId: 'your_asset_path_here/' + 'smap',
    region: aoi,
    scale: 10,
    maxPixels: 1e13
});
```

Code Checkpoint A18d. The book’s repository contains a script that shows what your code should look like at this point.

Question 5. What is the difference between the smap and cmap images?

Question 6. How does the FeatureCollection fCollectionDate relate to the change maps smap, cmap, and fmap?

39.2.4 Section 4: Filtering and Post-processing Alerts

As explained earlier in the Practicum, the smap results need to be filtered in order to eliminate possible false positives. The false positives are associated with the forest loss due to river morphology and the presence of muddy water bodies (Fig. 39.2).

In this section, we will explore different options to filter false positives and, therefore, post-process the results generated.

Like we did in previous sections, copy and paste the code below into a new script to import the study area and the exported smap image. Remember that our analysis covered 30 Sentinel-1 images from distinct dates.

```
//////////////////////////////////////////////////////
/// Section Four
//////////////////////////////////////////////////////

// Define the area of study.
var aoi = ee.FeatureCollection('projects/gee-
book/assets/A1-8/mdd');

// Center the map.
Map.centerObject(aoi, 10);

// Create an empty image.
var empty = ee.Image().byte();

// Convert the area of study to an EE image object so we
can visualize
// only the boundary.
var aoiOutline = empty.paint({
    featureCollection: aoi,
    color: 1,
    width: 2
});

// Select the satellite basemap view.
Map.setOptions('SATELLITE');
```

```
// Add the area of study boundary to the map.
Map.addLayer(aoiOutline, {
    palette: 'red'
}, 'Area of Study');

// Import the smap result from section 3.
var changeDetect = ee.Image('projects/gee-book/assets/A1-
8/smap');

// Visualization parameters.
var countDates = 30;
var jet = ['black', 'blue', 'cyan', 'yellow', 'red'];
var vis = {
    min: 0,
    max: countDates,
    palette: jet
};

// Add results to the map.
Map.addLayer(changeDetect, vis, 'Change Map Unfiltered');
```

Next, copy and paste the code below to import from the Earth Engine data catalog and add to the map all the sources of information for filtering false positives: Shuttle Radar Topography Mission (SRTM) digital elevation data, Hansen Global Forest Change data, and JRC Global Surface Water data (Fig. 39.9).

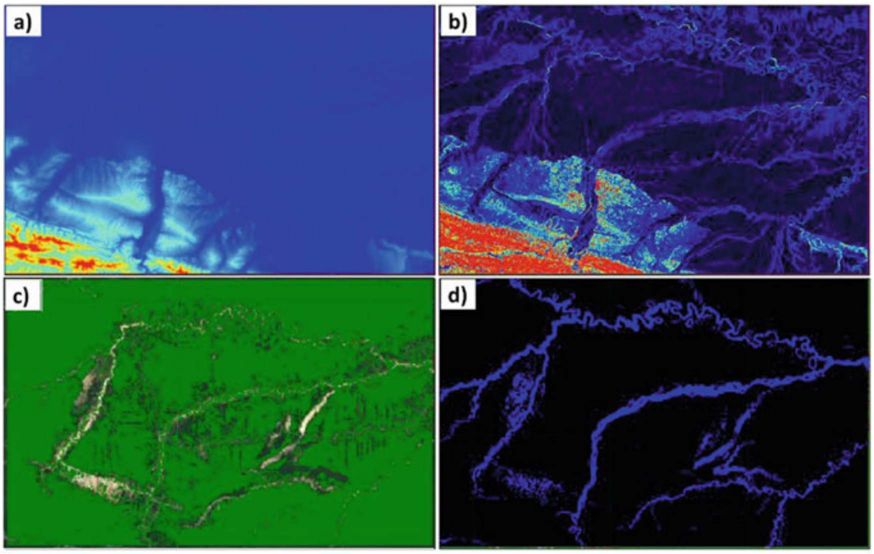


Fig. 39.9 Layers used to filter false positives alerts: (**a**) SRTM elevation, red color shows areas over 1000 m above sea level; (**b**) SRTM slope, red color shows areas with slope over 15°; (**c**) Hansen Global Forest Change, green color shows forested areas updated to 2020; (**d**) JRC Yearly Water Classification History, blue color shows the maximum extent of water surface detected from 1984 to 2020

```
// Digital Elevation Model SRTM.
// https://developers.google.com/earth-
engine/datasets/catalog/USGS_SRTMGL1_003
var srtm = ee.Image('USGS/SRTMGL1_003').clip(aoi);
var slope = ee.Terrain.slope(srtm);
var srtmVis = {
    min: 0,
    max: 1000,
    palette: ['black', 'blue', 'cyan', 'yellow', 'red']
};
Map.addLayer(srtm, srtmVis, 'SRTM Elevation');
var slopeVis = {
    min: 0,
    max: 15,
    palette: ['black', 'blue', 'cyan', 'yellow', 'red']
};
Map.addLayer(slope, slopeVis, 'SRTM Slope');
```

```
// Hansen Global Forest Change v1.8 (2000-2020)
// https://developers.google.com/earth-
engine/datasets/catalog/UMD_hansen_global_forest_change_20
20_v1_8
var gfc =
ee.Image('UMD/hansen/global_forest_change_2020_v1_8').clip
(
    aoi);
var forest2020 = gfc.select('treecover2000')
    .gt(0)
    .updateMask(gfc.select('loss')
        .neq(1))
    .selfMask();
Map.addLayer(forest2020,
    {
        min: 0,
        max: 1,
        palette: ['black', 'green']
    },
    'Forest cover 2020');

// JRC Yearly Water Classification History, v1.3 (Updated
until Dec 2020).
// https://developers.google.com/earth-
engine/datasets/catalog/JRC_GSW1_3_GlobalSurfaceWater
var waterJRC =
ee.Image('JRC/GSW1_3/GlobalSurfaceWater').select(
    'max_extent');
var waterVis = {
    min: 0,
    max: 1,
    palette: ['blue', 'black']
};
Map.addLayer(waterJRC.eq(0), waterVis, 'Water Bodies until
2020');
```

You can toggle these layers on and off, zoom in and out, and inspect pixel values to analyze them.

The SRTM elevation layer (`'SRTM Elevation'`) and the slope (`'SRTM Slope'`) derived from it with `ee.Terrain.slope` show red areas that correspond to an altitude over 1000 m above sea level and a slope over 15°, compared to blue areas, which are closer to sea level or to flat terrain. We chose these options because mining activity in this region is located in lowlands. Furthermore, the SAR Sentinel-1 data provided by Earth Engine are not radiometric-terrain corrected. So, steep slopes (> 15°) generate distortions in SAR images, and therefore, potential false changes between two or more images taken at different times.

The Hansen Global Forest Change data are composed of forested areas in 2000 (the `'treecover2000'` band) and the forest loss between 2001 and 2020. In this sense, the binary layer `'Forest cover 2020'` previously generated and added shows in green the forested areas updated until 2020, with all the non-forested and the forest loss between 2001 and 2020 areas masked.

The JRC Yearly Water Classification History data show surface water extent and change between 1984 and 2020. The binary layer `'Water Bodies until 2020'` previously added shows in blue the water bodies' maximum extent between 1984 and 2020. Non-water bodies are shown in black.

Note that alluvial mining expansion pattern in the study area is associated with primary forest loss and the appearance of new surface water patches (Fig. 39.2).

Now, we will filter the false positives based on thresholds. We will mask any pixel in `smap` marked as changeover areas greater than 1000 m above sea level, slope greater than 15° according to the SRTM data (classified as forest until 2020 by the Hansen data), and that are not classified as water bodies by the JRC dataset. Copy and paste below to add the filtered results to the map.

```
// Apply filters through masks.
var alertsFiltered = changeDetect
    .updateMask(srtm.lt(1000))
    .updateMask(slope.lt(15))
    .updateMask(forest2020.eq(1))
    .updateMask(waterJRC.eq(0))
    .selfMask();

// Add filtered results to the map.
Map.addLayer(alertsFiltered,
    {
        min: 0,
        max: countDates,
        palette: jet
    },
    'Change Map Filtered',
    1);
```

We can still improve the results a bit more. Copy and paste the code below to define and apply a function that eliminates small pixel patches and isolated pixels. We do this because we know that in this area, mining activities occur in areas of at least 0.5 ha.

```
// Function to filter small patches and isolated pixels.
function filterMinPatchs(alerts0, minArea0, maxSize0) {
    var pixelCount =
alerts0.gt(0).connectedPixelCount(maxSize0);
    var minPixelCount = ee.Image(minArea0).divide(ee.Image
    .pixelArea());
    return
alerts0.updateMask(pixelCount.gte(minPixelCount));
}

// Apply the function and visualize the filtered results.
var alertsFiltMinPatchs = filterMinPatchs(alertsFiltered,
10000, 200);

Map.addLayer(alertsFiltMinPatchs, vis,
    'Alerts Filtered - Minimum Patches');
```

Turn off all the other layers to visualize the filtered result. By analyzing the results without the filters and with the filters, we can see that we eliminated most of the false positives (Fig. 39.10).

Finally, we can export the outcome. Copy and paste the code below to export to the Drive.

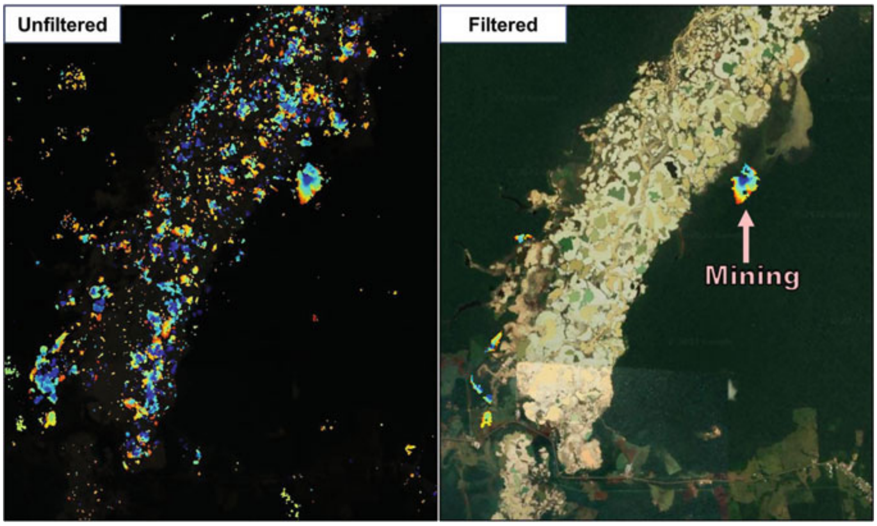


Fig. 39.10 A comparison of results, after (`'Alerts Filtered - Minimum Patches'`) and before filtering false positives (`'Change Map Unfiltered'`)

```
// Export filtered results to the Drive.
Export.image.toDrive({
    image: alertsFiltMinPatchs,
    description: 'alertsFiltered',
    folder: 'alertsFiltered',
    region: aoi,
    scale: 10,
});
```

Code Checkpoint A18e. The book's repository contains a script that shows what your code should look like at this point.

39.3 Synthesis

In this chapter, we mapped the changes generated by the alluvial mining activity over a forested area and between a period of time using Sentinel-1 SAR time series. For this, we separated the methodology into three steps. First, we were able to build a time series from Sentinel-1 mosaics. Second, we estimate all the changes based on the omnibus Q test change detection algorithm. Finally, we filter the detected changes based on existing forest/non-forest, water bodies, and elevation data, and a minimum mapping unit in order to retrieve the changes generated by the alluvial mining activity only.

Now, it's your turn to explore the use of the methodology.

Assignment 1. In this chapter, we applied the methodology for a given SAR orbit. Describe how we could identify the different SAR orbits over a specific area of study.

Assignment 2. Describe whether these alerts, which are generated by a change detection algorithm, are different for the ascending or descending orbit over our area of study.

39.4 Conclusion

In this chapter, you have learned how to analyze the changes generated by alluvial mining activity over forested areas based on the application of a SAR change detection methodology. This is possible because of the significant impact generated by this activity over the environment (that is, the deforestation and the use of water in the alluvial gold wash machine) that is reflected in the backscatter signal of SAR data. This methodology can be applied to other study cases since a good understanding of the principles of change detection has been achieved in this chapter and complemented by Chaps. 16 through 21.

References

Aguirre GA, Robles RRC, Duarez FMG et al (2021) Dinámica de la pérdida de bosques en el sureste de la Amazonia peruana: Un estudio de caso en Madre de Dios. Ecosistemas 30:2175. https://doi.org/10.7818/ECOS.2175

Álvarez-Berríos N, L'Roe J, Naughton-Treves L (2021) Does formalizing artisanal gold mining mitigate environmental impacts? Deforestation evidence from the Peruvian Amazon. Environ Res Lett 16:64052. https://doi.org/10.1088/1748-9326/abede9

Asner GP, Tupayachi R (2017) Accelerated losses of protected forests from gold mining in the Peruvian Amazon. Environ Res Lett 12:94004. https://doi.org/10.1088/1748-9326/aa7dab

Asner GP, Llactayo W, Tupayachi R, Luna ER (2013) Elevated rates of gold mining in the Amazon revealed through high-resolution monitoring. Proc Natl Acad Sci U S A 110:18454–18459. https://doi.org/10.1073/pnas.1318271110

Ballère M, Bouvet A, Mermoz S et al (2021) SAR data for tropical forest disturbance alerts in French Guiana: benefit over optical imagery. Remote Sens Environ 252:112159. https://doi.org/10.1016/j.rse.2020.112159

Caballero Espejo J, Messinger M, Román-Dañobeytia F et al (2018) Deforestation and forest degradation due to gold mining in the Peruvian Amazon: a 34-year perspective. Remote Sens 10:1903. http://doi.org/10.20944/preprints201811.0113.v1

Canty MJ, Nielsen AA, Conradsen K, Skriver H (2020) Statistical analysis of changes in Sentinel-1 time series on the Google Earth Engine. Remote Sens 12:46. https://doi.org/10.3390/rs12010046

Csillik O, Asner GP (2020) Near-real time aboveground carbon emissions in Peru. PLoS One 15:e0241418. https://doi.org/10.1371/journal.pone.0241418

Flores-Anderson AI, Herndon KE, Thapa RB, Cherrington E (2019) The SAR handbook: comprehensive methodologies for forest monitoring and biomass estimation

Nicolau AP, Herndon K, Flores-Anderson A, Griffin R (2019) A spatial pattern analysis of forest loss in the Madre de Dios region, Peru. Environ Res Lett 14:124045. https://doi.org/10.1088/1748-9326/ab57c3

Nicolau AP, Flores-Anderson A, Griffin R et al (2021) Assessing SAR C-band data to effectively distinguish modified land uses in a heavily disturbed Amazon forest. Int J Appl Earth Obs Geoinf 94:102214. https://doi.org/10.1016/j.jag.2020.102214

RAISG (2020) Amazonia under pressure. https://atlas2020.amazoniasocioambiental.org/en. Accessed 25 Feb 2022

Yard EE, Horton J, Schier JG et al (2012) Mercury exposure among artisanal gold miners in Madre de Dios, Peru: a cross-sectional study. J Med Toxicol 8:441–448

Part VIII
Aquatic and Hydrological Applications

Earth Engine's global scope and long time series allow analysts to understand the water cycle in new and unique ways. These include surface water in the form of floods and river characteristics, long-term issues of water balance, and the detection of subsurface ground water.

Groundwater Monitoring with GRACE

40

A. J. Purdy and J. S. Famiglietti

Overview

The following tutorial details how to use observations from the Gravity Recovery and Climate Experiment (GRACE) to evaluate changes in groundwater storage for a large river basin. Here, you will learn how to apply remote sensing estimates of total water storage anomalies, land surface model output, and in situ observations to resolve groundwater storage changes in California's Central Valley. The following method has been applied to study water storage changes around the world, and can be ported to quantify groundwater storage change for major river basins.

Learning Outcomes

- Plotting changes in total water storage using GRACE.
- Mapping trends in water storage.
- Resolving changes in groundwater storage for a river basin.
- Import image collections and create image collections from assets.
- Create charts by reducing an `ImageCollection` with a feature geometry.

A. J. Purdy (✉)
University of San Francisco, San Francisco, CA, USA
e-mail: adamjpurdy@gmail.com; apurdy@usca.edu; adpurdy@csumb.edu

California State University, Monterey Bay Seaside, CA, USA

J. S. Famiglietti
Global Institute for Water Security, University of Saskatchewan, Saskatoon, SK, Canada
e-mail: jay.famiglietti@usask.ca

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_40

Helps if you know how to

- Use expressions to perform calculations on image bands (Chap. 9).
- Write a function and `map` it over an ImageCollection (Chap. 12).
- Fit linear and nonlinear functions with regression in an `ImageCollection` time series (Chap. 18).
- Use `ee.Join` to join one `ImageCollection` to another to compute differences (Chap. 21).
- Filter a `FeatureCollection` to obtain a subset (Chaps. 22 and 23).

40.1 Introduction to Theory

Since 2002, GRACE and the follow-on mission, GRACE-FO, have provided a new vantage to track changes in water resources (Tapley et al. 2004). GRACE holds the unique ability to directly track changes in total water storage anomalies (TWSa), according to the following equation:

$$\text{TWSa} = \text{CANa} + \text{SWa} + \text{SMa} + \text{SWEa} + \text{GWa} \tag{40.1}$$

where CANa is canopy water storage anomaly, SWa is the surface water anomaly, SMa is the soil moisture anomaly, SWEa is the snow water equivalent anomaly, and GWa is the groundwater storage anomaly.

By utilizing supplemental observations from other remote sensing platforms and land surface models and rearranging Eq. 40.1, scientists have been able to resolve changes in groundwater storage within major river basins around the planet (Famiglietti 2014). From Bangladesh (Purdy et al. 2019) and India (Rodell et al. 2009) to the Middle East (Voss et al. 2013) and the American Southwest (Castle et al. 2014), the problem of declining groundwater storage has emerged with varying levels of severity (Richey et al. 2015). Along with many other regions around the world, California shares an overreliance on groundwater (Famiglietti et al. 2011). This tutorial demonstrates the analytical steps to resolve groundwater storage changes using GRACE for California's Central Valley.

40.2 Practicum

40.2.1 Section 1: Exploring the Study Area

Evaluating changes in hydrologic storage requires examining change within a hydrologically connected system. Watersheds and basins represent areas of land where precipitation drains to a common point. We will use already-generated basins from the Watershed Boundary Dataset (WBD) to delineate the drainage area for California's Central Valley. The WBD includes hydrologic unit codes (HUCs) to identify connected basins within the United States.

In the following sections of code, we will load three basins by their unique four-digit HUCs and merge the basins together. To accomplish this task, we will use the `ee.Filter.inList` function to filter the `basins` variable by the `'huc4'` property, extracting three to a variable `basin`.

```
// Import Basins.
var basins = ee.FeatureCollection('USGS/WBD/2017/HUC04');

// Extract the 3 HUC 04 basins for the Central Valley.
var codes = ['1802', '1803', '1804'];
var basin = basins.filter(ee.Filter.inList('huc4', codes));

// Add the basin to the map to show the extent of our
analysis.
Map.centerObject(basin, 6);
Map.addLayer(basin, {
    color: 'green'
}, 'Central Valley Basins', true, 0.5);
```

40.2.1.1 Section 1.1: Map the Extent of Agriculture in the Region

To get a sense for the extent of agriculture in California, we can visualize all cultivated land in the Central Valley. This will map where the greatest need for water occurs.

```
var landcover = ee.ImageCollection('USDA/NASS/CDL')
    .filter(ee.Filter.date('2019-01-01', '2019-12-31'))
    .select('cultivated');

Map.addLayer(landcover.first().clip(basin), {}, 'Cropland',
true,
    0.5);
```

The extent of cultivated lands shows up as a translucent purple (Fig. 40.1).

40.2.1.2 Section 1.2: Load Reservoir Locations

California has over 150 reservoirs distributed across the state. These reservoirs vary in size and capacity, and the regions of the state that they support. For the Central

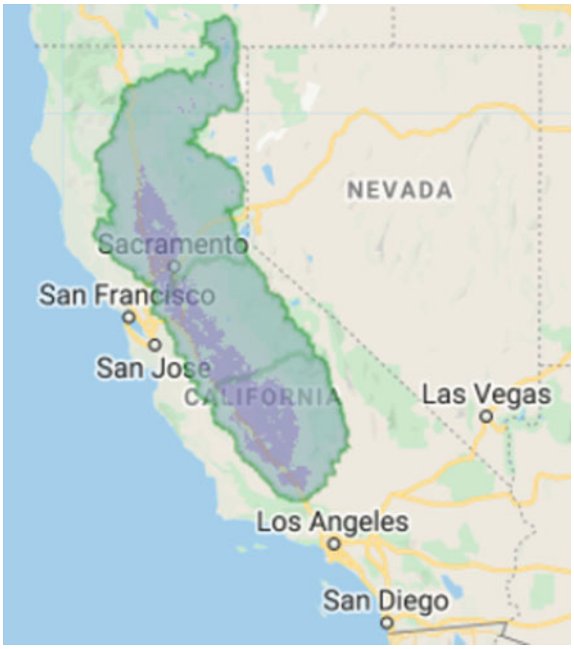


Fig. 40.1 California's Central Valley Basin, including agricultural lands

Valley, water conveyance infrastructure allows the transport of water from north to south. We will use our basin boundary to select the reservoirs within our basin to quantify changes in surface water storage. The list of reservoirs was gathered from the California Department of Water Resources' Data Exchange Center (CDEC). For an application in another study region, acquiring in situ surface water storage would be required to resolve that region's groundwater storage changes.

```
// This table was generated using the index from the CDEC
website
var res = ee.FeatureCollection(
    'projects/gee-book/assets/A2-1/ca_reservoirs_index');
// Filter reservoir locations by the Central Valley
geometry
var res_cv = res.filterBounds(basin);
Map.addLayer(res_cv, {
    'color': 'blue'
}, 'Reservoirs');
```

The blue dots that now appear on the map represent the distribution of water storage across the Central Valley. Water conveyance infrastructure and natural rivers deliver water to farms across the valley. Despite all these reservoirs, many water users continue to rely on groundwater to meet their needs. A 2011 study detailed the magnitude of this reliance using gravity-sensing satellites (Famiglietti et al. 2011). The next sections of this chapter reveal how to resolve groundwater storage changes using these methods.

Code Checkpoint A21a. The book's repository contains a script that shows what your code should look like at this point.

40.2.2 Section 2: Tracking Total Water Storage Changes in California with GRACE

GRACE can directly track changes in TWSa. Changes in TWSa indicate which regions are gaining or losing water.

40.2.2.1 Section 2.1: Import GRACE Data and Plot Changes in Total Water Storage in California

First, we will import the image collection and select the proper band to chart.

```
var GRACE =
ee.ImageCollection('NASA/GRACE/MASS_GRIDS/MASCON_CRI');
// Subset GRACE for liquid water equivalent dataset
var basinTWSa = GRACE.select('lwe_thickness');
```

The GRACE data imported here have already been processed to provide units of TWSa. The data contained in this dataset are units of "equivalent water thickness" anomalies. GRACE hydrologic data are presented as anomalies because GRACE does not directly observe the gravitational pull of only water. The gravity observed also includes Earth's surface (e.g., mountains). To disentangle the signal of water, we can look at changes relative to a longer term mean gravity signal. The anomalies represent the difference between a given month's observation and a multi-year mean. We will now plot TWSa for basins in a large part of California (Fig. 40.2).

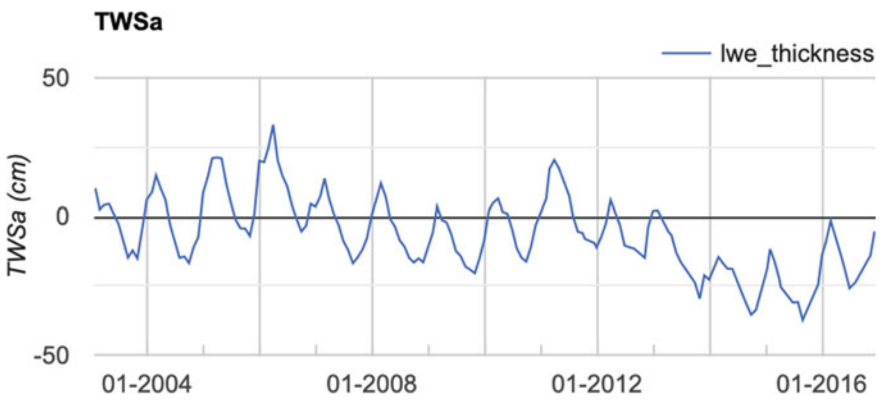


Fig. 40.2 TWSa for the Sacramento-San Joaquin Basin

```
// Make plot of TWSa for Basin Boundary
var TWSaChart = ui.Chart.image.series({
        imageCollection: basinTWSa.filter(ee.Filter.date(
            '2003-01-01', '2016-12-31')),
        region: basin,
        reducer: ee.Reducer.mean(),
    })
    .setOptions({
        title: 'TWSa',
        hAxis: {
            format: 'MM-yyyy'
        },
        vAxis: {
            title: 'TWSa (cm)'
        },
        lineWidth: 1,
    });
print(TWSaChart);
```

In the **Console**, you will see a plot of TWSa. Notice the seasonality and interannual variations in TWSa. Winter months reveal periods of maximum water storage due to snowpack, full reservoirs, and wet soil. Summer and early fall reveal less TWSa, as the snow has melted, reservoir water has been used, and soil is drying out. Additionally, summer months are periods when groundwater is extracted and used to supplement a limited surface water supply. Evidence of drought emerged through declining TWSa between 2006–2009 and 2012–2017.

Next, we will look at the trend in TWSa for the entire period of record.

40.2.2.2 Section 2.2: Estimate the Linear Trend in TWSa Over Time

As presented in Chap. 18, Earth Engine can fit linear models to time series data, with unique linear fits for each pixel based on the values through time. Consider the following linear model, where *et* is a random error:

$$pt = \beta 0 + \beta 1t + et(1) \tag{40.2}$$

This is the model behind the trendline added to the chart we just created. This model is useful for detrending data and reducing non-stationarity in the time series. The goal of the regression is to discover the values of the β's in each pixel.

To fit this trend model to the GRACE-based TWSa series using ordinary least squares, we can use the `linearRegression` reducer.

```
// Compute Trend for each pixel to map regions of most
change
var addVariables = function(image) {
    // Compute time in fractional years since the epoch.
    var date = ee.Date(image.get('system:time_start'));
    var years = date.difference(ee.Date('2003-01-01'),
'year');
    // Return the image with the added bands.
    return image
        // Add a time band.
        .addBands(ee.Image(years).rename('t').float())
        // Add a constant band.
        .addBands(ee.Image.constant(1));
};
var cvTWSa =
basinTWSa.filterBounds(basin).map(addVariables);
print(cvTWSa);
// List of the independent variable names
var independents = ee.List(['constant', 't']);

// Name of the dependent variable.
var dependent = ee.String('lwe_thickness');
// Compute a linear trend.  This will have two bands:
'residuals' and
// a 2x1 band called coefficients (columns are for
dependent variables).
var trend = cvTWSa.select(independents.add(dependent))

.reduce(ee.Reducer.linearRegression(independents.length(),
1));
```

The image of coefficients, computed below, is a two-band image in which each pixel contains values for $\beta 0$ and $\beta 1$. The $\beta 1$ value will represent the temporal slope for the GRACE mascon.

```
// Flatten the coefficients into a 2-band image
var coefficients = trend.select('coefficients')
    .arrayProject([0])
    .arrayFlatten([independents]);
```

Next, we can visualize the GRACE trends to capture the spatial scales on which GRACE can resolve TWSa. GRACE is adept at capturing these changes only for larger basins.

```
// Create a layer of the TWSa slope to add to the map
var slope = coefficients.select('t');
// Set visualization parameters to represent positive
(blue) & negative (red) trends
var slopeParams = {
    min: -3.5,
    max: 3.5,
    palette: ['red', 'white', 'blue']
};
Map.addLayer(slope.clip(basin), slopeParams, 'TWSa Trend',
true,
    0.75);
```

The slope layer reveals that the Tulare Basin (the southernmost basin in the Central Valley) experienced the largest negative changes in TWSa over the time period (Fig. 40.3). Darker reds indicate greater negative change, and blue represents positive change. This is a result of the region not receiving winter rain or snow and having the most junior surface water rights in the Central Valley.

The next steps in this chapter will review how to unpack the TWSa signal to resolve changes in groundwater storage anomalies for the basin.

Code Checkpoint A21b. The book's repository contains a script that shows what your code should look like at this point.

40.2.3 Section 3: Tracking Changes in Soil Water Storage and Snow Water Equivalent in California

The Global Land Data Assimilation System (GLDAS) utilizes multiple land surface models to globally resolve fluxes in storage of water (like soil moisture and

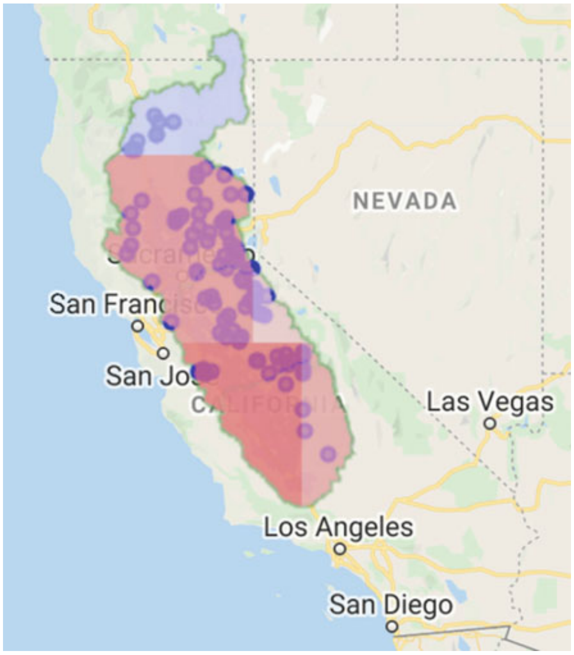


Fig. 40.3 Slope in TWSa for California. Darker reds indicate greater declines in total water storage. Blue represents increases in water storage. White represents no change in water storage

snow) and energy at a three-hour frequency (Rodell et al. 2004). An example of how to convert three-hourly GLDAS snow water equivalent to annual SWEa for 2003 can be found in script **A21s1** in the book's repository. Running the supplemental script is an optional part of this lab: it is added to provide clarity on how the image assets were created for each GLDAS variable in this chapter.

For the next analysis, you will be starting with a script that imports the GLDAS SMa and SWEa processed by the methods above. GLDAS estimates of soil moisture and snow water equivalent are resolved at a three-hour temporal frequency. Therefore, we have taken the time to reduce the GLDAS data to annual means from the three-hour estimates.

Additionally, we aggregated monthly GRACE observations to annual average estimates to improve the efficiency of running this analysis. More experienced users can adapt these methods to resolve monthly changes. However, it should be noted that to replicate the same methods at a monthly cadence would require the interpolation of missing months of GRACE observations. Please use the code starting point below, as the script imports the necessary assets to complete the final analysis.

40.2.3.1 Section 3.1: Load GLDAS Soil Moisture Images from an Asset to an Image Collection

Code Checkpoint A21c. The book's repository contains a script to use to begin this section. You will need to start with that script and paste code below into it.

When you run the script, you will see a number of assets being imported and an annual time series of GRACE. Additionally, the script is set to convert the list of annual mean soil moisture images to an `ImageCollection`.

```
var gldas_sm_list = ee.List([sm2003, sm2004, sm2005,
sm2006, sm2007,
    sm2008, sm2009, sm2010, sm2011, sm2012, sm2013, sm2014,
    sm2015, sm2016
]);
var sm_ic = ee.ImageCollection.fromImages(gldas_sm_list);
```

Before we compute groundwater storage anomalies from GRACE and GLDAS data following Eq. 40.1, we should inspect the units to ensure that our math is sound. In the **search bar** of Earth Engine, search for "GLDAS" and click on **GLDAS-2.1: Global Land Data Assimilation System**, then navigate to **Bands** to see what the units are for `'RootMoist_inst'` and `'SWE_inst'`.

The units for GLDAS are currently showing as kg/m^2. We need to convert the soil moisture and snow values to equivalent water depth units of centimeters. Define the following conversion variable and map this over the `ImageCollection`. As described in Chaps. 12 and 13, mapping over an `ImageCollection` is similar to running a loop: You apply the same function to each image and return the value back to the `ImageCollection`.

```
var kgm2_to_cm = 0.10;
var sm_ic_ts = sm_ic.map(function(img) {
    var date = ee.Date.fromYMD(img.get('year'), 1, 1);
    return
img.select('RootMoist_inst').multiply(kgm2_to_cm)
        .rename('SMa').set('system:time_start', date);
});
```

In addition to converting the units, the code renames the variable and sets properties such as `'system:time_start'`, which is necessary in Earth Engine to plot data and compare it with other image collections. Note that you might print out the variables `sm_ic` and `sm_ic_ts` to explore the differences between them. You should notice the new band name and properties (e.g., `'system:time_start'`).

Next, plot the data to evaluate soil moisture anomalies during the study period.

```
// Make plot of SMa for Basin Boundary
var SMaChart = ui.Chart.image.series({
        imageCollection: sm_ic_ts.filter(ee.Filter.date(
            '2003-01-01', '2016-12-31')),
        region: basin,
        reducer: ee.Reducer.mean(),
        scale: 25000
    })
    .setChartType('ScatterChart')
    .setOptions({
        title: 'Soil Moisture anomalies',
        trendlines: {
            0: {
                color: 'CC0000'
            }
        },
        hAxis: {
            format: 'MM-yyyy'
        },
        vAxis: {
            title: 'SMa (cm)'
        },
        lineWidth: 2,
        pointSize: 2
    });
print(SMaChart);
```

You may notice that SMa is of a similar magnitude to TWSa but is slightly out of phase with TWSa.

40.2.3.2 Section 3.2: Load GLDAS Snow Water Equivalent Images from an Asset to an Image Collection

Use similar code to load the snow water equivalent data to Earth Engine.

```
var gldas_swe_list = ee.List([swe2003, swe2004, swe2005,
swe2006,
    swe2007, swe2008, swe2009, swe2010, swe2011, swe2012,
    swe2013, swe2014, swe2015, swe2016
]);
var swe_ic = ee.ImageCollection.fromImages(gldas_swe_list);
```

Next, convert the snow values to equivalent water depth units of centimeters.

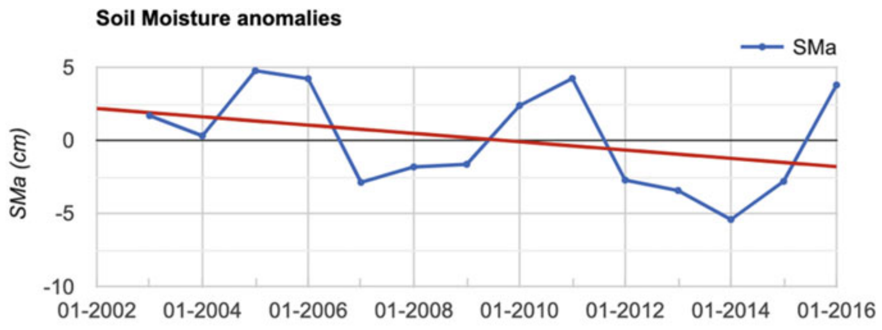
```
var swe_ic_ts = swe_ic.map(function(img) {
    var date = ee.Date.fromYMD(img.get('year'), 1, 1);
    return
img.select('SWE_inst').multiply(kgm2_to_cm).rename(
        'SWEa').set('system:time_start', date);
});
```

Next, we will visualize the new `ImageCollection`. If you did not do the previous step, your code here will not run.

```
// Make plot of SWEa for Basin Boundary
var SWEaChart = ui.Chart.image.series({
        imageCollection: swe_ic_ts.filter(ee.Filter.date(
            '2003-01-01', '2016-12-31')),
        region: basin,
        reducer: ee.Reducer.mean(),
        scale: 25000
    })
    .setChartType('ScatterChart')
    .setOptions({
        title: 'Snow Water Equivalent anomalies',
        trendlines: {
            0: {
                color: 'CC0000'
            }
        },
        hAxis: {
            format: 'MM-yyyy'
        },
        vAxis: {
            title: 'SWEa (cm)'
        },
        lineWidth: 2,
        pointSize: 2
    });
print(SWEaChart);
```

You successfully plotted soil moisture and snow water equivalent (Fig. 40.4). You may notice that SWEa is much smaller in magnitude than the other two variables.

Code Checkpoint A21d. The book's repository contains a script that shows what your code should look like at this point.



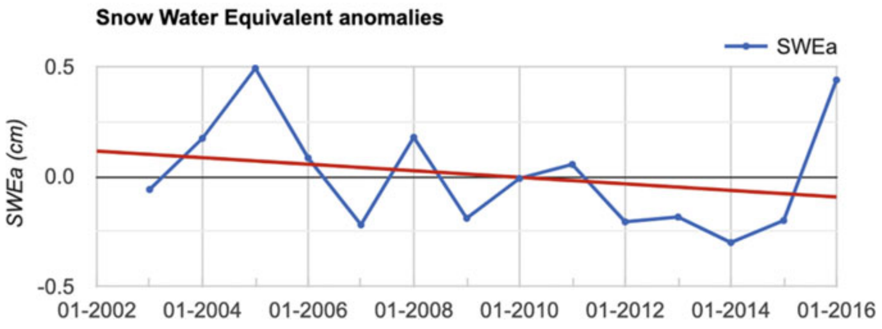


Fig. 40.4 Time-series charts of SMa and SWEa in units of equivalent water height (centimeters)

40.2.4 Section 4: Importing a Table of Surface Water Storage

Reservoir storage data from the California Data Exchange Center (CDEC) facilitated computing Surface Water storage anomalies (SWa) for the Sacramento-San Joaquin Basin. Surface water storage, unlike the other components of water storage, is not represented in land surface models. Instead, SWa is sourced from in situ observations. Here, the reservoir storage observations were summed for the Central Valley and then total converted to annual anomalies to directly compare with SWEa and SMa. Prior to reading the table of reservoir storage, we compute the area from the combined HUC8 basins.

```
// Extract geometry to convert time series of anomalies in
km3 to cm
var area_km2 = basin.geometry().area().divide(1000 * 1000);
var km_2_cm = 100000;
```

Next, the imported table `res_table` is converted to an `ImageCollection` of constant values to facilitate combining the data with GRACE and GLDAS-resolved water storage anomalies.

```
// Convert csv to image collection
var res_list = res_table.toList(res_table.size());
var yrs = res_list.map(function(ft) {
    return ee.Date.fromYMD(ee.Feature(ft).get('YEAR'), 1,
1);
});
var SWanoms = res_list.map(function(ft) {
    return
ee.Image.constant(ee.Feature(ft).get('Anom_km3'));
});
var sw_ic_ts = ee.ImageCollection.fromImages(
    res_list.map(
        function(ft) {
            var date =
ee.Date.fromYMD(ee.Feature(ft).get('YEAR'),
                1, 1);
            return ee.Image.constant(ee.Feature(ft).get(
                'Anom_km3')).divide(area_km2).multiply(
                km_2_cm).rename('SWa').set(
                'system:time_start', date);
        }
    )
);
```

Plot SWa in equivalent units of centimeters per year (Fig. 40.5).

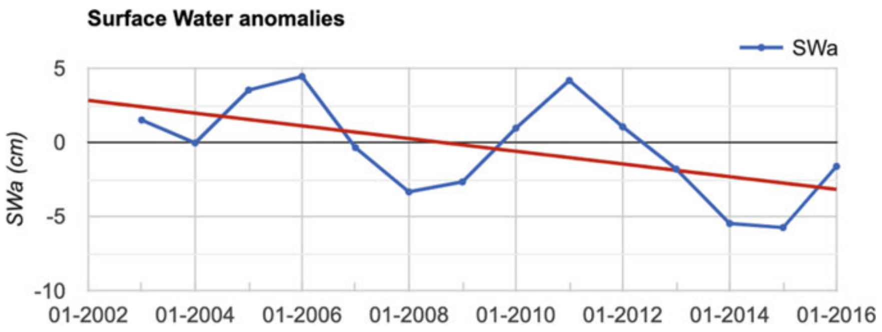


Fig. 40.5 Time-series chart of SWa in units of equivalent water height (centimeters)

```
// Create a time series of Surface Water Anomalies
var SWaChart = ui.Chart.image.series({
        imageCollection: sw_ic_ts.filter(ee.Filter.date(
            '2003-01-01', '2016-12-31')),
        region: basin,
        reducer: ee.Reducer.mean(),
        scale: 25000
    })
    .setChartType('ScatterChart')
    .setOptions({
        title: 'Surface Water anomalies',
        trendlines: {
            0: {
                color: 'CC0000'
            }
        },
        hAxis: {
            format: 'MM-yyyy'
        },
        vAxis: {
            title: 'SWa (cm)'
        },
        lineWidth: 2,
        pointSize: 2
    });
print(SWaChart);
```

The chart shows that surface water anomalies are of a similar magnitude to soil moisture anomalies. As expected, SWa decreases during each drought period in California (2006–2008 and 2012–2016). These reservoir storage declines show use is greater than inputs during each period.

Now, we will combine the previous datasets to resolve changes in groundwater during the period of record. Unfortunately, it's still hard to quantify change without having all the variables on one plot. It might be best to compute the differences via Eq. 40.1 from the introductory paragraph at the top of the document.

Code Checkpoint A21e. The book's repository contains a script that shows what your code should look like at this point.

40.2.5 Section 5: Combining Image Collections

Here, you will see how to combine multiple image collections and compute differences via an expression. We will start by joining the GLDAS image collections together. This is accomplished with the `ee.Join.inner` function.

```
// Combine GLDAS & GRACE Data to compute change in human
accessible water
var filter = ee.Filter.equals({
    leftField: 'system:time_start',
    rightField: 'system:time_start'
});
// Create the join.
var joindata = ee.Join.inner();
// Join GLDAS data
var firstJoin =
ee.ImageCollection(joindata.apply(swe_ic_ts, sm_ic_ts,
    filter));
var join_1 = firstJoin.map(function(feature) {
    return ee.Image.cat(feature.get('primary'),
feature.get(
        'secondary'));
});
print('Joined', join_1);
```

Next, we join the reservoir data.

```
// Repeat to append Reservoir Data now
var secondJoin = ee.ImageCollection(joindata.apply(join_1,
sw_ic_ts,
    filter));
var res_GLDAS = secondJoin.map(function(feature) {
    return ee.Image.cat(feature.get('primary'),
feature.get(
        'secondary'));
});
```

Lastly, we need to repeat this step by joining GRACE to the output from the last join.

```
// Repeat to append GRACE now
var thirdJoin =
ee.ImageCollection(joindata.apply(res_GLDAS, GRACE_yr,
    filter));
var GRACE_res_GLDAS = thirdJoin.map(function(feature) {
    return ee.Image.cat(feature.get('primary'),
feature.get(
        'secondary'));
});
```

Take a moment to print out the ImageCollection GRACE_res_GLDAS.

To resolve groundwater storage changes in the basin, one can rearrange Eq. 40.1 to solve for GWa. Here we assume canopy storage anomalies are very small relative to other storage components and ignore them in the equation below.

$$\mathrm{GWa} = \mathrm{TWSa} - \mathrm{SWa} - \mathrm{SMa} - \mathrm{SWEa} \tag{40.3}$$

To execute this step, we map an expression across an ImageCollection to produce a new variable named GWa.

```
// Compute groundwater storage anomalies
var GWa =
ee.ImageCollection(GRACE_res_GLDAS.map(function(img) {
    var date = ee.Date.fromYMD(img.get('year'), 1, 1);
    return img.expression(
        'TWSa - SWa - SMa - SWEa', {
            'TWSa': img.select('TWSa'),
            'SMa': img.select('SMa'),
            'SWa': img.select('SWa'),
            'SWEa': img.select('SWEa')
        }).rename('GWa').copyProperties(img, [
        'system:time_start'
    ]);
}));
print('GWa', GWa);
```

You can see how the variable img is used to extract bands from the combined ImageCollection and create a new one with just one band.

We'll plot this to see how groundwater storage is changing (Fig. 40.6).

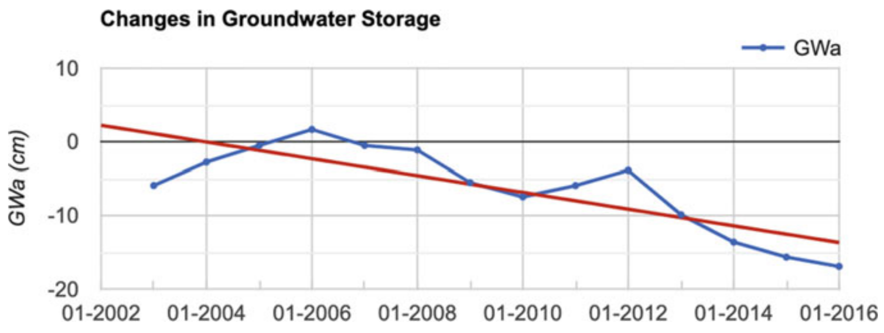


Fig. 40.6 Time-series chart of GWa in units of equivalent water height (centimeters)

```
// Chart Results
var GWaChart = ui.Chart.image.series({
        imageCollection: GWa.filter(ee.Filter.date('2003-
01-01',
            '2016-12-31')),
        region: basin,
        reducer: ee.Reducer.mean(),
        scale: 25000
    })
    .setChartType('ScatterChart')
    .setOptions({
        title: 'Changes in Groundwater Storage',
        trendlines: {
            0: {
                color: 'CC0000'
            }
        },
        hAxis: {
            format: 'MM-yyyy'
        },
        vAxis: {
            title: 'GWa (cm)'
        },
        lineWidth: 2,
        pointSize: 2
    });
print(GWaChart);
```

You can see how reliant California is on groundwater. The chart shows large declines during recent drought periods. Using the chart, you can estimate how much groundwater was used during the 2012–2016 drought period. In the **Console**, hover your mouse over the chart and jot down the value for GWa in 2012 and in 2016. You will use this information, in addition to the area and unit conversion, to estimate groundwater usage during this period in cubic kilometers.

```
// Now look at the values from the start of 2012 to the end
of 2016 drought.
// 2012 -3.874 cm --> 2016 -16.95 cm
// This is a ~13 cm / 100000 (cm/km) * Area 155407 km2 =
var loss_km3 = ee.Number(-3.874).subtract(-
16.95).divide(km_2_cm)
    .multiply(area_km2);
print('During the 2012-2016 drought, CA lost ', loss_km3,
    'km3 in groundwater');
```

Code Checkpoint A21f. The book's repository contains a script that shows what your code should look like at this point.

40.3 Synthesis

Assignment 1. This chapter provides a roadmap to monitor changes in groundwater storage at a basin scale using observations of TWSa from the GRACE satellites and hydrologic data from GLDAS. Now you can apply these methods to another river basin anywhere in the world.

40.4 Conclusion

In this chapter, we reviewed how GRACE observations can be used to estimate changes in water storage for a region of interest like California's Central Valley. Specifically, this chapter demonstrated how to combine equivalent water thickness observations from GRACE with model simulations of soil moisture, snow water equivalent, and in situ reservoir storage observations to quantify groundwater storage declines. Along the way, some advanced Earth Engine skills were explored, including creating an `ImageCollection` from a table and joining multiple image collections. Earth Engine users now have the skills and the knowledge of GRACE observations to apply this methodology to other regions around the world.

References

Castle SL, Thomas BF, Reager JT et al (2014) Groundwater depletion during drought threatens future water security of the Colorado River Basin. Geophys Res Lett 41:5904–5911. https://doi.org/10.1002/2014GL061055

Famiglietti JS (2014) The global groundwater crisis. Nat Clim Change 4:945–948. https://doi.org/10.1038/nclimate2425

Famiglietti JS, Lo M, Ho SL et al (2011) Satellites measure recent rates of groundwater depletion in California's Central Valley. Geophys Res Lett 38. https://doi.org/10.1029/2010GL046442

Purdy AJ, David CH, Sikder MS et al (2019) An open-source tool to facilitate the processing of GRACE observations and GLDAS outputs: an evaluation in Bangladesh. Front Environ Sci 7. https://doi.org/10.3389/fenvs.2019.00155

Richey AS, Thomas BF, Lo MH et al (2015) Quantifying renewable groundwater stress with GRACE. Water Resour Res 51:5217–5237. https://doi.org/10.1002/2015WR017349

Rodell M, Houser PR, Jambor U et al (2004) The global land data assimilation system. Bull Am Meteorol Soc 85:381–394. https://doi.org/10.1175/BAMS-85-3-381

Rodell M, Velicogna I, Famiglietti JS (2009) Satellite-based estimates of groundwater depletion in India. Nature 460:999–1002. https://doi.org/10.1038/nature08238

Tapley BD, Bettadpur S, Ries JC, Thompson PF, Watkins MM (2004) GRACE measurements of mass variability in the Earth system. Science 305(5683):503–505. https://doi.org/10.1126/science.1099192

Voss KA, Famiglietti JS, Lo M et al (2013) Groundwater depletion in the Middle East from GRACE with implications for transboundary water management in the Tigris-Euphrates-Western Iran region. Water Resour Res 49:904–914. https://doi.org/10.1002/wrcr.20078

Benthic Habitats

41

Dimitris Poursanidis, Aurélie C. Shapiro, and Spyridon Christofilakos

Overview

Shallow-water coastal benthic habitats, which can comprise seagrasses, sandy soft bottoms, and coral reefs are essential ecosystems, supporting fisheries, providing coastal protection, and sequestering 'blue' carbon. Multispectral satellite imagery, particularly with blue and green spectral bands, can penetrate clear, shallow water, allowing us to identify what lies on the seafloor. In terrestrial habitats, atmospheric and topographic corrections are important, whereas in shallow waters, it is essential to correct the effects of the water column, as different depths can change the reflectance observed by the satellite sensor. Once you know the water depth, you can accurately assess benthic habitats such as seagrass, sand, and coral. In this chapter, we will describe how to estimate water depth from high-resolution Planet data and map benthic habitats.

D. Poursanidis (✉)
Institute of Applied and Computational Mathematics, Foundation for Research and Technology Hellas, The Remote Sensing Lab, 100 N. Plastira Str., Vassilika Vouton, 70013, Heraklion, Greece
e-mail: dpoursanidis@iacm.forth.gr

A. C. Shapiro
Here+There Mapping Solutions, Berlin, Germany

S. Christofilakos
German Aerospace Center (DLR), Remote Sensing Technology Institute, Department of Photogrammetry and Image Analysis, Rutherfordstraße 2, 12489 Berlin, Germany

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_41

Learning Outcomes

- Separating land from water with a supervised classification.
- Removing surface sunglint and wave glint.
- Correcting for water depth to derive bottom surface reflectance through a regression approach.
- Identifying and classifying benthic habitats using machine learning.
- Developing regression models to estimate water depth using training data.
- Evaluating training data and model accuracy.

Helps if you know how to

- Import images and image collections, filter, and visualize (Part 1).
- Perform basic image analysis: select bands, compute indices, create masks, classify images (Part 2).
- Use `normalizedDifference` to calculate vegetation indices (Chap. 5).
- Use drawing tools to create points, lines, and polygons (Chap. 6).
- Perform a supervised Random Forest image classification (Chap. 6).
- Obtain accuracy metrics from classifications (Chap. 7).
- Use reducers to implement linear regression between image bands (Chap. 8).
- Filter a `FeatureCollection` to obtain a subset (Chaps. 22 and 23).
- Convert between raster and vector data (Chap. 23).

41.1 Introduction to Theory

Coastal benthic habitats include several ecosystems across the globe. One common element is the seagrass meadow, a significant component of coastal marine ecosystems (UNEP/MAP 2009). Seagrass meadows are among the most productive habitats in the coastal zone, performing essential ecosystem functions and providing essential ecosystem services (Duarte et al. 2013), such as water oxygenation and nutrient provision, seafloor and beach stabilization (as sediment is controlled and trapped within the rhizomes of the meadows), carbon burial, and nursery areas and refuge for commercial and endemic species (Boudouresque et al. 2012; Vassallo et al. 2013; Campagne et al. 2014).

However, seagrass meadows are experiencing a global decline due to intensive human activities and climate change (Boudouresque et al. 2009). Threats from climate change include sea surface temperature increases and sea level rise, as well as more frequent and intensive storms (Pergent et al. 2014). These threats represent a pressing challenge for coastal management and are predicted to have deleterious effects on seagrasses.

Several programs have focused on coastal seabed mapping, and a wide range of methods have been utilized for mapping seagrasses (Poursanidis et al. 2021; Borfecchia et al. 2013; Eugenio et al. 2015). Satellite remote sensing has been employed for the mapping of seagrass meadows and coral reefs in several areas (Goodman et al. 2011; Hedley et al. 2016; Knudby and Nordlund 2011; Koedsin et al. 2016; Lyons et al. 2012).

In this chapter, we will show you how to map coastal habitats using high-resolution Earth observation data from Planet with updated field data collected in the same period as the imagery acquisition. You will learn how to calculate coastal bathymetry using high-quality field data and machine learning regression methods. In the end, you will be able to adapt the code and use your own data to work in your own coastal area of interest, which can be tropical or temperate as long as there are clear waters up to 30 m deep.

41.2 Practicum

41.2.1 Section 1: Inputting Data

The first step is to define the data that you will work on. By uploading raster and vector data via the Earth Engine asset inventory, you will be able to analyze and process them through the Earth Engine API. The majority of satellite data are stored with values scaled by 10,000 and truncated in order to occupy less memory. In this setting, it is crucial to scale them back to their physical values before processing them.

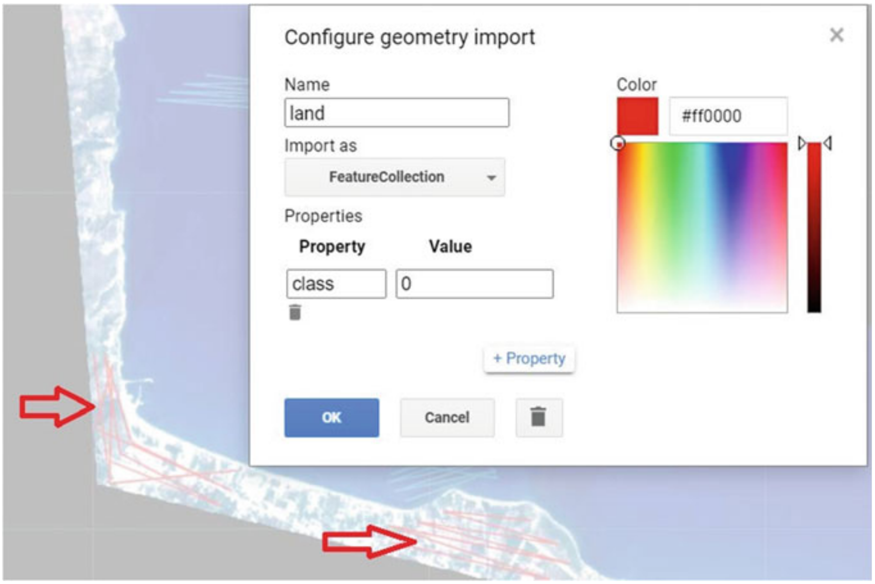
```
// Section 1
// Import and display satellite image.
var planet = ee.Image('projects/gee-book/assets/A2-
2/20200505_N2000')
    .divide(10000);

Map.centerObject(planet, 12);
var visParams = {
    bands: ['b3', 'b2', 'b1'],
    min: 0.17,
    max: 0.68,
    gamma: 0.8
};
Map.addLayer({
    eeObject: planet,
    visParams: visParams,
    name: 'planet initial',
    shown: true
});
```

Code Checkpoint A22a. The book's repository contains a script that shows what your code should look like at this point.

41.2.2 Section 2: Preprocessing Functions

Code Checkpoint A22b. The book's repository contains a script to use to begin this section. You will need to start with that script and paste code below into it. When you run the script, you will see several assets being imported for water, land, sunglint, and sandy patches to correct for the water column.

With the Planet imagery imported into Earth Engine, we now need to prepare for the classification and bathymetry procedures that will follow. The aim of the preprocessing is to correct or minimize spectral alterations due to physical conditions like sunglint, waves, and the water column. The `landmask` function below will remove the land area from our image in order to focus on the marine area of interest. This prevents the land reflectance values, which are relatively high, from biasing the main process of classification over dark water with low reflectance. To implement this step, we will perform a supervised classification to obtain the land/water mask. To prepare for the classification, we drew water and land geometry imports and set them as a `FeatureCollection` with property `'class'` and values 1 and 0, respectively (Fig. 41.1). We will employ the Normalized Difference Water Index (NDWI) (Gao 1996) using a Random Forest classifier (Breiman 2001). Due to the distinct spectral reflectances of terrain and water surfaces, there is no need to ensure a balanced dataset between the two classes. However, the size of the training dataset is still important and therefore, the more the better. For this task, we create 'line' geometries because we can get more training points than 'point' geometries with fewer clicks. For more information regarding generating training lines, points or polygons, please see Chap. 6.


Configure geometry import

Name: land

Import as: FeatureCollection

Properties

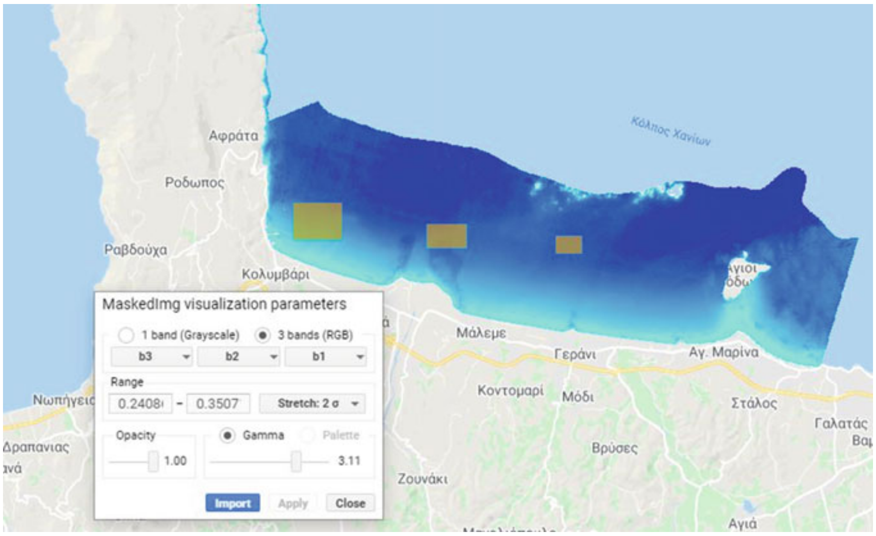
Property	Value
class	0

Color: #ff0000

Fig. 41.1 Labeling of land and water regions using lines in preparation for the Random Forests classification

```
// Section 2
// Mask based to NDWI and RF.
function landmask(img) {
    var ndwi = img.normalizedDifference(['b2', 'b4']);
    var training = ndwi.sampleRegions(land.merge(water),
['class'],
        3);
    var trained = ee.Classifier.smileRandomForest(10)
        .train(training, 'class');
    var classified = ndwi.classify(trained);
    var mask = classified.eq(1);

    return img.updateMask(mask);
}

var maskedImg = landmask(planet);

Map.addLayer(maskedImg, visParams, 'maskedImg', false);
```

Sunglint is a phenomenon that occurs when the sun angle and the sensor are positioned such that there is a mirror-like reflection at the water surface. In areas of glint, we cannot detect reflectance from the ocean floor. This will affect image



MaskedImg visualization parameters

○ 1 band (Grayscale) ● 3 bands (RGB)

b3 b2 b1

Range 0.2408 - 0.3507 Stretch: 2 σ

Opacity 1.00 ● Gamma ○ Palette 3.11

Import Apply Close

Fig. 41.2 Identification of areas with sunglint. Raising the gamma parameter during the visualization makes the phenomenon appear more intense, making it easier to draw the glint polygons

processing and needs to be corrected. The user adds polygons identifying areas of glint (Fig. 41.2), and these areas aid the linear model to remove sunglint areas in the whole image extend (Hedley et al. 2005).

```
// Sun-glint correction.
function sunglintRemoval(img) {
    var linearFit1 = img.select(['b4',
'b1']).reduceRegion({
        reducer: ee.Reducer.linearFit(),
        geometry: sunglint,
        scale: 3,
        maxPixels: 1e12,
        bestEffort: true,
    });
    var linearFit2 = img.select(['b4',
'b2']).reduceRegion({
        reducer: ee.Reducer.linearFit(),
        geometry: sunglint,
        scale: 3,
        maxPixels: 1e12,
        bestEffort: true,
    });
    var linearFit3 = img.select(['b4',
'b3']).reduceRegion({
        reducer: ee.Reducer.linearFit(),
        geometry: sunglint,
        scale: 3,
        maxPixels: 1e12,
        bestEffort: true,
    });

    var slopeImage = ee.Dictionary({
        'b1': linearFit1.get('scale'),
        'b2': linearFit2.get('scale'),
        'b3': linearFit3.get('scale')
    }).toImage();

    var minNIR = img.select('b4').reduceRegion({
        reducer: ee.Reducer.min(),
        geometry: sunglint,
        scale: 3,
        maxPixels: 1e12,
        bestEffort: true,
    }).toImage(['b4']);

    return img.select(['b1', 'b2', 'b3'])

.subtract(slopeImage.multiply((img.select('b4')).subtract(
            minNIR)))
        .addBands(img.select('b4'));
}
var sgImg = sunglintRemoval(maskedImg);
Map.addLayer(sgImg, visParams, 'sgImg', false);
```

Question 1. If you design more polygons for sunglint correction, can you see any improvement in the results visually and by examining values in pixels? Keep in mind that in already drawn rectangles (sunglint polygons), you cannot draw free-shaped polygons or lines. Try instead adding some more points or rectangles.

You can design more polygons, or select areas with severe or moderate sunglint to see how the site selection influences the final results.

The Depth Invariant Index (DIV) is a tool that creates a proxy image that helps minimize the bias of the spectral values due to the water column during classification and bathymetry procedures. Spectral signatures tend to be affected by the depth of the water column due to suspended material and the absorption of light. A typical result is that shallow seagrasses and deeper sandy seafloors will have similar spectral signatures. The correction of that error is based on the correlation between depth and logged bands (Lyzenga 1981).

In our example, the correction of water column alterations is based on the ratio of the green and blue bands, because of their higher penetrating properties compared to the red and near-infrared (NIR) bands. As in the previous functions, a requirement for this procedure is to identify sandy patches in different depth ranges. If needed, you can use the satellite image layer to do so.

Since we will use log values in the current step, it is crucial to transform all the negative values to positive before estimating DIV and since the majority of the values are in optically deep waters, the value 0.0001 will be assigned to values less than 0 in an attempt to avoid altering the pixels with positive values at the coastal zone. Moreover, a low-pass filter will be applied to normalize the observed noise that occurred during the previous steps (see Chap. 10).

```
// DIV procedure.
function kernel(img) {
    var boxcar = ee.Kernel.square({
        radius: 2,
        units: 'pixels',
        normalize: true
    });
    return img.convolve(boxcar);
}

function makePositive(img) {
    return img.where(img.lte(0), 0.0001);
}
```

```
function div(img) {
    var band1 = ee.List(['b1', 'b2', 'b3', 'b1', 'b2']);
    var band2 = ee.List(['b3', 'b3', 'b2', 'b2', 'b1']);
    var nband = ee.List(['b1b3', 'b2b3', 'b3b2', 'b1b2',
'b2b1']);

    for (var i = 0; i < 5; i += 1) {
        var x = band1.get(i);
        var y = band2.get(i);
        var z = nband.get(i);

        var imageLog = img.select([x, y]).log();

        var covariance = imageLog.toArray().reduceRegion({
            reducer: ee.Reducer.covariance(),
            geometry: DIVsand,
            scale: 3,

            maxPixels: 1e12,
            bestEffort: true,
        });

        var covarMatrix =
ee.Array(covariance.get('array'));
        var var1 = covarMatrix.get([0, 0]);
        var var2 = covarMatrix.get([1, 1]);
        var covar = covarMatrix.get([0, 1]);

        var a =
var1.subtract(var2).divide(covar.multiply(2));
        var attenCoeffRatio =
a.add(((a.pow(2)).add(1)).sqrt());

        var depthInvariantIndex = img.expression(
            'image1 - (image2 * coeff)', {
                'image1': imageLog.select([x]),
                'image2': imageLog.select([y]),
                'coeff': attenCoeffRatio
            });
```

Question 2. How can the selection of sandy patches influence the final result?

Sandy patches can also include signals from sparse seagrass or other types of benthic cover. Select different areas of variable depths and therefore different spectral reflectance to explore the changes in the final DIV result.

Code Checkpoint A22c. The book's repository contains a script that shows what your code should look like at this point.

41.2.3 Section 3: Supervised Classification

To create accurate classifications, it is beneficial for the training data set to be created from in situ data. This is especially important in marine remote sensing, due to dynamic, varying ecosystems. In our example, all the reference data were acquired during scientific dives. The reference data consist of three classes: SoftBottom for sandy patches, rockyBottom for rocky patches, and pO for posidonia-seagrass patches. Here, we provide a dataset that will be split using a 70–30% partitioning strategy into training and validation data (see Chap. 6) and is already pre-loaded in the assets of this book.

```
// Section 3, classification
// Import of reference data and split.
var softBottom = ee.FeatureCollection(
    'projects/gee-book/assets/A2-2/SoftBottom');
var rockyBottom = ee.FeatureCollection(
    'projects/gee-book/assets/A2-2/RockyBottom');
var pO = ee.FeatureCollection('projects/gee-book/assets/A2-
2/PO');

var sand = ee.FeatureCollection.randomPoints(softBottom,
150).map(
    function(s) {
        return s.set('class', 0);
    }).randomColumn();
var sandT = sand.filter(ee.Filter.lte('random',
0.7)).aside(print,
    'sand training');
var sandV = sand.filter(ee.Filter.gt('random',
0.7)).aside(print,
    'sand validation');
Map.addLayer(sandT, {
    color: 'yellow'
}, 'Sand Training', false);
Map.addLayer(sandV, {
    color: 'yellow'
}, 'Sand Validation', false);

var hard = ee.FeatureCollection.randomPoints(rockyBottom,
79).map(
    function(s) {
        return s.set('class', 1);
    }).randomColumn();
var hardT = hard.filter(ee.Filter.lte('random',
0.7)).aside(print,
    'hard training');
var hardV = hard.filter(ee.Filter.gt('random',
0.7)).aside(print,
    'hard validation');
Map.addLayer(hardT, {
    color: 'red'
}, 'Rock Training', false);
Map.addLayer(hardV, {
    color: 'red'
}, 'Rock Validation', false);
```

```
var posi = pO.map(function(s) {
        return s.set('class', 2);
    })
    .randomColumn('random');
var posiT = posi.filter(ee.Filter.lte('random',
0.7)).aside(print,
    'posi training');
var posiV = posi.filter(ee.Filter.gt('random',
0.7)).aside(print,
    'posi validation');
Map.addLayer(posiT, {
    color: 'green'
}, 'Posidonia Training', false);
Map.addLayer(posiV, {
    color: 'green'
}, 'Posidonia Validation', false);
```

For this procedure, we chose the `ee.Classifier.libsvm` classifier because of its established performance in aquatic environments (Poursanidis et al. 2018; da Silveira et al. 2021). The function below does the classification and also estimates overall user's and producer's accuracy (see Chap. 7):

```
// Classification procedure.
function classify(img) {
    var mergedT = ee.FeatureCollection([sandT, hardT,
posiT])
    .flatten();
    var training = img.sampleRegions(mergedT, ['class'],
3);
    var trained = ee.Classifier.libsvm({
        kernelType: 'RBF',
        gamma: 1,
        cost: 500
    }).train(training, 'class');
    var classified = img.classify(trained);

    var mergedV = ee.FeatureCollection([sandV, hardV,
posiV])
        .flatten();
    var accuracyCol = classified.unmask().reduceRegions({
        collection: mergedV,
        reducer: ee.Reducer.first(),
        scale: 10
    });
    var classificationErrorMatrix =
accuracyCol.errorMatrix({
        actual: 'class',
        predicted: 'first',
        order: [0, 1, 2]
    });
    var classNames = ['soft_bot', 'hard_bot', 'seagrass'];
    var accuracyOA = classificationErrorMatrix.accuracy();
    var accuraccyCons = ee.Dictionary.fromLists({
        keys: classNames,
        values:
classificationErrorMatrix.consumersAccuracy()
            .toList()
            .flatten()
    });
    var accuracyProd = ee.Dictionary.fromLists({
        keys: classNames,
        values:
classificationErrorMatrix.producersAccuracy()
            .toList()
            .flatten()
    });

    var classificationErrormatrixArray =
classificationErrorMatrix
        .array();

    var arrayToDatatable = function(array) {
        var classesNames = ee.List(classNames);

        function toTableColumns(s) {
            return {
                id: s,
                label: s,
                type: 'number'
            };
        }
        var columns = classesNames.map(toTableColumns);

        function featureToTableRow(f) {
            return {
                c: ee.List(f).map(function(c) {
                    return {
                        v: c
                    };
                })
            };
        }
        var rows = array.toList().map(featureToTableRow);
        return ee.Dictionary({
            cols: columns,
            rows: rows
        });
    };

    var dataTable =
arrayToDatatable(classificationErrormatrixArray)
        .evaluate(function(dataTable) {
            print('------------- Error matrix -------------
',
                ui.Chart(dataTable, 'Table')
                .setOptions({
                    pageSize: 15
                }),
                'rows: reference, cols: mapped');
        });
    print('Overall Accuracy', accuracyOA);
    print('Users accuracy', accuraccyCons);
    print('Producers accuracy', accuracyProd);
    return classified;
}

var svmClassification = classify(divImg);
var svmVis = {
    min: 0,
    max: 2,
    palette: ['ffffbf', 'fc8d59', '91cf60']
};
Map.addLayer(svmClassification, svmVis, 'classification');
```

Code Checkpoint A22d. The book's repository contains a script that shows what your code should look like at this point.

41.2.4 Section 4: Bathymetry by Random Forests Regression

For the bathymetry procedure, we will exploit the `setOutputMode('REGRESSION')` option of `ee.Classifier.smile RandomForest`. For this example, reference data came from a sonar that was mounted on a boat. In contrast to the classification accuracy assessment, the accuracy assessment of bathymetry is based on R^2 and the root-mean-square error (RMSE).

With regard to visualization of the resulting bathymetry, we have to consider the selection of colors and their physical meanings. In the classification, which is a categorical image, we use a diverging palette, while in bathymetry, which shows a continuous value, we should use a sequential palette. Tip: ‘cold’ colors better convey depth. For the satellite-derived bathymetry, we use pre-loaded assets with the in situ depth measurement. The quality of these measurements is crucial to the success of the classifier.

```
// Section 4, Bathymetry
// Import and split training and validation data for the
bathymetry.
var depth = ee.FeatureCollection(
    'projects/gee-book/assets/A2-
2/DepthDataTill09072020_v2');
depth = depth.randomColumn();
var depthT = depth.filter(ee.Filter.lte('random', 0.7));
var depthV = depth.filter(ee.Filter.gt('random', 0.7));
Map.addLayer(depthT, {
    color: 'black'
}, 'Depth Training', false);
Map.addLayer(depthV, {
    color: 'gray'
}, 'Depth Validation', false);
```

So that every pixel contains at most one measurement, the vector depth assets are rasterized prior to using them for the regression.

```
function vector2image(vector) {
    var rasterisedVectorData = vector
        .filter(ee.Filter.neq('Depth',
            null)) // Filter out NA depth values.
        .reduceToImage({
            properties: ['Depth'],
            reducer: ee.Reducer.mean()
        });
    return (rasterisedVectorData);
}

var depthTImage = vector2image(depthT)
    .aside(Map.addLayer, {
        color: 'white'
    }, 'Depth Training2', false);
var depthVImage = vector2image(depthV)
    .aside(Map.addLayer, {
        color: 'white'
    }, 'Depth Validation2', false);
```

Finally, we need to enter down and execute the function to calculate the satellite derived bathymetry function, based on the Random Forest classifier.

```
function rfbathymetry(img) {
    var training = img.sampleRegions({
        collection: depthT,
        scale: 3
    });

    var regclass = ee.Classifier.smileRandomForest(15)
        .train(training, 'Depth');
    var bathyClass = img

.classify(regclass.setOutputMode('REGRESSION')).rename(
            'Depth');
    var sdbEstimate = bathyClass.clip(depthV);

    // Prepare data by putting SDB estimated data and in
situ data
    // in one image to compare them afterwards.
    var imageI = ee.Image.cat([sdbEstimate, depthVImage]);
    // Calculate covariance.
    // Print together, so that they appear in the same
output.
    print('R²', rSqr, 'RMSE', rmse);
    return bathyClass;
}

var rfBathymetry = rfbathymetry(divImg);
var bathyVis = {
    min: -50,
    max: 0,
    palette: ['084594', '2171b5', '4292c6', '6baed6',
        '9ecae1', 'c6dbef', 'deebf7', 'f7fbff'
    ]
};
Map.addLayer(rfBathymetry, bathyVis, 'bathymetry');
    var covariance = imageI.toArray().reduceRegion({
        reducer: ee.Reducer.covariance(),
        geometry: depthV,
        scale: 3,
        bestEffort: true,
        maxPixels: 1e9
    });
    var covarMatrix = ee.Array(covariance.get('array'));
    var rSqr = covarMatrix.get([0, 1]).pow(2)
        .divide(covarMatrix.get([0, 0])
            .multiply(covarMatrix.get([1, 1])));
    var deviation = depthVImage.select('mean')
        .subtract(sdbEstimate.select('Depth')).pow(2);
    var rmse = ee.Number(deviation.reduceRegion({
            reducer: ee.Reducer.mean(),
            geometry: depthV,
            scale: 3,
            bestEffort: true,
            maxPixels: 1e12
        }).get('mean'))
        .sqrt();
```

Question 3. Does the selection of different color ramps lead to misinterpretations?

By choosing different color ramps for the same data set, you can see how visual interpretation can be changed based on color. Trying different color ramps will reveal how some can better visualize the final results for further use in maritime spatial planning activities, while others, including commonly seen rainbow ramps, can lead to erroneous decisions.

Code Checkpoint A22e. The book’s repository contains a script that shows what your code should look like at this point.

41.3 Synthesis

With what you learned in this chapter, you can analyze Earth observation data—here, specifically from the Planet Cubesat constellation—to create your own map of coastal benthic habitats and coastal bathymetry for a specific case study. Feel free to test out the approach in another part of the world using your own data or open-access data, or use your own training data for a more refined classification model.

You can add your own point data to the map, collected via a fieldwork campaign or by visually interpreting the imagery, and merge with the training data to improve a classification, or clean areas that need to be removed by drawing polygons and masking them in the classification.

For the bathymetry, you can select different calibration/validation ratio approaches and test the optimum ratio of splitting to see how it influences the final bathymetry map. You can also add a smoothing filter to create a visually smoother image of coastal bathymetry.

41.4 Conclusion

Many coastal habitats, especially seagrass meadows, are dynamic ecosystems that change over time and can be lost through natural and anthropogenic causes.

The power of Earth Engine lies in its cloud-based, lightning-fast, automated approach to workflows, especially its processing power. This process would take days when performed offline in traditional remote sensing software, especially over large areas. And the Earth Engine approach is not only fast but also consistent: The same method can be applied to images from different dates to assess habitat changes over time, both gain and loss.

The availability of Planet imagery allows us to use a high-resolution product. Natively, Earth Engine hosts the archives of Sentinel-2 and Landsat data. The Landsat archive spans from 1984 to the present, while Sentinel-2, a higher-resolution product, is available from 2017 to today. All of this imagery can be used in the same workflow in order to map benthic habitats and monitor their changes over time, allowing us to understand the past of the coastal seascape and envision its future.

References

Borfecchia F, Micheli C, Carli F et al (2013) Mapping spatial patterns of Posidonia oceanica meadows by means of Daedalus ATM airborne sensor in the coastal area of Civitavecchia (Central Tyrrhenian Sea, Italy). Remote Sens 5:4877–4899. https://doi.org/10.3390/rs5104877

Boudouresque CF, Bernard G, Pergent G et al (2009) Regression of Mediterranean seagrasses caused by natural processes and anthropogenic disturbances and stress: a critical review. Bot Mar 52:395–418. http://doi.org/10.1515/BOT.2009.057

Boudouresque CF, Bernard G, Bonhomme P et al (2012) Protection and conservation of Posidonia Oceanica meadows. RAMOGE and RAC/SPA

Breiman L (2001) Random forests. Mach Learn 45:5–32. https://doi.org/10.1023/A:1010933404324

Campagne CS, Salles JM, Boissery P, Deter J (2014) The seagrass Posidonia oceanica: ecosystem services identification and economic evaluation of goods and benefits. Mar Pollut Bull 97:391–400. https://doi.org/10.1016/j.marpolbul.2015.05.061

da Silveira CBL, Strenzel GMR, Maida M et al (2021) Coral reef mapping with remote sensing and machine learning: a nurture and nature analysis in marine protected areas. Remote Sens 13:2907. https://doi.org/10.3390/rs13152907

Duarte CM, Kennedy H, Marbà N, Hendriks I (2013) Assessing the capacity of seagrass meadows for carbon burial: current limitations and future strategies. Ocean Coast Manag 83:32–38. https://doi.org/10.1016/j.ocecoaman.2011.09.001

Eugenio F, Marcello J, Martin J (2015) High-resolution maps of bathymetry and benthic habitats in shallow-water environments using multispectral remote sensing imagery. IEEE Trans Geosci Remote Sens 53:3539–3549. https://doi.org/10.1109/TGRS.2014.2377300

Gao BC (1996) NDWI—a normalized difference water index for remote sensing of vegetation liquid water from space. Remote Sens Environ 58:257–266. https://doi.org/10.1016/S0034-4257(96)00067-3

Goodman J, Purkis S, Phinn SR (2011) Coral reef remote sensing: a guide for mapping, monitoring and management. Springer, Berlin

Hedley JD, Harborne AR, Mumby PJ (2005) Simple and robust removal of sun glint for mapping shallow-water benthos. Int J Remote Sens 26:2107–2112. https://doi.org/10.1080/01431160500034086

Hedley JD, Roelfsema CM, Chollett I et al (2016) Remote sensing of coral reefs for monitoring and management: a review. Remote Sens 8:118. https://doi.org/10.3390/rs8020118

Knudby A, Nordlund L (2011) Remote sensing of seagrasses in a patchy multi-species environment. Int J Remote Sens 32:2227–2244. https://doi.org/10.1080/01431161003692057

Koedsin W, Intararuang W, Ritchie RJ, Huete A (2016) An integrated field and remote sensing method for mapping seagrass species, cover, and biomass in Southern Thailand. Remote Sens 8:292. https://doi.org/10.3390/rs8040292

Lyons MB et al (2012) Long term land cover and seagrass mapping using Landsat and object-based image analysis from 1972 to 2010 in the coastal environment of South East Queensland, Australia. ISPRS J Photogrammetry Remote Sens 71:34–46. https://doi.org/10.1016/j.isprsjprs.2012.05.002

Lyzenga DR (1981) Remote sensing of bottom reflectance and water attenuation parameters in shallow water using aircraft and Landsat data. Int J Remote Sens 2:71–82. https://doi.org/10.1080/01431168108948342

Pergent G, Bazairi H, Bianchi CN et al (2014) Climate change and Mediterranean seagrass meadows: a synopsis for environmental managers. Mediterr Mar Sci 15:462–473. http://doi.org/10.12681/mms.621

Poursanidis D, Topouzelis K, Chrysoulakis N (2018) Mapping coastal marine habitats and delineating the deep limits of the Neptune's seagrass meadows using very high resolution Earth observation data. Int J Remote Sens 39:8670–8687. https://doi.org/10.1080/01431161.2018.1490974

Poursanidis D, Traganos D, Teixeira L, Shapiro A, Muaves L (2021) Cloud-native seascape mapping of Mozambique's Quirimbas National Park with Sentinel-2. Remote Sens Ecol Conserv 7(2):275–291. https://zslpublications.onlinelibrary.wiley.com/doi/full/10.1002/rse2.187

UNEP/MAP (2009) State of the Mediterranean marine and coastal environment. In: Ecological applications, pp 1047–1056

Vassallo P, Paoli C, Rovere A, Montefalcone M, Morri C, Bianchi CN (2013) The value of the seagrass Posidonia oceanica: a natural capital assessment. Mar Pollut Bull 75(1–2):157–167. https://doi.org/10.1016/j.marpolbul.2013.07.044

Surface Water Mapping

42

K. Markert, G. Donchyts, and A. Haag

Overview

In this chapter, you will learn the step-by-step implementation of an efficient and robust approach for mapping surface water. You will also learn how the extracted surface water information can be used in conjunction with historical surface water information to extract flooded areas. This chapter will focus mostly on the use of Sentinel-1 Synthetic Aperture Radar (SAR) data, but the approaches apply to both SAR and optical remotely sensed data.

Learning Outcomes

- Applying Otsu thresholding techniques for surface water mapping.
- Understanding the considerations of global versus adaptive histogram sampling.
- Implementing an adaptive histogram sampling approach.
- Extracting flooded areas from a surface water map.

Helps if you know how to

- Import images and image collections, filter, and visualize (Chaps. 2 and 3).
- Create a graph using `ui.Chart` (Chap. 4).

K. Markert · G. Donchyts
Google, Mountain View, USA
e-mail: kmarkert@google.com

G. Donchyts
e-mail: dgena@google.com

A. Haag (✉)
Deltares, Delft, The Netherlands
e-mail: arjen.haag@deltares.nl

- Write a function and `map` it over an `ImageCollection` (Chap. 12).
- Understand basics of working with Synthetic Aperture Radar images (Chap. 39).

42.1 Introduction to Theory

Flooding impacts more people than any other environmental hazard, and flood exposure is expected to increase in the future (Tellman et al. 2021). Remote sensing data plays a pivotal role in mapping historical flood zones and producing spatial maps of flood events that can be used to guide response efforts (Oddo and Bolten 2019). Oftentimes, flood maps need to be created and delivered to disaster managers within hours of image acquisition. Thus, computationally efficient approaches are required to reduce latency. Furthermore, these approaches need to produce accurate results without increasing processing time.

Image thresholding is an efficient method for mapping surface water (Schumann et al. 2009). Among the numerous methods available for image thresholding, a popular one is 'Otsu's method' (Otsu 1979). Otsu's method is a histogram-based thresholding approach where the inter-class variance between two classes, a foreground class and a background class, is maximized. As this approach assumes that only two classes are present within an image, which is rarely the case, methods have been developed (Donchyts et al. 2016; Cao et al. 2019) to constrain histogram sampling to areas that are more likely to represent a bimodal histogram of water/no water. Otsu's method applied on such a constrained histogram provides a more accurate estimation of a water threshold without sacrificing the computational efficiency of the method.

This chapter explores surface water mapping using Otsu's method and walks through an adaptive thresholding technique initially developed by Donchyts et al. 2016 and applied on optical imagery, then adapted by Markert et al. 2020 for detecting surface water in SAR imagery. Furthermore, the resulting surface water map will be compared with the Joint Research Centre's (JRC) Global Surface Water dataset (Pekel et al. 2016) to extract the flooded areas. This chapter will focus on the use of Sentinel-1 (S1) SAR data, but the concepts apply to other satellite imagery where we can distinguish water.

42.2 Practicum

42.2.1 Otsu Thresholding

Otsu's method is widely used for determining the optimal threshold of an image with two classes. In this section, we will explore the use of Otsu's method for segmenting water using an image-wide histogram (also known as a global histogram).

We will start by accessing Sentinel-1 data. We will focus our analysis on Southeast Asia, which experiences yearly flooding and so provides plenty of good test cases. To do this, we will assign the Sentinel-1 collection to a variable and filter by space, time, and metadata properties to get our image for processing.

```
// Define a point in Cambodia to filter by location.
var point = ee.Geometry.Point(104.9632, 11.7686);

Map.centerObject(point, 11);

// Get the Sentinel-1 collection and filter by space/time.
var s1Collection = ee.ImageCollection('COPERNICUS/S1_GRD')
    .filterBounds(point)
    .filterDate('2019-10-05', '2019-10-06')
    .filter(ee.Filter.eq('orbitProperties_pass',
'ASCENDING'))
    .filter(ee.Filter.eq('instrumentMode', 'IW'));

// Grab the first image in the collection.
var s1Image = s1Collection.first();
```

Now, we can add our image to the map.

```
// Add the Sentinel-1 image to the map.
Map.addLayer(s1Image, {
    min: -25,
    max: 0,
    bands: 'VV'
}, 'Sentinel-1 image');
```

The map should now have a Sentinel-1 image in Cambodia that looks like Fig. 42.1.

Now that we have our image, we can begin our processing to extract surface water information using Otsu's threshold. Otsu's thresholding algorithm uses histograms, so we will need to create one for the pixels of the image. Otsu's method works on a single band of values, so we will use the VV band from Sentinel-1, as seen in Fig. 42.1. Earth Engine allows us to easily calculate a histogram using the reducer `ee.Reducer.histogram` applied across the image using the `reduceRegion` operation:

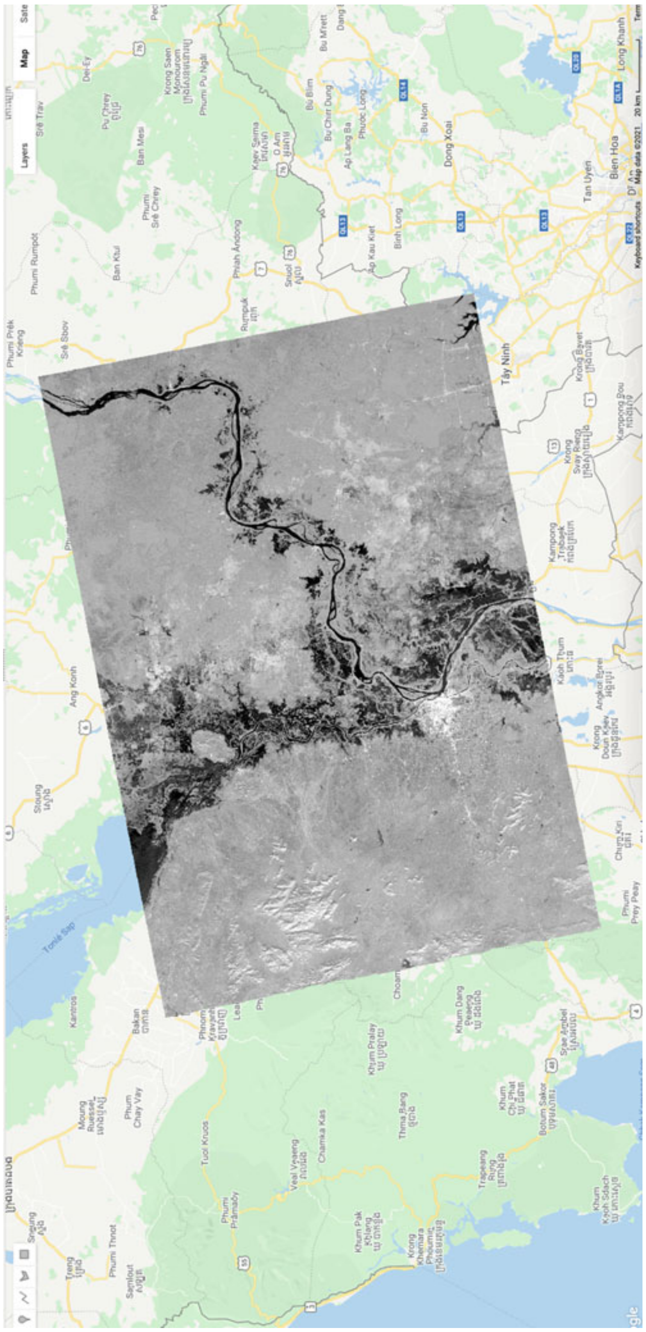


Fig. 42.1 Sentinel-1 VV image from October 5, 2019 highlighting a flooding event in Cambodia

```
// Specify band to use for Otsu thresholding.
var band = 'VV';

// Define a reducer to calculate a histogram of values.
var histogramReducer = ee.Reducer.histogram(255, 0.1);

// Reduce all of the image values.
var globalHistogram = ee.Dictionary(
    s1Image.select(band).reduceRegion({
        reducer: histogramReducer,
        geometry: s1Image.geometry(),
        scale: 90,
        maxPixels: 1e10
    }).get(band)
);

// Extract out the histogram buckets and counts per bucket.
var x = ee.List(globalHistogram.get('bucketMeans'));
var y = ee.List(globalHistogram.get('histogram'));

// Define a list of values to plot.
var dataCol = ee.Array.cat([x, y], 1).toList();

// Define the header information for data.
var columnHeader = ee.List([
    [
    {
        label: 'Backscatter',
        role: 'domain',
        type: 'number'
    },
    {
        label: 'Values',
        role: 'data',
        type: 'number'
    }, ]
]);

// Concat the header and data for plotting.
var dataTable = columnHeader.cat(dataCol);
```

```
// Create plot using the ui.Chart function with the
dataTable.
// Use 'evaluate' to transfer the server-side table to the
client.
// Define the chart and print it to the console.
dataTable.evaluate(function(dataTableClient) {
    var chart = ui.Chart(dataTableClient)
        .setChartType('AreaChart')
        .setOptions({
            title: band + ' Global Histogram',
            hAxis: {
                title: 'Backscatter [dB]',
                viewWindow: {
                    min: -35,
                    max: 15
                }
            },
            vAxis: {
                title: 'Count'
            }
        });
    print(chart);
});
```

After running that code, you should see the chart in Fig. 42.2 printed in the **Console**. This is the histogram of values across the image.

In the histogram, the data points are heavily concentrated around −9 dB. We can see a small peak of low backscatter values around −22 dB; these are the likely

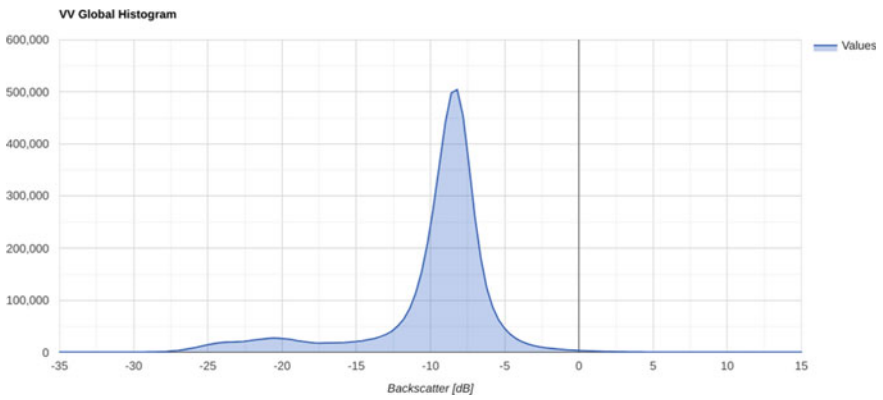


Fig. 42.2 Histogram of values from the Sentinel-1 VV image from October 5, 2019

open-water values. The SAR signal bounces off large, smooth surfaces like water, so likely open-water values are low, and non-water values are high.

How should we split the histogram into two parts to create a high-quality partition of the image into two classes? That is the job of Otsu's thresholding algorithm. Earth Engine does not have a built-in function for Otsu's method, so we will create a function that implements the algorithm's logic. This function takes in a histogram as input, applies the algorithm, and returns a single value where Otsu's method suggests breaking the histogram into two parts. (More information about Otsu's method can be found in the 'For Further Reading' section of this book.)

```
function otsu(histogram) {
    // Make sure histogram is an ee.Dictionary object.
    histogram = ee.Dictionary(histogram);
    // Extract relevant values into arrays.
    var counts = ee.Array(histogram.get('histogram'));
    var means = ee.Array(histogram.get('bucketMeans'));
    // Calculate single statistics over arrays
    var size = means.length().get([0]);
    var total = counts.reduce(ee.Reducer.sum(),
[0]).get([0]);
    var sum =
means.multiply(counts).reduce(ee.Reducer.sum(), [0])
        .get([0]);
    var mean = sum.divide(total);
    // Compute between sum of squares, where each mean
partitions the data.

    var indices = ee.List.sequence(1, size);
    var bss = indices.map(function(i) {
        var aCounts = counts.slice(0, 0, i);
        var aCount = aCounts.reduce(ee.Reducer.sum(), [0])
            .get([0]);
        var aMeans = means.slice(0, 0, i);
        var aMean = aMeans.multiply(aCounts)
            .reduce(ee.Reducer.sum(), [0]).get([0])
            .divide(aCount);
        var bCount = total.subtract(aCount);
        var bMean = sum.subtract(aCount.multiply(aMean))
            .divide(bCount);
        return aCount.multiply(aMean.subtract(mean).pow(2))
            .add(

bCount.multiply(bMean.subtract(mean).pow(2)));
    });
    // Return the mean value corresponding to the maximum
BSS.
    return means.sort(bss).get([-1]);
}
```

When the threshold is calculated, it can be applied to the imagery, and we can inspect where it falls within our histogram. The following code creates a new array of values and checks where the threshold is so that we can see how the algorithm performed.

```
// Apply otsu thresholding.
var globalThreshold = otsu(globalHistogram);
print('Global threshold value:', globalThreshold);

// Create list of empty strings that will be used for
annotation.
var thresholdCol = ee.List.repeat('', x.length());
// Find the index where the bucketMean equals the
threshold.
var threshIndex = x.indexOf(globalThreshold);
// Set the index to the annotation text.
thresholdCol = thresholdCol.set(threshIndex, 'Otsu
Threshold');
```

```
// Redefine the column header information with annotation
column.
columnHeader = ee.List([
    [
    {
        label: 'Backscatter',
        role: 'domain',
        type: 'number'
    },
    {
        label: 'Values',
        role: 'data',
        type: 'number'
    },
    {
        label: 'Threshold',
        role: 'annotation',
        type: 'string'
    }]
]);

// Loop through the data rows and add the annotation
column.
dataCol = ee.List.sequence(0,
x.length().subtract(1)).map(function(
i) {
    i = ee.Number(i);
    var row = ee.List(dataCol.get(i));
    return row.add(ee.String(thresholdCol.get(i)));
});

// Concat the header and data for plotting.
dataTable = columnHeader.cat(dataCol);

// Create plot using the ui.Chart function with the
dataTable.
// Use 'evaluate' to transfer the server-side table to the
client.
// Define the chart and print it to the console.
dataTable.evaluate(function(dataTableClient) {
    // loop through the client-side table and set empty
strings to null
    for (var i = 0; i < dataTableClient.length; i++) {
        if (dataTableClient[i][2] === '') {
            dataTableClient[i][2] = null;
        }
    }
    var chart = ui.Chart(dataTableClient)
        .setChartType('AreaChart')
        .setOptions({
            title: band +
                ' Global Histogram with Threshold
annotation',
            hAxis: {
                title: 'Backscatter [dB]',
                viewWindow: {
                    min: -35,
                    max: 15
                }
            },
            vAxis: {
                title: 'Count'
            },
            annotations: {
                style: 'line'
            }
        });
    print(chart);
});
```

Once you have run the above code, you should see another histogram chart that looks like Fig. 42.3. Note the addition of the text and light vertical line indicating the location of the threshold value.

We can see that the threshold is around −15 dB, which is about halfway between the two peaks. We can now apply that threshold on the imagery and inspect how the extracted water looks compared to the original image. Using the code below, we apply the threshold and add the water image to the map (Fig. 42.4).

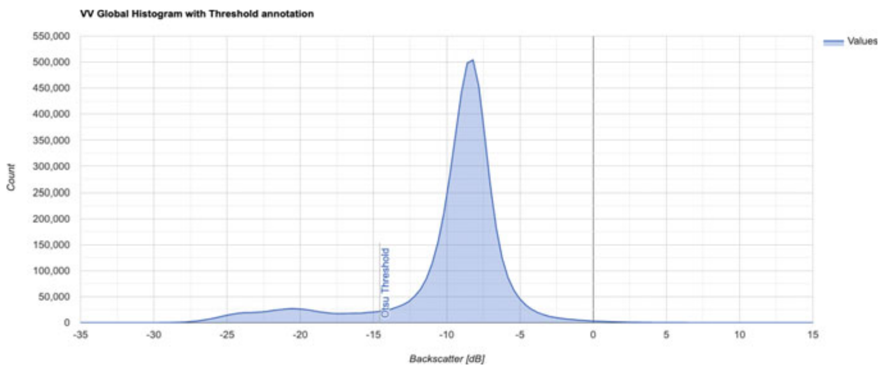


Fig. 42.3 Histogram of values from the Sentinel-1 VV image with the threshold calculated using Otsu's method

```
// Apply the threshold on the image to extract water.
var globalWater = s1Image.select(band).lt(globalThreshold);

// Add the water image to the map and mask 0 (no-water)
values.
Map.addLayer(globalWater.selfMask(),
    {
        palette: 'blue'
    },
    'Water (global threshold)');
```

The results look promising. The blue areas overlap with the low backscatter (specular reflectance) that is representative of open water in C-band SAR imagery. However, upon closer inspection we can see that the extracted water overestimates in some areas (Fig. 42.5).

We see an overestimation as large local errors may be introduced when calculating a constant threshold for distinguishing water from land when using an image-wide histogram. It is due to this issue that algorithms have been developed to constrain the histogram sampling and estimate a more locally contextual threshold.

Code Checkpoint A23a. The book's repository contains a script that shows what your code should look like at this point.

Question 1. Do some reading on Otsu's method (the Wikipedia page has a good description). How does Otsu's threshold work? What underlying assumptions does Otsu's threshold make?

Question 2. Based on the results and your understanding of Otsu's thresholding, why do you think the calculated histogram overestimated water areas?

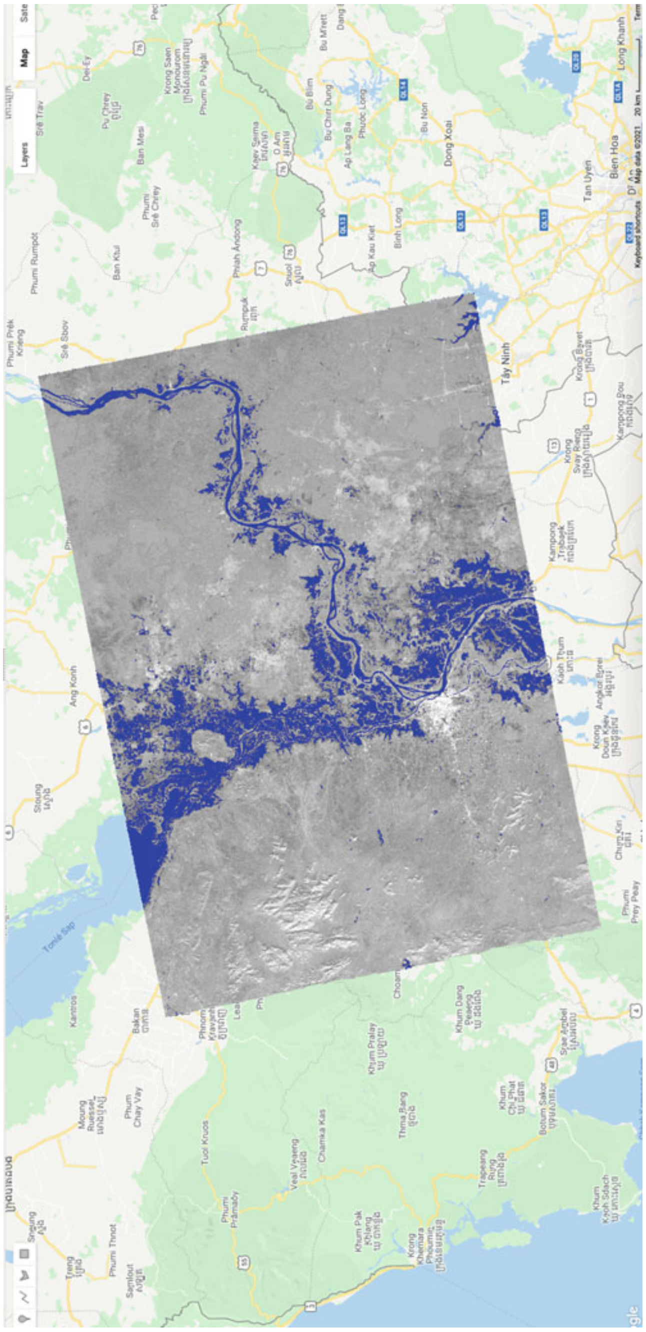


Fig. 42.4 Extracted surface water from Otsu's method for the Sentinel-1 VV image from October 5, 2019, highlighting a flooding event in Cambodia

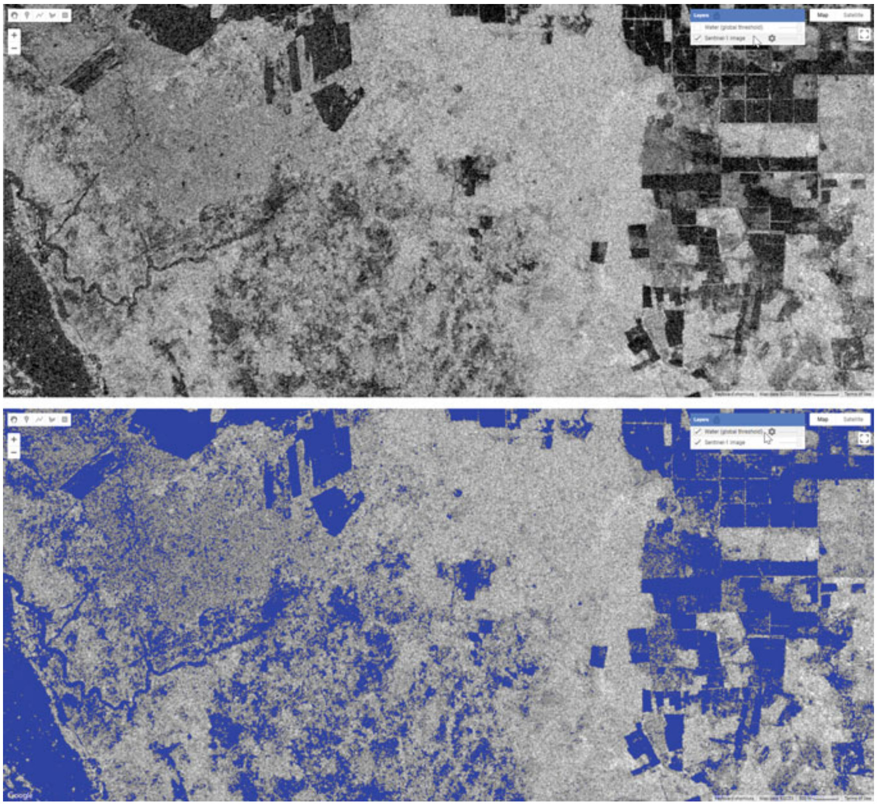


Fig. 42.5 Close-up inspection of extracted surface water. The extracted water is toggled on and off to illustrate where global Otsu thresholding overestimated water

42.2.2 Adaptive Thresholding

Surface water usually constitutes only a small fraction of the overall land cover within an Earth observation image. This makes it harder to apply threshold-based methods to extract water. The challenge is to establish a varying threshold that can be derived automatically. In images that show flooding, like the one from the previous section, this limitation is not so significant. Nevertheless, this section walks through an adaptive thresholding technique designed to overcome the challenges of using a global threshold.

The method we will discuss was developed by Donchyts et al. (2016) and applied to the Modified Normalized Difference Water Index (MNDWI) from Landsat 8 imagery. The algorithm finds edges within the image, buffers the areas around the identified edges, and uses the buffered area to sample a histogram for Otsu thresholding. This approach assumes that the edges detected are from water. The result is a bimodal histogram from the area around water edges that can be used to calculate a refined threshold. The overall workflow of the algorithm is shown in Fig. 42.6.

This approach was refined by Markert et al. (2020), where the main change is that instead of calculating the edges on the raw values (from an index or otherwise), an initial segmentation threshold is provided to create a binary image as input for the edge detection. This overcomes issues with SAR speckle and other artifacts, as well as with any edges being defined from other classes that are present in imagery (e.g., urban areas or forests). The defined edges are then filtered by length to omit small edges that can occur and can skew the histogram sampling. This requires that a few parameters be tuned, namely the initial threshold, edge length, and buffer size. Here, we define a few of those parameters.

```
// Define parameters for the adaptive thresholding.
// Initial estimate of water/no-water for estimating the
edges
var initialThreshold = -16;
// Number of connected pixels to use for length
calculation.
var connectedPixels = 100;
// Length of edges to be considered water edges.
var edgeLength = 20;
// Buffer in meters to apply to edges.
var edgeBuffer = 300;
// Threshold for canny edge detection.
var cannyThreshold = 1;
// Sigma value for gaussian filter in canny edge detection.
var cannySigma = 1;
// Lower threshold for canny detection.
var cannyLt = 0.05;
```

With these parameters defined, we can begin the process of constraining the histogram sampling.

```
// Get preliminary water.
var binary = s1Image.select(band).lt(initialThreshold)
    .rename('binary');

// Get projection information to convert buffer size to
pixels.
var imageProj = s1Image.select(band).projection();
```

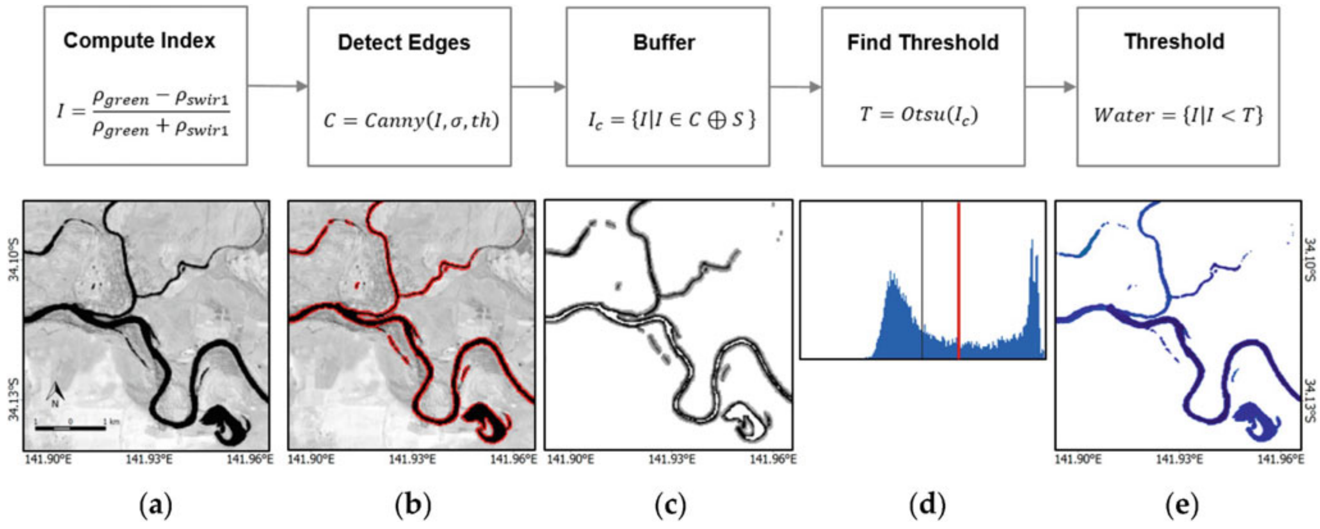


Fig. 42.6 Workflow of adaptive thresholding techniques. Figure taken from Donchyts et al. (2016) under the Creative Commons Attribution License

```
// Get canny edges.
var canny = ee.Algorithms.CannyEdgeDetector({
    image: binary,
    threshold: cannyThreshold,
    sigma: cannySigma
});

// Process canny edges.

// Get the edges and length of edges.
var connected = canny.updateMask(canny).lt(cannyLt)
    .connectedPixelCount(connectedPixels, true);

// Mask short edges that can be noise.
var edges = connected.gte(edgeLength);

// Calculate the buffer in pixel size.
var edgeBufferPixel =
ee.Number(edgeBuffer).divide(imageProj
    .nominalScale());

// Buffer the edges using a dilation operation.
var bufferedEdges =
edges.fastDistanceTransform().lt(edgeBufferPixel);

// Mask areas not within the buffer .
var edgeImage =
s1Image.select(band).updateMask(bufferedEdges);
```

Now that we have the edge information and the data to sample processed, we can visually inspect what the algorithm is doing. Here, we will display the calculated edges as well as the buffered edges to highlight which data is being sampled.

```
// Add the detected edges and buffered edges to the map.
Map.addLayer(edges, {
    palette: 'red'
}, 'Detected water edges');
var edgesVis = {
    palette: 'yellow',
    opacity: 0.5
};
Map.addLayer(bufferedEdges.selfMask(), edgesVis,
    'Buffered water edges');
```

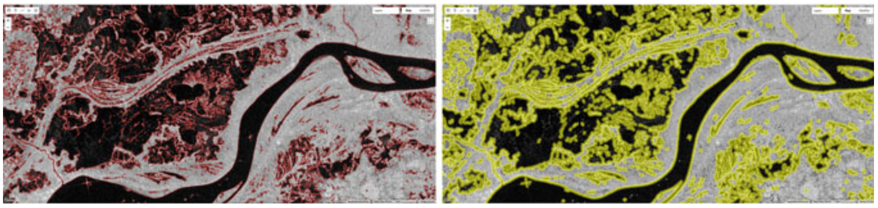


Fig. 42.7 Results from the water edge detection process, where the edges are shown in red (top image) and the buffered edges highlighting the sampling regions in yellow (bottom image)

You should now have the data added to the map, which should look like the images in Fig. 42.7 when zoomed in.

At this point, we have our regions that we want to sample that are more representative of a bimodal histogram, and we have masked out areas that we don't want to sample. Now, we can calculate the histogram as before and make a plot (Fig. 42.8).

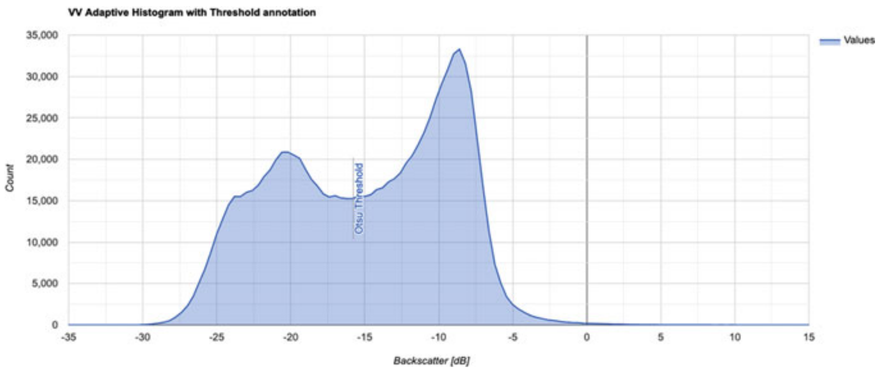


Fig. 42.8 Histogram of values from the Sentinel-1 VV image using the adaptive thresholding

```
// Reduce all of the image values.
var localHistogram = ee.Dictionary(
    edgeImage.reduceRegion({
        reducer: histogramReducer,
        geometry: s1Image.geometry(),
        scale: 90,
        maxPixels: 1e10
    }).get(band)
);

// Apply otsu thresholding.
var localThreshold = otsu(localHistogram);
print('Adaptive threshold value:', localThreshold);

// Extract out the histogram buckets and counts per bucket.
var x = ee.List(localHistogram.get('bucketMeans'));
var y = ee.List(localHistogram.get('histogram'));

// Define a list of values to plot.
var dataCol = ee.Array.cat([x, y], 1).toList();

// Concat the header and data for plotting.
var dataTable = columnHeader.cat(dataCol);

// Create list of empty strings that will be used for
annotation.
var thresholdCol = ee.List.repeat('', x.length());
// Find the index that bucketMean equals the threshold.
var threshIndex = x.indexOf(localThreshold);
// Set the index to the annotation text.
thresholdCol = thresholdCol.set(threshIndex, 'Otsu
Threshold');

// Redefine the column header information now with
annotation col.
columnHeader = ee.List([
    [
    {
        label: 'Backscatter',
        role: 'domain',
        type: 'number'
    },
    {
        label: 'Values',
        role: 'data',
        type: 'number'
    },
    {
        label: 'Threshold',
        role: 'annotation',
        type: 'string'
    }]
]);

// Loop through the data rows and add the annotation col.
dataCol = ee.List.sequence(0,
x.length().subtract(1)).map(function(
i) {
    i = ee.Number(i);
    var row = ee.List(dataCol.get(i));
    return row.add(ee.String(thresholdCol.get(i)));
});

// Concat the header and data for plotting.
dataTable = columnHeader.cat(dataCol);

// Create plot using the ui.Chart function with the
dataTable.
// Use 'evaluate' to transfer the server-side table to the
client.
// Define the chart and print it to the console.
dataTable.evaluate(function(dataTableClient) {
    // Loop through the client-side table and set empty
strings to null.
    for (var i = 0; i < dataTableClient.length; i++) {
        if (dataTableClient[i][2] === '') {
            dataTableClient[i][2] = null;
        }
    }
    var chart = ui.Chart(dataTableClient)
        .setChartType('AreaChart')
        .setOptions({
            title: band +
                ' Adaptive Histogram with Threshold
annotation',
            hAxis: {
                title: 'Backscatter [dB]',
                viewWindow: {
                    min: -35,
                    max: 15
                }
            },
            vAxis: {
                title: 'Count'
            },
            annotations: {
                style: 'line'
            }
        });
    print(chart);
});
```

We can see from the histogram that we have two distinct peaks. This meets the assumption of Otsu thresholding that only two classes are present within an image, which allows the algorithm to more accurately calculate the threshold for water. The last thing left to do is to apply the calculated adaptive threshold on the imagery and add it to the map (Fig. 42.9).

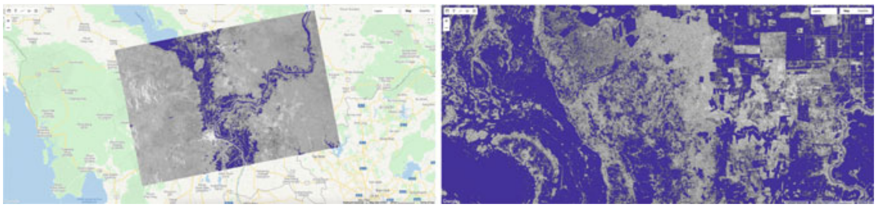


Fig. 42.9 Extracted surface water using the adaptive Otsu thresholding method for Sentinel-1 VV image (top image) and close-up inspection of extracted surface water for almost the same area as in Sect. 42.2.1 (bottom image)

```
// Apply the threshold on the image to extract water.
var localWater = s1Image.select(band).lt(localThreshold);

// Add the water image to the map and mask 0 (no-water)
values.
Map.addLayer(localWater.selfMask(),
    {
        palette: 'darkblue'
    },
    'Water (adaptive threshold)');
```

It can be seen from the resulting images that the adaptive thresholding technique produces a reasonable surface water map. Furthermore, the resulting threshold value was −15.799, as compared to −14.598 from the global thresholding. A lower threshold in this case means less surface water area extracted. However, less surface water area does not necessarily mean more accuracy. There needs to be a balance between producer's and user's accuracy.

Now that we have a surface water map and we are moderately confident that it represents the actual surface water for that day, we can begin to identify flooded areas by differencing our map with historical information.

Code Checkpoint A23b. The book's repository contains a script that shows what your code should look like at this point.

Question 3. Why do we apply an initial threshold to the SAR imagery prior to detecting edges? What happens to the detected edges if we do not apply an initial threshold? Explain why you are or are not getting a difference in detected edges. Recall that we defined some parameters for the Canny edge detection. Show a comparison.

Question 4. Compare the threshold calculated with the adaptive technique to the global threshold. In your own words, explain why the two thresholds are different.

Question 5. Change the parameters used for the adaptive thresholding to see how the results change. Which parameters is the algorithm most sensitive to?

42.2.3 Extracting Flood Areas

Up to this point, we have been mapping surface water, which includes permanent and seasonal water that was observed by the sensor. What we need to do now is to identify areas from our image that are considered permanent water. There are typically two approaches to mapping flooded areas with a thematic surface water map: (1) comparing pre- and post-event images to estimate changes; or (2) comparing extracted surface water with historically observed permanent water.

To achieve the goal of flood mapping, we will use the historical JRC Global Surface Water dataset to define permanent water and then find the difference to extract flooded areas. We already have our post-event surface water map. Now, we need to access and use the JRC data.

```
// Get the previous 5 years of permanent water.

// Get the JRC historical yearly dataset.
var jrc = ee.ImageCollection('JRC/GSW1_3/YearlyHistory')
    // Filter for historical data up to date of interest.
    .filterDate('1985-01-01', s1Image.date())
    // Grab the 5 latest images/years.
    .limit(5, 'system:time_start', false);
```

Because this data is a yearly classification of permanent and seasonal water, we need to reclassify the imagery to just permanent water.

```
var permanentWater = jrc.map(function(image) {
        // Extract out the permanent water class.
        return image.select('waterClass').eq(3);
        // Reduce the collection to get information on if a
pixel has
        // been classified as permanent water in the past 5
years.
    }).sum()
    // Make sure we have a value everywhere.
    .unmask(0)
    // Get an image of 1 if permanent water in the past 5
years, otherwise 0.
    .gt(0)
    // Mask for only the water image we just calculated.
    .updateMask(localWater.mask());
```

```
// Add the permanent water layer to the map.
Map.addLayer(permanentWater.selfMask(),
    {
        palette: 'royalblue'
    },
    'JRC permanent water');
```

The final thing we need to do is apply a simple differencing between the surface water map from Sentinel-1 and the JRC permanent water (Fig. 42.10).

```
// Find areas where there is not permanent water, but water
is observed.
var floodImage = permanentWater.not().and(localWater);

// Add flood image to map.
Map.addLayer(floodImage.selfMask(), {
    palette: 'firebrick'
}, 'Flood areas');
```

There are nuances associated with comparing optically derived water information (like from JRC) with SAR water maps. For example, any surface that is large enough and smooth can 'look' like water in SAR imagery because of specular reflectance and can be wrongly classified as a flooded area. Examples of this are airports, exposed channel beds, and highways. It should also be noted that there is a component of seasonal flooding—flooding that occurs every year and is expected. Currently, our flood map contains areas of seasonal flooding as well. Therefore, to accurately map the "abnormal" area of a flood, we'd also have to account for seasonal patterns. Lastly, rivers are in constant flux, changing patterns, and even change due to flooding events, so comparing historical observations with flooding events may yield some areas that have changed. Therefore, comparing pre- and post-event imagery from the same sensor is best. However, it is challenging to define events in seasonal flooding (such as this case), making a pre- and post-event comparison a little more complicated.

Code Checkpoint A23c. The book's repository contains a script that shows what your code should look like at this point.

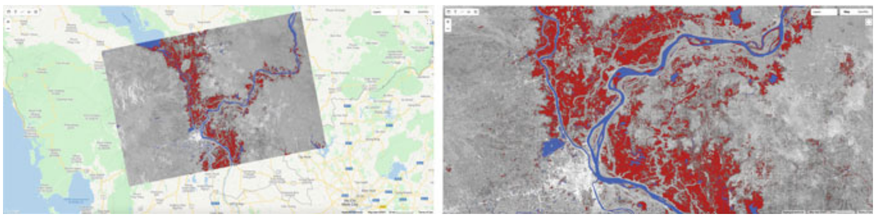


Fig. 42.10 Extracted flood areas by comparing the calculated surface water map against the JRC permanent water data. Bottom image shows a close-up of the flood map around the confluence of the Mekong and Tonle Sap rivers near Phnom Penh, Cambodia

42.3 Synthesis

In this chapter, we covered a common image segmentation method, Otsu's thresholding, and applied it to map surface water using Sentinel-1 SAR imagery. Furthermore, we illustrated an image processing technique to constrain the histogram sampling for input into the Otsu thresholding method. Lastly, we created a flood map from the segmented surface water map using historical permanent surface water data. You should now have a good grasp on the Otsu thresholding technique for surface water mapping, understand the considerations of global versus localized histogram sampling, be able to implement an adaptive histogram sampling approach, and take a surface water map and convert it to a flood map.

Assignment 1. Identify a flooding event of interest (a good source is https://floodlist.com) and walk through the process of creating a flood map for the event you chose. Do you notice anything different with the resulting flood map? Make note of the identified threshold value. Does the threshold value represent water versus no-water areas? Keep in mind the physical properties of SAR—some areas naturally have low backscatter like water does.

Assignment 2. In your own words, describe the difference between a surface water map and a flood map. Conceptually, what do you need to take into consideration when extracting flood areas?

Assignment 3. Find a pre-event Sentinel-1 image for the case we have gone through, extract surface water from the pre-event image, and compare it to the post-event image. Do you find differences in the flood areas derived from pre/post-comparison versus historical comparison?

Assignment 4. Refactor the code to make the adaptive thresholding algorithm and flood mapping into a callable function that can be mapped over an image collection.

42.4 Conclusion

It should be noted that Sentinel-1 SAR imagery ideally should undergo preprocessing to remove the effects of terrain and speckle in imagery, as in Mullissa et al. (2021), before applying surface water mapping algorithms. However, SAR preprocessing is outside the scope of this chapter.

There are many more sophisticated algorithms for mapping surface water from satellite imagery (such as Mayer et al. 2021). The application illustrated in this chapter is meant to highlight a practical workflow for mapping surface water and floods that can be implemented in Earth Engine.

References

Cao H, Zhang H, Wang C, Zhang B (2019) Operational flood detection using Sentinel-1 SAR data over large areas. Water (switzerland) 11:786. https://doi.org/10.3390/w11040786

Donchyts G, Schellekens J, Winsemius H et al (2016) A 30 m resolution surface water mask including estimation of positional and thematic differences using Landsat 8, SRTM and OpenStreetMap: a case study in the Murray-Darling basin, Australia. Remote Sens 8:386. https://doi.org/10.3390/rs8050386

Markert KN, Markert AM, Mayer T et al (2020) Comparing Sentinel-1 surface water mapping algorithms and radiometric terrain correction processing in Southeast Asia utilizing Google Earth Engine. Remote Sens 12:2469. https://doi.org/10.3390/RS12152469

Mayer T, Poortinga A, Bhandari B et al (2021) Deep learning approach for Sentinel-1 surface water mapping leveraging Google Earth Engine. ISPRS Open J Photogramm Remote Sens 2:100005. https://doi.org/10.1016/j.ophoto.2021.100005

Mullissa A, Vollrath A, Odongo-Braun C et al (2021) Sentinel-1 SAR backscatter analysis ready data preparation in Google Earth Engine. Remote Sens 13:1954. https://doi.org/10.3390/rs13101954

Oddo PC, Bolten JD (2019) The value of near real-time Earth observations for improved flood disaster response. Front Environ Sci 7:127. https://doi.org/10.3389/fenvs.2019.00127

Otsu N (1979) A threshold selection method from gray-level histograms. IEEE Trans Syst Man Cybern SMC-9:62–66. https://doi.org/10.1109/tsmc.1979.4310076

Pekel JF, Cottam A, Gorelick N, Belward AS (2016) High-resolution mapping of global surface water and its long-term changes. Nature 540:418–422. https://doi.org/10.1038/nature20584

Schumann G, Di Baldassarre G, Bates PD (2009) The utility of spaceborne radar to render flood inundation maps based on multialgorithm ensembles. IEEE Trans Geosci Remote Sens 47:2801–2807. https://doi.org/10.1109/TGRS.2009.2017937

Tellman B, Sullivan JA, Kuhn C et al (2021) Satellite imaging reveals increased proportion of population exposed to floods. Nature 596:80–86. https://doi.org/10.1038/s41586-021-03695-w

43 River Morphology

Xiao Yang, Theodore Langhorst, and Tamlin M. Pavelsky

Overview
The purpose of this chapter is to showcase Earth Engine's application in fluvial hydrology and geomorphology. Specifically, we show examples demonstrating how to use Earth Engine to extract a river's centerline and width, and how to calculate the bank erosion rate. At the end of this chapter, you will be able to distinguish rivers from other water bodies, perform basic morphological analyses, and detect changes in river form over time.

Learning Outcomes

- Working with Landsat surface water products.
- Calculating river centerline location and width.
- Quantifying river bank erosion.

Helps if you know how to

- Import images and image collections, filter, and visualize (Part I).
- Perform basic image analysis: select bands, compute indices, create masks (Part II).
- Perform image morphological operations (Chap. 10).
- Write a function and `map` it over an ImageCollection (Chap. 12).

X. Yang (✉)
Southern Methodist University, Dallas, USA
e-mail: xnayang@smu.edu

T. Langhorst · T. M. Pavelsky
University of North Carolina at Chapel Hill, Chapel Hill, USA

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_43

- Use `reduceRegions` to summarize an image with zonal statistics in irregular shapes (Chaps. 22 and 24).
- Work with vector data (Chap. 23)

43.1 Introduction to Theory

The shape of a river viewed from above, known as its "planview geometry," can reveal many things about the river, including its morphological evolution and the flow of water and sediment within its channel. For example, hydraulic geometry establishes that a river's width and its discharge satisfy a power-law relation (rating curve). Thus, one can use such a relationship to monitor a river's discharge from river widths derived from remote sensing images (Smith et al. 1996). Similarly, in addition to the natural variability of river size, rivers also adjust their courses on the landscape as water flows from the headwaters toward lowland downstream regions. These adjustments result in meandering, lateral variations in a river's course resulting from the erosion and accretion of sediment along the banks. Many tools have been developed to study morphological changes of rivers using remotely sensed images.

Early remote sensing of river form was done by manual interpretation of aerial imagery, but advances in computing power have facilitated and the volume of imagery from satellites has necessitated automated processing. The RivWidth software (Pavelsky and Smith 2008) first presented an automated method for river width extraction, and RivWidthCloud (RWC) (Yang et al. 2020) later applied and expanded these methods to Earth Engine. Similarly, methods for detecting changes in river form have evolved from simple tools that track hand-drawn river centerlines (Shields et al. 2000) to automated methods that can process entire basins (Constantine et al. 2014; Rowland et al. 2016). This progression toward automated software for studies in fluvial geomorphology has paired well with the capabilities of Earth Engine, and as a result, many tools are being built for large-scale analysis in the cloud (Boothroyd et al. 2021).

43.2 Practicum

43.2.1 Creating and Analyzing a Single River Mask

In this section, we will prepare an image and calculate some simple morphological attributes of a river. To do this, we will use a pre-classified image of surface water occurrence, identify which pixels represent the river and channel bars, and finally calculate the centerline, width, and bank characteristics.

43.2.1.1 Isolate River from Water Surface Occurrence Map

Code Checkpoint A24a. The book's repository contains a script to use to begin this section. You will need to start with that script and paste code below into it.

The script includes our example area of interest (in the variable `aoi`) and two helper functions for reprojecting data to the local UTM coordinates. We force this projection and scale for many of our map layers because we are trying to observe and measure the river morphology. As data is viewed at different zoom levels, the shapes and apparent connectivity of many water bodies will change. To allow a given dataset to be viewed with the same detail at multiple scales, we can force the data to be reprojected, as we do here.

The Joint Research Centre's surface water occurrence dataset (Pekel et al. 2016) classified the entire Landsat 5, 7, and 8 history and produced annual maps that identify seasonal and permanent water classes. Here, we will include both seasonal and permanent water classes (represented by pixel values of ≥ 2) as water pixels (with value = 1) and the rest as non-water pixels (with value = 0). In this section, we will look at only one image at a time by choosing the image from the year 2000 (Fig. 1a). In the code below, `bg` serves as a dark background layer for other map layers to be seen easily.

```
// IMPORT AND VISUALIZE SURFACE WATER MASK.
// Surface water occurrence dataset from the JRC (Pekel et
al., 2016).
var jrcYearly =
ee.ImageCollection('JRC/GSW1_3/YearlyHistory');

// Select the seasonal and permanent pixels image
representing the year 2000
var watermask = jrcYearly.filter(ee.Filter.eq('year',
2000)).first()
    .gte(2).unmask(0)
    .clip(aoi);

Map.centerObject(aoi);
Map.addLayer(ee.Image.constant(0), {
    min: 0,
    palette: ['black']
}, 'bg', false);
Map.addLayer(watermask, {}, 'watermask', false);
```

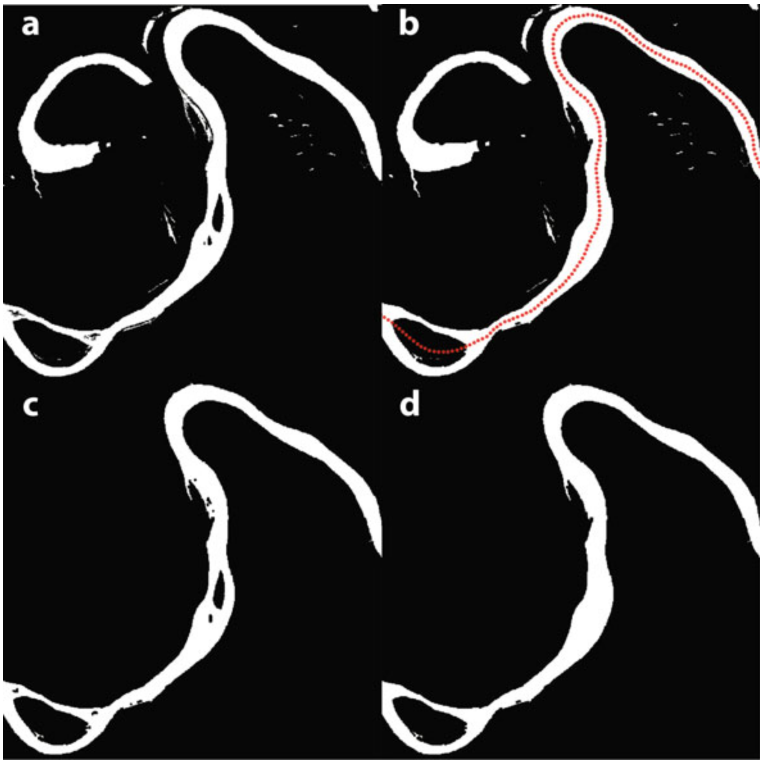


Fig. 43.1 **a** Water mask; **b** filled water mask with prior centerline points; **c** channel mask; **d** river mask

Next, we clean up the water mask by filling in small gaps by performing a closing operation (dilation followed by erosion). Areas of non-water pixels inside surface water bodies in the water mask may represent small channel bars, which we will fill in to create a simplified water mask. We identify these bars using a vectorization; however, you could do a similar operation with the `connectedPixelCount` method for bars up to 256 pixels in size (Fig. 1b). Filling in these small bars in the river mask improves the creation of a new centerline later in the lab.

```
// REMOVE NOISE AND SMALL ISLANDS TO SIMPLIFY THE TOPOLOGY.

// a. Image closure operation to fill small holes.
watermask = watermask.focal_max().focal_min();

// b. Identify small bars and fill them in to create a
filled water mask.
var MIN_SIZE = 2E3;
var barPolys = watermask.not().selfMask()
    .reduceToVectors({
        geometry: aoi,
        scale: 30,
        eightConnected: true
    })
    .filter(ee.Filter.lte('count', MIN_SIZE)); // Get small
polys.
var filled = watermask.paint(barPolys, 1);

Map.addLayer(rpj(filled), {
    min: 0,
    max: 1
}, 'filled water mask', false);
```

Note here that we forced reprojection of the map layer using the helper function `rpj`. This means we have to be careful to keep our domain small enough to be processed at the set scale when doing the calculation on the fly in the Code Editor; otherwise, we will run out of memory. The reprojection may not be necessary when exporting the output using a task.

In the following step, we extract water bodies in the water mask that correspond to rivers. We will define a river mask (Fig. 1d) to be pixels that are connected to the river centerline according to the filled water mask. The channel mask (Fig. 1c) is defined also by connectivity but excludes the small bars, which will give us more accurate widths and areas for change detection in Sects. 43.2.1.2 and 43.2.1.3.

We can extract the river mask by checking the water pixels' connectivity to a provided river location database. Specifically, we use the Earth Engine method `cumulativeCost` to identify connectivity between the filled water mask and the pixels corresponding to the river dataset. By inverting the filled mask, the cost to traverse water pixels is 0, and the cost over land pixels is 1. Pixels in the cost map with a value of 0 are entirely connected to the Surface Water and Ocean Topography (SWOT) Mission River Database (SWORD) centerline points by water, and pixels with values greater than 0 are separated from SWORD by land. The SWORD data, which were loaded as assets in the starter script, have some points located on land, either because the channel bifurcates or because the channel has migrated, so we must exclude those from our cumulative cost parameter `source`, or they will appear as single pixels of 0 in our cost map.

The maxDistance parameter must be set to capture maximum distance between centerline points and river pixels. In a single-threaded river with an accurate centerline, the ideal maxDistance value would be about half the river width. However, in reality, the centerlines are not perfect, and large islands may separate pixels from their nearest centerline. Unfortunately, increasing maxDistance has a large computational penalty, so some tweaking is required to get an optimal value. We can set geodeticDistance to **false** to regain some computational efficiency, because we are not worried about the accuracy of the distances.

```
// IDENTIFYING RIVERS FROM OTHER TYPES OF WATER BODIES.
// Cumulative cost mapping to find pixels connected to a
reference centerline.
var costmap = filled.not().cumulativeCost({
    source: watermask.and(ee.Image().toByte().paint(sword,
        1)),
    maxDistance: 3E3,
    geodeticDistance: false
});

var rivermask = costmap.eq(0).rename('riverMask');
var channelmask = rivermask.and(watermask);

Map.addLayer(sword, {
    color: 'red'
}, 'sword', false);
Map.addLayer(rpj(costmap), {
    min: 0,
    max: 1E3
}, 'costmap', false);
Map.addLayer(rpj(rivermask), {}, 'rivermask', false);
Map.addLayer(rpj(channelmask), {}, 'channelmask', false);
```

Code Checkpoint A24b. The book's repository contains a script that shows what your code should look like at this point.

43.2.1.2 Obtain River Centerline and Width

After processing the image to create a river mask, we will use existing functions from RivWidthCloud to process the image further to obtain river centerlines and widths. Here, we will call RivWidthCloud functions directly, taking advantage of the ability to use functions from another Earth Engine script (using the require functionality to load another script as a module). We will explain the usage and purpose of the RivWidthCloud functions used here.

There are three major steps involved in obtaining river widths from a given river mask:

1. Calculate one-pixel-width river centerlines.
2. Estimate the direction orthogonal to the flow direction for each centerline pixel.
3. Quantify river width on the channel mask along the orthogonal directions.

Extract River Centerline

We rely on morphological image analysis techniques to extract a river centerline. This process involves three steps:

1. Using distance transform to enhance pixels near the centerline of the river.
2. Using gradient to further isolate the centerline pixel having local minimal gradient values.
3. Cleaning the raw centerline by removing spurious centerlines.

First, a distance transform is applied to the river mask, resulting in a raster image where the value of each water pixel in the river mask is replaced by the closest distance to the shore. This step is done by using the `CalcDistanceMap` function from RWC. From Fig. 2a, we can see that, in the distance transform, the center of the river has the highest values.

```
// Import existing functions from RivWidthCloud.
var riverFunctions = require(

'users/eeProject/RivWidthCloudPaper:functions_river.js');
var clFunctions = require(

'users/eeProject/RivWidthCloudPaper:functions_centerline_wi
dth.js'
    );

//Calculate distance from shoreline using distance
transform.

var distance = clFunctions.CalcDistanceMap(rivermask, 256,
scale);
Map.addLayer(rpj(distance), {
    min: 0,
    max: 500
}, 'distance raster', false);
```

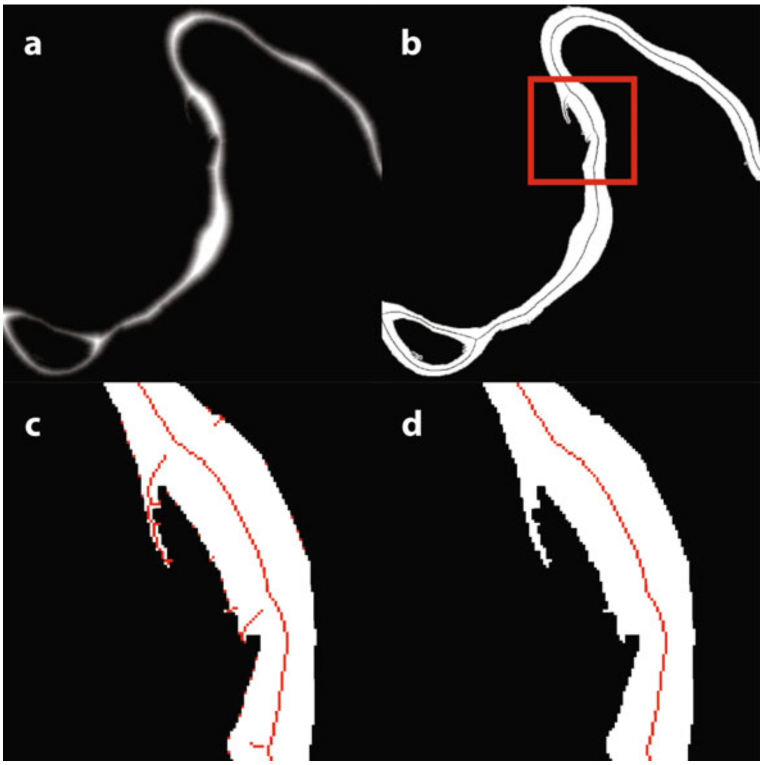


Fig. 43.2 Steps extracting river centerline: **a** distance transform of a river mask; **b** gradient of the distance map (a); **c** raw centerline after skeletonization; **d** centerline after pruning

Second, to isolate the centerline of the river, we apply a gradient calculation to the distance raster. If we treat the distance raster as a digital elevation model (DEM), then the locations of the river centerline can be visualized as ridgelines. They will thus have minimal gradient value. The gradient calculation is important, as it converts a local property of the centerline (local maximum distance) to a global property (global minimal gradient) to allow extraction of the centerline with a fixed gradient threshold (Fig. 2b). We use a 0.9 threshold (recommended for RivWidth (Pavelsky and Smith 2008) and RWC) to extract the centerline pixels from the gradient image. However, the resulting initial centerline is not always one pixel wide. To ensure a one-pixel-wide centerline, iterative image skeletonization is applied to thin the initial centerline (Fig. 2c).

```
// Calculate gradient of the distance raster.
// There are three different ways (kernels) to calculate
the gradient.
// By default, the function used the second approach.
// For details on the kernels, please see the source code
for this function.
var gradient = clFunctions.CalcGradientMap(distance, 2,
scale);
Map.addLayer(rpj(gradient), {}, 'gradient raster', false);

// Threshold the gradient raster and derive 1px width
centerline using skeletonization.

var centerlineRaw =
clFunctions.CalcOnePixelWidthCenterline(rivermask,
    gradient, 0.9);
var raw1pxCenterline = rpj(centerlineRaw).eq(1).selfMask();
Map.addLayer(raw1pxCenterline, {
    palette: ['red']
}, 'raw 1px centerline', false);
```

Third, the centerline from the previous step will have noise along the shoreline and will have spurious branches resulting from side channels or irregular channel forms that need to be pruned. The pruning function in RWC, `CleanCenterline`, works by first identifying end pixels of the centerline (i.e., centerline pixels with only one neighboring pixel) and then erasing pixels along the centerline pixels starting from the end pixels for a distance specified by `MAXDISTANCE_BRANCH_REMOVAL`. It will stop if the specified distance is reached or the erasing encounters a joint pixel (i.e., pixels having more than two neighboring pixels). After pruning, the final centerline should look like Fig. 2d.

```
// Prune the centerline to remove spurious branches.
var MAXDISTANCE_BRANCH_REMOVAL = 500;
// Note: the last argument of the CleanCenterline function
enables removal of the pixels so that the resulting
centerline will have 1px width in an 8-connected way. Once
it is done, it doesn't need to be done the second time
(thus it equals false)
var cl1px = clFunctions
    .CleanCenterline(centerlineRaw,
MAXDISTANCE_BRANCH_REMOVAL, true);
var cl1px = clFunctions
    .CleanCenterline(cl1px, MAXDISTANCE_BRANCH_REMOVAL,
false);
var final1pxCenterline = rpj(cl1px).eq(1).selfMask();
Map.addLayer(final1pxCenterline, {
    palette: ['red']
}, 'final 1px centerline', false);
```

Estimate Cross-Sectional Direction

Now we will use the centerline we obtained from the previous step to help us measure the widths of the river. River width is often measured along the direction perpendicular to the flow, which we will approximate using the course of its centerline. To estimate cross-sectional directions, we convolve the centerline image with a customized kernel. The square 9×9 kernel has been designed so that each pixel on its rim has the radian value of the angle between the line connecting the rim pixel and the center of the kernel and the horizontal x-axis (radian angle 0). The convolution works by overlapping the center of the kernel with the centerline and calculating the average of the values of the rim pixels that overlap the centerline pixels, which corresponds to the cross-sectional direction of the particular centerline point under consideration. Here, we use the function `CalculateAngle` to estimate the cross-sectional angles. The resulting raster will replace each centerline pixel with the value of the cross-sectional directions in degrees.

```
// Calculate perpendicular direction for the cleaned
centerline.
var angle = clFunctions.CalculateAngle(cl1px);
var angleVis = {
    min: 0,
    max: 360,
    palette: ['#ffffd4', '#fed98e', '#fe9929', '#d95f0e',
        '#993404'
    ]
};
Map.addLayer(rpj(angle), angleVis, 'cross-sectional
directions',
    false);
```

Quantify River Widths

To estimate river width, we will be using the RWC function `rwGen_waterMask`. This function can take any binary water mask image as input to calculate river widths, so long as the band name is "waterMask" and contains the following three properties: (1) crs—UTM projection code, (2) scale—native spatial resolution, and (3) image_id—acting as an identifier for the output widths. This function works by first processing the input water mask to create all the intermediate images mentioned before (channel mask, river mask, centerline, and angle image). Then, it creates a `FeatureCollection` of cross-sectional lines, each centered on one centerline pixel (from the centerline raster) along the direction estimated in the "Estimate Cross-Sectional Direction" section (from the angle raster) and with a length three times longer than the distance from the centerline point to the closest shoreline pixel (obtained from the distance raster). This `FeatureCollection` is then used in the `Image.reduceRegions` method as the `FeatureCollection` input. With a mean reducer, the result denotes the ratio between the actual river width and the length of the line segment (which is known). Thus, the final river width can be estimated by multiplying the ratio with the length of each line segment in the `FeatureCollection`. However, the scaling factor of 3 is chosen empirically, and can over- or underestimate the maximum extent of river width. This is because the width, scaled by 3, is the minimal distance from centerline pixels to the nearest shoreline pixels. When aligning line segments along the directions orthogonal to the river centerline, we might encounter situations when the length of these segments is too short to cover the width of the river (underestimation) or too long that they overlap with neighboring river reaches (overestimation). In both cases, the end(s) of the line segment overlaps with a pixel identified as "water" in the channel mask. Thus, additional steps are taken to flag these measurements.

The `rwGen_waterMask` takes four arguments—maximum search distance (unit: meter) to label river pixels, maximum size of islands (unit: pixel) to be filled in to calculate river mask, distance (unit: meter) to be pruned to clean the raw centerline, and the area of interest to carry out the width calculation. The output of the `rwc` function is a `FeatureCollection` with each feature having the properties listed in Table 43.1.

```
// Estimate width.
var rwcFunction = require(
    'users/eeProject/RivWidthCloudPaper:rwc_watermask.js');
var rwc = rwcFunction.rwGen_waterMask(4000, 333, 500, aoi);
watermask =
ee.Image(watermask.rename(['waterMask']).setMulti({
    crs: crs,
    scale: 30,
    image_id: 'aoi'
}));

var widths = rwc(watermask);
print('example width output', widths.first());
```

Table 43.1 Output variables from the `rwc` function

`longitude`	Longitude of the centerline point
`latitude`	Latitude of the centerline point
`width`	Wetted river width measured at the centerline point
`orthogonalDirection`	Angle of the cross-sectional direction at the centerline point
`flag_elevation`	Mean elevation across the river surface (unit: meter) based on MERIT DEM
`image_id`	Image ID of the input image
`crs`	The projection of the input image
`endsInWater`	Indicates inaccurate width due to the insufficient length of the cross-sectional segment that was used to measure the river width
`endsOverEdge`	Indicates calculated width too close to the edge of the image such that the width can be inaccurate

43.2.1.3 Bank Morphology

In addition to a river's centerline and width, we can also extract information about the banks of the river, such as their aspect and total length. To identify the banks, we simply dilate the channel mask and compare it to the original channel mask. The difference in these images represents the land pixels adjacent to the channel mask.

```
var bankMask = channelmask.focal_max(1).neq(channelmask);
```

Next, we will calculate the aspect, or compass direction, of the bank faces. We use the `Image.cumulativeCost` method with the entire river channel as our source to create a new image (bankDistance) with increasing values away from the river channel, similar to an elevation map of river banks. In this image, the banks will "slope" toward the river channel and we can take advantage of the terrain methods in EE. We will call the `Terrain.aspect` method on the bank distance and select the bank pixels by applying the bank mask. In the end, our bank aspect data will give us the direction from each bank pixel toward the center of the channel. These data could be useful for interpreting any directional preferences in erosion as a result of geological features or thawed permafrost soils from solar radiation.

```
var bankDistance = channelmask.not().cumulativeCost({
    source: channelmask,
    maxDistance: 1E2,
    geodeticDistance: false
});

var bankAspect = ee.Terrain.aspect(bankDistance)
    .multiply(Math.PI).divide(180)
    .mask(bankMask).rename('bankAspect');
```

Last, we calculate the length represented by each bank pixel by convolving the bank mask with a Euclidean distance kernel. Sections of bank oriented along the pixel edges will have a value of 30 m per pixel, whereas a diagonal section will have a value of $\sqrt{2} * 30$ m per pixel.

```
var distanceKernel = ee.Kernel.euclidean({
    radius: 30,
    units: 'meters',
    magnitude: 0.5
});
var bankLength = bankMask.convolve(distanceKernel)
    .mask(bankMask).rename('bankLength');

var radianVis = {
    min: 0,
    max: 2 * Math.PI,
    palette: ['red', 'yellow', 'green', 'teal', 'blue',
'magenta',
        'red'
    ]
};
Map.addLayer(rpj(bankAspect), radianVis, 'bank aspect',
false);
Map.addLayer(rpj(bankLength), {
    min: 0,
    max: 60
}, 'bank length', false);
```

Code Checkpoint A24c. The book's repository contains a script that shows what your code should look like at this point.

43.2.2 Multitemporal River Width

Refresh the Code Editor to begin with a new script for this section.

In Sect. 43.2.1.2, we walked through the process of extracting the river centerline and width from a given water mask. In that section, we intentionally unpacked the different steps used to extract river centerline and width so that readers can: (1) get an intuitive idea of how the image processes work step by step and see the resulting images at each stage; (2) combine these functions to answer different questions (e.g., readers might only be interested in river centerlines instead of getting all the way to widths). In this section, we will walk you through how to use some high-level functions in RivWidthCloud to more efficiently implement these steps across multiple water mask images to extract time series of widths at a given location. To do this, we need to provide two inputs: a point of interest (longitude, latitude) and a collection of binary water masks. The code below re-introduces a helper function to convert between projections, then accesses other data and functionality.

```
var getUTMProj = function(lon, lat) {
    // Given longitude and latitude in decimal degrees,
    // return EPSG string for the corresponding UTM
projection. See:
    // https://apollomapping.com/blog/gtm-finding-a-utm-
zone-number-easily
    // https://sis.apache.org/faq.html
    var utmCode =
ee.Number(lon).add(180).divide(6).ceil().int();
    var output = ee.Algorithms.If({
        condition: ee.Number(lat).gte(0),
        trueCase: ee.String('EPSG:326').cat(utmCode
            .format('%02d')),
        falseCase: ee.String('EPSG:327').cat(utmCode
            .format('%02d'))
    });
    return (output);
};

// IMPORT AND VISUALIZE SURFACE WATER MASK
// Surface water occurrence dataset from the JRC (Pekel et
al., 2016).
var jrcYearly =
ee.ImageCollection('JRC/GSW1_3/YearlyHistory');
var poi = ee.Geometry.LineString([
    [110.77450764660864, 30.954167027937988],
    [110.77158940320044, 30.950633845897112]
]);

var rwcFunction = require(
    'users/eeProject/RivWidthCloudPaper:rwc_watermask.js');
```

Remember that the widths from Sect. 43.2.1.2 are stored in a `FeatureCollection` with multiple width values from different locations along a centerline. To extract the multitemporal river width for a particular location along a river, we only need one width measurement from each water mask. Here, we choose the width for the centerline pixel that is nearest to the given point of interest using the function `getNearestCl`. This function takes the width `FeatureCollection` from Sect. 43.2.1.2 as input and returns a feature corresponding to the width closest to the point of interest.

```
// Function to identify the nearest river width to a given
location.
var GetNearestClGen = function(poi) {
    var temp = function(widths) {
        widths = widths.map(function(f) {
            return f.set('dist2cl', f.distance(poi,
                30));
        });

        return ee.Feature(widths.sort('dist2cl', true)
            .first());
    };
    return temp;
};
var getNearestCl = GetNearestClGen(poi);
```

Then, we will need to use the `map` method on the input collection of water masks to apply the `rwc` to all the water mask images. This will result in a `FeatureCollection`, each feature of which will contain the width quantified from one image (Fig. 43.3).

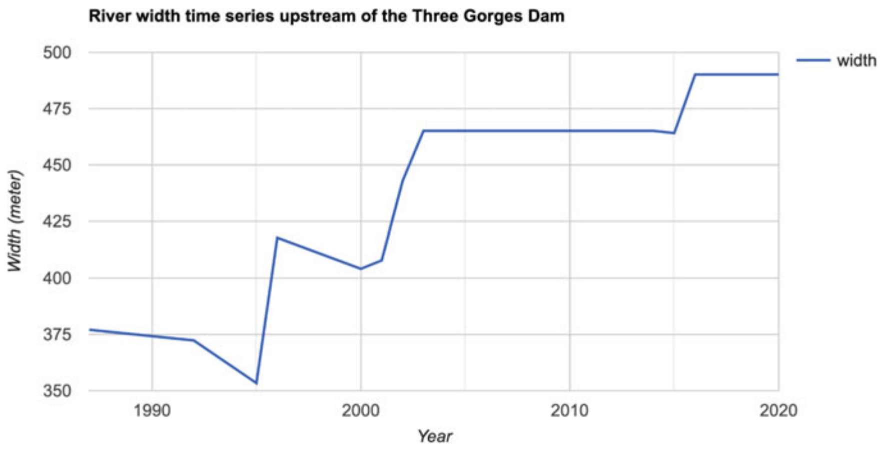


Fig. 43.3 River width time series upstream of the Three Gorges Dam in China. The series shows the abrupt increase in river width around the year 2003, when the dam was completed

```
// Multitemporal width extraction.
var polygon = poi.buffer(2000);
var coords = poi.centroid().coordinates();
var lon = coords.get(0);
var lat = coords.get(1);
var crs = getUTMProj(lon, lat);
var scale = ee.Number(30);

var multiwidths =
ee.FeatureCollection(jrcYearly.map(function(i) {
    var watermask = i.gte(2).unmask(0);

    watermask = ee.Image(watermask.rename(['waterMask'])
        .setMulti({
            crs: crs,
            scale: scale,
            image_id: i.getNumber('year')
        }));
    var rwc = rwcFunction.rwGen_waterMask(2000, 333, 300,
        polygon);
    var widths = rwc(watermask)
        .filter(ee.Filter.eq('endsInWater', 0))
        .filter(ee.Filter.eq('endsOverEdge', 0));

    return ee.Algorithms.If(widths.size(), getNearestCl(
        widths), null);
}, true));

var widthTs = ui.Chart.feature.byFeature(multiwidths,
'image_id', [
        'width'
    ])
    .setOptions({
        hAxis: {
            title: 'Year',
            format: '####'
        },
        vAxis: {
            title: 'Width (meter)'
        },
        title: 'River width time series upstream of the
Three Gorges Dam'
```

```
    });
print(widthTs);

Map.centerObject(polygon);
Map.addLayer(polygon, {}, 'area of width calculation');
```

Code Checkpoint A24d. The book's repository contains a script that shows what your code should look like at this point.

43.2.3 Riverbank Erosion

In this section, we will apply the methods we developed in Sect. 43.2.1 to multiple images, calculate the amount of bank erosion, and summarize our results back onto our centerline. Before doing so, we will create a new script that wraps the masking and morphology code in Sects. 43.2.1.1 and 43.2.1.3 into a function called `makeChannelmask` that has one argument for the year. We return an image with bands for all of the masks and bank calculations, plus a property named `'year'` that contains the year argument. If you have time, you could try to create this function on your own and then compare with our implementation of it, in the next code checkpoint. Note that we would not expect that your code would look the same, but it should ideally have the same functionality.

Code Checkpoint A24e. The book's repository contains a script to use to begin this section. You will need to start with that script and paste code below into it.

Change Detection

We will use a section of the Madre de Dios River as our study area for this example because it migrates very quickly, more than 30 m per year in some locations. Our methods will work best if the two channel masks partially overlap everywhere along the length of the river; if there is a gap between the two masks, we will underestimate the amount of change and not be able to calculate the direction of change. As such, we will pick the years 2015 and 2020 for our example. However, in other locations, you may want to increase the time span in order to observe more change. We first create these two sets of channel masks and add them to the map (Fig. 4a).

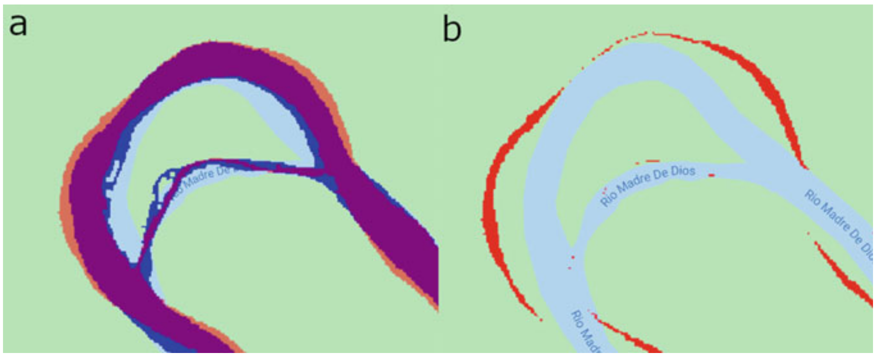


Fig. 43.4 Single meander bend of the Madre de Dios River in Bolivia, showing areas of erosion and accretion: **a** channel mask from 2015 in blue and channel mask from 2020 in red at 50% transparency; **b** pixels that represent erosion between 2015 and 2020

```
var masks1 = makeChannelmask(2015);
var masks2 = makeChannelmask(2020);
Map.centerObject(aoi, 13);
var year1mask =
rpj(masks1.select('channelmask').selfMask());
Map.addLayer(year1mask, {
    palette: ['blue']
}, 'year 1');
var year2mask =
rpj(masks2.select('channelmask').selfMask());
Map.addLayer(year2mask, {
    palette: ['red']
}, 'year 2', true, 0.5);
```

Next, we create an image to represent the eroded area (Fig. 4b). We can quickly calculate this by comparing the channel mask in year 2 to the inverse water mask from year 1. In alluvial river systems, avulsions and meander cutoffs can leave fragments of old channels near the river. If the river meanders back into these water bodies, we want to be careful not to count these as fully eroded, which is why we need to compare our river pixels in year 2 (channel mask) to the land pixels in year 1 (inverse water mask). If you were to compare only the channel masks from year to year, water in the floodplains that is captured by the channel migration would be falsely counted as erosion.

```
// Pixels that are now the river channel but were
previously land.
var erosion = masks2.select('channelmask')

.and(masks1.select('watermask').not()).rename('erosion');
Map.addLayer(rpj(erosion).selfMask(), {}, 'erosion',
false);
```

Now we are going to approximate the direction of erosion. We will define the direction of erosion by the shortest path through the eroded area from each bank pixel in year 1 to any of the bank pixels in year 2. In reality, meandering rivers often translate their shape downvalley, which breaks our definition of the shortest path between banks. However, the shortest path produces a reasonable approximation in most cases and is easy to calculate. We will again use `Image.cumulativeCost` to measure the distance using the erosion image as our cost surface. The erosion image has to be dilated by 1 pixel to compensate for the missing edge pixels in the gradient calculations and masked in order to limit the cost paths to within the eroded area.

```
// Erosion distance assuming the shortest distance between
banks.
var erosionEndpoints =
erosion.focal_max(1).and(masks2.select(
    'bankMask'));
var erosionDistance = erosion.focal_max(1).selfMask()
    .cumulativeCost({
        source: erosionEndpoints,
        maxDistance: 1E3,
        geodeticDistance: true
    }).rename('erosionDistance');
Map.addLayer(rpj(erosionDistance),
    {
        min: 0,
        max: 300
    },
    'erosion distance',
    false);
```

Now we can use the same `Terrain.aspect` method that we used for the bank aspect to calculate the direction of the shortest path along our cost surface. You could also calculate this direction (and the bank aspect in Sect. 43.2.1.3) using the `Image.gradient` method and then calculating the tangent of the resulting *x* and *y* components.

```
// Direction of the erosion following slope of distance.
var erosionDirection = ee.Terrain.aspect(erosionDistance)
    .multiply(Math.PI).divide(180)
    .clip(aoi)
    .rename('erosionDirection');
erosionDistance = erosionDistance.mask(erosion);
Map.addLayer(rpj(erosionDirection),
    {
        min: 0,
        max: Math.PI
    },
    'erosion direction',
    false);
```

Connecting to the Centerline

We now have all of our change metrics calculated as images in Earth Engine. We could export these and make maps and figures using these data. However, when analyzing a lot of river data, we often want to look at long profiles of a river or tributary networks in a watershed. In order to do this, we will use reducers to summarize our raster data back onto our vector centerline. The first step is to identify which pixels should be assigned to which centerline points. We will start by calculating a single image representing the distance to any SWORD centerline point with the `FeatureCollection.distance` method. Next, we will use a convolution with the Laplacian kernel (Chap. 10) as an edge detection method on our distance raster. By convolving the distance to the nearest SWORD node with the Laplacian kernel, we are calculating the second derivative of distance and can find the locations where the distance surface starts sloping toward another SWORD point.

```
// Distance to nearest SWORD centerline point.
var distance = sword.distance(2E3).clip(aoi);

// Second derivatives of distance.
// Finding the 0s identifies boundaries between centerline
points.
var concavityBounds =
distance.convolve(ee.Kernel.laplacian8())
    .gte(0).rename('bounds');

Map.addLayer(rpj(distance), {
    min: 0,
    max: 1E3
}, 'distance', false);
Map.addLayer(rpj(concavityBounds), {}, 'bounds', false);
```

Next, we need to create an image where each pixel's value is set to the unique node identifier of the nearest SWORD centerline point. We will create a two-band image, where the first band is the concavity boundaries found in the last step, and the second band has the unique node identifiers painted on their location. When we reduce this image using the `Image.reduceConnectedComponents` method, we set all pixels in each region with the corresponding node ID. Last, we need to dilate these pixels to fill in the boundary gaps using a call to the `Image.focalMode` method (Fig. 43.5).

```
// Reduce the pixels according to the concavity boundaries,
// and set the value to SWORD node ID.  Note that focalMode
is used
// to fill in the empty pixels that were the boundaries.
var swordImg = ee.Image(0).paint(sword,
'node_id').rename('node_id')
    .clip(aoi);
var nodePixels = concavityBounds.addBands(swordImg)
    .reduceConnectedComponents({
        reducer: ee.Reducer.max(),
        labelBand: 'bounds'
    }).focalMode({
        radius: 3,
        iterations: 2
    });
Map.addLayer(rpj(nodePixels).randomVisualizer(),
    {},
    'node assignments',
    false);
```

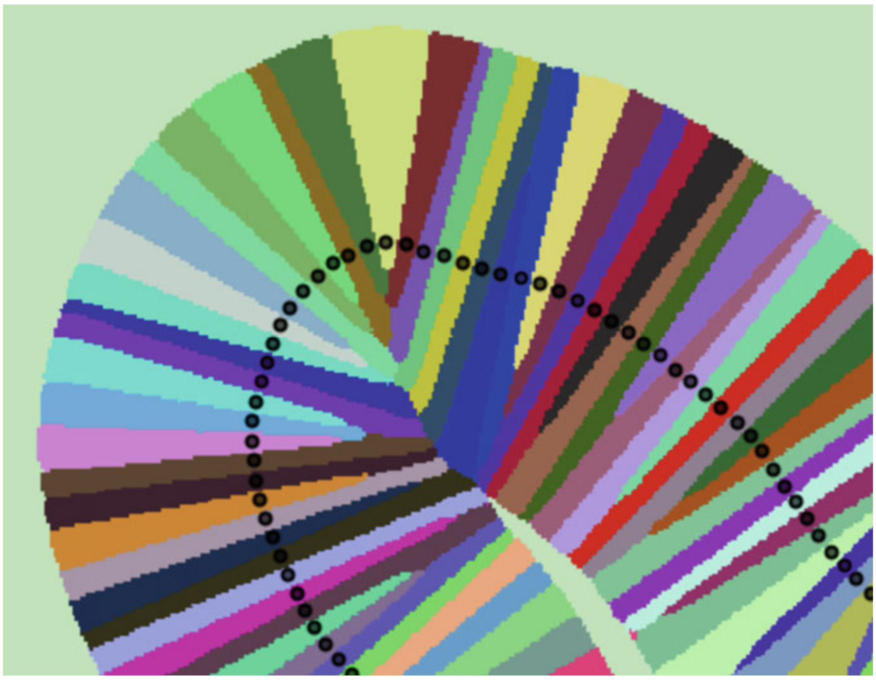


Fig. 43.5 Section of the Madre de Dios River where each pixel is assigned to its closest centerline node

Summarizing the Data

The final step in this section is to apply a reducer that uses our `nodePixels` image from the previous step to group our raster data. We will combine the `reducer.forEach` and `reducer.group` methods into our own custom function that we can use with different reducers to get our final results. The `reducer.forEach` method sets up a different reducer and output for each band in our image, which is necessary when we use the `reducer.group` method. The `reducer.group` method is conceptually similar to `reducer.reduceRegions`, except our regions are defined by an image band instead of by polygons. In some cases, the group method is much faster than the `reducer.reduceRegions` method, particularly if you were to have to convert your regions to polygons in order to provide the input to `reducer.reduceRegions`. The grouped reducers in our function return a list of dictionaries. However, it is much easier to work with feature collections, so we will map over the list and create a `FeatureCollection` before returning from the function.

```
// Set up a custom reducing function to summarize the data.
var groupReduce = function(dataImg, nodeIds, reducer) {
    // Create a grouped reducer for each band in the data
image.
    var groupReducer = reducer.forEach(dataImg.bandNames())
        .group({
            groupField: dataImg.bandNames().length(),
            groupName: 'node_id'
        });

    // Apply the grouped reducer.
    var statsList = dataImg.addBands(nodeIds).clip(aoi)
        .reduceRegion({
            reducer: groupReducer,
            scale: 30,
        }).get('groups');

    // Convert list of dictionaries to FeatureCollection.
    var statsOut = ee.List(statsList).map(function(dict) {
        return ee.Feature(null, dict);
    });
    return ee.FeatureCollection(statsOut);
};
```

For some variables—such as the erosion, the channel mask, or the bank length—we want the total number of pixels or bank length, so we will use the `Reducer.sum` method with our grouped reducer function. For our aspect and directional variables, we need to use the `Reducer.circularMean` method to find the mean direction. The returned variables `sumStats` and `angleStats` are feature collections with properties for our reduced data and the corresponding node ID.

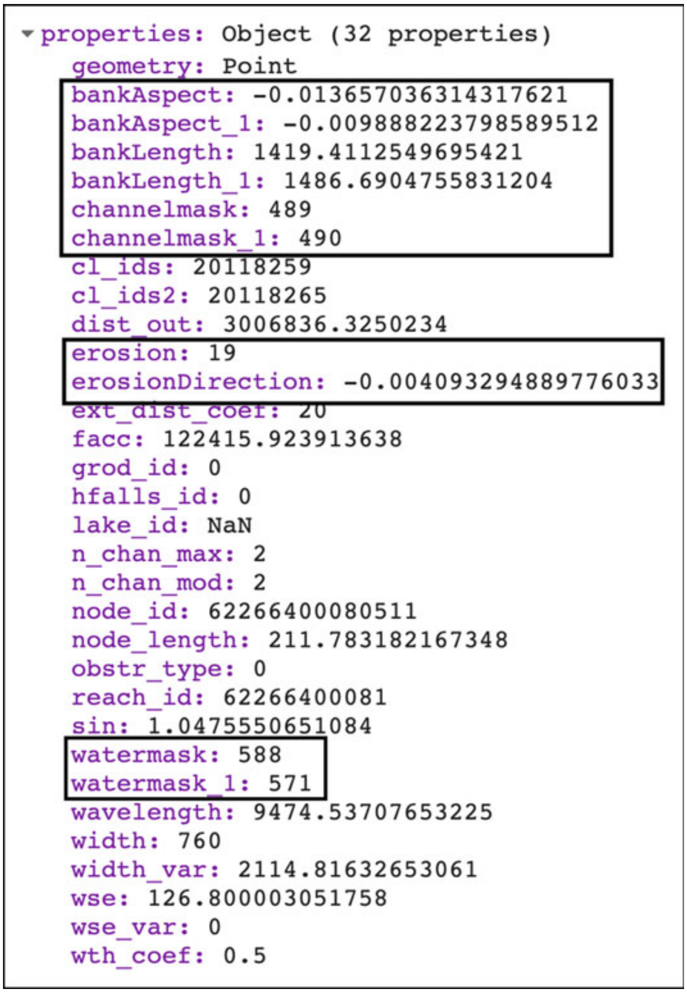
```
var dataMask = masks1.addBands(masks2).reduce(ee.Reducer
    .anyNonZero());

var sumBands = ['watermask', 'channelmask', 'bankLength'];
var sumImg = erosion
    .addBands(masks1, sumBands)
    .addBands(masks2, sumBands);
var sumStats = groupReduce(sumImg, nodePixels,
ee.Reducer.sum());

var angleImg = erosionDirection
    .addBands(masks1, ['bankAspect'])
    .addBands(masks2, ['bankAspect']);
var angleStats = groupReduce(angleImg, nodePixels,
ee.Reducer
    .circularMean());
```

Finally, we will join these two new feature collections to our original centerline data and print the results (Fig. 43.6).

```
var vectorData = sword.filterBounds(aoi).map(function(feat)
{
    var nodeFilter = ee.Filter.eq('node_id', feat.get(
        'node_id'));
    var sumFeat = sumStats.filter(nodeFilter).first();
    var angleFeat = angleStats.filter(nodeFilter).first();
    return feat.copyProperties(sumFeat).copyProperties(
        angleFeat);
});

print(vectorData);
Map.addLayer(vectorData, {}, 'final data');
```

Code Checkpoint A24f. The book’s repository contains a script that shows what your code should look like at this point.

This workflow can be used to add many new properties to the river centerlines based on raster calculations. For example, we calculated the amount of erosion between these two years, but you could use very similar code to calculate the amount of accretion that occurred. Other interesting properties of the river, like the slope of the banks from a DEM, could be calculated and added to our centerline dataset.



```
▾ properties: Object (32 properties)
    geometry: Point
    bankAspect: -0.013657036314317621
    bankAspect_1: -0.009888223798589512
    bankLength: 1419.4112549695421
    bankLength_1: 1486.6904755831204
    channelmask: 489
    channelmask_1: 490
    cl_ids: 20118259
    cl_ids2: 20118265
    dist_out: 3006836.3250234
    erosion: 19
    erosionDirection: -0.004093294889776033
    ext_dist_coef: 20
    facc: 122415.923913638
    grod_id: 0
    hfalls_id: 0
    lake_id: NaN
    n_chan_max: 2
    n_chan_mod: 2
    node_id: 62266400080511
    node_length: 211.783182167348
    obstr_type: 0
    reach_id: 62266400081
    sin: 1.0475550651084
    watermask: 588
    watermask_1: 571
    wavelength: 9474.53707653225
    width: 760
    width_var: 2114.81632653061
    wse: 126.800003051758
    wse_var: 0
    wth_coef: 0.5
```

Fig. 43.6 Updated list of properties in our centerline dataset; new properties are outlined in black. The erosion and mask fields are in units of pixels, but you could convert to area using the `Image.pixelArea` method on the masks

43.3 Synthesis

Assignment 1. RivWidthCloud can estimate individual channel width in the case of multichannel rivers. Change the AOI to a multichannel river and observe the resulting centerline and width data. Note down things you think are different from the single-channel case.

Assignment 2. Answer the following question. When rivers experience both variable width over time and bank migration, how can we apply the methods in this chapter to distinguish these two types of changes?

43.4 Conclusion

In this chapter, we provide ways in which Earth Engine can be used to aid river planview morphological studies. In the first half of the chapter, we show how to distinguish river pixels from other types of water bodies, as well as how to extract river centerline, river width, bank aspect, and length. In the second half of the chapter, we give examples of how to apply these methods to multitemporal image collections to estimate changes in river widths for rivers that have stable channels and to estimate bank erosion for rivers that tend to meander quickly. The analysis makes use of both raster- and vector-based methods provided by Earth Engine to help quantify river morphology. More importantly, these methods can be applied at scale.

References

Boothroyd RJ, Williams RD, Hoey TB et al (2021) Applications of Google Earth Engine in fluvial geomorphology for detecting river channel change. Wiley Interdiscip Rev Water 8:e21496. https://doi.org/10.1002/wat2.1496

Constantine JA, Dunne T, Ahmed J et al (2014) Sediment supply as a driver of river meandering and floodplain evolution in the Amazon Basin. Nat Geosci 7:899–903. https://doi.org/10.1038/ngeo2282

Pavelsky TM, Smith LC (2008) RivWidth: a software tool for the calculation of river widths from remotely sensed imagery. IEEE Geosci Remote Sens Lett 5:70–73. https://doi.org/10.1109/LGRS.2007.908305

Pekel JF, Cottam A, Gorelick N, Belward AS (2016) High-resolution mapping of global surface water and its long-term changes. Nature 540:418–422. https://doi.org/10.1038/nature20584

Rowland JC, Shelef E, Pope PA et al (2016) A morphology independent methodology for quantifying planview river change and characteristics from remotely sensed imagery. Remote Sens Environ 184:212–228. https://doi.org/10.1016/j.rse.2016.07.005

Shields FD Jr, Simon A, Steffen LJ (2000) Reservoir effects on downstream river channel migration. Environ Conserv 27:54–66. https://doi.org/10.1017/S0376892900000072

Smith LC, Isacks BL, Bloom AL, Murray AB (1996) Estimation of discharge from three braided rivers using synthetic aperture radar satellite imagery: potential application to ungaged basins. Water Resour Res 32:2021–2034. https://doi.org/10.1029/96WR00752

Yang X, Pavelsky TM, Allen GH, Donchyts G (2020) RivWidthCloud: an automated Google Earth Engine algorithm for river width extraction from remotely sensed imagery. IEEE Geosci Remote Sens Lett 17:217–221. https://doi.org/10.1109/LGRS.2019.2920225

Water Balance and Drought

44

Ate Poortinga, Quyen Nguyen, Nyein Soe Thwal, and Andréa Puzzi Nicolau

Overview
In this chapter, you will learn simple water balance calculations using remote sensing-derived products related to precipitation and evapotranspiration. You will work at the river basin scale and perform time-series analysis, while comparing the data series with remote sensing vegetation and drought indices using the Earth Engine platform. You will also overlay the various indices with a land cover map to estimate potential drought impacts throughout the region.

Learning Outcomes

- Understanding the basics of remote sensing-derived precipitation and evapotranspiration products.
- Calculating monthly aggregate statistics.
- Performing time-series analysis.
- Calculating vegetation and drought indices.

A. Poortinga (✉) · A. P. Nicolau
Spatial Informatics Group, Pleasanton, CA, USA
e-mail: apoortinga@sig-gis.com

A. P. Nicolau
e-mail: apnicolau@sig-gis.com

Q. Nguyen · N. S. Thwal
Asian Disaster Preparedness Center, Bangkok, Thailand
e-mail: nguyen.quyen@adpc.net

N. S. Thwal
e-mail: nyein.thwal@adpc.net

A. Poortinga · Q. Nguyen · N. S. Thwal
SERVIR-Southeast Asia, Bangkok, Thailand

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_44

Helps if you know how to:

- Import images and image collections, filter, and visualize (Part 1).
- Create a graph using `ui.Chart` (Chap. 4).
- Perform basic image analysis: select bands, compute indices, create masks (Part 2).
- Write a function and map it over an `ImageCollection` (Chap. 12).
- Summarize an `ImageCollection` with reducers (Chaps. 12 and 13).
- Aggregate data to build a time series (Chap. 14).
- Work with CHIRPS rainfall data (Chap. 14)
- Mask cloud, cloud shadow, snow/ice, and other undesired pixels (Chap. 15).

44.1 Introduction to Theory

Water is vital for sustaining human life, ensuring food security, generating power, and supporting industrial processes in river basins. Both terrestrial and aquatic ecosystems are dependent on water to provide valuable ecosystem services, not only for the current generation but also for generations in the future. Managing the complex flow paths of water to and from these different water-use sectors requires a quantitative understanding of hydrological processes. Quantitative insights are necessary to help manage water consumption more efficiently by means of retention, withdrawals, and land use change. Managers need background data to help them optimize water allocation to sectors without further depleting the natural capital in the basin.

The water balance (Fig. 44.1) is the key concept in understanding the availability of water resources in a hydrological system. The water balance includes both input and extractions of water. In its simplest form, the water balance can be defined as Eq. (44.1). Inputs to the hydrological system are defined by precipitation (P; rainfall and snow). Extractions for the system are from runoff (Q) and evapotranspiration (ET), with evapotranspiration denoting the sum of evaporation from the land surface plus transpiration from plants. Water balance changes in groundwater and soil storage are indicated by ΔS.

$$P = Q + \mathrm{ET} + \Delta S \tag{44.1}$$

A hydrological system, also referred to as river basin or drainage basin, is any land area where precipitation collects and drains off into a common outlet. The hydrological processes between upstream and downstream are interconnected: For example, extractions of water resources upstream will impact the amount of available downstream water resources. Similarly, upstream activities such as logging and swidden agriculture might impact the quality of downstream water resources. Figure 44.2 shows the boundaries of the Lower Mekong River Basin, which covers parts of Laos, Thailand, Cambodia, Myanmar, and Vietnam. Water is a shared

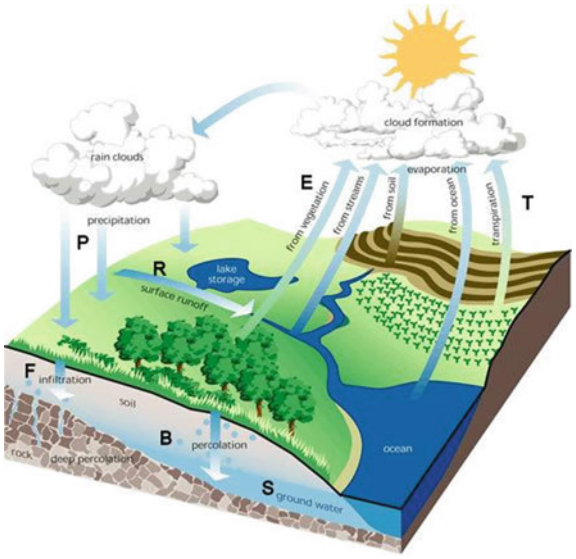


Fig. 44.1 The key components of the hydrological cycle

resource among the countries, although each country has its own legal framework to maintain and protect water supplies. A quantitative understanding of this shared resource is imperative to formulating effective management strategies.

In the following exercises, we will calculate the main components of the water balance and link them with vegetation growth, drought information, and land cover information in the Lower Mekong Basin.

44.2 Practicum

44.2.1 Section 1: Calculating Monthly Precipitation

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.

Precipitation has been measured for many centuries (Strangeways 2010). The traditional method is point measurement, which was standardized in the previous century to make measurements comparable in space and time. Figure 44.3 shows a conventional weather station used to measure various weather-related parameters, including the amount of rainfall. Although statistical methods exist to calculate area averaged rainfall from weather stations, the limited number of data points remains a constraint, especially in developing countries and sparsely populated regions where the density of weather stations is low. Satellites can fill

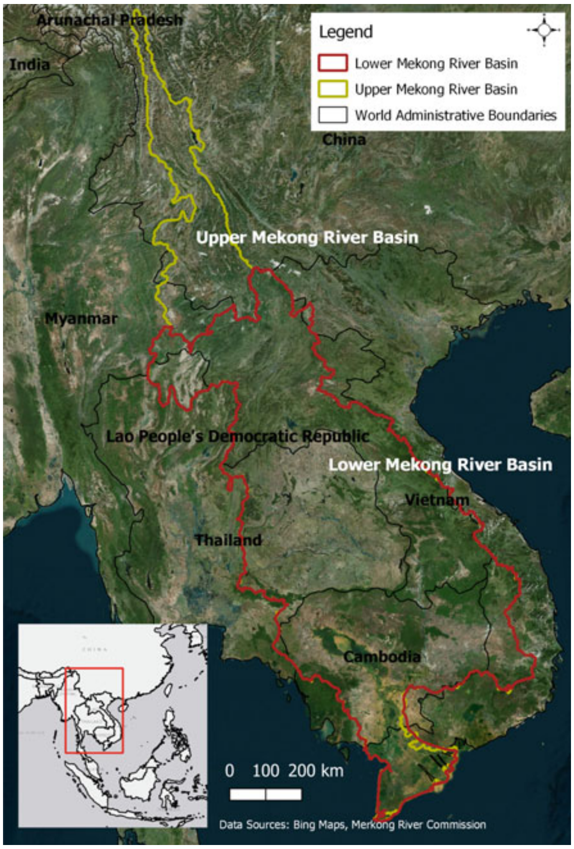

Legend
🟥 Lower Mekong River Basin
🟨 Upper Mekong River Basin
⬜ World Administrative Boundaries
Arunachal Pradesh
India
China
Upper Mekong River Basin
Myanmar
Lao People's Democratic Republic
Lower Mekong River Basin
Vietnam
Thailand
Cambodia
0 100 200 km
Data Sources: Bing Maps, Merkong River Commission

Fig. 44.2 The upper and lower Mekong River Basin

this information gap, as they observe the planet at a regular interval with calibrated sensors.

The Tropical Rainfall Measuring Mission (TRMM), a joint mission of the Japan Aerospace Exploration Agency (JAXA) and NASA, was a notable effort to monitor and study tropical rainfall (Kummerow et al. 1998). The satellite, which operated for 17 years, contained various instruments to measure clouds and cloud structures in order to advance understanding of the global energy and water cycles. The Global Precipitation Measurement (GPM; Fig. 44.4) mission is the successor, with the primary aim of making frequent (every 2–3 h) observations of Earth's precipitation (Hou et al. 2014). A wide variety of data products are available for both TRMM and GPM. Precipitation estimates are derived from the Precipitation Radar (PR), TRMM Microwave Imager (TMI), Visible Infrared Scanner (VIRS), Clouds and Earth's Radiant Energy System (CERES), and Lightning Imaging Sensor (LSI). Frequently used products of TRMM have a spatial resolution of 0.25° (~ 25 km), whereas GPM has a higher resolution of 0.1° (~ 10 km). Data is available on a 3-h time interval. The data can be obtained through NASA but also can be accessed from the Earth Engine data repository.

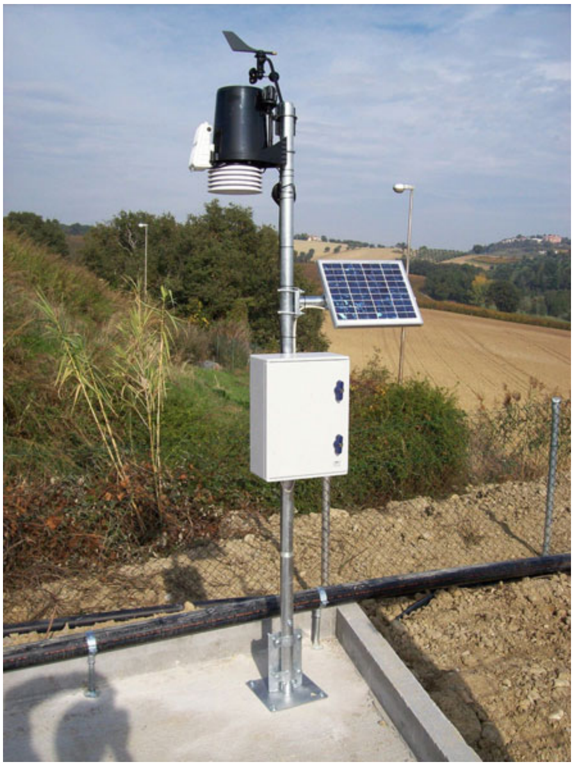


Fig. 44.3 Conventional weather station that measures various parameters, including precipitation

Fig. 44.4 The satellite for the Global Precipitation Measurement (GPM) mission

The Climate Hazards Group InfraRed Precipitation with Station (CHIRPS) data is a quasi-global rainfall dataset (Funk et al. 2015) covering more than 35 years. CHIRPS provides precipitation information at a 0.5° (~ 5 km) spatial resolution. The dataset estimates precipitation by combining data from observing meteorological stations with satellite data. CHIRPS data is available at intervals from daily to annual and can be very valuable in hydrology studies, as it provides a long and consistent time series with precipitation estimates at a relatively high spatial resolution.

Below is the code and an exercise to calculate monthly precipitation using these satellite data.

We begin by importing our area of interest, the Lower Mekong River Basin (Fig. 44.5).

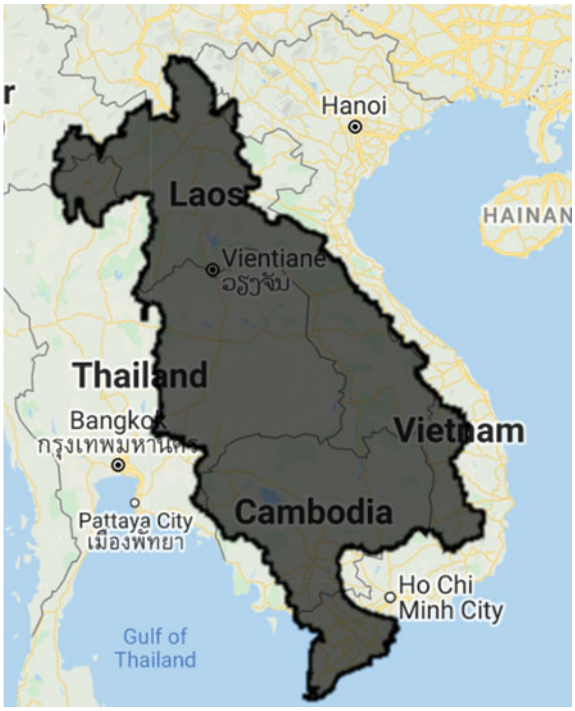


Fig. 44.5 The lower Mekong Basin

```
// Import the Lower Mekong boundary.
var mekongBasin = ee.FeatureCollection(
    'projects/gee-book/assets/A2-5/lowerMekongBasin');

// Center the map.
Map.centerObject(mekongBasin, 5);

// Add the Lower Mekong Basin boundary to the map.
Map.addLayer(mekongBasin, {}, 'Lower Mekong basin');
```

In the next step, we set the start and end dates for our analysis. We create a list for both years and months, which we will later use to iterate over.

```
// Set start and end years.
var startYear = 2010;
var endYear = 2020;

// Create two date objects for start and end years.
var startDate = ee.Date.fromYMD(startYear, 1, 1);
var endDate = ee.Date.fromYMD(endYear + 1, 1, 1);

// Make a list with years.
var years = ee.List.sequence(startYear, endYear);

// Make a list with months.
var months = ee.List.sequence(1, 12);
```

We import the CHIRPS `ImageCollection` and select the imagery for the relevant dates, as presented in Chap. 14. Note that we used the pentad time series; each image in this collection contains the accumulated rainfall for five days. The daily product is also available in Earth Engine. The pentad dataset was used rather than the daily data product to reduce the number of computations needed to aggregate the data.

```
// Import the CHIRPS dataset.
var CHIRPS = ee.ImageCollection('UCSB-CHG/CHIRPS/PENTAD');

// Filter for the relevant time period.
CHIRPS = CHIRPS.filterDate(startDate, endDate);
```

The year and month lists are used in the function below to calculate the monthly rainfall. We use a server-side nested loop where we first map over the years (2010, 2011, … 2020) and then map over the months (1, 2, … 12). This returns an image with the total rainfall for each month. We set the year, month, and timestamp ('system:time_start') for each image and flatten the image to turn the object into a single ImageCollection.

```
// We apply a nested loop where we first map over
// the relevant years and then map over the relevant
// months. The function returns an image with the total
(sum)
// rainfall for each month. A flatten is applied to convert
a
// feature collection of features into a single feature
collection.
var monthlyPrecip = ee.ImageCollection.fromImages(
    years.map(function(y) {
        return months.map(function(m) {
            var w = CHIRPS.filter(ee.Filter
                    .calendarRange(y, y, 'year'))
                .filter(ee.Filter.calendarRange(m, m,
                    'month'))
                .sum();
            return w.set('year', y)
                .set('month', m)
                .set('system:time_start', ee.Date
                    .fromYMD(y, m, 1));

        });
    }).flatten()
);
```

Add a layer with the monthly mean precipitation to the map and calculate a chart with monthly mean precipitation (Fig. 44.6).

```
// Add the layer with monthly mean. Note that we clip for
the Mekong river basin.
var precipVis = {
    min: 0,
    max: 250,
    palette: 'white, blue, darkblue, red, purple'
};

Map.addLayer(monthlyPrecip.mean().clip(mekongBasin),
    precipVis,
    '2015 precipitation');

// Set the title and axis labels for the chart.
var title = {
    title: 'Monthly precipitation',
    hAxis: {
        title: 'Time'
    },
    vAxis: {
        title: 'Precipitation (mm)'
    },
};

// Plot the chart using the Mekong boundary.
var chartMonthly = ui.Chart.image.seriesByRegion({
        imageCollection: monthlyPrecip,
        regions: mekongBasin.geometry(),
        reducer: ee.Reducer.mean(),
        band: 'precipitation',
        scale: 5000,
        xProperty: 'system:time_start'
    }).setSeriesNames(['P'])
    .setOptions(title)
    .setChartType('ColumnChart');

// Print the chart.
print(chartMonthly);
```

Code Checkpoint A25a. The book's repository contains a script that shows what your code should look like at this point.

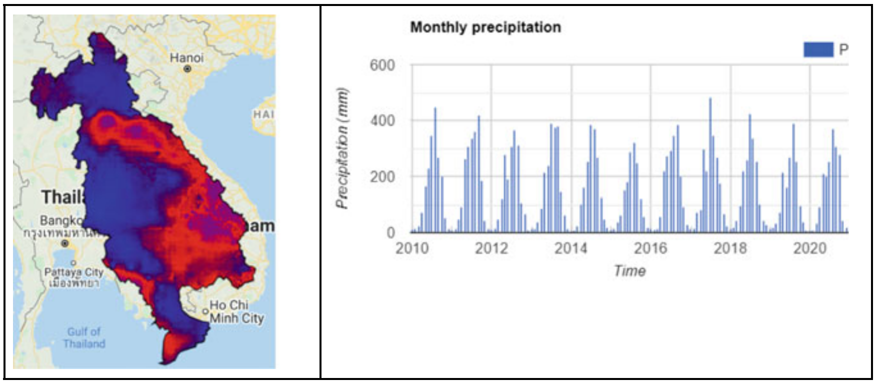


Fig. 44.6 Mean precipitation in the Lower Mekong Basin (left) and the monthly average precipitation (right)

44.2.2 Section 2: Calculating Monthly Evapotranspiration

Measuring evapotranspiration at large scales is important for assessing climate and anthropogenic effects on natural and agricultural ecosystems (Kustas and Norman 1996). Methods exist to measure ET at a field scale, but those methods cannot be extrapolated to larger areas. Traditional ways of estimating ET have been to use reference ET, derived from various weather-related parameters, with a crop coefficient. However, there are large uncertainties due to, for example, spatial and temporal heterogeneity and data gaps. Satellite information can be very useful as it provides spatially and temporally dense information for ET estimation.

Different methods exist to map ET from remote sensing data, including simple empirical models that relate spectral reflectance with ET, vegetation index models, energy budget, and deterministic models (Courault et al. 2005). Fundamentally, however, ET is governed by the energy budget and driving variables such as surface temperature. The total amount of available net radiant energy is divided into a soil heat flux and the atmospheric convective fluxes, which are the sensible heat flux (H) and latent energy exchanges (LE). This essentially means that the temperature decreases when energy is used for ET. Indeed, there are different ways to further quantify the different energy fluxes in more detail, and there is a wide body of scientific literature that describes those methods.

There are different readily available ET products derived from the Moderate Resolution Imaging Spectroradiometer (MODIS), including Atmosphere-Land Exchange Inverse (ALEXI; Anderson et al. 1997; Mecikalski et al. 1999), the operational Simplified Surface Energy Balance (SSEB; Senay et al. 2013), CSIRO MODIS Rescaled Evapotranspiration (CMRSET; Guerschman et al. 2009), and MOD16. The MOD16 algorithm is based on the logic of the Penman–Monteith equation, which uses daily meteorological reanalysis data and eight-day remotely sensed vegetation property dynamics from MODIS as inputs. The MOD16 product is available in the Earth Engine assets, and we will use this product for ET estimation in the next exercise.

First, we start a new script and repeat the first two steps of the previous section by copying and pasting the following code:

```
// Import the Lower Mekong boundary.
var mekongBasin = ee.FeatureCollection(
    'projects/gee-book/assets/A2-5/lowerMekongBasin');

// Center the map.
Map.centerObject(mekongBasin, 5);

// Add the Lower Mekong Basin boundary to the map.
Map.addLayer(mekongBasin, {}, 'Lower Mekong basin');

// Set start and end years.
var startYear = 2010;
var endYear = 2020;

// Create two date objects for start and end years.
var startDate = ee.Date.fromYMD(startYear, 1, 1);
var endDate = ee.Date.fromYMD(endYear + 1, 1, 1);

// Make a list with years.
var years = ee.List.sequence(startYear, endYear);

// Make a list with months.
var months = ee.List.sequence(1, 12);
```

We import the MOD16 dataset and select the ET band, which represents total evapotranspiration.

```
// Import the MOD16 dataset.
var mod16 =
ee.ImageCollection('MODIS/006/MOD16A2').select('ET');

// Filter for the relevant time period.
mod16 = mod16.filterDate(startDate, endDate);
```

We use the same function to calculate monthly values as in the previous section. Note that we multiply by 0.1 as a scaling factor. The scaling factor can be found in the description of the dataset in Earth Engine. Scaling factors are applied to reduce the required storage capacity by changing the data type.

```
// We apply a nested loop where we first map over
// the relevant years and then map over the relevant
// months. The function returns an image with the total
(sum)
// evapotranspiration for each month. A flatten is applied
to convert a
// collection of collections into a single collection.
// We multiply by 0.1 because of the ET scaling factor.
var monthlyEvap = ee.ImageCollection.fromImages(
    years.map(function(y) {
        return months.map(function(m) {
            var w = mod16.filter(ee.Filter
                    .calendarRange(y, y, 'year'))
                .filter(ee.Filter.calendarRange(m, m,
                    'month'))
                .sum()
                .multiply(0.1);
            return w.set('year', y)
                .set('month', m)
                .set('system:time_start', ee.Date
                    .fromYMD(y, m, 1));

        });
    }).flatten()
);
```

We use the code below to visualize the results (Fig. 44.7). Note that we changed the color of the bar chart and applied the reducer on a 500 m spatial resolution.

```
// Add the layer with monthly mean. Note that we clip for
the Mekong river basin.
var evapVis = {
    min: 0,
    max: 140,
    palette: 'red, orange, yellow, blue, darkblue'
};

Map.addLayer(monthlyEvap.mean().clip(mekongBasin),
    evapVis,
    'Mean monthly ET');

// Set the title and axis labels for the chart.
var title = {
    title: 'Monthly evapotranspiration',
    hAxis: {
        title: 'Time'
    },
    vAxis: {
        title: 'Evapotranspiration (mm)'
    },
    colors: ['red']
};

// Plot the chart using the Mekong boundary.
var chartMonthly = ui.Chart.image.seriesByRegion({
        imageCollection: monthlyEvap,
        regions: mekongBasin.geometry(),
        reducer: ee.Reducer.mean(),
        band: 'ET',
        scale: 500,
        xProperty: 'system:time_start'
    }).setSeriesNames(['ET'])
    .setOptions(title)
    .setChartType('ColumnChart');

// Print the chart.
print(chartMonthly);
```

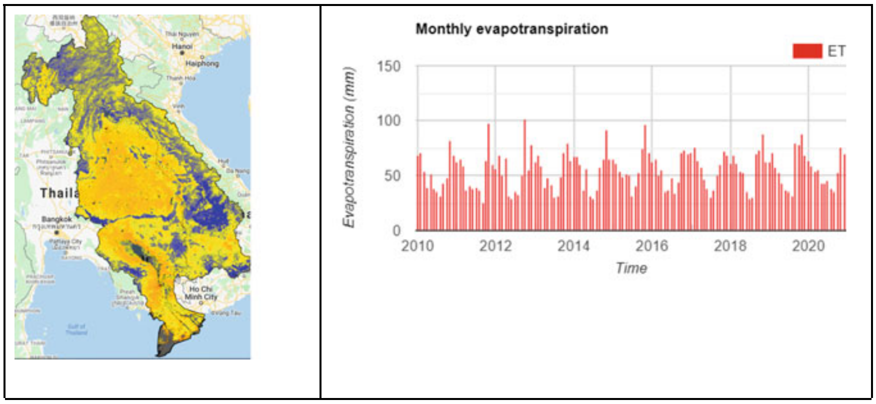


Fig. 44.7 Mean ET in the lower Mekong basin (left) and the monthly average ET (right)

Code Checkpoint A25b. The book's repository contains a script that shows what your code should look like at this point.

44.2.3 Section 3: Monthly Water Balance

We learned that the water balance is calculated using precipitation, evapotranspiration, runoff, and storage changes (Eq. 44.1). In the previous two sections, we calculated the monthly precipitation (P) on a 5 km spatial resolution and the monthly evapotranspiration (ET) on a 500 m spatial resolution. In Eq. (44.2), we rearrange Eq. (44.1) so that we can calculate the portion of Q and ΔS on a pixel level and aggregate that information to a basin level.

$$P - ET = Q + \Delta S \tag{44.2}$$

In this section, we will use the previous data to calculate the monthly water balance. First, we set the dates and import the relevant `ImageCollection`, as shown in the previous sections. Copy and paste the code below in a new script.

```
// Import the Lower Mekong boundary.
var mekongBasin = ee.FeatureCollection(
    'projects/gee-book/assets/A2-5/lowerMekongBasin');

// Center the map.
Map.centerObject(mekongBasin, 5);

// Add the Lower Mekong Basin boundary to the map.
Map.addLayer(mekongBasin, {}, 'Lower Mekong basin');

// Set start and end years.
var startYear = 2010;
var endYear = 2020;

// Create two date objects for start and end years.
var startDate = ee.Date.fromYMD(startYear, 1, 1);
var endDate = ee.Date.fromYMD(endYear + 1, 1, 1);

// Make a list with years.
var years = ee.List.sequence(startYear, endYear);

// Make a list with months.
var months = ee.List.sequence(1, 12);

// Import the CHIRPS dataset.
var CHIRPS = ee.ImageCollection('UCSB-CHG/CHIRPS/PENTAD');

// Filter for relevant time period.
CHIRPS = CHIRPS.filterDate(startDate, endDate);

// Import the MOD16 dataset.
var mod16 =
ee.ImageCollection('MODIS/006/MOD16A2').select('ET');

// Filter for relevant time period.
mod16 = mod16.filterDate(startDate, endDate);
```

Now we use the function that we used earlier to calculate monthly ET and *P* to calculate the water balance.

```
// We apply a nested loop where we first map over
// the relevant years and then map over the relevant
// months. The function returns an image with P - ET
// for each month. A flatten is applied to convert an
// collection of collections into a single collection.
var waterBalance = ee.ImageCollection.fromImages(
    years.map(function(y) {
        return months.map(function(m) {

            var P = CHIRPS.filter(ee.Filter
                    .calendarRange(y, y, 'year'))
                .filter(ee.Filter.calendarRange(m, m,
                    'month'))
                .sum();

            var ET = mod16.filter(ee.Filter
                    .calendarRange(y, y, 'year'))
                .filter(ee.Filter.calendarRange(m, m,
                    'month'))
                .sum()
                .multiply(0.1);

            var wb = P.subtract(ET).rename('wb');

            return wb.set('year', y)
                .set('month', m)
                .set('system:time_start', ee.Date
                    .fromYMD(y, m, 1));

        });
    }).flatten()
);
```

Next, we add the monthly mean water balance to the map and calculate the monthly water balance (Fig. 44.8). Note that negative numbers in the map indicate regions with an overall surplus of ET, whereas negative monthly water balances indicate a surplus ET for the whole region.

```
// Add layer with monthly mean. note that we clip for the
Mekong river basin.
var balanceVis = {
    min: -50,
    max: 200,
    palette: 'red, orange, yellow, blue, darkblue, purple'
};

Map.addLayer(waterBalance.mean().clip(mekongBasin),
    balanceVis,
    'Mean monthly water balance');

// Set the title and axis labels for the chart.
var title = {
    title: 'Monthly water balance',
    hAxis: {
        title: 'Time'
    },
    vAxis: {
        title: 'Evapotranspiration (mm)'
    },
    colors: ['green']
};

// Plot the chart using the Mekong boundary.
var chartMonthly = ui.Chart.image.seriesByRegion({
        imageCollection: waterBalance,
        regions: mekongBasin.geometry(),
        reducer: ee.Reducer.mean(),
        band: 'wb',
        scale: 500,
        xProperty: 'system:time_start'
    }).setSeriesNames(['WB'])
    .setOptions(title)
    .setChartType('ColumnChart');

// Print the chart.
print(chartMonthly);
```

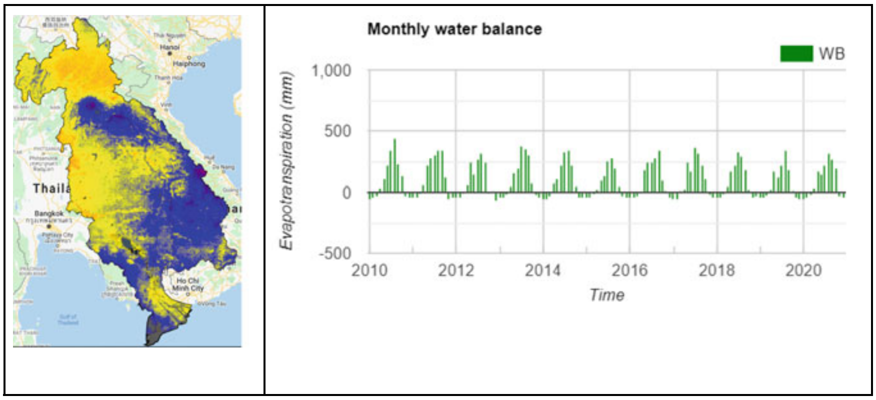


Fig. 44.8 Mean water balance in the lower Mekong basin (left) and the monthly average water balance (right)

Code Checkpoint A25c. The book's repository contains a script that shows what your code should look like at this point.

44.2.4 Section 4: Vegetation and Drought Indices

Calculating vegetation indices is a common practice when working with remote sensing data. The Normalized Difference Vegetation Index (NDVI; Rouse et al. 1973) and Enhanced Vegetation Index (EVI; Chap. 9) (Huete et al. 1994) are among the most commonly used. Vegetation indices rely on the absorption and reflection spectra of chlorophyll, often including the red band, where absorption is high, and near infrared, where reflection is high. Vegetation indices are often used to measure crop health and density, but they can also be used to measure, for example, biophysical health over longer time periods (Poortinga et al. 2018). Vegetation indices can be calculated from the spectral reflectance, but readily available products can be used as well. The latter have been processed and contain, for example, corrections for outliers and artifacts.

Besides vegetation indices, there are many other indices to describe specific natural phenomena or detect specific land surface features. These indices often rely on simple band ratios or more sophisticated formulas containing multiple bands. Drought indices are another category of important indicators as they enable us to depict spatiotemporal drought patterns in great detail. This can be particularly useful for remote areas where people rely on local agriculture for their livelihoods but production data is scarce. In the next exercise, we will be using the Moisture Stress Index (MSI; Vogelmann and Rock 1985), as this index has been shown to be highly related to soil moisture. Equation (44.3) shows that MSI is calculated from the shortwave infrared (SWIR) and near infrared (NIR) bands.

$$\text{MSI} = \text{SWIR}/\text{NIR} \tag{44.3}$$

In this section, we use the EVI product for the vegetation index from MODIS. This data is collected daily and can be used for calculation of the monthly vegetation index. You may note that we import the readily available EVI product and also import the spectral reflectance dataset.

We first import all the relevant datasets and define the period of interest, as in the other exercises, starting a new script.

```
// Import the Lower Mekong boundary.
var mekongBasin = ee.FeatureCollection(
    'projects/gee-book/assets/A2-5/lowerMekongBasin');

// Center the map.
Map.centerObject(mekongBasin, 5);

// Add the Lower Mekong Basin boundary to the map.
Map.addLayer(mekongBasin, {}, 'Lower Mekong basin');

// Set start and end years.
var startYear = 2010;
var endYear = 2020;

// Create two date objects for start and end years.
var startDate = ee.Date.fromYMD(startYear, 1, 1);
var endDate = ee.Date.fromYMD(endYear + 1, 1, 1);

// Make a list with years.
var years = ee.List.sequence(startYear, endYear);

// Make a list with months.
var months = ee.List.sequence(1, 12);
```

```
// Import the CHIRPS dataset.
var CHIRPS = ee.ImageCollection('UCSB-CHG/CHIRPS/PENTAD');

// Filter for relevant time period.
CHIRPS = CHIRPS.filterDate(startDate, endDate);

// Import the MOD16 dataset.
var mod16 =
ee.ImageCollection('MODIS/006/MOD16A2').select('ET');

// Filter for relevant time period.
mod16 = mod16.filterDate(startDate, endDate);

// Import and filter the MOD13 dataset.
var mod13 = ee.ImageCollection('MODIS/006/MOD13A1');
mod13 = mod13.filterDate(startDate, endDate);

// Select the EVI.
var EVI = mod13.select('EVI');

// Import and filter the MODIS Terra surface reflectance
dataset.
var mod09 = ee.ImageCollection('MODIS/006/MOD09A1');
mod09 = mod09.filterDate(startDate, endDate);
```

Two steps are needed to calculate the MSI. First, we need to remove the clouds and cloud shadows. This can be done by using the StateQA quality band that comes with the MOD09 product. After all the artifacts have been removed, we can calculate the MSI in the next function.

```
// We use a function to remove clouds and cloud shadows.
// We map over the mod09 image collection and select the
StateQA band.
// We mask pixels and return the image with clouds and
cloud shadows masked.
mod09 = mod09.map(function(image) {
    var quality = image.select('StateQA');
    var mask = image.and(quality.bitwiseAnd(1).eq(
            0)) // No clouds.
        .and(quality.bitwiseAnd(2).eq(0)); // No cloud
shadow.

    return image.updateMask(mask);
});
```

```
// We use a function to calculate the Moisture Stress
Index.
// We map over the mod09 image collection and select the
NIR and SWIR bands
// We set the timestamp and return the MSI.
var MSI = mod09.map(function(image) {
    var nirband = image.select('sur_refl_b02');
    var swirband = image.select('sur_refl_b06');

    var msi = swirband.divide(nirband).rename('MSI')
        .set('system:time_start', image.get(
            'system:time_start'));
    return msi;
});
```

We use the same nested loop as the previous sections, but now we calculate all layers and return an image where the layers are included as bands. Note that EVI and MOD16 are multiplied by the scale factor. For *P*, ET, and WB, we calculate the sum; for MSI and EVI, we calculate the mean.

```
// We apply a nested loop where we first map over
// the relevant years and then map over the relevant
// months. The function returns an image with bands for
// water balance (wb), rainfall (P), evapotranspiration
(ET),
// EVI and MSI for each month. A flatten is applied to
// convert an collection of collections
// into a single collection.
var ic = ee.ImageCollection.fromImages(
    years.map(function(y) {
        return months.map(function(m) {
            // Calculate rainfall.
            var P = CHIRPS.filter(ee.Filter
                    .calendarRange(y, y, 'year'))
                .filter(ee.Filter.calendarRange(m, m,
                    'month'))
                .sum();

            // Calculate evapotranspiration.
            var ET = mod16.filter(ee.Filter
                    .calendarRange(y, y, 'year'))
                .filter(ee.Filter.calendarRange(m, m,
                    'month'))
                .sum()
                .multiply(0.1);
            // Calculate EVI.
            var evi = EVI.filter(ee.Filter
                    .calendarRange(y, y, 'year'))
                .filter(ee.Filter.calendarRange(m, m,
                    'month'))
                .mean()
                .multiply(0.0001);

            // Calculate MSI.
            var msi = MSI.filter(ee.Filter
                    .calendarRange(y, y, 'year'))
                .filter(ee.Filter.calendarRange(m, m,
                    'month'))
                .mean();

            // Calculate monthly water balance.
            var wb = P.subtract(ET).rename('wb');

            // Return an image with all images as bands.
            return ee.Image.cat([wb, P, ET, evi, msi])
                .set('year', y)
                .set('month', m)
                .set('system:time_start', ee.Date
                    .fromYMD(y, m, 1));

        });
    }).flatten()
);
```

We display the monthly mean EVI and MSI and show the time series (Fig. 44.9).

```
// Add the mean monthly EVI and MSI to the map.
var eviVis = {
    min: 0,
    max: 0.7,
    palette: 'red, orange, yellow, green, darkgreen'
};

Map.addLayer(ic.select('EVI').mean().clip(mekongBasin),
    eviVis,
    'EVI');

var msiVis = {
    min: 0.25,
    max: 1,
    palette: 'darkblue, blue, yellow, orange, red'
};

Map.addLayer(ic.select('MSI').mean().clip(mekongBasin),
    msiVis,
    'MSI');

// Define the water balance chart and print it to the
console.
var chartWB =
    ui.Chart.image.series({
        imageCollection: ic.select(['wb', 'precipitation',
'ET']),
        region: mekongBasin,
        reducer: ee.Reducer.mean(),
        scale: 5000,
        xProperty: 'system:time_start'
    })
    .setSeriesNames(['wb', 'P', 'ET'])
    .setOptions({
        title: 'water balance',
        hAxis: {
            title: 'Date',
            titleTextStyle: {
                italic: false,
                bold: true
            }
        },
        vAxis: {
            title: 'Water (mm)',
            titleTextStyle: {
                italic: false,
                bold: true
            }
        },
        lineWidth: 1,
        colors: ['green', 'blue', 'red'],
        curveType: 'function'
    });

// Print the water balance chart.
print(chartWB);

// Define the indices chart and print it to the console.
var chartIndices =
    ui.Chart.image.series({
        imageCollection: ic.select(['EVI', 'MSI']),
        region: mekongBasin,
        reducer: ee.Reducer.mean(),
        scale: 5000,
        xProperty: 'system:time_start'
    })
    .setSeriesNames(['EVI', 'MSI'])
    .setOptions({
        title: 'Monthly indices',
        hAxis: {
            title: 'Date',
            titleTextStyle: {
                italic: false,
                bold: true
            }
        },
        vAxis: {
            title: 'Index',
            titleTextStyle: {
                italic: false,
                bold: true
            }
        },
        lineWidth: 1,
```

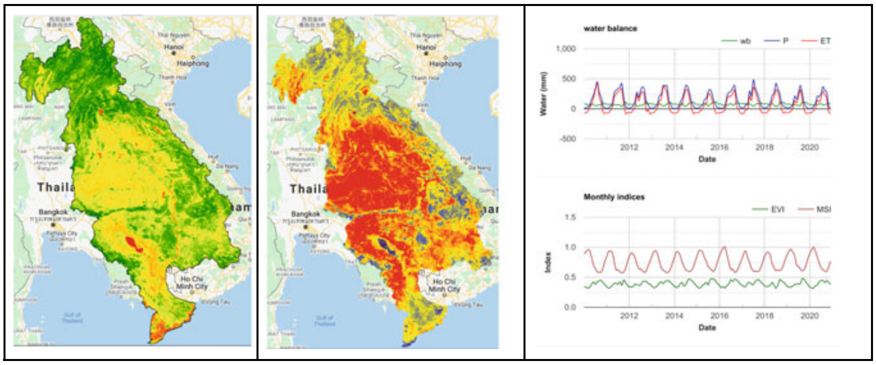


Fig. 44.9 Monthly mean EVI (left), monthly mean MSI (middle), and the time series for the water balance (top right) and indices (bottom right)

```
        colors: ['darkgreen', 'brown'],
        curveType: 'function'
    });

// Print the indices chart.
print(chartIndices);
```

Code Checkpoint A25d. The book's repository contains a script that shows what your code should look like at this point.

44.2.5 Section 5: Partitioning Water Resources and Mapping Drought Impacts

Historical and near-real-time remote-sensing-derived data can be very useful to assess the impact on the ground. For example, overlaying information layers on water resources with land cover information enables us to partition water resources by land cover and investigate consumptive use of, for example, different agricultural commodities. Overlaying land cover with vegetation and drought indicators helps to investigate land cover categories that are most impacted by water shortage. It also helps to assess the impacts on crop production and food security (Poortinga et al. 2019), as well as potential environmental impacts, including impacts on biodiversity. However, in many cases, maps are produced and updated only infrequently and are often not accompanied by appropriate accuracy assessment information and documentation (Saah et al. 2019). Therefore, SERVIR-Mekong developed a time series of yearly land cover maps covering the greater Mekong region, including the Lower Mekong Basin (Saah et al. 2020; Potapov

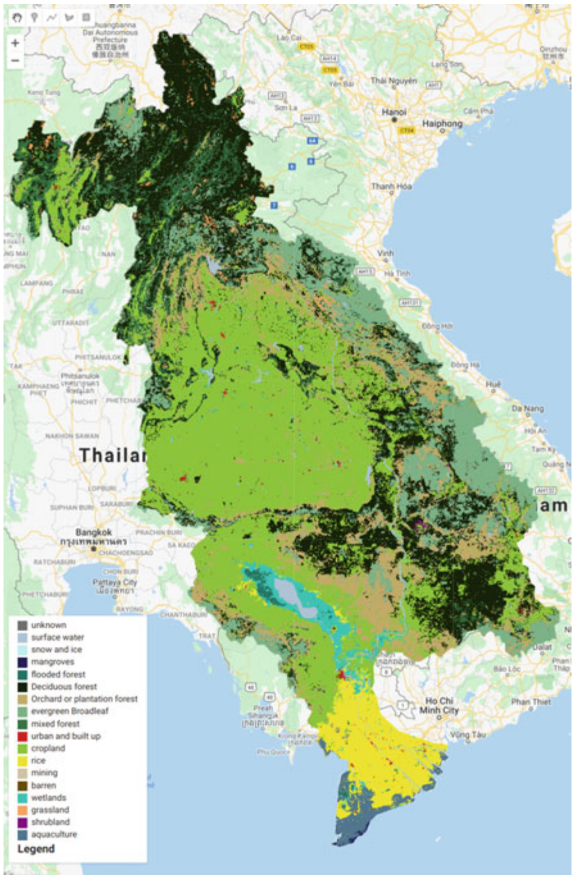


Fig. 44.10 Land cover map of the Lower Mekong Basin

et al. 2019; Poortinga et al. 2020). Figure 44.10 shows the land cover map of 2018.

44.2.5.1 Section 5.1: Annual Classifications

In the following subsections, we illustrate annual land cover maps and a variety of interpretations of them. The annual maps can be accessed using script **A25s1—Annual** in the book repository.

Note: if you get an error related to "too many concurrent aggregations" in this last script link or the following ones, try re-running the script.

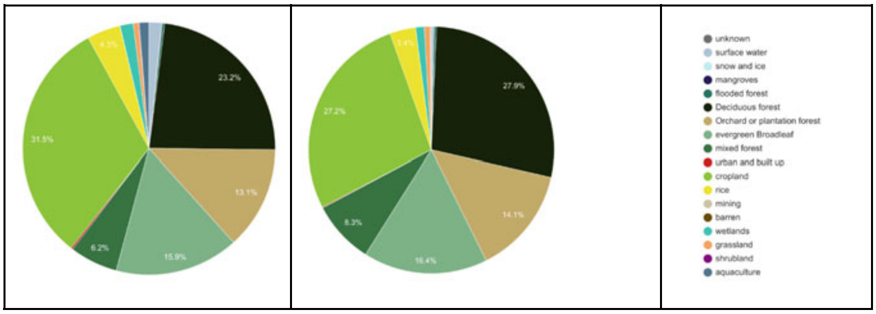


Fig. 44.11 The amount of *P* (left) and ET (right) per land cover category

44.2.5.2 Section 5.2: Precipitation and Evapotranspiration

An example of partitioning water inputs and consumption is shown in Fig. 44.11. Here we calculate the mean *P* and ET for each land cover category using the land cover map. We can see that the largest portions of water are being used by cropland and agriculture; plantations also consume a large portion of the water. The pie charts can be created and viewed using script **A25s2—PET** in the book repository.

44.2.5.3 Section 5.3: Monthly Water Balance

We can apply a similar overlaying method of partitioning for the water balance. Figure 44.12 shows the monthly water balance for deciduous forest, evergreen broadleaf, cropland, and rice. It can be seen that a large portion of the water stored in the wet season is used by forest and cropland in the dry season. The area of paddy rice is smaller, so the total water consumption in the dry season of those categories is smaller. The monthly water balance charts can be found in script **A25s3—Monthly** in the book repository.

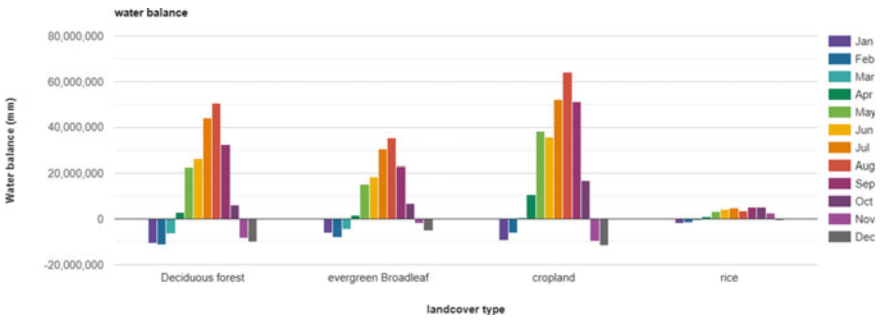


Fig. 44.12 Water balance for four different land cover categories: deciduous forest, evergreen broadleaf, cropland, and rice

44.2.5.4 Section 5.4: Per-class Water Balance Across Seasons

Partitioning can also be done per land cover category. In Fig. 44.13, we calculated the EVI and MSI for four land cover categories. There are very distinct patterns: we see little variation in the signal from the EVI but large variations for the deciduous forest. For cropland, we found a signal that closely corresponds to the yearly dry and wet seasons, whereas we see multiple cropping seasons per year. The long time series enables us to investigate deviations from the mean, which in turn provides valuable information on, for example, potential drought impacts. The per-class water balance charts can be found in script **A25s4—Per Class Balance** in the book repository.

44.3 Synthesis

With what you learned in this chapter, you can analyze large-scale hydrological processes in a river basin. The approach can be applied for any river basin in the world using your own data or the open-access data in the exercises.

Assignment 1. Test the approach in another part of the world using your own data or open-access data, or use your own training data for a more refined classification model.

Assignment 2. For further analysis, we encourage you to (1) replace MODIS with data from the Visible Infrared Imaging Radiometer Suite (VIIRS); (2) replace the CHIRPS data with the Integrated Multi-satellite Retrievals for GPM (IMERG) data and change the time intervals; and (3) use a different land cover map for partitioning the water resources.

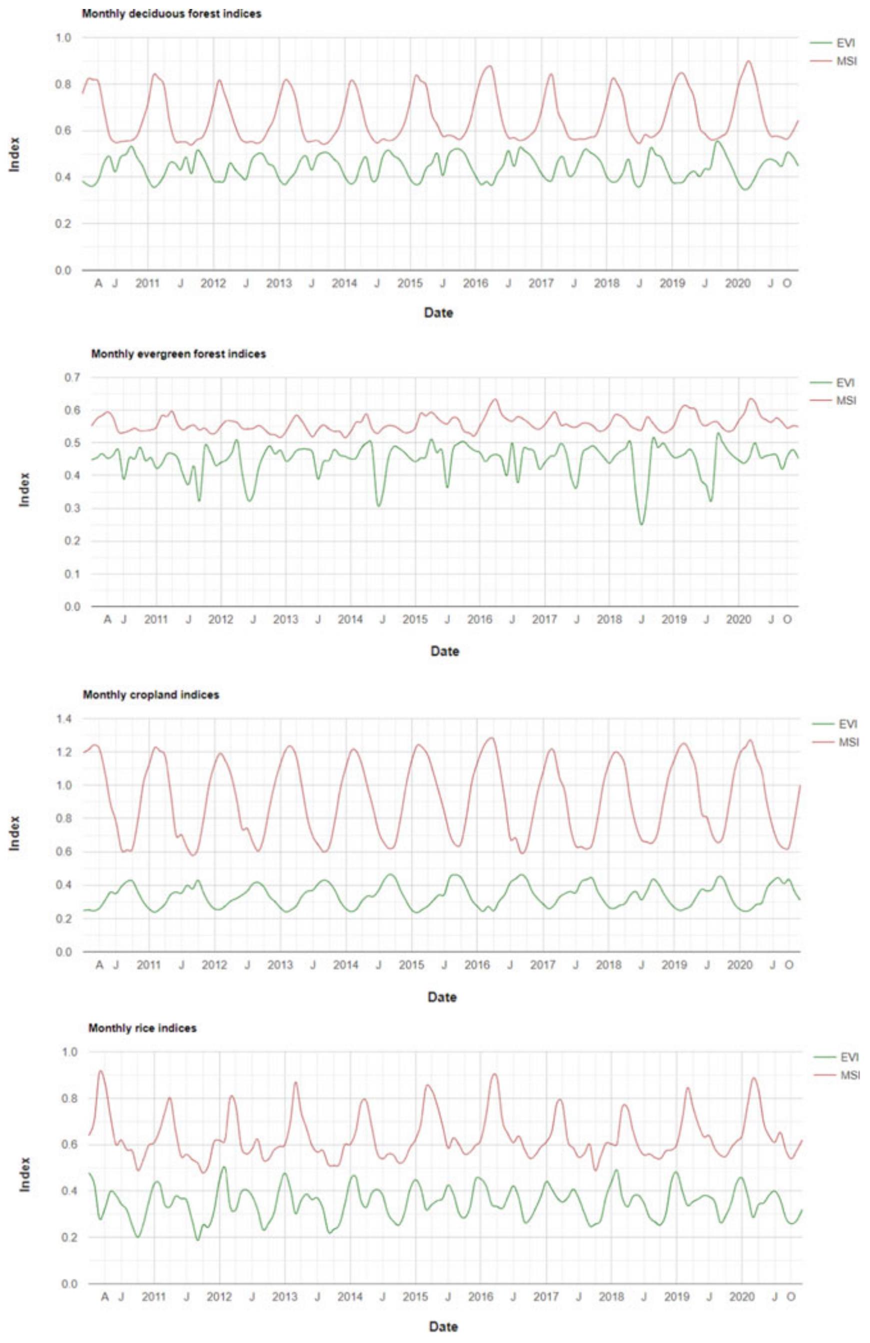


Fig. 44.13 EVI and MSI per land cover category

44.4 Conclusion

A safe and sustainable supply of water is essential for drinking, washing, cleaning, cooking, and growing food. However, water is often a scarce resource that needs to be managed in a sustainable and equitable way. Satellite data products in Earth Engine can help us to map the quantity of water in space and time. It enables us to partition water according to its consumptive use and evaluate how this affects a wide variety of important functions within a water basin.

References

Anderson MC, Norman JM, Diak GR et al (1997) A two-source time-integrated model for estimating surface fluxes using thermal infrared remote sensing. Remote Sens Environ 60:195–216. https://doi.org/10.1016/S0034-4257(96)00215-5

Courault D, Seguin B, Olioso A (2005) Review on estimation of evapotranspiration from remote sensing data: from empirical to numerical modeling approaches. Irrig Drain Syst 19:223–249. https://doi.org/10.1007/s10795-005-5186-0

Funk C, Peterson P, Landsfeld M et al (2015) The climate hazards infrared precipitation with stations—a new environmental record for monitoring extremes. Sci Data 2:1–21. https://doi.org/10.1038/sdata.2015.66

Guerschman JP, Van Dijk AIJM, Mattersdorf G et al (2009) Scaling of potential evapotranspiration with MODIS data reproduces flux observations and catchment water balance observations across Australia. J Hydrol 369:107–119. https://doi.org/10.1016/j.jhydrol.2009.02.013

Hou AY, Kakar RK, Neeck S et al (2014) The global precipitation measurement mission. Bull Am Meteorol Soc 95:701–722. https://doi.org/10.1175/BAMS-D-13-00164.1

Huete A, Justice C, Liu H (1994) Development of vegetation and soil indices for MODIS-EOS. Remote Sens Environ 49:224–234. https://doi.org/10.1016/0034-4257(94)90018-3

Kummerow C, Barnes W, Kozu T et al (1998) The tropical rainfall measuring mission (TRMM) sensor package. J Atmos Ocean Technol 15:809–817. https://doi.org/10.1175/1520-0426(1998)015%3c0809:TTRMMT%3e2.0.CO;2

Kustas WP, Norman JM (1996) Use of remote sensing for evapotranspiration monitoring over land surfaces. Hydrol Sci J 41:495–516. https://doi.org/10.1080/02626669609491522

Mecikalski JR, Diak GR, Anderson MC, Norman JM (1999) Estimating fluxes on continental scales using remotely sensed data in an atmospheric-land exchange model. J Appl Meteorol 38:1352–1369. https://doi.org/10.1175/1520-0450(1999)038%3c1352:EFOCSU%3e2.0.CO;2

Poortinga A, Clinton N, Saah D et al (2018) An operational before-after-control-impact (BACI) designed platform for vegetation monitoring at planetary scale. Remote Sens 10:760. https://doi.org/10.3390/rs10050760

Poortinga A, Nguyen Q, Tenneson K et al (2019) Linking Earth observations for assessing the food security situation in Vietnam: a landscape approach. Front Environ Sci 7:186. https://doi.org/10.3389/fenvs.2019.00186

Poortinga A, Aekakkararungroj A, Kityuttachai K et al (2020) Predictive analytics for identifying land cover change hotspots in the Mekong region. Remote Sens 12:1472. https://doi.org/10.3390/RS12091472

Potapov P, Tyukavina A, Turubanova S et al (2019) Annual continuous fields of woody vegetation structure in the Lower Mekong region from 2000–2017 Landsat time-series. Remote Sens Environ 232:111278. https://doi.org/10.1016/j.rse.2019.111278

Rouse Jr JW, Haas RH, Schell JA, Deering DW (1973) Paper a 20. In: Third earth resources technology Satellite-1 symposium: Section AB. Technical presentations, p 309

Saah D, Tenneson K, Matin M et al (2019) Land cover mapping in data scarce environments: challenges and opportunities. Front Environ Sci 7:150. https://doi.org/10.3389/fenvs.2019.00150

Saah D, Tenneson K, Poortinga A et al (2020) Primitives as building blocks for constructing land cover maps. Int J Appl Earth Obs Geoinf 85:101979. https://doi.org/10.1016/j.jag.2019.101979

Senay GB, Bohms S, Singh RK et al (2013) Operational evapotranspiration mapping using remote sensing and weather datasets: a new parameterization for the SSEB approach. J Am Water Resour Assoc 49:577–591. https://doi.org/10.1111/jawr.12057

Strangeways I (2010) A history of rain gauges. Weather 65:133–138. https://doi.org/10.1002/wea.548

Vogelmann JE, Rock BN (1985) Spectral characterization of suspected acid deposition damage in red spruce (Picea Rubens) stands from Vermont. In: Proceedings of the Airborne imaging spectrometer data analysis workshop

45 Defining Seasonality: First Date of No Snow

Amanda Armstrong, Morgan Tassone, and Justin Braaten

Overview

The purpose of this chapter is to demonstrate how to produce annual maps representing the first day within a year on which a given pixel reaches 0% snow cover. It also provides suggestions for summarizing and visualizing the results over time and space.

Learning Outcomes

- Generating and using a date band in image compositing.
- Applying temporal filtering to an `ImageCollection`.
- Identifying patterns of seasonal snowmelt.

A. Armstrong (✉)
Biospheric Sciences Laboratory, University of Maryland Baltimore County, GESTAR II NASA's Goddard Space Flight Center Code 618, Greenbelt, MD 20771, USA
e-mail: amanda.h.armstrong@nasa.gov

M. Tassone
Department of Environmental Sciences, University of Virginia, 290 McCormick Road, Charlottesville, VA 22902, USA
e-mail: mms3sh@virginia.edu

J. Braaten
Google Inc., 1600 Amphitheater Parkway, Mountain View, CA 94043, USA
e-mail: braaten@google.com

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_45

Helps if you know how to:

- Import images and image collections, filter, and visualize (Part 1).
- Perform basic image analysis: select bands, compute indices, create masks (Part 2).
- Work with time-series data in Earth Engine (Part 4).
- Fit linear and nonlinear functions with regression in an `ImageCollection` time series (Chap. 18).

45.1 Introduction to Theory

The timing of annual seasonal snowmelt (Fig. 45.1) and any potential change in that timing have broad ecological implications and thus impact human livelihoods, particularly in and around high-latitude and mountainous systems. The annual melting of accumulated winter snowfall, one of the most important phases of the hydrologic cycle within these regions, provides the dominant source of water for streamflow and groundwater recharge for approximately one sixth of the global population (Musselman et al. 2017; Barnhart et al. 2016; Bengtsson 1976). The timing of snowmelt in the Arctic and Antarctic influences the length of the growing season, and consistent snow cover throughout the winter insulates vegetation from harsh temperatures and wind (Duchesne et al. 2012; Kudo et al. 1999). In mountainous regions, such as the Himalayas, snowmelt is a major source of freshwater downstream (Barnhart et al. 2016) and is essential in recharging groundwater.

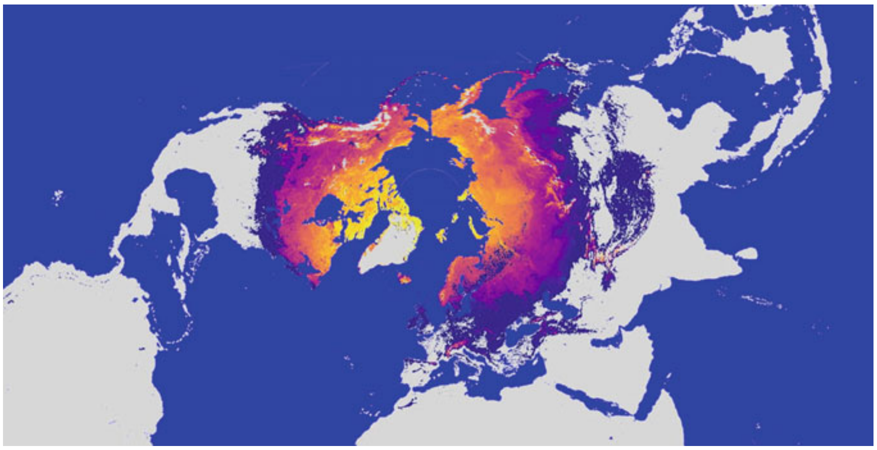


Fig. 45.1 Arctic polar stereographic projection showing the pattern of snowmelt timing in the Northern Hemisphere. The image shows the first day in 2018 on which each pixel no longer contained snow, as detected by the Moderate Resolution Imaging Spectroradiometer (MODIS) Snow Cover Daily Global product. The color grades from purple (earlier) to yellow (later)

This seasonal water resource is one of the fastest-changing hydrologic systems under Earth's warming climate, and these changes will broadly impact regional economies and ecosystem functioning and increase the potential for flood hazards (Musselman et al. 2017; IPCC 2007; Beniston 2012; Allan and Casillo 2007; Barnett and Lettenmaier 2005). An analysis focusing on the Yamal Peninsula in the northwestern Siberian tundra found that the timing of snowmelt (calculated using the methods outlined here) was an important predictor of differences in ecosystem functioning across the landscape (Tassone et al. 2020).

The anticipated warmer temperatures will alter the type and onset of precipitation. Multiple regions, including the Rocky Mountains of North America, have already measured a reduction in snowpack volume, and warmer temperatures have shifted precipitation from snowfall to rain, causing snowmelt to occur earlier (Barnhart et al. 2016; Clow 2010; Harpold et al. 2012).

This tutorial demonstrates how to calculate the first day of no snow annually at the pixel level, providing the user with the ability to track the seasonal and interannual variability in the timing of snowmelt toward a better understanding of how the hydrological cycles of higher-latitude and mountainous regions are responding to climate change.

45.2 Practicum

45.2.1 Section 1: Identifying the First Day of 0% Snow Cover

This section covers building an `ImageCollection` where each image is a mosaic of pixels that describe the first day in a given year that 0% snow cover was recorded. Snow cover is defined by the MODIS Normalized Difference Snow Index (NDSI) Snow Cover Daily Global product. The general workflow is as follows.

1. Define the date range to consider for analysis.
2. Define a function that adds date information as bands to snow cover images.
3. Define an analysis mask.
4. For each year:
   a. Filter the `ImageCollection` to observations from the given year.
   b. Add date bands to the filtered images.
   c. Identify the first day of the year without any snow per pixel.
   d. Apply an analysis mask to the image mosaic.
   e. Summarize the findings with a series of visualizations.

45.2.1.1 Section 1.1: Define the Date Range

First, we specify the day of year (DOY) on which to start the search for the first day with 0% snow cover. For applications in the Northern Hemisphere, you will likely want to start with January 1 (DOY 1). However, if you are studying snowmelt timing in the Southern Hemisphere (e.g., the Andes), where snowmelt can occur on dates on either side of January 1, it is more appropriate to start the year on July 1 (DOY 183), for instance. In this calculation, a year is defined as the 365 days beginning from the specified `startDoy`.

```
var startDoy = 1;
```

Then, we define the years to start and end tracking snow cover fraction. All years in the range will be included in the analysis.

```
var startYear = 2000;
var endYear = 2019;
```

45.2.1.2 Section 1.2: Define the Date Bands

Next, we will define a function to add several date bands to the images; the added bands will be used in a future step. Each image has a metadata timestamp property, but since we will be creating annual image mosaics composed of pixels from many different images, the date needs to be encoded per pixel as a value in an image band so that it is retained in the final mosaics. The function encodes:

- Calendar DOY (`calDoy`): enumerated DOY from January 1.
- Relative DOY (`relDoy`): enumerated DOY from a given `startDoy`.
- Milliseconds elapsed since the Unix epoch (`millis`).
- Year (`year`): Note that the year is tied to the `startDoy`. For example, if the `startDoy` is set at 183, the analysis will cross into the next calendar year, and the `year` given to all pixels will be the earlier year, even if a particular image was collected on or after January 1 of the subsequent year.

Additionally, two global variables are initialized (`startDate` and `startYear`) that will be redefined iteratively in a subsequent step.

```
var startDate;
var startYear;

function addDateBands(img) {
    // Get image date.
    var date = img.date();
    // Get calendar day-of-year.
    var calDoy = date.getRelative('day', 'year');
    // Get relative day-of-year; enumerate from user-
defined startDoy.
    var relDoy = date.difference(startDate, 'day');
    // Get the date as milliseconds from Unix epoch.
    var millis = date.millis();
    // Add all of the above date info as bands to the snow
fraction image.
    var dateBands = ee.Image.constant([calDoy, relDoy,
millis,
            startYear
        ])
        .rename(['calDoy', 'relDoy', 'millis', 'year']);
    // Cast bands to correct data type before returning
the image.
    return img.addBands(dateBands)
        .cast({
            'calDoy': 'int',
            'relDoy': 'int',
            'millis': 'long',
            'year': 'int'
        })
        .set('millis', millis);
}
```

45.2.1.3 Section 1.3: Define an Analysis Mask

Here is the opportunity to define a mask for your analysis. This mask can be used to constrain the analysis to certain latitudes, land cover types, geometries, etc. In this case, we will (1) mask out water so that the analysis is confined to pixels over landforms only, (2) mask out pixels that have very few days of snow cover, and (3) mask out pixels that are snow covered for a good deal of the year (e.g., glaciers).

Import the MODIS Land/Water Mask dataset, select the `'water_mask'` band, and set all land pixels to value 1.

```
var waterMask =
ee.Image('MODIS/MOD44W/MOD44W_005_2000_02_24')
    .select('water_mask')
    .not();
```

Import the MODIS Snow Cover Daily Global 500 m product and select the 'NDSI_Snow_Cover' band.

```
var completeCol = ee.ImageCollection('MODIS/006/MOD10A1')
    .select('NDSI_Snow_Cover');
```

Mask pixels based on the frequency of snow cover.

```
// Pixels must have been 10% snow covered for at least 2
weeks in 2018.
var snowCoverEphem = completeCol.filterDate('2018-01-01',
        '2019-01-01')
    .map(function(img) {
        return img.gte(10);
    })
    .sum()
    .gte(14);

// Pixels must not be 10% snow covered more than 124 days
in 2018.
var snowCoverConst = completeCol.filterDate('2018-01-01',
        '2019-01-01')
    .map(function(img) {
        return img.gte(10);
    })
    .sum()
    .lte(124);
```

Combine the water mask and the snow cover frequency masks.

```
var analysisMask =
waterMask.multiply(snowCoverEphem).multiply(
    snowCoverConst);
```

45.2.1.4 Section 1.4: Identify the First Day of the Year Without Snow per Pixel, per Year

Make a list of the years to process. The input variables were defined in Sect. 45.2.1.1.

```
var years = ee.List.sequence(startYear, endYear);
```

Map the following function over the list of years. For each year, identify the first day with 0% snow cover.

1. Define the start and end dates to filter the dataset for the given year.
2. Filter the ImageCollection by the date range.
3. Add the date bands to each image in the filtered collection.
4. Sort the filtered collection by date. (Note: To determine the first day with snow accumulation in the fall, reverse sort the filtered collection.)
5. Make a mosaic using the min reducer to select the pixel with 0 (minimum) snow cover. Since the collection is sorted by date, the first image with 0 snow cover is selected. This operation is conducted per pixel to build the complete image mosaic.
6. Apply the analysis mask to the resulting mosaic.

An ee.List of images is returned.

```
var annualList = years.map(function(year) {
    // Set the global startYear variable as the year being
worked on so that
    // it will be accessible to the addDateBands mapped to
the collection below.
    startYear = year;
    // Get the first day-of-year for this year as an
ee.Date object.
    var firstDoy = ee.Date.fromYMD(year, 1, 1);
    // Advance from the firstDoy to the user-defined
startDay; subtract 1 since
    // firstDoy is already 1. Set the result as the global
startDate variable so
    // that it is accessible to the addDateBands mapped to
the collection below.
    startDate = firstDoy.advance(startDoy - 1, 'day');
    // Get endDate for this year by advancing 1 year from
startDate.
    // Need to advance an extra day because end date of
filterDate() function
    // is exclusive.
    var endDate = startDate.advance(1, 'year').advance(1,
        'day');
    // Filter the complete collection by the start and end
dates just defined.
```

```
    var yearCol = completeCol.filterDate(startDate,
endDate);
    // Construct an image where pixels represent the first
day within the date
    // range that the lowest snow fraction is observed.
    var noSnowImg = yearCol
        // Add date bands to all images in this particular
collection.
        .map(addDateBands)
        // Sort the images by ascending time to identify
the first day without
        // snow. Alternatively, you can use
.sort('millis', false) to
        // reverse sort (find first day of snow in the
fall).
        .sort('millis')
        // Make a mosaic composed of pixels from images
that represent the
        // observation with the minimum percent snow cover
(defined by the
        // NDSI_Snow_Cover band); include all associated
bands for the selected
        // image.
        .reduce(ee.Reducer.min(5))
        // Rename the bands - band names were altered by
previous operation.
        .rename(['snowCover', 'calDoy', 'relDoy',
'millis',
            'year'
        ])
        // Apply the mask.
        .updateMask(analysisMask)
        // Set the year as a property for filtering by
later.
        .set('year', year);

    // Mask by minimum snow fraction - only include pixels
that reach 0
    // percent cover. Return the resulting image.
    return
noSnowImg.updateMask(noSnowImg.select('snowCover')
        .eq(0));
});
```

Convert the `ee.List` of images to an `ImageCollection`.

```
var annualCol = ee.ImageCollection.fromImages(annualList);
```

Code Checkpoint A26a. The book's repository contains a script that shows what your code should look like at this point.

45.2.2 Section 2: Data Summary and Visualization

The following is a series of examples for how to display and explore the "first DOY with no snow" dataset you just generated.

- These examples refer to the calendar date (`calDoy` band) when displaying and incorporating date information in calculations. If you are using a date range that begins on any day other than January 1 (DOY 1) you may want to replace `calDoy` with `relDoy` in all cases below.
- Results may appear different as you zoom in and out of the **Map** because of tile aggregation, which is described in the Earth Engine documentation. It is best to view **Map** data interactively with a relatively high zoom level. Additionally, for any analysis where a function provides a scale parameter (e.g., region reduction, exporting results), it is best to define it with the native resolution of the dataset (500 m).
- MODIS cloud masking can influence results. If there are a number of sequentially masked image pixel observations (e.g., clouds, poor quality), the actual date of the first observation with 0% snow cover may be earlier than identified in the image time series. Regional patterns may be less influenced by this bias than local results. For local results, please inspect image masks to understand their influence on the dates near snowmelt timing.

45.2.2.1 Section 2.1: Single-Year Map

Filter a single year from the collection (2018 in the example below) and display the image to the **Map** to see spatial patterns of snowmelt timing. Setting the `min` and `max` parameters of the `visArgs` variable to a narrow range around expected snowmelt timing is important for getting a good color stretch.

```
// Define a year to visualize.
var thisYear = 2018;

// Define visualization arguments.
var visArgs = {
    bands: ['calDoy'],
    min: 150,
    max: 200,
    palette: [
        '0D0887', '5B02A3', '9A179B', 'CB4678', 'EB7852',
        'FBB32F', 'F0F921'
    ]
};

// Subset the year of interest.
var firstDayNoSnowYear =
annualCol.filter(ee.Filter.eq('year',
    thisYear)).first();

// Display it on the map.
Map.setCenter(-95.78, 59.451, 5);
Map.addLayer(firstDayNoSnowYear, visArgs,
    'First day of no snow, 2018');
```

Running this code produces something similar to Fig. 45.2. The color represents the DOY when 0% snow cover was first observed per pixel (blue is earlier, yellow is later).

One can notice a number of interesting patterns. Frozen lakes have been shown to decrease air temperatures in adjacent pixels, resulting in delayed snowmelt (Rouse et al. 1997; Salomonson and Appel 2004; Wang and Derksen 2008). Additionally, the protected estuaries of the Northwest Passages have earlier dates of no snow compared to the landscapes exposed to the currents and winds of the Northern Atlantic. Latitude, elevation, and proximity to ocean currents are the strongest determinants in this region.

Note that pixels representing glaciers that did not get removed by the analysis mask can produce anomalies in the data. Since glaciers are generally snow covered, the DOY with the least snow cover according to the MODIS Snow Cover Daily Global product is presented in the **Map**. In Fig. 45.2, this is evident in the abrupt transition within alpine areas of Baffin Island (white pixels represent glaciers in this case).

45.2.2.2 Section 2.2: Year-to-Year Difference Map

Compare year-to-year difference in snowmelt timing by selecting two years of interest from the collection and subtracting them. Here, we are calculating the difference in snowmelt timing between 2005 and 2015.

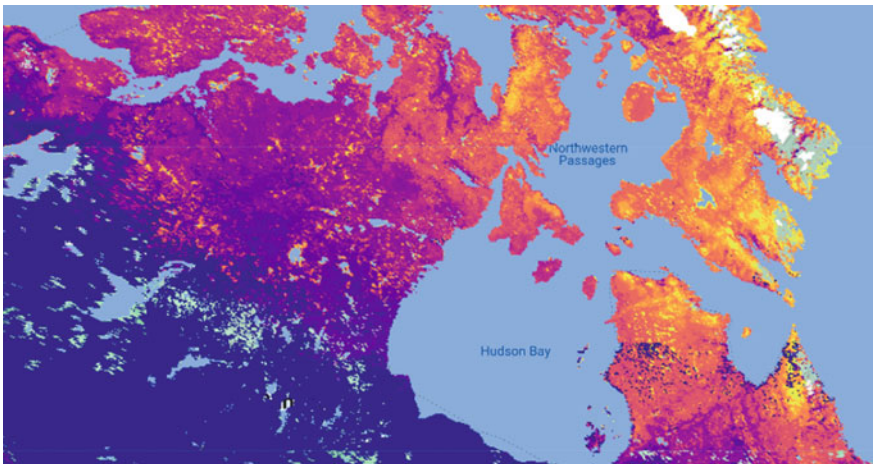


Fig. 45.2 Thematic map representing the first DOY with 0% snow cover. Color grades from blue to yellow as the DOY increases

```
// Define the years to difference.
var firstYear = 2005;
var secondYear = 2015;

// Calculate difference image.
var firstImg = annualCol.filter(ee.Filter.eq('year',
firstYear))
    .first().select('calDoy');
var secondImg = annualCol.filter(ee.Filter.eq('year',
secondYear))
    .first().select('calDoy');
var dif = secondImg.subtract(firstImg);

// Define visualization arguments.
var visArgs = {
    min: -15,
    max: 15,
    palette: ['b2182b', 'ef8a62', 'fddbc7', 'f7f7f7',
'd1e5f0',
        '67a9cf', '2166ac'
    ]
};

// Display it on the map.
Map.setCenter(95.427, 29.552, 8);
Map.addLayer(dif, visArgs, '2015-2005 first day no snow
dif');
```

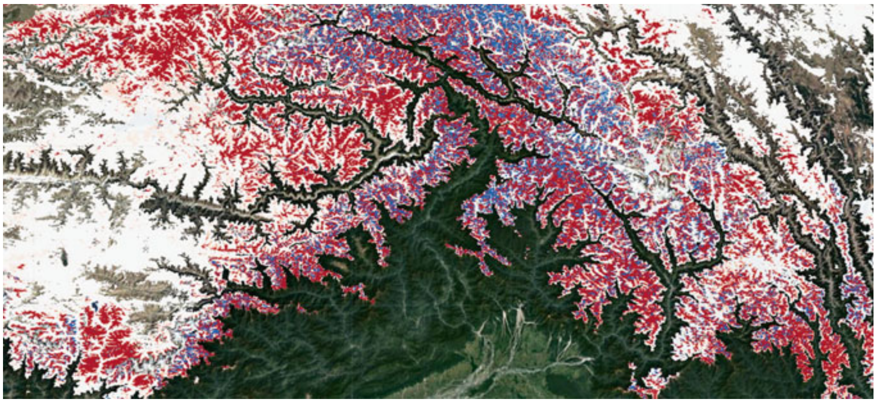


Fig. 45.3 Year-to-year (2005–2015) difference map of the Himalayas on the Nepal–China border. Color grades from red to blue, with red indicating an earlier date of no snow in 2015 and blue indicating a later date of no snow in 2015. White areas indicate little or no change

Running this code produces something similar to Fig. 45.3. The color represents the difference, in each pixel, between the 2005 and 2015 DOY when 0% snow cover was first observed. Red represents a negative change (an earlier no-snow date in 2015), blue represents a positive change (a later no-snow date in 2015), and white represents a negligible or no change in the no-snow dates for 2005 and 2015.

45.2.2.3 Section 2.3: Trend Analysis Mapping

It is also possible to identify trends in the shifting first DOY with no snow by calculating the slope through a pixel's time-series points. Here, the slope for each pixel is calculated with year as the *x* variable and the first DOY with no snow as the *y* variable.

```
// Calculate slope image.
var slope = annualCol.sort('year').select(['year',
'calDoy'])
    .reduce(ee.Reducer.linearFit()).select('scale');

// Define visualization arguments.
var visArgs = {
    min: -1,
    max: 1,
    palette: ['b2182b', 'ef8a62', 'fddbc7', 'f7f7f7',
        'd1e5f0', '67a9cf', '2166ac'
    ]
};

// Display it on the map.
Map.setCenter(11.25, 59.88, 6);
Map.addLayer(slope, visArgs, '2000-2019 first day no snow
slope');
```

The result is a map (Fig. 45.4) where red represents a negative slope (progressively earlier first DOY with no snow), white represents a slope of 0, and blue represents a positive slope (progressively later first DOY with no snow). In southern Norway and Sweden, the trend in the first DOY with no snow between 2002 and 2019 appears to be influenced by various factors. Coastal areas exhibit progressively earlier first DOY of no snow than inland areas; however, high variability in slopes can be observed around fjords and in mountainous regions. This map reveals the complexity of seasonal snow dynamics in these areas.

Note that goodness-of-fit calculations are not measured here, nor is significance considered. While a slope can indicate regional trends, more local trends should be investigated using a time-series chart (see the next section). Interannual variability can be influenced by masked pixels, as described above.

45.2.2.4 Section 2.4: Time-Series Chart

To visually understand the temporal patterns of the first DOY with no snow through time, we can display our results in a time-series chart. In this case, we have defined a circle with a radius of 500 m around a point of interest and calculated the annual mean first DOY with no snow for pixels within that circle.

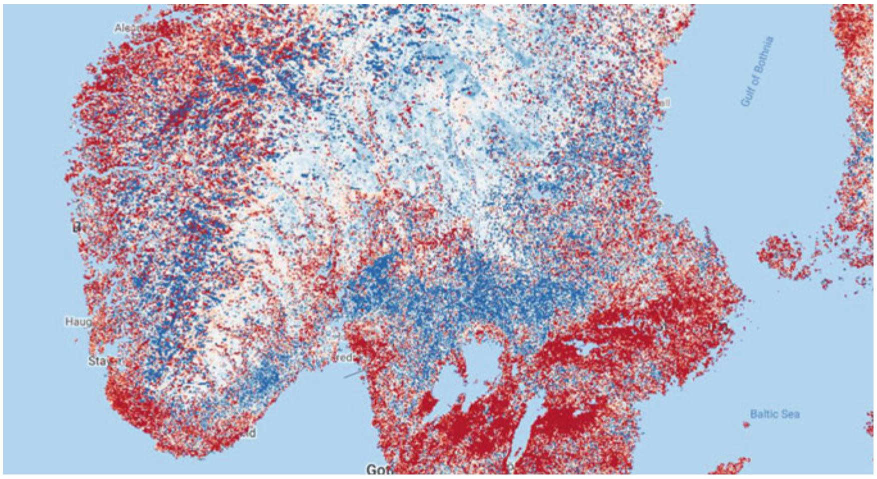


Fig. 45.4 Map representing the slope of the first DOY with no snow cover between 2000 and 2019. The slope represents the overall trend. Color grades from red (negative slope) to blue (positive slope). White indicates areas of little to no change

```
// Define an AOI.
var aoi = ee.Geometry.Point(-94.242, 65.79).buffer(1e4);
Map.addLayer(aoi, null, 'Area of interest');

// Calculate annual mean DOY of AOI.
var annualAoiMean =
annualCol.select('calDoy').map(function(img) {
    var summary = img.reduceRegion({
        reducer: ee.Reducer.mean(),
        geometry: aoi,
        scale: 1e3,
        bestEffort: true,
        maxPixels: 1e14,
        tileScale: 4,
    });
    return ee.Feature(null, summary).set('year', img.get(
        'year'));
});
```

```
// Print chart to console.
var chart = ui.Chart.feature.byFeature(annualAoiMean,
'year',
        'calDoy')
    .setOptions({
        title: 'Regional mean first day of year with no
snow cover',
        legend: {
            position: 'none'
        },
        hAxis: {
            title: 'Year',
            format: '####'
        },
        vAxis: {
            title: 'Day-of-year'
        }
    });
print(chart);
```

Code Checkpoint A26b. The book's repository contains a script that shows what your code should look like at this point.

As is evident in the displayed results (Fig. 45.5), the first DOY with no snow was mostly stable from 2000 to 2012; following this, it has become more erratic.

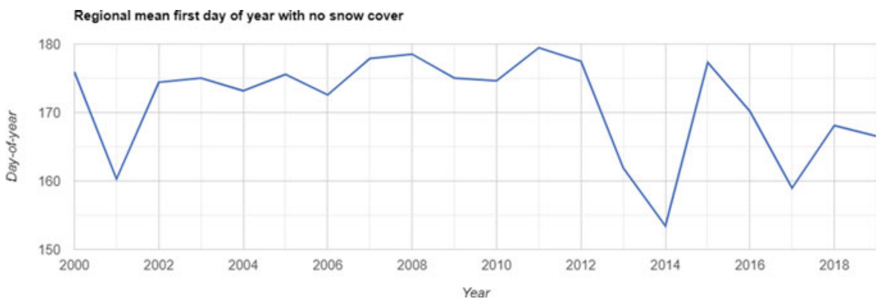


Fig. 45.5 Annual mean first DOY with no snow time series for pixels within a small region of interest

45.3 Synthesis

Assignment 1. In Sect. 45.2.2.4, a time-series chart for a region of interest was generated. Suppose you wanted to compare several regions in the same chart; how would you change the code to achieve this? For instance, try plotting the first DOY of no snow at points (− 69.271, 65.532) and (− 104.484, 65.445) in the same chart. Some helpful functions include `ee.Image.reduceRegions`, `ee.FeatureCollection.flatten`, and `ui.Chart.feature.groups`.

Assignment 2. The objective of this chapter was to identify the first DOY of 0% snow cover. However, Sect. 45.2.1.4 provides a suggestion for altering the code to identify the last day of 0% snow cover. Can you modify the code to achieve this result?

Assignment 3. How could you determine regions that are often masked during the time of year when snowmelt occurs? (The results from such regions might not be reliable.) Hint: Think about how you can use the `mask` function on images and investigate the `'NDSI_Snow_Cover_Class'` and `'Snow_Albedo_Daily_Tile_Class'` bands.

45.4 Conclusion

In this chapter, we provided a method to identify the annual, per pixel, first DOY with no snow from MODIS snow cover image data. The result can be used to investigate patterns of seasonal snowmelt spatially and temporally. The method relied on adding a date band to each image in the collection, temporal sorting, and `ImageCollection` reduction. We demonstrated several different ways to analyze the results using map and chart interpretation, image differencing, and linear regression.

References

Allan JD, Castillo MM (2007) Stream ecology: structure and function of running waters. Springer Nature

Barnhart TB, Molotch NP, Livneh B et al (2016) Snowmelt rate dictates streamflow. Geophys Res Lett 43:8006–8016. https://doi.org/10.1002/2016GL069690

Barnett TP, Adam JC, Lettenmaier DP (2005) Potential impacts of a warming climate on water availability in snow-dominated regions. Nature 438(7066):303–309. https://doi.org/10.1038/nature04141

Bengtsson L (1976) Snowmelt estimated from energy budget studies. Nord Hydrol 7:3–18. https://doi.org/10.2166/nh.1976.0001

Beniston M (2012) Impacts of climatic change on water and associated economic activities in the Swiss Alps. J Hydrol 412–413:291–296. https://doi.org/10.1016/j.jhydrol.2010.06.046

Clow DW (2010) Changes in the timing of snowmelt and streamflow in Colorado: a response to recent warming. J Clim 23:2293–2306. https://doi.org/10.1175/2009JCLI2951.1

Duchesne L, Houle D, D'Orangeville L (2012) Influence of climate on seasonal patterns of stem increment of balsam fir in a boreal forest of Québec, Canada. Agric for Meteorol 162–163:108–114. https://doi.org/10.1016/j.agrformet.2012.04.016

Harpold A, Brooks P, Rajagopal S et al (2012) Changes in snowpack accumulation and ablation in the intermountain west. Water Resour Res 48. https://doi.org/10.1029/2012WR011949

Kudo G, Nordenhäll U, Molau U (1999) Effects of snowmelt timing on leaf traits, leaf production, and shoot growth of alpine plants: comparisons along a snowmelt gradient in northern Sweden. Ecoscience 6:439–450. https://doi.org/10.1080/11956860.1999.11682543

Musselman KN, Clark MP, Liu C et al (2017) Slower snowmelt in a warmer world. Nat Clim Chang 7:214–219. https://doi.org/10.1038/nclimate3225

Rouse WR, Douglas MSV, Hecky RE et al (1997) Effects of climate change on the freshwaters of arctic and subarctic North America. Hydrol Process 11:873–902. https://doi.org/10.1002/(SICI)1099-1085(19970630)11:8%3c873::AID-HYP510%3e3.0.CO;2-6

Salomonson VV, Appel I (2004) Estimating fractional snow cover from MODIS using the normalized difference snow index. Remote Sens Environ 89:351–360. https://doi.org/10.1016/j.rse.2003.10.016

Solomon S, Manning M, Marquis M et al (2007) Climate change 2007—the physical science basis: working group I contribution to the fourth assessment report of the IPCC. Cambridge University Press

Tassone M, Epstein HE (2020) Drivers of spatial and temporal variability in vegetation productivity on the Yamal Peninsula, Siberia, Russia. In: AGU fall meeting abstracts, pp B084-04

Wang L, Derksen C, Brown R (2008) Detection of pan-Arctic terrestrial snowmelt from QuikSCAT, 2000–2005. Remote Sens Environ 112:3794–3805. https://doi.org/10.1016/j.rse.2008.05.017

Part IX
Terrestrial Applications

Earth’s terrestrial surface is analyzed regularly by satellites, in search of both change and stability. These are of great interest to a wide cross-section of Earth Engine users, and projects across large areas illustrate both the challenges and opportunities for life on Earth. Chapters in this part illustrate the use of Earth Engine for disturbance, understanding long-term changes of rangelands, and creating optimum study sites.

Active Fire Monitoring

Morgan A. Crowley and Tianjia Liu

Overview

Fire monitoring across the world benefits from raw satellite imagery and processed fire mapping datasets. Google Earth Engine supports fire monitoring throughout fire seasons with satellite data from sources like Landsat 8, Sentinel-2, and Moderate Resolution Imaging Spectroradiometer (MODIS), and by hosting multiple fire datasets from the Geostationary Operational Environmental Satellite (GOES) and the Fire Information for Resource Management System (FIRMS). In this chapter, you will access, process, and explore three fire monitoring datasets available in the data catalog. By the end of this chapter, you will learn how to use the Code Editor and user apps to summarize and compare the characteristics of fires, fire seasons, and fire monitoring datasets.

Learning Outcomes

- Accessing and visualizing fire monitoring datasets in the JavaScript UI.
- Adjusting previously drafted code to calculate fire characteristics in the JavaScript UI for a fire of your choice.
- Exploring fire metrics and visualization with user apps.

Morgan A. Crowley and Tianjia Liu: Shared first-authorship.

M. A. Crowley (✉)
Canadian Forest Service—Great Lakes Forestry Centre, Natural Resources Canada, 1219 Queen Street E, Sault Ste. Marie, ON, Canada
e-mail: morgan.crowley@nrcan-rncan.gc.ca

T. Liu
Department of Earth and Planetary Sciences, Harvard University, 20 Oxford Street, Cambridge, MA, USA
e-mail: tianjia.liu@columbia.edu

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_46

- Identifying pros and cons of different fire datasets for a variety of monitoring objectives.

Helps if you know how to:

- Import images and image collections, filter, and visualize (Part 1).
- Write a function and map it over an `ImageCollection` (Chap. 12).
- Filter a `FeatureCollection` to obtain a subset (Chaps. 22 and 23).
- Design user interfaces for an Earth Engine App (Chap. 30).
- Access and interact with previously made Earth Engine user apps (Chap. 30).

46.1 Introduction to Theory

Fire has many roles around the world. It is both a naturally occurring ecological process in fire-prone regions and a tool used by humans for land and resource management. However, fires and their emissions continue to have more extreme impacts as human settlements expand, climatic conditions become less predictable, and fire seasons lengthen (Jolly et al. 2015). To better identify and quantify the effects of fires across the globe, it is vital to monitor fires using various methods, including hand-drawn maps, ground-based sensors, GPS tracking, aerial surveys, imagery collection, and satellite-based data (Andela et al. 2017; Archibald et al. 2009; Nogueira et al. 2017; Stinson et al. 2011; Veraverbeke et al. 2014).

Different sources of satellite imagery can be used to visualize fire conditions and progressions, calculate band ratios reflecting disturbance and fire severity, and map burned areas with training data-informed classification algorithms (Crowley et al. 2019a, b; Hawbaker et al. 2017; Hermosilla et al. 2018; Parks 2014; Parks et al. 2019; Veraverbeke et al. 2014). Many premade fire datasets are readily available for monitoring global fire locations, extents, and progressions (Andela et al. 2019; Chuvieco et al. 2016; Giglio et al. 2016; Humber et al. 2019). In the case of existing fire datasets available for large spatial and temporal extents, remote sensing scientists apply their robust classification algorithms on satellite imagery and other geospatial data. Earth Engine makes fire monitoring more accessible by sharing multiple data sources in the data catalog so users can easily access and process these data to meet their desired objectives (Liu and Crowley 2021).

46.2 Practicum

46.2.1 Section 1: Fire Datasets in Google Earth Engine

In the following example, we use MODIS and GOES datasets (Table 46.1) to map the Bobcat Fire, a megafire that burned 115,796 acres in Los Angeles County, California, in September 2020.

First, we need to define temporal and spatial variables to filter the datasets, namely the approximate ignition coordinates and start date of the fire.

```
// Define the location of the fire.
var lon = -117.868;
var lat = 34.241;
var zoom = 9;

// Filter datasets to a specific date range:
// start date of fire.
var inYear = 2020;
var inMonth = 9;
var inDay = 6;
```

Using the ignition date (September 6, 2020), we can define separate date ranges to filter the active fire and burned area datasets to account for differences in the temporal structure of the datasets, i.e., daily versus monthly. Here, we set the temporal filter range for active fire datasets as the two-week period starting from the ignition date and for the burned area dataset as the month of September. The duration variables can be modified according to the fire of interest.

Table 46.1 MODIS and GOES fire mapping datasets available in the Google Earth Engine data catalog

Satellite/sensor	Dataset	Variable	Resolution	Dataset coverage
MODIS Terra, MODIS Aqua*	MOD/MYD14A1	Active fires	1 km, daily	Global, 2001 to present
	MCD64A1	Burned area	500 m, monthly**	Global, 2001 to present
GOES-16, GOES-17*	FDCF	Active fires	2 km, every 15 min	North/South America, 2017 to present

* MODIS Terra (from 2000), MODIS Aqua (from 2002), GOES-16 (from 2016), GOES-17 (from 2017)
** Can be disaggregated into daily resolution using the MCD64A1 burn date variable

```
var durationAF = 15; // in days
var durationBA = 1; // in months

// Date range for active fires.
var startDateAF = ee.Date.fromYMD(inYear, inMonth, inDay);
var endDateAF = startDateAF.advance(durationAF, 'day');

// Date range for burned area.
var startDateBA = ee.Date.fromYMD(inYear, inMonth, 1);
var endDateBA = startDateBA.advance(durationBA, 'month');
```

With these input variables defined, we can preprocess the fire datasets. We will upload a high-resolution reference perimeter provided by the U.S. National Interagency Fire Center (NIFC) and then add the MODIS and GOES datasets from the Earth Engine data catalog.

Reference Fire Perimeter

The NIFC produces wildland fire perimeters using information from local fire agencies. For this tutorial, we uploaded the NIFC perimeter for the Bobcat Fire, converting it from a shapefile to an Earth Engine asset. We will access the asset from the book repository and use it as a reference layer to compare with the MODIS and GOES datasets.

```
// -------------------------------
// 1. Reference Perimeter (WFIGS)
// -------------------------------
// Note: each fire has multiple versions, so here we are
// filtering WFIGS by the name of the fire, sorting the
// area of the filtered polygons in descending order,
// and retrieving the polygon with the highest area.
var WFIGS = ee.FeatureCollection(
    'projects/gee-book/assets/A3-1/WFIGS');
var reference =
ee.Feature(WFIGS.filter(ee.Filter.eq('irwin_In_1',
        'BOBCAT'))
    .sort('poly_Acres', false).first());
```

MODIS Active Fire Products

The gridded 1 km MODIS active fire datasets, MOD14A1 (Terra) and MYD14A1 (Aqua), have daily collection rates and global coverage (Giglio et al. 2016; Giglio 2010). The MODIS sensor is mounted on the two separate satellites, Terra and Aqua, both operated by NASA for environmental monitoring. Here, we define two functions to process the fire mask and fire radiative power (FRP) variables. The fire mask provides a categorical classification of the confidence in fire detection, where values ≥ 7 indicate that fire is present. FRP is a continuous variable that is a proxy for fire intensity, in units of megawatts (MW). Note that MODIS FRP must be multiplied by 0.1 to be in units of MW.

```
// -------------------------------
// 2. MODIS active fire datasets
// -------------------------------
// MOD14A1, MYD14A1 = MODIS/Terra and Aqua active fires and
thermal anomalies
// resolution: daily, gridded at 1km in sinusoidal
projection (SR-ORG:6974)
// variables: fire mask (FireMask), fire radiative power in
MW (MaxFRP)
// satellite overpasses: Terra (10:30am/pm local time),
Aqua (1:30am/pm local time)

// Define the Earth Engine paths for MOD14A1 and MYD14A1,
collection 6.
var mod14a1 = ee.ImageCollection('MODIS/006/MOD14A1');
var myd14a1 = ee.ImageCollection('MODIS/006/MYD14A1');

// Filter the datasets according to the date range.
var mod14a1Img = mod14a1.filterDate(startDateAF,
endDateAF);
var myd14a1Img = myd14a1.filterDate(startDateAF,
endDateAF);

var getFireMask = function(image) {
    // Fire Mask (FireMask): values ≥ 7 are active fire
pixels
    return image.select('FireMask').gte(7);
};

var getMaxFRP = function(image) {
    // FRP (MaxFRP): MaxFRP needs to be scaled by 0.1 to be
in units of MW.
    return image.select('MaxFRP').multiply(0.1);
};

// Define the active fire mask (count of active fire
pixels).
var mod14a1ImgMask = mod14a1Img.map(getFireMask).sum();
var myd14a1ImgMask = myd14a1Img.map(getFireMask).sum();
```

```
// Define the total FRP (MW).
var mod14a1ImgFrp = mod14a1Img.map(getMaxFRP).sum();
var myd14a1ImgFrp = myd14a1Img.map(getMaxFRP).sum();
```

MODIS Burned Area Product

The gridded 500 m MODIS burned area dataset, MCD64A1, is monthly with global coverage but can be disaggregated to daily resolution with its burn date variable (Giglio et al. 2016; Humber et al. 2019). Here, we define a function to retrieve the burn date.

```
// ------------------------------
// 3. MODIS burned area dataset
// ------------------------------
// MCD64A1 = MODIS/Terra and Aqua combined burned area
// resolution: monthly, gridded at 500m in sinusoidal
projection (SR-ORG:6974),
// can be disaggregated to daily resolution
// variables: burn date as day of year (BurnDate)

// Define the Earth Engine paths for MCD64A1, collection 6.
var mcd64a1 = ee.ImageCollection('MODIS/006/MCD64A1');

var getBurnDate = function(image) {
    // burn day of year (BurnDate)
    return image.select('BurnDate');
};

// Define the burned area mask.
var mcd64a1Img = mcd64a1.filterDate(startDateBA,
endDateBA);
var mcd64a1ImgMask = mcd64a1Img.map(getBurnDate).min();
```

GOES Active Fire Products

The gridded 2 km GOES-16 (East) and GOES-17 (West) active fire datasets cover North and South America in the full disk version (FDCF) with a temporal revisit rate of 15-min increments (Hall et al. 2019; Schroeder et al. 2008). The two GOES satellites are operated by the National Oceanic and Atmospheric Administration (NOAA) and are primarily used for meteorological monitoring. Note that the pixel orientation and shape differ between GOES-16 and GOES-17 because of the different viewing angles of the two satellites.

```
// 4. GOES 16/17 active fires
// ------------------------------
// GOES-16/17 - geostationary satellites over North/South
America
// resolution: every 10-30 minutes, 2 km
// variables: fire mask (Mask), FRP (Power)

// Define the Earth Engine paths for GOES-16/17.
var goes16 = ee.ImageCollection('NOAA/GOES/16/FDCF');
var goes17 = ee.ImageCollection('NOAA/GOES/17/FDCF');

var filterGOES = ee.Filter.calendarRange(0, 0, 'minute');

// Filter the datasets according to the date range.
var goes16Img = goes16.filterDate(startDateAF, endDateAF)
    .filter(filterGOES);
var goes17Img = goes17.filterDate(startDateAF, endDateAF)
    .filter(filterGOES);

var getFireMask = function(image) {
    // fire mask (Mask): values from 10-35 are active fire
pixels,
    // see the description for QA values to filter out low
confidence fires
    return
image.select('Mask').gte(10).and(image.select('Mask')
        .lte(35));
};

var getFRP = function(image) {
    // FRP (Power), in MW
    return image.select('Power');
};

// Define the active fire mask (count of active fire
pixels).
var goes16ImgMask = goes16Img.map(getFireMask).sum();
var goes17ImgMask = goes17Img.map(getFireMask).sum();

// Define the total FRP (MW).
var goes16ImgFrp = goes16Img.map(getFRP).sum();
var goes17ImgFrp = goes17Img.map(getFRP).sum();
```

Now, we will visualize the three datasets, along with the reference Bobcat Fire perimeter, and plot the layers on the Earth Engine interactive map.

```
// -------------------------------
// 5. Map Visualization - Layers
// -------------------------------
// Use the 'Layers' dropdown menu on the map panel to
toggle on and off layers.
Map.addLayer(mod14a1ImgMask.selfMask(), {
    palette: 'orange'
}, 'MOD14A1');
Map.addLayer(myd14a1ImgMask.selfMask(), {
    palette: 'red'
}, 'MYD14A1');

Map.addLayer(mcd64a1ImgMask.selfMask(), {
    palette: 'black'
}, 'MCD64A1');

Map.addLayer(goes16ImgMask.selfMask(), {
    palette: 'skyblue'
}, 'GOES16', false);
Map.addLayer(goes17ImgMask.selfMask(), {
    palette: 'purple'
}, 'GOES17', false);

Map.setCenter(lon, lat, 9);
```

We can also visualize the datasets side by side as shown in Fig. 46.1 by using the `ui.Panel` and `ui.Map.Linker` using the code provided by the "Linked Maps" script under **Examples** > **User Interface** in the **Scripts** panel.

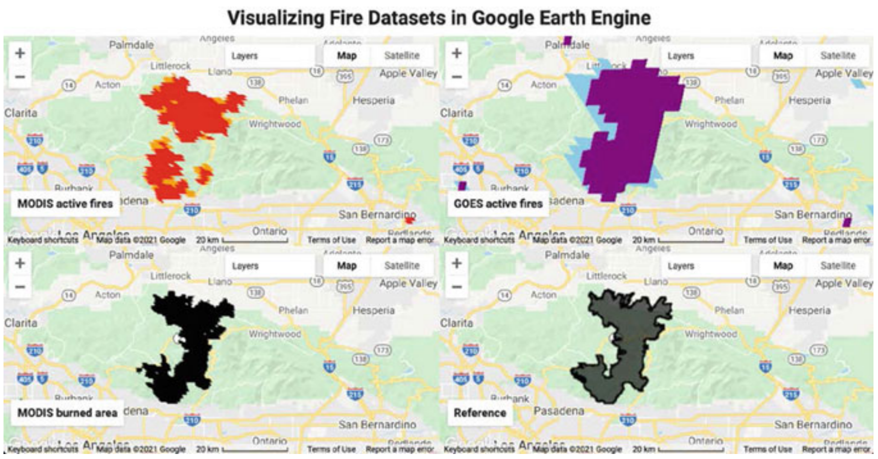


Fig. 46.1 Side-by-side panel comparison of the Bobcat Fire as seen in (clockwise from top left) MODIS active fires, GOES-16/17 active fires, MODIS burned area, and a reference fire perimeter from the WFIGS dataset

```
// ------------------------------------
// 6. Map Visualization - Panel Layout
// ------------------------------------

// Define the panel layout.
var panelNames = [
    'MODIS active fires', // panel 0 - top left
    'MODIS burned area', // panel 1 - bottom left
    'GOES active fires', // panel 2 - top right
    'Reference' // panel 3 - bottom right
];

// Create a map for each visualization option.
var maps = [];
panelNames.forEach(function(name, index) {
    var map = ui.Map();
    map.setControlVisibility({
        fullscreenControl: false
    });
    if (index === 0) {
        map.addLayer(mod14a1ImgMask.selfMask(), {
            palette: 'orange'
        }, 'MOD14A1');
        map.addLayer(myd14a1ImgMask.selfMask(), {
            palette: 'red'
        }, 'MYD14A1');
        map.add(ui.Label(panelNames[0], {
            fontWeight: 'bold',
            position: 'bottom-left'
        }));
    }
    if (index == 1) {
        map.addLayer(mcd64a1ImgMask.selfMask(), {
            palette: 'black'
        }, 'MCD64A1');
        map.add(ui.Label(panelNames[1], {
            fontWeight: 'bold',
            position: 'bottom-left'
        }));
    }
    if (index == 2) {

        map.addLayer(goes16ImgMask.selfMask(), {
            palette: 'skyblue'
        }, 'GOES16');
        map.addLayer(goes17ImgMask.selfMask(), {
            palette: 'purple'
        }, 'GOES17');
        map.add(ui.Label(panelNames[2], {
            fontWeight: 'bold',
            position: 'bottom-left'
        }));
    }
    if (index == 3) {
        map.addLayer(reference, {}, 'Reference');
        map.add(ui.Label(panelNames[3], {
            fontWeight: 'bold',
            position: 'bottom-left'
        }));
    }
    maps.push(map);
});
```

```
var linker = ui.Map.Linker(maps);

// Make a label for the main title of the app.
var title = ui.Label(
    'Visualizing Fire Datasets in Google Earth Engine', {
        stretch: 'horizontal',
        textAlign: 'center',
        fontWeight: 'bold',
        fontSize: '24px'
    });

// Define a map grid of 2x2 sub panels.
var mapGrid = ui.Panel(
    [
        ui.Panel([maps[0], maps[1]], null, {
            stretch: 'both'
        }),
        ui.Panel([maps[2], maps[3]], null, {
            stretch: 'both'
        })
    ],
    ui.Panel.Layout.Flow('horizontal'), {
        stretch: 'both'
    }
);
maps[0].setCenter(lon, lat, zoom);

// Add the maps and title to the ui.root().
ui.root.widgets().reset([title, mapGrid]);
ui.root.setLayout(ui.Panel.Layout.Flow('vertical'));
```

Code Checkpoint A31a. The book's repository contains a script that shows what your code should look like at this point.

Question 1. How does the burned area classification (i.e., burned versus unburned) and the spatial resolution (i.e., pixel size) differ across the three datasets?

Question 2. There appears to be a gap in fire activity between the MODIS incident data and burned area map for the Bobcat Fire, as shown in the split-panel app. What differences in the datasets might account for the mismatched fire classifications? Hint: Use https://worldview.earthdata.nasa.gov/ to examine raw MODIS imagery of the fire location and date.

Question 3. How does the temporal resolution of 15 min for GOES impact monitoring fires in the event of smoke and haze?

46.2.2 Section 2: In-Depth Visualization and Analysis of Fires in Earth Engine Apps

Earth Engine Apps help to curate in-depth visualization and analysis of fires. Here we present two apps using datasets from the Earth Engine public data catalog.

In the remainder of this chapter, you will use these two apps to learn more about the Bobcat Fire and to explore findings from the two datasets.

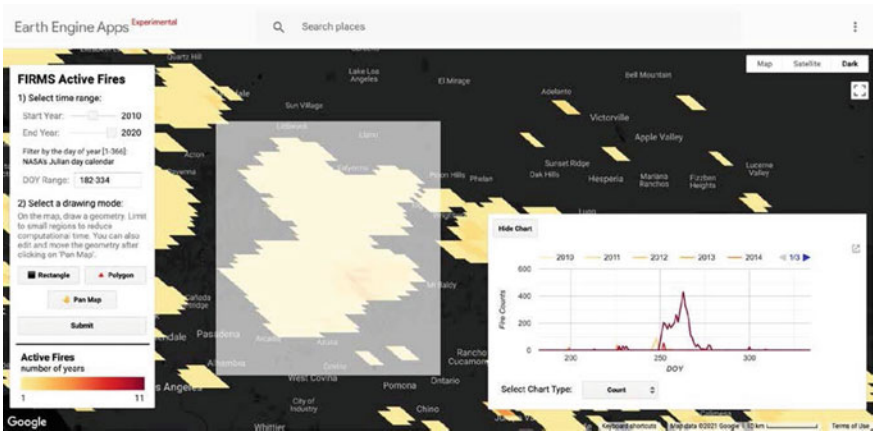
App 1: FIRMS Active Fires

FIRMS currently monitors active fires detected by MODIS, Visible Infrared Imaging Radiometer Suite (VIIRS), and NOAA-20 in near real time. FIRMS retains the coordinates of the centroid of pixels where one or more active fires are detected. The FIRMS dataset in the Earth Engine data catalog includes only MODIS active fires, gridded at 1 km spatial resolution. Note that the FIRMS dataset is meant for exploratory rather than rigorous scientific analyzes.

The "FIRMS Active Fires" Earth Engine app allows users to visualize the spatial and temporal variation in FIRMS active fires within a defined region.

Code Checkpoint A31b. The book's repository contains information about accessing the app.

Using the control panel, you can specify the date range (start year, end year, and day-of-year range) and draw a region of interest using either a rectangle or a polygon. The map shows the number of years that one or more active fires were detected in each pixel. The chart panel shows the total daily fire counts within the region as a timeseries, where each color represents a different year (Fig. 46.2). You can also change the chart type to display the cumulative active fire count.



FIRMS Active Fires

1) Select time range:

Start Year: 2010

End Year: 2020

Filter by the day of year [1-366]: NASA's Julian day calendar

DOY Range: 182-334

2) Select a drawing mode:

On the map, draw a geometry. Limit to small regions to reduce computational time. You can also edit and move the geometry after clicking on 'Pan Map'.

Active Fires
number of years
1 11

Select Chart Type: Count

Fig. 46.2 Bobcat Fire in the Earth Engine app FIRMS Active Fires. The chart panel on the lower right displays the history of active fires in the Bobcat Fire area from July–November in the years 2010–2020. The map layer shows the number of years in each pixel that had at least one active fire

Question 4. Navigate to the Bobcat Fire using the ignition coordinates (longitude, latitude). Using the satellite or map base layer, draw a polygon similar to the one shown in a Fig. 46.2. Submit your task and confirm your results with the above details.

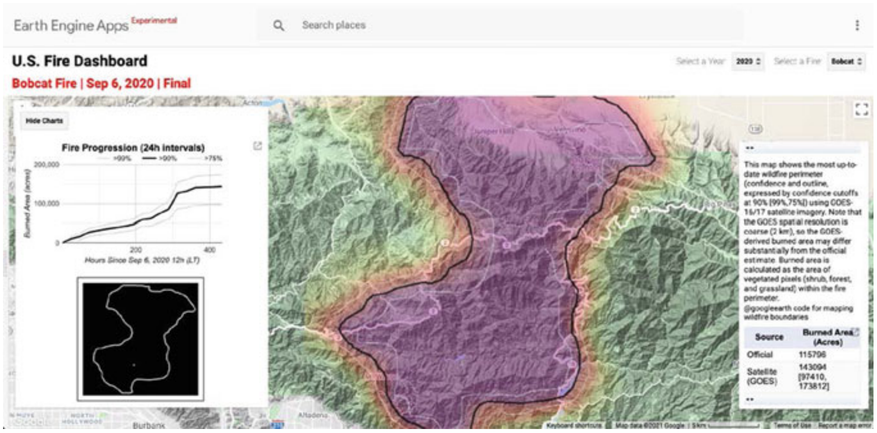
Code Checkpoint A31c. The book's repository contains information about how your app should look at this point.

Question 5. Examine the chart. Approximately how many days did the fire actively burn? Hint: Hover over the chart and compare the first DOY value and the final DOY value.

App 2: U.S. Fire Dashboard

In a more advanced app, the "U.S. Fire Dashboard", GOES-16/17 gridded active fires are used to calculate a smoothed burn perimeter for the various wildfires in 2020 by modifying the code from the Google Earth Engine Medium article, "How to generate wildfire boundary maps with Earth Engine" (Restif and Hoffman 2020). The code takes advantage of the different GOES-16 and GOES-17 pixel orientation and shape to downscale the burn classification to a finer resolution than that of the native GOES imagery. In the following example, we will use this app to visualize how the Bobcat Fire progressed in space and time from ignition (Fig. 46.3).

Code Checkpoint A31d. The book's repository contains information about accessing the app.



This map shows the most up-to-date wildfire perimeter (confidence and outline, expressed by confidence cutoffs at 90% [99%,75%]) using GOES-16/17 satellite imagery. Note that the GOES spatial resolution is coarse (2 km), so the GOES-derived burned area may differ substantially from the official estimate. Burned area is calculated as the area of vegetated pixels (shrub, forest, and grassland) within the fire perimeter.
@googleearth code for mapping wildfire boundaries

Source	Burned Area (Acres)
Official	115796
Satellite (GOES)	143094 [97410, 173812]

Fig. 46.3 Bobcat Fire in the Earth Engine app U.S. Fire Dashboard. The map displays the smoothed burn perimeter derived from GOES active fires, where the color gradient represents the confidence in the classification of the burned area. The black line shows the burn perimeter derived from pixels with > 90% confidence. The charts on the left-hand panel show the evolution of the Bobcat Fire in terms of its cumulative area and spatial growth

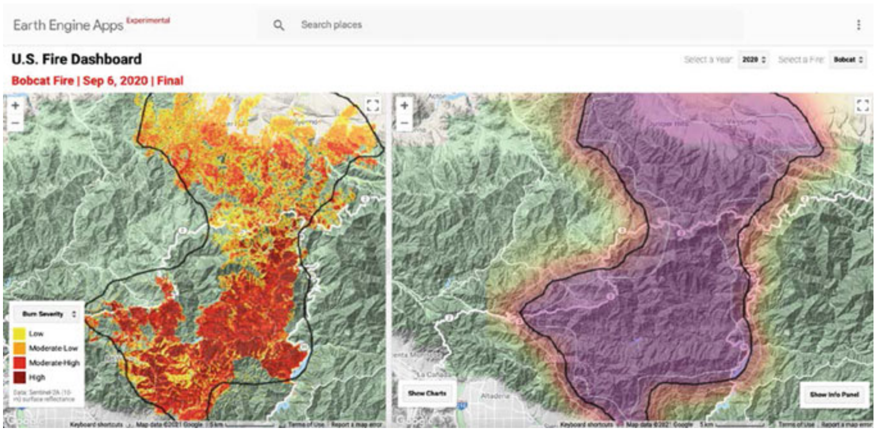


Fig. 46.4 Bobcat Fire in the Earth Engine app U.S. Fire Dashboard (https://globalfires.earthengine.app/view/us-fire-dashboard). The left-hand panel shows the burn severity derived from Sentinel-2A surface reflectance. The right-hand panel shows the confidence of the burned area classification derived from GOES active fires. The black line in both panels shows the burn perimeter derived from pixels having > 90% confidence

Question 6. Navigate to the Bobcat Fire using the dropdown panel at the top right corner of the app and wait a few minutes for the results to load. Examine the fire progression chart drawn at the top of the left panel. How many hours did it take for the fire to reach its maximum burned area? Now examine the animated fire progression GIF at the bottom of the left panel. What direction did the Bobcat Fire burn?

Question 7. Now drag the split panel from the left to reveal a second panel. Use the drop-down menu on the legend to navigate to the burn severity option. Let the results load and you will see a burn severity map calculated from Sentinel-2 imagery in the left-hand panel. For this question, consider all burn severities as burned area. Where do the Sentinel-2 map and the GOES map for the Bobcat Fire agree on the burned area (Fig. 46.4)? Where do they disagree?

Question 8. Having explored the different datasets and the two apps, what features would you include in your own wildfire mapping app? Explore additional data sources in the Synthesis section to get more inspiration for your app.

46.3 Synthesis

Assignment 1. You are now familiar with three fire datasets available in Earth Engine. Table 46.2 presents some additional fire datasets that are also available in the data catalog. Use the example codes written in each dataset's description in the data catalog to load and explore the datasets in the Code Editor.

Table 46.2 Additional external fire mapping datasets that can be uploaded into Google Earth Engine

Dataset	Variable	Resolution	Geographic coverage	Temporal extent
FireCCI51	Burned area	250 m raster, daily	Global	2001–2019
GlobFire fire event	Fire boundaries	Polygon, daily and final	Global	2001–2021
MODIS FIRMS near-real-time hotspot	Active fires	1 km, daily	Global	2000–present

For each dataset, we describe their variable type, resolution, geographic coverage, and temporal extent features

Assignment 2. While Earth Engine provides access to many existing fire datasets, there are other commonly used fire mapping datasets that are also quite useful for examining active and past fires. Select and import one of the external fire datasets shown in Table 46.3 into Earth Engine as a personal asset to compare it with the active fire datasets already loaded in your script.

Assignment 3. In addition to comparing active fire maps using the datasets suggested in Tables 46.2 and 46.3, you can examine fire conditions and impacts using ancillary datasets readily available in Earth Engine. For example, you can overlay the fire datasets with vegetation conditions and fire regimes from the LANDFIRE program to better understand the ecological context of active fires.

Select one ancillary dataset to load from the data catalog and explore it alongside an active fire dataset. Land cover datasets such as MODIS, the USGS National Land Cover Database, and Copernicus Global Land Cover can help indicate types of fires and where they are occurring. By examining active fire maps with aerosol and other emission data from Sentinel-5P and MODIS Multi-Angle Implementation of Atmospheric Correction, you can begin to identify relationships between

Table 46.3 Additional external fire-related datasets that can be uploaded into Google Earth Engine

Dataset	Variable	Resolution	Geographic coverage	Temporal extent
Monitoring trends in burn severity	Burned severity and perimeters	30 m, final	United States	1984–2019
Canadian national fire database	Fire locations, perimeters, and burned area	30 m, final	Canada	1980–2020
Landsat burned area	Burned area	30 m, 8-day	United States	1984–present

For each dataset, we describe their variable type, resolution, geographic coverage, and temporal extent features fires and air quality. These are just some of the many analyzes you can explore using ancillary datasets that are already on hand in the data catalog.

46.4 Conclusion

Earth Engine provides access to multiple fire monitoring datasets that are useful for tracking active fires throughout fire seasons and retrospectively for prior years. In this chapter, you examined one fire using three datasets (MODIS active fire, MODIS burned areas, and GOES active fire). You learned how to access, visualize, and analyze the raster datasets by adjusting code in the Code Editor and interacting with the data in premade user apps. By comparing the fire mapping characteristics of each dataset, you learned how to weigh the pros and cons of existing datasets for meeting fire mapping objectives. Now that you understand the basics of what we look for in fire mapping, you can compare additional fire datasets available in Earth Engine, or explore how to make your dataset using satellite and other geospatial data.

References

Andela N, Morton DC, Giglio L et al (2017) A human-driven decline in global burned area. Science 356:1356–1362. https://doi.org/10.1126/science.aal4108

Andela N, Morton DC, Giglio L et al (2019) The Global Fire Atlas of individual fire size, duration, speed and direction. Earth Syst Sci Data 11:529–552. https://doi.org/10.5194/essd-11-529-2019

Archibald S, Roy DP, van Wilgen BW, Scholes RJ (2009) What limits fire? An examination of drivers of burnt area in Southern Africa. Glob Chang Biol 15:613–630. https://doi.org/10.1111/j.1365-2486.2008.01754.x

Chuvieco E, Yue C, Heil A et al (2016) A new global burned area product for climate assessment of fire impacts. Glob Ecol Biogeogr 25:619–629. https://doi.org/10.1111/geb.12440

Crowley MA, Cardille JA, White JC, Wulder MA (2019a) Generating intra-year metrics of wildfire progression using multiple open-access satellite data streams. Remote Sens Environ 232:111295. https://doi.org/10.1016/j.rse.2019.111295

Crowley MA, Cardille JA, White JC, Wulder MA (2019b) Multi-sensor, multi-scale, Bayesian data synthesis for mapping within-year wildfire progression. Remote Sens Lett 10:302–311. https://doi.org/10.1080/2150704X.2018.1536300

Giglio L (2010) MODIS collection 5 active fire product user's guide version 2.4. Science Systems and Applications, Inc. University of Maryland

Giglio L, Schroeder W, Justice CO (2016) The collection 6 MODIS active fire detection algorithm and fire products. Remote Sens Environ 178:31–41. https://doi.org/10.1016/j.rse.2016.02.054

Hall JV, Zhang R, Schroeder W et al (2019) Validation of GOES-16 ABI and MSG SEVIRI active fire products. Int J Appl Earth Obs Geoinf 83:101928. https://doi.org/10.1016/j.jag.2019.101928

Hawbaker TJ, Vanderhoof MK, Beal YJ et al (2017) Mapping burned areas using dense time-series of Landsat data. Remote Sens Environ 198:504–522. https://doi.org/10.1016/j.rse.2017.06.027

Hermosilla T, Wulder MA, White JC et al (2018) Disturbance-informed annual land cover classification maps of Canada's forested ecosystems for a 29-year Landsat time series. Can J Remote Sens 44:67–87. https://doi.org/10.1080/07038992.2018.1437719

Humber ML, Boschetti L, Giglio L, Justice CO (2019) Spatial and temporal intercomparison of four global burned area products. Int J Digit Earth 12:460–484. https://doi.org/10.1080/17538947.2018.1433727

Jolly WM, Cochrane MA, Freeborn PH et al (2015) Climate-induced variations in global wildfire danger from 1979 to 2013. Nat Commun 6:1–11. https://doi.org/10.1038/ncomms8537

Liu T, Crowley MA (2021) Detection and impacts of tiling artifacts in MODIS burned area classification. IOP SciNotes 2:014003. https://doi.org/10.1088/2633-1357/abd8e2

Nogueira JMP, Ruffault J, Chuvieco E, Mouillot F (2017) Can we go beyond burned area in the assessment of global remote sensing products with fire patch metrics? Remote Sens 9:7. https://doi.org/10.3390/rs9010007

Parks SA (2014) Mapping day-of-burning with coarse-resolution satellite fire-detection data. Int J Wildl Fire 23:215–223. https://doi.org/10.1071/WF13138

Parks SA, Holsinger LM, Koontz MJ et al (2019) Giving ecological meaning to satellite-derived fire severity metrics across North American forests. Remote Sens 11:1735. https://doi.org/10.3390/rs11141735

Restif BC, Hoffman A (2020) How to generate wildfire boundary maps with Earth Engine. In: Google Earth and Earth Engine. https://medium.com/google-earth/how-to-generate-wildfire-boundary-maps-with-earth-engine-b38eadc97a38. Accessed 1 Oct 2020

Schroeder W, Prins E, Giglio L et al (2008) Validation of GOES and MODIS active fire detection products using ASTER and ETM+ data. Remote Sens Environ 112:2711–2726. https://doi.org/10.1016/j.rse.2008.01.005

Stinson G, Kurz WA, Smyth CE et al (2011) An inventory-based analysis of Canada's managed forest carbon dynamics, 1990 to 2008. Glob Chang Biol 17:2227–2244. https://doi.org/10.1111/j.1365-2486.2010.02369.x

Veraverbeke S, Sedano F, Hook SJ et al (2014) Mapping the daily progression of large wildland fires using MODIS active fire data. Int J Wildl Fire 23:655–667. https://doi.org/10.1071/WF13015

Mangroves

47

Aurélie Shapiro

Overview
Mangrove ecosystems are tropical coastal forests that are adapted to saltwater environments. Their unique qualities of existing primarily in moist environments at low elevation along shorelines, lack of seasonality, and compact pattern make them relatively easy to identify in satellite images. In this chapter, we present a series of automated steps, including water masking, to extract mangroves from a fusion of optical and active radar data. Furthermore, as global mangrove datasets are readily available in Google Earth Engine, we present an approach to automatically extract training data from existing information, saving time and effort in your supervised classification. The method can be adapted to create subsequent maps from your own results to produce changes in mangrove ecosystems over time.

Learning Outcomes

- Fusing Sentinel-1 and -2 optical/radar sensors.
- Sampling points on an image to create training and testing datasets.
- Calculating additional indices to add to the image stack.
- Applying an automatic water masking function and buffering to focus classification on coastal areas likely to have mangroves.
- Understanding supervised classification with random forests using automatically derived training data.
- Evaluating training and model accuracy.

A. Shapiro (✉)
Here+There Mapping Solutions, Berlin, Germany
e-mail: aurelie@here-there-mapping.com

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_47

Helps if you know how to

- Import images and image collections, filter, and visualize (Part I).
- Perform basic image analysis: select bands, compute indices, create masks (Part II).
- Perform pixel-based supervised or unsupervised classification (Chap. 6).
- Use expressions to do calculations on image bands (Chap. 9).
- Perform image morphological operations (Chap. 10).
- Create or access image mosaics (Chap. 15).
- Interpret Otsu's method for partitioning a histogram (Chap. 42).

47.1 Introduction to Theory

Mangrove forests consist of specialized species adapted to saltwater environments located in tropical and subtropical latitudes. They are highly productive environments that provide essential services, notably the storage of blue carbon, stabilization and protection of coastlines from storms and coastal events, and the provision of nurseries for fish (Alongi, 2002). Mangroves also enhance associated coral reef ecosystems, which are crucial to supporting local livelihoods (Bryan-Brown et al. 2020). Over 1.3 billion people live in tropical coastal areas and rely on mangrove and associated ecosystems for their health, safety, and livelihood. It is therefore important to map, monitor, and quantify their change over time in order to properly conserve and restore them.

In this chapter, we will review the basic process for mapping mangroves using Sentinel-1 and -2 imagery. Mangroves are particularly recognizable in satellite imagery by their wetness—these ecosystems thrive at the water/land interface, making them easy to distinguish with satellite sensors that are sensitive to vegetation and water. We use sensor fusion to combine the advantages of radar and optical satellite imagery. For optical data, we extract relevant vegetation and water indices to discern mangroves, and we use active radar data for its capacity to mask water and derive canopy texture information.

In this chapter, we will show you how to evaluate mangroves at 10 m resolution using a fusion of optical and radar sensors (Sentinel-1 and -2) and derived indices. We will also implement automatic water masking (see also Chap. 42) and a random forest machine-learning supervised (see Chaps. 40 and 41) classification, to which you can potentially add your own improvements as needed.

If you are interested in learning more about mangrove mapping with Earth Engine, a thorough workflow for analyzing Landsat imagery for mangroves with a light graphic user interface is presented as the Google Earth Engine Mangrove Mapping Methodology (GEMMM) in Yancho et al. (2020). The advantages of this approach are the evaluation of the shoreline buffer areas, assessment of high- and low-tide imagery, a user-friendly interface, and the freely available and well-explained code.

47.2 Practicum

Several assets are provided for you to work with. As a first step, define the area of interest (aoi) and view it. In this case, we choose the Sundarbans ecosystem on the border of India and Bangladesh, which is an iconic mangrove forest (also known for its mysterious native tiger population) and a simple example to showcase water masking and mangrove mapping:

```
// Create an ee.Geometry.
var aoi = ee.Geometry.Polygon([
    [
        [88.3, 22.61],
        [90, 22.61],
        [90, 21.47],
        [88.3, 21.47]
    ]
]);

// Locate a coordinate in the aoi with land and water.
var point = ee.Geometry.Point([89.2595, 21.7317]);

// Position the map.
Map.centerObject(point, 13);
Map.addLayer(aoi, {}, 'AOI');

// Sentinel-1 wet season data.
var wetS1 = ee.Image(
    'projects/gee-book/assets/A3-
2/wet_season_tscan_2020');
// Sentinel-1 dry season data.
var dryS1 = ee.Image(
    'projects/gee-book/assets/A3-
2/dry_season_tscan_2020');
// Sentinel-2 mosaic.
var S2 = ee.Image('projects/gee-book/assets/A3-
2/Sundarbans_S2_2020');
```

We will fuse radar and optical data at 10 m resolution for this exercise using multi-temporal composites that were developed using the Food and Agriculture Organization (FAO) freely available SEPAL (https://sepa.io), which is a cloud-based image and data processing platform that has several modules built on Earth Engine. The available recipes let you choose dates and processing parameters and export composites directly to your Earth Engine account. For radar, we used

SEPAL to produce two composite “timescan” images - one for the wet season and one for dry season) created from multi-temporal statistics derived from multiple Sentinel-1 images, a compilation of all filtered, terrain-corrected images that are available in Earth Engine for a certain time period (Esch et al. 2018, Mulissa et al. 2021, Vollrath et al. 2020). Additionally, the SEPAL platform calculates statistics (standard deviation, minimum, maximum) of different bands. We developed two timescan images for the 2020 dry (March to September) and wet (October to April) seasons of the study area.

The Sentinel-2 optical composite was also generated in SEPAL, applying the bidirectional reflectance distribution function (BRDF) correction to surface reflectance corrected images and producing the median value of all cloud-free pixels for 2020.

You will now access both exported SEPAL composites using the Code Editor.

```
//Visualize the input data.
var s1VisParams = {
    bands: ['VV_min', 'VH_min', 'VVVH_ratio_min'],
    min: -36,
    max: 3
};
var s2VisParams = {
    bands: ['swir1', 'nir', 'red'],
    min: 82,
    max: 3236
};

Map.addLayer(dryS1, s1VisParams, 'S1 dry', false);
Map.addLayer(wetS1, s1VisParams, 'S1 wet', false);
Map.addLayer(S2, s2VisParams, 'S2 2020');
```

47.2.1 Section 1: Deriving Additional Indices

It is a good idea to complement your data stack with additional indices (see Chap. 5) that are relevant to mangroves—such as indices sensitive to water, greenness, and vegetation (see Wang et al. 2018). You can add these via band calculations and equations, and we will present several here. But the list of indices is virtually endless; what is useful can depend on the location.

To compute the normalized vegetation index (NDVI) using an existing Earth Engine NDVI function, add this line:

```
var NDVI = S2.normalizedDifference(['nir',
'red']).rename(['NDVI']);
```

You can also use an image expression (see Chap. 9) for any calculation, such as a band ratio:

```
var ratio_swir1_nir = S2.expression(
        'swir1/(nir+0.1)', {
            'swir1': S2.select('swir1'),
            'nir': S2.select('nir')
        })
    .rename('ratio_swir1_nir_wet');
```

You add the `rename` function so you can recognize the band more easily in your data stack. You create a data stack by adding the different indices to the input image by using `addBands` and the name of the index or expression. Don't forget to add your Sentinel-1 data too:

```
var data_stack =
S2.addBands(NDVI).addBands(ratio_swir1_nir).addBands(
    dryS1).addBands(wetS1).addBands(S2);
```

And finally, you can see the names of all your bands by entering:

```
print(data_stack);
```

Question 1. What other indices could be useful for mapping mangroves?

There are a number of useful articles on mangrove mapping that provide assess the value of additional indices; if you know how they are calculated, you can add them to your data stack with image expressions.

Code Checkpoint A32a. The book's repository contains a script that shows what your code should look like at this point.

47.2.2 Section 2: Automatic Water Masking and Buffering

As explained above, mangroves are found close to coastlines, which tend to be at low elevations. The next steps involve automatic water masking to delineate land from sea; then we will use an existing dataset to buffer the area of interest so that we are only mapping mangroves where we would expect to find them.

We will use the Canny edge detector and Otsu thresholding (Donchyts et al. 2016) approach to automatically detect water. For this, we use the point provided at the beginning of the script that is located near land and water. The function will then automatically identify an appropriate threshold that delineates land pixels from water, based on the calculation of edges in a selected region with both land and water. This approach is also demonstrated in Chap. 42 using different parameters and settings, where it is described in detail.

Paste the code below to add functionality that can compute the threshold, detect edges, and create the water mask:

```
/***
 * This script computes surface water mask using
 * Canny Edge detector and Otsu thresholding.
 * See the following paper for details:
 * http://www.mdpi.com/2072-4292/8/5/386
 *
 * Author: Gennadii Donchyts
 * Contributors: Nicholas Clinton
 *
 */

/***
 * Return the DN that maximizes interclass variance in B5
(in the region).
 */
var otsu = function(histogram) {
    histogram = ee.Dictionary(histogram);
    var counts = ee.Array(histogram.get('histogram'));
    var means = ee.Array(histogram.get('bucketMeans'));
    var size = means.length().get([0]);
    var total = counts.reduce(ee.Reducer.sum(),
[0]).get([0]);
    var sum =

means.multiply(counts).reduce(ee.Reducer.sum(), [0])
        .get([0]);
    var mean = sum.divide(total);

    var indices = ee.List.sequence(1, size);

    // Compute between sum of squares, where each mean
partitions the data.
    var bss = indices.map(function(i) {
        var aCounts = counts.slice(0, 0, i);
        var aCount = aCounts.reduce(ee.Reducer.sum(), [0])
            .get([0]);
        var aMeans = means.slice(0, 0, i);
        var aMean = aMeans.multiply(aCounts)
            .reduce(ee.Reducer.sum(), [0]).get([0])
            .divide(aCount);
        var bCount = total.subtract(aCount);
        var bMean = sum.subtract(aCount.multiply(aMean))
            .divide(bCount);
        return aCount.multiply(aMean.subtract(mean).pow(
            2)).add(
            bCount.multiply(bMean.subtract(mean).pow(
                2)));
    });

    // Return the mean value corresponding to the maximum
BSS.
    return means.sort(bss).get([-1]);
};

/***
 * Compute a threshold using Otsu method (bimodal).
 */
```

```
function computeThresholdUsingOtsu(image, scale, bounds,
    cannyThreshold,
    cannySigma, minValue, debug) {

    // Clip image edges.
    var mask = image.mask().gt(0)
        .focal_min(ee.Number(scale).multiply(3), 'circle',
'meters');

    // Detect sharp changes.
    var edge = ee.Algorithms.CannyEdgeDetector(image,
cannyThreshold,
        cannySigma);
    edge = edge.multiply(mask);

    // Buffer around NDWI edges.
    var edgeBuffer = edge
        .focal_max(ee.Number(scale).multiply(1), 'square',
'meters');
    var imageEdge = image.mask(edgeBuffer);

    // Compute threshold using Otsu thresholding.
    var buckets = 100;
    var hist = ee.Dictionary(ee.Dictionary(imageEdge
            .reduceRegion({
                reducer: ee.Reducer.histogram(buckets),
                geometry: bounds,
                scale: scale,
                maxPixels: 1e9
            }))
        .values()
        .get(0));

    var threshold = ee.Number(ee.Algorithms.If({
        condition: hist.contains('bucketMeans'),
        trueCase: otsu(hist),
        falseCase: 0.3
    }));

    if (debug) {
        Map.addLayer(edge.mask(edge), {

            palette: ['ff0000']
        }, 'edges', false);
        print('Threshold: ', threshold);
        print(ui.Chart.image.histogram(image, bounds,
scale,
            buckets));
        print(ui.Chart.image.histogram(imageEdge, bounds,
scale,
            buckets));
    }

    return minValue !== 'undefined' ?
threshold.max(minValue) :
        threshold;
}

var bounds = ee.Geometry(Map.getBounds(true));

var image = data_stack;
print('image', image);

var ndwi_for_water = image.normalizedDifference(['green',
'nir']);
var debug = true;
var scale = 10;
var cannyThreshold = 0.9;
var cannySigma = 1;
var minValue = -0.1;
var th = computeThresholdUsingOtsu(ndwi_for_water, scale,
bounds,
    cannyThreshold, cannySigma, minValue, debug);

print('th', th);

function getEdge(mask) {
    return mask.subtract(mask.focal_min(1));
}


var water_mask =
ndwi_for_water.mask(ndwi_for_water.gt(th));
```

```
th.evaluate(function(th) {
    Map.addLayer(water_mask, {
        palette: '0000ff'
    }, 'water mask (th=' + th + ')');
});
```

You'll notice that new layers are loaded in the map, which include the edge detection and a water mask that identifies all marine and surface water (Fig. 47.1).

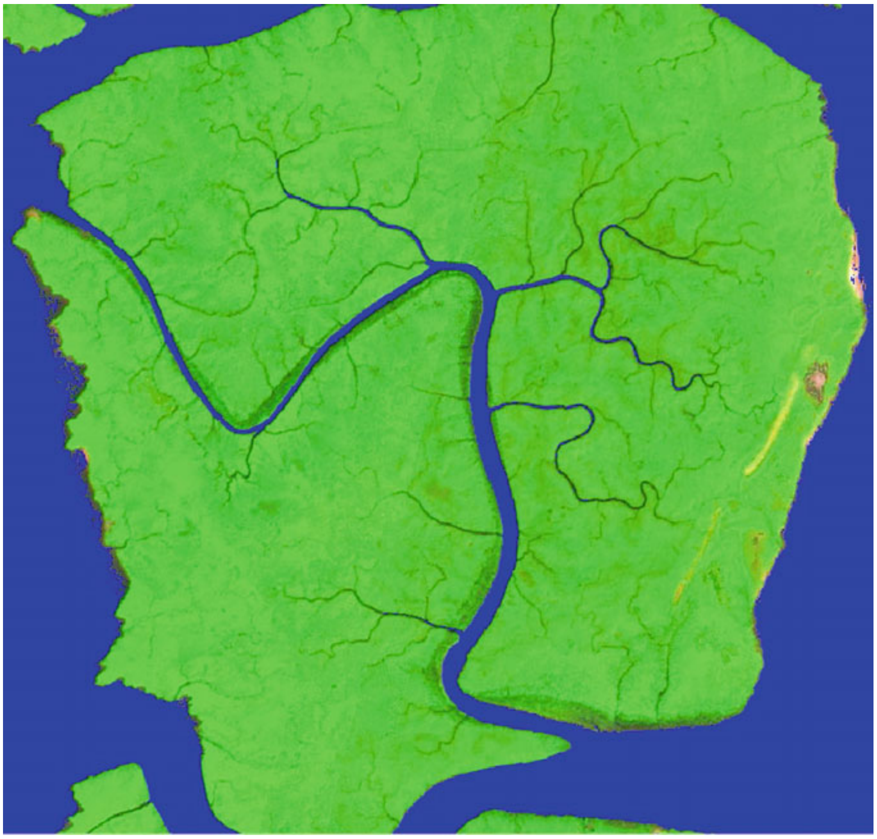


Fig. 47.1 Automatic water mask identifies all open and surface water pixels

Question 2. Is the point well located to appropriately identify the water mask? What happens with turbid water?

Move the point around and see if it improves the automatic water masking. Turbid water can be an issue and may not be detected by the mask. Be certain that these muddy waters are not classified as mangrove later on.

Next, we create the land mask by inverting the water mask, and removing any areas with elevation greater than 40 m above sea level using the NASADEM (Digital Elevation Model from NASA) data collection. This will ensure we aren't erroneously mapping mangroves far inland, where they don't occur. You can of course change the elevation threshold according to your study area.

```
// Create land mask area.
var land = water_mask.unmask();
var land_mask = land.eq(0);
Map.addLayer(land_mask, {}, 'Land mask', false);

// Remove areas with elevation greater than mangrove
elevation threshold.
var elev_thresh = 40;
var dem =
ee.Image('NASA/NASADEM_HGT/001').select('elevation');
var elev_mask = dem.lte(elev_thresh);
var land_mask = land_mask.updateMask(elev_mask);
```

Next, we will buffer the area of interest to restrict the analysis only to areas where mangroves might realistically be found. For this, we will use the Global Mangrove Dataset from 2000 available in Earth Engine (note: this is one of several mangrove datasets; you could also have used other datasets such as Global Mangrove Watch or any other available raster data.)

The Global Mangrove Dataset was derived from Landsat 2000 (Giri et al. 2011); we will buffer 1000 m around it. The 1000 m buffer allows for the possibility that some mangroves were missed in the original map, and that mangroves might have expanded in some areas since 2000. You can change the buffer distance to any value suitable for your study area.

```
// Load global mangrove dataset as reference for training.
var mangrove_ref =
ee.ImageCollection('LANDSAT/MANGROVE_FORESTS')
    .filterBounds(aoi)
    .first()
    .clip(aoi);
Map.addLayer(mangrove_ref, {
    palette: 'Green'
}, 'Reference Mangroves', false);

// Buffer around known mangrove area with a specified
distance.
var buffer_dist = 1000;
var mang_buffer = mangrove_ref
    .focal_max(buffer_dist, 'square', 'meters')
    .rename('mangrove_buffer');
Map.addLayer(mang_buffer, {}, 'Mangrove Buffer', false);
```

Question 3. Can the buffer distance or elevation threshold be changed to capture more mangroves or remove extra areas where mangroves aren't likely to be found—and that we don't need to classify? We don't want to miss any mangrove areas, but we also want an efficient code that does not process large areas that can't be mangroves, which can add processing complexity and commit easily avoidable errors. To restrict the processing to the potential mangrove area, can you change the buffer distance or elevation threshold to best capture the area you are interested in?

We will now mask the mangrove buffer, create the area to classify, and mask it from the data stack.

```
// Mask land from mangrove buffer.
var area_to_classify =
mang_buffer.updateMask(land_mask).selfMask();
Map.addLayer(area_to_classify,
    {},
    'Mangrove buffer with water and elevation mask',
    false);
var image_to_classify =
data_stack.updateMask(area_to_classify);
Map.addLayer(image_to_classify,
    {
        bands: ['swir1', 'nir', 'red'],
        min: 82,
        max: 3236
    },
    'Masked Data Stack',
    false);
```

Code Checkpoint A32b. The book's repository contains a script that shows what your code should look like at this point.

47.2.3 Section 3: Creating Training Data and Running and Evaluating a Random Forest Classification

We will now automatically select mangrove and non-mangrove locations as training data for our model. We use morphological image processing (see Chap. 10) to select areas deep inside the reference mangrove dataset as areas we can be sure are mangroves, because mangrove forests tend to be lost or deforested at edges rather than in the interior. Using the same theory, we select areas far away from mangrove edges as our non-forest areas. This approach allows us to use a relatively older dataset from 2000 for current data training. We will extract mangrove and non-mangrove from the reference data. First we create the feature layers to store our training data.

```
// Create training data from existing data
// Class values: mangrove = 1, not mangrove = 0
var ref_mangrove = mangrove_ref.unmask();
var mangroveVis = {
    min: 0,
    max: 1,
    palette: ['grey', 'green']
};
Map.addLayer(ref_mangrove, mangroveVis, 'mangrove = 1');

// Class values: not mangrove = 1 and mangrove = 0
var notmang = ref_mangrove.eq(0);
var notMangroveVis = {
    min: 0,
    max: 1,
    palette: ['grey', 'red']
};
Map.addLayer(notmang, notMangroveVis, 'not mangrove = 1',
false);
```

We then use erosion and dilation (as described in Chap. 10) to identify areas at the center of mangrove forests and far from outside edges. We put everything together in a training layer where mangroves = 1, non-mangroves = 2, and everything else = 0.

```
// Define a kernel for core mangrove areas.
var kernel = ee.Kernel.circle({
    radius: 3
});

// Perform a dilation to identify core mangroves.
var mang_dilate = ref_mangrove
    .focal_min({
        kernel: kernel,
        iterations: 3
    });
var mang_dilate = mang_dilate.updateMask(mang_dilate);
var mang_dilate =
mang_dilate.rename('auto_train').unmask();
Map.addLayer(mang_dilate, {}, 'Core mangrove areas to
sample', false);
```

```
// Do the same for non-mangrove areas.
var kernel1 = ee.Kernel.circle({
    radius: 3
});
var notmang_dilate = notmang
    .focal_min({
        kernel: kernel1,
        iterations: 2
    });
var notmang_dilate =
notmang_dilate.updateMask(notmang_dilate);
var notmang_dilate =
notmang_dilate.multiply(2).unmask().rename(
    'auto_train');
Map.addLayer(notmang_dilate, {}, 'Not mangrove areas to
sample',
    false);

// Core mangrove = 1, core non mangrove = 2, neither = 0.
var train_labels =
notmang_dilate.add(mang_dilate).clip(aoi);
var train_labels =
train_labels.int8().updateMask(area_to_classify);
var trainingVis = {
    min: 0,
    max: 2,
```

Question 4. How do the kernel radius and iteration parameters identify or miss important core mangrove areas?

To obtain good training data, we want samples located throughout the area of interest. Sometimes, if the mangroves occur in very small patches, if the radius is too large, or if there are too many iterations, we don't end up with enough core forest to sample.

Change the parameters to see what works best for you. You may need to zoom in to see the core mangrove areas.

The next step is the supervised classification (see Chap. 6). We will collect random samples from the training areas to train and run the random forest classifier. We will conduct two classifications. One is for creating the map and another for validation, to obtain the test accuracy and determine how consistent the training areas are between two random samples.

```
// Begin Classification.
// Get image and bands for training - including automatic
training band.
var trainingImage =
image_to_classify.addBands(train_labels);
var trainingBands = trainingImage.bandNames();
print(trainingBands, 'training bands');

// Get training samples and classify.
// Select the number of training samples per class.
var numPoints = 2000;
var numPoints2 = 2000;

var training = trainingImage.stratifiedSample({
    numPoints: 0,
    classBand: 'auto_train',
    region: aoi,
    scale: 100,
    classValues: [1, 2],
    classPoints: [numPoints, numPoints2],
    seed: 0,
    dropNulls: true,
    tileScale: 16,
});

var validation = trainingImage.stratifiedSample({
    numPoints: 0,
    classBand: 'auto_train',
    region: aoi,
    scale: 100,
    classValues: [1, 2],
    classPoints: [numPoints, numPoints2],
    seed: 1,
    dropNulls: true,
    tileScale: 16,
});

// Create a random forest classifier and train it.
var nTrees = 50;
var classifier = ee.Classifier.smileRandomForest(nTrees)
    .train(training, 'auto_train');

 var classified = image_to_classify.classify(classifier);
```

```
// Classify the test set.
var validated = validation.classify(classifier);

// Get a confusion matrix representing resubstitution
accuracy.
var trainAccuracy = classifier.confusionMatrix();
print('Resubstitution error matrix: ', trainAccuracy);
print('Training overall accuracy: ',
trainAccuracy.accuracy());
var testAccuracy = validated.errorMatrix('mangrove',
    'classification');
```

The training accuracy is over 99%, which is very good. According to the substitution matrix, only a few training points seem to be confused when they are randomly replaced.

In addition, we can estimate variable importance, or how much each band in the data stack contributes to the final random forest model. These are always good metrics to observe, as you can remove the least important bands from the training image if you need to improve the classification.

```
var dict = classifier.explain();
print('Explain:', dict);
var variable_importance = ee.Feature(null,
ee.Dictionary(dict).get(
    'importance'));

// Chart variable importance.
var chart =
ui.Chart.feature.byProperty(variable_importance)
    .setChartType('ColumnChart')
    .setOptions({
        title: 'Random Forest Variable Importance',
        legend: {
            position: 'none'
        },
        hAxis: {
            title: 'Bands'
        },
        vAxis: {
            title: 'Importance'
        }
    });
print(chart);
```

Question 5. What are the most important bands in the classification model?

Question 6. Based on the chart, is one of the sensors more important in the classification model? Which bands? Why do you think that is?

Next, we will visualize the final classification. We can apply a filter to remove individual pixels of one class, which effectively applies a minimum mapping unit (MMU). In this case, any areas with fewer than 25 connected pixels are filtered out.

```
var classificationVis = {
    min: 1,
    max: 2,
    palette: ['green', 'grey']
};
Map.addLayer(classified, classificationVis,
    'Mangrove Classification');

// Clean up results to remove small patches/pixels.
var mang_only = classified.eq(1);
// Compute the number of pixels in each connected mangrove
patch
// and apply the minimum mapping unit (number of pixels).
var mang_patchsize = mang_only.connectedPixelCount();

//mask pixels based on the number of connected neighbors
var mmu = 25;
var mang_mmu = mang_patchsize.gte(mmu);
var mang_mmu = classified.updateMask(mang_mmu).toInt8();
Map.addLayer(mang_mmu, classificationVis, 'Mangrove Map
MMU');
```

Your map window should resemble Fig. 47.2 with mangroves in green, and non-mangroves in gray.

Code Checkpoint A32c. The book's repository contains a script that shows what your code should look like at this point.

In the **Console** window, you'll see the substitution matrix, training accuracy, and a chart of variable importance.

Question 7. Do you notice any errors in the map? Omissions or commissions? How does the map compare to 2000?

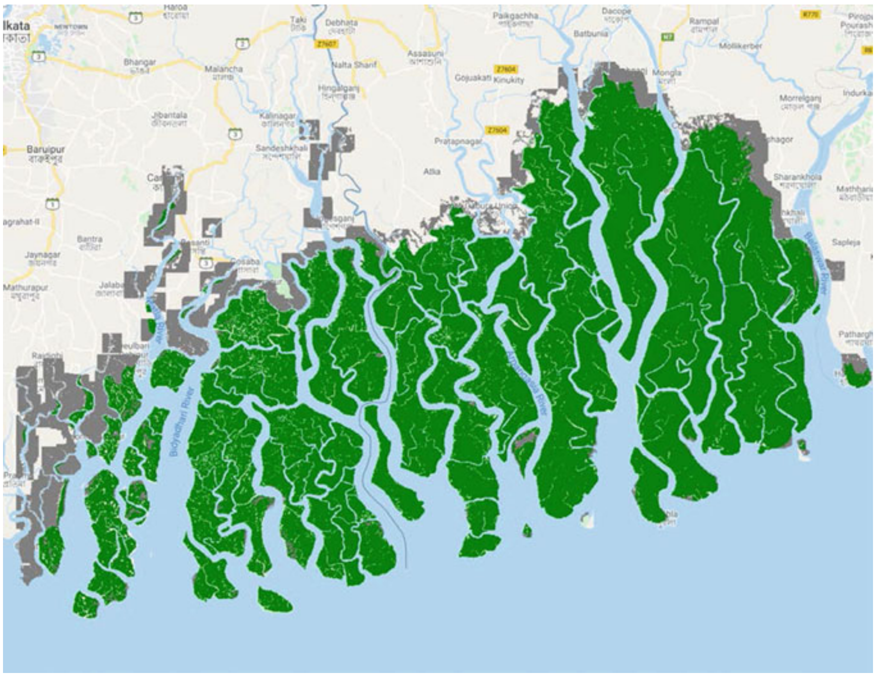


Fig. 47.2 Final mangrove classification

Question 8. You should be able to see small differences in the mangrove extent since 2000. What do you see? What could be the causes for the changes?

Question 9. How does the MMU parameter change the output map?

Try out different values and see what the results look like.

47.3 Synthesis

Assignment 1. With what you learned in this chapter, you can fuse Sentinel-1 and -2 data to create your own map of mangroves for anywhere in the world. You might now test out the approach in another part of the world, or use your own training data for a more refined classification model. You can add your own point data in the map and merge with the training data to improve a classification, or clean areas that need to be removed by drawing polygons and masking them in the classification.

Assignment 2. You can also add more indices, remove ones that aren't as informative, create a map using earlier imagery, use the later date classification as reference data, and look for differences via post-classification change. Additional indices can be found in script **A32s1—Supplemental** in the book's repository. Using these other indices, does that change your estimation of what areas are stable mangrove, and where have we observed gains or losses?

47.4 Conclusion

Mangroves are dynamic ecosystems that can expand over time, and can also be lost through natural and anthropogenic causes. The power of Earth Engine lies in the cloud-based, lightning-fast, automated approach to workflows, particularly with automated training data collection. This process would take days when performed offline in traditional remote sensing software, especially over large areas. The Earth Engine approach is not only fast but also consistent. The same method can be applied to images from different dates to assess mangrove changes over time—both gain and loss.

References

Donchyts G, Schellekens J, Winsemius H et al (2016) A 30 m resolution surface water mask including estimation of positional and thematic differences using Landsat 8, SRTM and OpenStreetMap: a case study in the Murray-Darling basin. Australia. Remote Sens 8:386. https://doi.org/10.3390/rs8050386

Esch T, Üreyen S, Zeidler J et al (2018) Exploiting big Earth data from space–first experiences with the TimeScan processing chain. Big Earth Data 2:36–55. https://doi.org/10.1080/20964471.2018.1433790

Giri C, Ochieng E, Tieszen LL et al (2011) Status and distribution of mangrove forests of the world using Earth observation satellite data. Glob Ecol Biogeogr 20:154–159. https://doi.org/10.1111/j.1466-8238.2010.00584.x

Mullissa A, Vollrath A, Odongo-Braun C et al (2021) Sentinel-1 SAR backscatter analysis ready data preparation in Google Earth Engine. Remote Sens 13:1954. https://doi.org/10.3390/rs13101954

Vollrath A, Mullissa A, Reiche J (2020) Angular-based radiometric slope correction for Sentinel-1 on Google Earth Engine. Remote Sens 12:1867. https://doi.org/10.3390/rs12111867

Wang D, Wan B, Qiu P et al (2018) Evaluating the performance of Sentinel-2, Landsat 8 and Pléiades-1 in mapping mangrove extent and species. Remote Sens 10:1468. https://doi.org/10.3390/rs10091468

Yancho JMM, Jones TG, Gandhi SR et al (2020) The Google Earth Engine mangrove mapping methodology (GEEMMM). Remote Sens 12:1–35. https://doi.org/10.3390/rs12223758

Bryan-Brown D, Connolly RM, Richards DR et al (2020) Global trends in mangrove forest fragmentation Abstr Sci Rep 10(1). https://doi.org/10.1038/s41598-020-63880-1

Alongi DM (2002) Present state and future of the world's mangrove forests. Environ Conserv 29:331–349

Mangroves II—Change Mapping

48

Celio de Sousa, David Lagomasino, and Lola Fatoyinbo

Overview
The purpose of this chapter is to present two methods of land cover extent change detection: map-to-map and map-to-image using anomaly data. In a map-to-map approach, changes are extracted by subtracting a two-date pair of land cover extent maps: that is, time 2 (T2) extent minus time 1 (T1) extent. By comparison, a map-to-image approach uses a baseline extent map within which changes are calculated based on a T2 image where the change classes are defined by threshold values. In this chapter, we will perform a map-to-map change detection between two mangrove extent maps from 2000 and 2020 and also perform a vegetation index anomaly analysis to detect changes within a mangrove extent map from the year 2000 in Guinea, West Africa.

Learning Outcomes

- Performing change detection by contrasting two pre-existent mangrove extent maps.
- Harmonizing across Landsat generations to create consistent long-term series.
- Calculating the anomaly of a given vegetation index based on its long-term average value.

C. de Sousa (✉)
University of Maryland Baltimore County/NASA Goddard Space Flight Center, Greenbelt, USA
e-mail: celio.h.resendedesousa@nasa.gov

D. Lagomasino
East Carolina University, Greenville, USA
e-mail: lagomasinod19@ecu.edu

L. Fatoyinbo
NASA Goddard Space Flight Center, Greenbelt, USA
e-mail: lola.fatoyinbo@nasa.gov

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_48

- Performing change detection by interpreting vegetation index anomalies.

Helps if you know how to

- Recognize similarities and differences among satellite spectral bands (Part I, Part II, Part III).
- Perform basic image analysis: select bands, compute indices, create masks (Part II).
- Use `normalizedDifference` to calculate vegetation indices (Chap. 5).
- Perform a supervised image classification (Chap. 6).
- Work with array images (Chaps. 9 and 18).
- Use expressions to perform calculations on image bands (Chap. 9).
- Summarize an image with `reduceRegion` (Chap. 9).
- Write a function and `map` it over an `ImageCollection` (Chap. 12).
- Mask cloud, cloud shadow, snow/ice, and other undesired pixels (Chap. 15).
- Perform a two-period change detection (Chap. 16).

48.1 Introduction to Theory

Mangrove forests are among the most productive ecosystems on Earth, providing a wide range of critical economic, ecological, and societal services. However, mangroves worldwide have undergone intense conversion, with human-related disturbances being one of the main causes of global mangrove loss in recent decades (Goldberg et al. 2020). Thus, regular monitoring of the dynamics of mangrove ecosystems worldwide is key for establishing better coastal management policies and mangrove conservation initiatives.

Readily available, low-cost, and repeated-coverage remote sensing data provides numerous advantages for monitoring mangrove forests and detecting changes in the landscape over time. In this context, the Landsat data archive, which covers more than 35 years, has been widely used for mapping land cover dynamics worldwide. Most notably, the Landsat mission coupled with Google Earth Engine's computing infrastructure have been leveraged to develop widely used global datasets at 30 m spatial resolution, such as the global forest cover (Hansen et al. 2013) and global mangrove extent (Bunting et al. 2018) datasets.

However, as extensively explored in the literature, change detection using remotely sensed data is challenged by several factors, including (but not limited to) the following: spatial, spectral, thematic, and temporal constraints, atmospheric conditions, high cloud-coverage, and differences in sensors' spectral characteristics. These differences may have a direct effect on the ability to accurately detect and monitor changes in the landscape, including mangrove forests. As explored by Roy et al. (2016), the Landsat TM/ETM + and OLI sensors present small but potentially significant differences between their spectral characteristics, with the greatest differences in the near-infrared (NIR) and the shortwave infrared (SWIR) bands—the most important spectral bands for vegetation studies.

In this chapter, we take advantage of the statistical functions presented in Roy et al. (2016) to transform between the comparable TM/ETM + and OLI bands in order to ensure inter-sensor harmonized spectral information and temporal continuity. We will use this harmonized Landsat TM/ETM + /OLI collection to derive anomaly-based changes in mangrove extent over 21 years (2000–2020) in Guinea, West Africa.

48.2 Practicum

48.2.1 Section 1: Map-To-Map Change Detection

Previous chapters demonstrated how to perform a general supervised classification, as well as how to apply those classifiers and functions to create a mangrove extent map (Chap. 47). In this section, we will build upon those techniques to perform a map-to-map change analysis. Paste the code below into a new script, which will access pre-created assets for two time periods: 2000 and 2020.

```
var areaOfstudy = ee.FeatureCollection(
    'projects/gee-book/assets/A3-3/Border5km');
var mangrove2000 = ee.Image(
    'projects/gee-book/assets/A3-3/MangroveGuinea2000_v2');
var mangrove2020 = ee.Image(
    'projects/gee-book/assets/A3-3/MangroveGuinea2020_v2');
```

Start by setting the map center around Conakry, Guinea, and adding your 2000 and 2020 mangrove extent to the map using a color of your choice and naming them `'Mangrove Extent 2000'` and `'Mangrove Extent 2020'`:

```
Map.setCenter(-13.6007, 9.6295, 10);
// Sets the map center to Conakry, Guinea
Map.addLayer(areaOfstudy, {}, 'Area of Study');
Map.addLayer(mangrove2000, {
    palette: '#16a596'
}, 'Mangrove Extent 2000');
Map.addLayer(mangrove2020, {
    palette: '#9ad3bc'
}, 'Mangrove Extent 2020');
```

Because the assets have the value of 1 (one) assigned to mangrove pixels and 0 (zero) to everything else, you can derive losses and gains from both mangrove extent maps with a simple subtraction of T1 (2000) from T2 (2020). The value of 1 has been assigned to mangrove pixels and everything else has been masked. For the mathematical operation between the two layers to work, every pixel has to have a value assigned to it. In this case, you can unmask previously masked pixels and assign the value 0 to them using `unmask(0)`. Finally, subtract T1 from T2 into a new variable `change`:

```
var mang2020 = mangrove2020.unmask(0);
var mang2000 = mangrove2000.unmask(0);
var change = mang2020.subtract(mang2000)
    .clip(areaOfstudy);
```

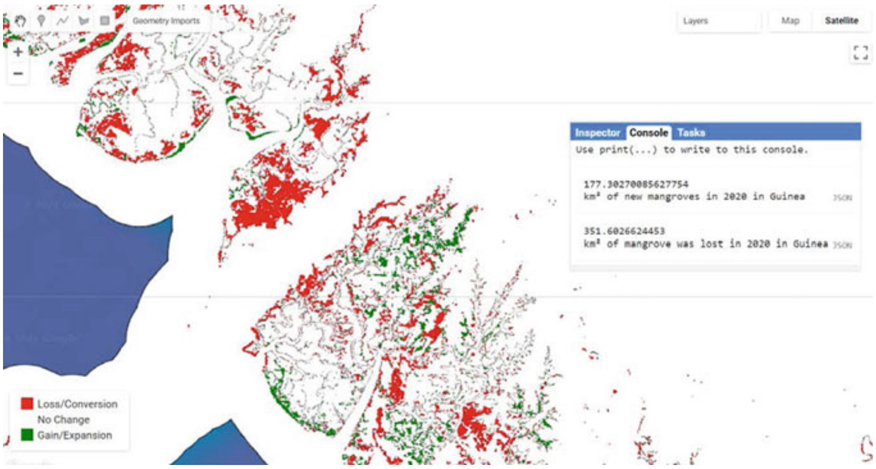
Pixels in the raster `change` will have values of -1, 0, or 1, which represent loss/conversion, no change, and gains, respectively:

- -1 = no mangroves in 2020, mangroves in 2000 ($0-1=-1$);
- 0 = mangroves in 2020, mangroves in 2000 ($1-1=0$);
- 1 = mangroves in 2020, no mangroves in 2000 ($1-0=1$)

Finally, add `change` to the map:

```
var paletteCHANGE = [
    'red', // Loss/conversion
    'white', // No Change
    'green', // Gain/Expansion
];

Map.addLayer(change, {
    min: -1,
    max: 1,
    palette: paletteCHANGE
}, 'Changes 2000-2020');
```

You can calculate the area of expansion/conversion by isolating the pixels of gain/loss from the `change` into `gain` and `loss`. Then, calculate the area of each pixel using ee.Image.pixelArea and multiplying by the count of pixels in `gain` and `loss` using `multiply`. The default unit is square meters (m^2). You can use `divide` to transform into square kilometers `(divide(1000000))` or hectares `(divide(10000))`. Finally, use `ee.Reducer.sum` to sum all area values for both `gain` and `loss` and print them to the **Console** (Fig. 48.1).



```
Use print(...) to write to this console.

177.30270085627754
km² of new mangroves in 2020 in Guinea                 JSON

351.6026624453
km² of mangrove was lost in 2020 in Guinea             JSON
```

Fig. 48.1 Map-to-map changes in mangrove extent in Guinea from 2000–2020

```
// Calculate the area of each pixel
var gain = change.eq(1);
var loss = change.eq(-1);

var gainArea =
gain.multiply(ee.Image.pixelArea().divide(1000000));
var lossArea =
loss.multiply(ee.Image.pixelArea().divide(1000000));

// Sum all the areas
var statsgain = gainArea.reduceRegion({
    reducer: ee.Reducer.sum(),
    scale: 30,
    maxPixels: 1e14
});

var statsloss = lossArea.reduceRegion({
    reducer: ee.Reducer.sum(),
    scale: 30,
    maxPixels: 1e14
});

print(statsgain.get('classification'),
    'km² of new mangroves in 2020');
print(statsloss.get('classification'),
    'of mangrove was lost in 2020');
```

```
Map.addLayer(gain.selfMask(), {
    palette: 'green'
}, 'Gains');
Map.addLayer(loss.selfMask(), {
    palette: 'red'
}, 'Loss');
```

Code Checkpoint A33a. The book's repository contains a script that shows what your code should look like at this point.

Question 1. What are some of the issues that may arise when using a map-to-map change detection? Explain how these different dates may affect the classification output and, consequently, the change output.

48.2.2 Section 2: Map-To-Image Change Detection

Using the mangrove extent you created in the previous section, we will look at another way to detect changes without having to classify an image. In this case, change classes are defined by threshold values of a specific metric, such as a vegetation index or a spectral image band. In this section, we will take advantage of the Landsat archive to create an average reference value of a given vegetation index at an earlier date T1 and see how it compares to its later value at T2.

The first assumption of this approach is that changes will happen within a buffer zone from the baseline extent. Start by setting the baseline extent and buffer zone using `focal_max`:

```
var buffer = 1000; // In meters
var extentBuffer = mangrove2000.focal_max(buffer, 'circle',
'meters');
Map.addLayer(mangrove2000, {
    palette: '#000000'
}, 'Baseline', false);
Map.addLayer(extentBuffer, {
    palette: '#0e49b5',
    opacity: 0.3
}, 'Mangrove Buffer', false);
```

48.2.2.1 Harmonizing Landsat 5/7/8 Image Collections

As described in the beginning of this chapter, the Landsat TM/ETM + and OLI sensors present differences between their spectral characteristics. Thus, to ensure inter-sensor harmonized spectral information and temporal continuity, we will harmonize the entire Landsat image archive using the statistical functions presented in Roy et al. (2016). For that, start by defining the temporal parameters for the harmonization:

```
var startYear = 1984;
var endyear = 2020;
var startDay = '01-01';
var endDay = '12-31';
```

Next, we will create several functions for the harmonization of the Landsat archive:

Harmonization Function: `harmonizationRoy` uses the regression coefficients (slopes and intercepts) retrieved from Roy et al. (2016) and performs a linear transformation of ETM + spectral space to OLI spectral space:

```
var harmonizationRoy = function(oli) {
    var slopes = ee.Image.constant([0.9785, 0.9542, 0.9825,
        1.0073, 1.0171, 0.9949
    ]);
    var itcp = ee.Image.constant([-0.0095, -0.0016, -
0.0022, -
        0.0021, -0.0030, 0.0029
    ]);
    var y = oli.select(['B2', 'B3', 'B4', 'B5', 'B6',
'B7'], [
            'B1', 'B2', 'B3', 'B4', 'B5', 'B7'
        ])
        .resample('bicubic')
        .subtract(itcp.multiply(10000)).divide(slopes)
        .set('system:time_start',
oli.get('system:time_start'));
    return y.toShort();
};
```

Retrieve a Particular Sensor Function: getSRcollection will be used to retrieve individual sensor collections based on the temporal parameters and the harmonization function above. Additionally, this function will mask cloud, cloud shadow, and snow based on the Landsat quality assessment bands:

```
var getSRcollection = function(year, startDay, endYear,
endDay,
    sensor) {
    var srCollection = ee.ImageCollection('LANDSAT/' +
sensor +
            '/C01/T1_SR')
        .filterDate(year + '-' + startDay, endYear + '-' +
endDay)
        .map(function(img) {
            var dat;
            if (sensor == 'LC08') {
                dat = harmonizationRoy(img.unmask());
            } else {
                dat = img.select(['B1', 'B2', 'B3', 'B4',
                        'B5', 'B7'
                    ])
                    .unmask()
                    .resample('bicubic')
                    .set('system:time_start', img.get(
                        'system:time_start'));
            }
            // Cloud, cloud shadow and snow mask.
            var qa = img.select('pixel_qa');
            var mask = qa.bitwiseAnd(8).eq(0).and(
                qa.bitwiseAnd(16).eq(0)).and(
                qa.bitwiseAnd(32).eq(0));
            return dat.mask(mask);
        });
    return srCollection;
};
```

Combining the Collections Function: getCombinedSRcollection will merge all the individual L5/L7/L8 collections into one:

```
var getCombinedSRcollection = function(year, startDay,
endYear,
    endDay) {
    var lt5 = getSRcollection(year, startDay, endYear,
endDay,
        'LT05');
    var le7 = getSRcollection(year, startDay, endYear,
endDay,
        'LE07');
    var lc8 = getSRcollection(year, startDay, endYear,
endDay,
        'LC08');
    var mergedCollection =
ee.ImageCollection(le7.merge(lc8)
        .merge(lt5));
    return mergedCollection;
};
```

Vegetation Indices: `addIndices` calculates several vegetation/spectral indices based on the harmonized Landsat bands. In this example, we are including the Normalized Difference Vegetation Index (NDVI), the Enhanced Vegetation Index (EVI), the Soil Adjusted Vegetation Index (SAVI), the Normalized Difference Mangrove Index (NDMI), the Normalized Difference Water index (NDWI), and the Modified Normalized Difference Water Index (MNDWI). Here is where you can include your own vegetation indices:

```
var addIndices = function(image) {
   var ndvi = image.normalizedDifference(['B4',
'B3']).rename(
      'NDVI');
  var evi = image.expression(
      '2.5*((NIR-RED)/(NIR+6*RED-7.5*BLUE+1))', {
          'NIR': image.select('B4'),
          'RED': image.select('B3'),
          'BLUE': image.select('B1')
      }).rename('EVI');
  var savi = image.expression(
      '((NIR - RED) / (NIR + RED + 0.5) * (0.5 + 1))', {
          'NIR': image.select('B4'),
          'RED': image.select('B3'),
          'BLUE': image.select('B1')
      }).rename('SAVI');
  var ndmi = image.normalizedDifference(['B7','B2']).rename(
      'NDMI');
  var ndwi = image.normalizedDifference(['B5','B4']).rename(
      'NDWI');
```

```
    var mndwi = image.normalizedDifference(['B2',
'B5']).rename(
        'MNDWI');
    return image.addBands(ndvi)
        .addBands(evi)
        .addBands(savi)
        .addBands(ndmi)
        .addBands(ndwi)
        .addBands(mndwi);
};
```

Finally, `collectionSR_wIndex` will include the final harmonized collection with all the sensors based on the temporal parameters defined previously, with all spectral bands and vegetation/spectral indices:

```
var collectionSR_wIndex =
getCombinedSRcollection(startYear, startDay,
    endyear, endDay).map(addIndices);
```

Filter this collection by the bounds of the area of study:

```
var collectionL5L7L8 =
collectionSR_wIndex.filterBounds(areaOfstudy);
```

Question 2. Based on your knowledge and the functions described above, what are the main fundamental differences between Landsat TM, ETM+, and OLI?

Question 3. What are the issues that may arise when using the same function for calculating vegetation indices using surface reflectance data acquired by ETM + and OLI sensors?

Vegetation Index Anomaly

By definition, anomaly is anything that deviates from what is standard, normal, or expected. Usually, anomalies are calculated by subtracting a long-term average of a variable from the actual value of that variable at a given time. For example, if X = actual value of average NDVI for mangroves in 2020, and Y = long-term average NDVI of mangroves (an average over many years), then the anomaly = $X - Y$. If the anomaly values are zero (or very close to zero), it means that NDVI remained relatively stable in that period, which indicates that there has not been any significant disturbance in that area. On the other hand, a positive anomaly means that the NDVI signal is greater than its long-term average, which indicates that vegetation has shown growth in that area; similarly, a negative anomaly means that the NDVI signal is weaker than its long-term average, indicating a potential loss in the area.

To calculate the anomaly, start by defining the index you want to compute the anomaly for and the reference period to get the average value. In this example, we are going use the 16 years before the mangrove extent baseline in 2000:

```
var index = 'NDVI';
var ref_start = '1984-01-01'; // Start of the period
var ref_end = '1999-12-31'; // End of the period
```

Next, create the reference collection using `collectionL5L7L8` and the parameters above. You can print the size of this reference collection to the **Console** using `print` and `size`:

```
var reference = collectionL5L7L8
    .filterDate(ref_start, ref_end)
    .select(index)
    .sort('system:time_start', true);
print('Number of images in Reference Collection',
reference.size());
```

You can now calculate the mean value (and other statistics) for the reference collection `reference`. Mask the results by the baseline mangrove extent using `extentBuffer`:

```
var mean = reference.mean().mask(extentBuffer);
var median = reference.median().mask(extentBuffer);
var max = reference.max().mask(extentBuffer);
var min = reference.min().mask(extentBuffer);
```

Now that we have our long-term reference metrics, you can define the period for which you want to compute the gains and losses. In this example, we will use the full period of 2000–2020. However, any combination of years is possible depending on what period you are interested in. Then, an anomaly function can be created to subtract the metric from the average of your period of interest:

```
var period_start = '2000-01-01'; // Full period
var period_end = '2020-12-31';

var anomalyfunction = function(image) {

    return image.subtract(mean)
        .set('system:time_start',
image.get('system:time_start'));
};
```

```
Finally, map the anomalyfunction to the Landsat collection
filtered by your period_start and period_end:
```

```
var series = collectionL5L7L8.filterDate(period_start,
period_end)
    .map(anomalyfunction);
```

Finally, map the `anomalyfunction` to the Landsat collection filtered by your period_start and `period_end`:

```
var seriesSum =
series.select(index).sum().mask(extentBuffer);
var numImages =
series.select(index).count().mask(extentBuffer);
var anomaly = seriesSum.divide(numImages);
```

The object `series` will have all the spectral bands and vegetation/spectral indices for the time period defined above. Their values, however, will be different from the original collection since we subtracted the average value of the reference period. The next step is to sum all the values for the index from `series` and divide by the number of images available:

Add the anomaly layer to the map using a color ramp of your choice (Fig. 48.2):

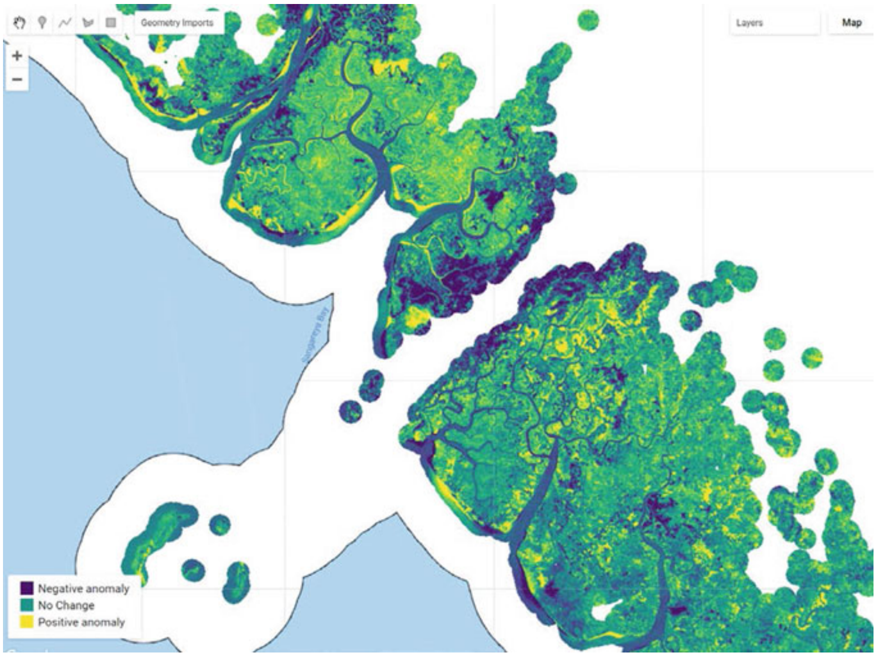


Fig. 48.2 NDVI anomaly for the period of 2000–2000

```
var visAnon = {
    min: -0.20,
    max: 0.20,
    palette: ['#481567FF', '#482677FF', '#453781FF',
'#404788FF',
        '#39568CFF', '#33638DFF', '#2D708EFF', '#287D8EFF',
        '#238A8DFF',
        '#1F968BFF', '#20A387FF', '#29AF7FFF', '#3CBB75FF',
        '#55C667FF',
        '#73D055FF', '#95D840FF', '#B8DE29FF', '#DCE319FF',
        '#FDE725FF'
    ]
};
Map.addLayer(anomaly, visAnon, index + ' anomaly');
```

You can then extract loss areas by selecting a value threshold on the anomaly (Fig. 48.3):

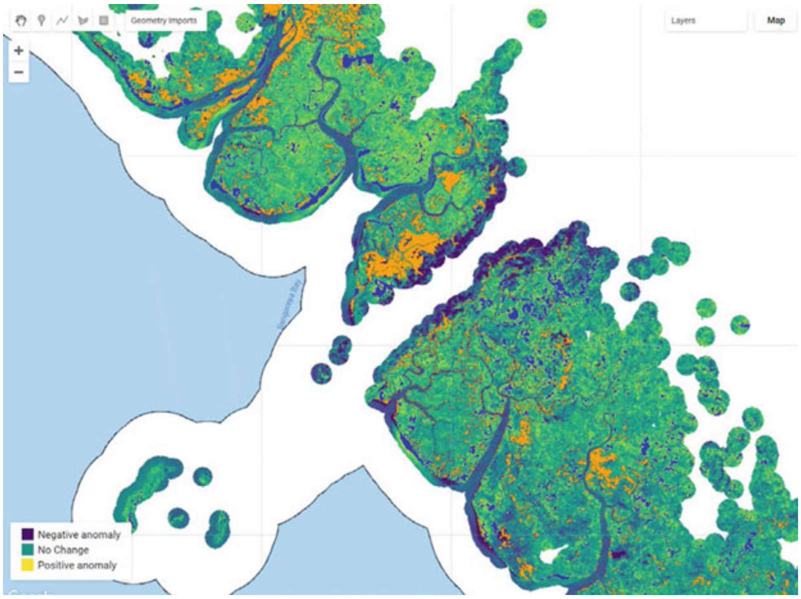


Fig. 48.3 NDVI anomaly losses (orange) and gains (blue) based on change threshold values

```
var thresholdLoss = -0.05;
var lossfromndvi = anomaly.lte(thresholdLoss)
    .selfMask()
    .updateMask(
        mangrove2000
    ); // Only show the losses within the mangrove extent
of year 2000

Map.addLayer(lossfromndvi, {
    palette: ['orange']
}, 'Loss from Anomaly 00-20');

var thresholdGain = 0.20;
var gainfromndvi = anomaly.gte(thresholdGain)
    .selfMask()
    .updateMask(
        extentBuffer
    ); // Only show the gains within the mangrove extent
buffer of year 2000

Map.addLayer(gainfromndvi, {
    palette: ['blue']
}, 'Gain from Anomaly 00-20');
```

Code Checkpoint A33b. The book's repository contains a script that shows what your code should look like at this point.

Question 4. What are the main challenges of a map-to-image change detection approach? Think from a perspective of baseline extent, spectral index, and metric used (e.g., mean versus median).

48.3 Synthesis

With the content covered in this chapter, you will be able to reproduce a map-to-map and a map-to-image change detection to your own area of interest. Additionally, the anomaly analysis can be used to detect changes in other land cover types, including (but not limited to) forests (forest cover loss) and agricultural land (crop harvest).

Assignment 1. Practice your land cover classification skills by using the code in script **A33s1—Supplemental** in the book's repository. In this code, we show how to create a mangrove extent map for Guinea using manually and automatically selected samples. The code covers masking techniques and using data from Google Earth Engine Catalog to assist in the classification workflow. Using the techniques you have learned, create two 30 m Landsat-based mangrove extent maps for the year of 2000 and 2020 for Guinea, West Africa.

Assignment 2. Using what you learned in this chapter, compile the areas (in km^2) of mangrove change in Guinea derived from the anomaly analysis using: (a) vegetation spectral index versus a water-based spectral index; (b) mean versus median; and (c) five-year intervals (2000–2005, 2005–2010, 2010–2015, and 2015–2020.

Assignment 3. Practice the classification and compare the map-to-map and map-to-image change approaches for another land cover type/area of your choice, such as the following:

- Agricultural land expansion in the Nile Delta, Egypt.
- Forest burn/loss near Mount Hood, Oregon, United States.

48.4 Conclusion

Google Earth Engine's computing infrastructure has been revolutionizing time-consuming remote sensing processes, creating a new way for rapid land cover classification and change detection at large scales. In the case of mangrove forests—a highly dynamic ecosystem that has been under increasing anthropogenic pressure—these approaches allow for a rapid and consistent monitoring of change in extent over time. These approaches using Earth Engine may be particularly useful for developing countries where, until very recently, the high computational power and the difficulties of distributing nontrivial classification algorithms across multiple computational workstations has challenged classification and change detection at large scales.

References

Bunting P, Rosenqvist A, Lucas RM et al (2018) The global mangrove watch—a new 2010 global baseline of mangrove extent. Remote Sens 10:1669. https://doi.org/10.3390/rs10101669

Goldberg L, Lagomasino D, Thomas N, Fatoyinbo T (2020) Global declines in human-driven mangrove loss. Glob Chang Biol 26:5844–5855. https://doi.org/10.1111/gcb.15275

Hansen MC, Potapov PV, Moore R et al (2013) High-resolution global maps of 21st-century forest cover change. Science 342:850–853. https://doi.org/science.1244693

Roy DP, Kovalskyy V, Zhang HK et al (2016) Characterization of Landsat-7 to Landsat-8 reflective wavelength and normalized difference vegetation index continuity. Remote Sens Environ 185:57–70. https://doi.org/10.1016/j.rse.2015.12.024

Forest Degradation and Deforestation

49

Carlos Souza Jr., Karis Tenneson, John Dilger, Crystal Wespestad, and Eric Bullock

Overview

Tropical forests are being disturbed by deforestation and forest degradation at an unprecedented pace (Hansen et al. 2013; Bullock et al. 2020). Deforestation completely removes the original forest cover and replaces it with another land cover type, such as pasture or agriculture fields. Generally speaking, forest degradation is a temporary or permanent disturbance, often caused by predatory logging, fires, or forest fragmentation, where the tree loss does not entirely change the land cover type. Forest degradation leads to a more complex environment with a mixture of vegetation, soil, tree trunks and branches, and fire ash. Defining a boundary between deforestation and forest degradation is not straightforward; at the time this chapter was

C. Souza Jr. (✉)
Amazon Institute of People and the Environment, Belém, Brazil
e-mail: souzajr@imazon.org.br

K. Tenneson · J. Dilger · C. Wespestad
Spatial Informatics Group, Pleasanton, CA, USA
e-mail: ktenneson@sig-gis.com

J. Dilger
e-mail: jdilger@sig-gis.com

C. Wespestad
e-mail: cwespestad@sig-gis.com

E. Bullock
Boston University, Boston, MA, USA
e-mail: bullocke@bu.edu

K. Tenneson · J. Dilger · C. Wespestad
SERVIR-Amazonia, Cali, Colombia

J. Dilger
Astraea, Charlottesville, VA, USA

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_49

written, there was no universally accepted definition for forest degradation (Aryal et al. 2021). Furthermore, the signal of forest degradation often disappears within one to two years, making degraded forests spectrally similar to undisturbed forests. Due to these factors, detecting and mapping forest degradation with remotely sensed optical data is more challenging than mapping deforestation.

The purpose of this chapter is to present a spectral unmixing algorithm and the normalized difference fraction index (NDFI) to detect and map both forest degradation and deforestation in tropical forests. This spectral unmixing model uses a set of generic endmembers (Souza et al. 2005) to process any Landsat Surface Reflectance (Tier 1) scene available in Google Earth Engine. We present two examples of change detection applications: one comparing a pair of images acquired at different times a year apart by making a temporal color composite and an empirically defined change threshold, and another using a more extensive and dense time series approach.

Learning Outcomes

- Calculating NDFI, the Normalized Difference Fraction Index.
- Interpreting fraction images and NDFI using a temporal color composite.
- Analyzing deforestation and forest degradation with NDFI.
- Running a time-series change detection to detect forest change.

Helps if you know how to

- Import images and image collections, filter, and visualize (Part I).
- Perform basic image analysis: select bands, compute indices, and create masks (Part II).
- Use drawing tools to create points, lines, and polygons (Chap. 6).
- Run and interpret spectral unmixing models (Chap. 9).
- Use expressions to perform calculations on image bands (Chap. 9).
- Aggregate data to build a time series (Chap. 14).
- Perform a two-period change detection (Chap. 16).

49.1 Introduction to Theory

Landsat imagery has been extensively used to monitor deforestation (Woodcock et al. 2020). However, detecting and mapping forest degradation associated with selective logging is more intricate and challenging. First efforts involved the application of spectral and textural indices to enhance the detection of canopy damage created by logging (Asner et al. 2002; Souza et al. 2005), but these turned out to be more helpful in enhancing logging infrastructure using Landsat shortwave infrared bands (i.e., roads and log landings; Matricardi et al. 2007).

Alternatively, spectral mixture analysis (SMA) has been proposed to overcome the challenge of using whole-pixel information to detect and classify forest degradation. Landsat pixels typically contain a mixture of land cover components (Adams et al. 1995). The SMA method is based on the linear spectral unmixing model, as described in Chap. 9. The identification of the nature and number of pure spectra (often referred to in this context as "endmembers") in the image scene is an important step in obtaining correct SMA models. In logged forests (and also in burned forest and forest edges), mixed pixels predominate and are expected to have a combination of green vegetation (GV), soil, non-photosynthetic vegetation (NPV), and shade-covered materials. Therefore, fraction images derived from SMA analyses are more suitable to enhance the detectability of logging infrastructure and canopy damage within degraded forests. For example, soil fractions reveal log landings and logging roads (Souza and Barreto 2000), while the NPV highlights forest canopy damage (Souza et al. 2003), and the areas decreasing in GV indicate forest canopy gaps (Asner et al. 2004).

A study has shown that it is possible to generalize the SMA model to Landsat sensors (including TM, ETM+, and OLI) (Small 2004). Souza et al. (2005) expanded the generalized SMA to handle five endmembers—GV, NPV, soil, shade, and cloud—expected within forest degradation areas and proposed a novel compositional index based on SMA fractions, the normalized difference fraction index (NDFI).

The NDFI is computed as:

$$\mathrm{NDFI} = \frac{\mathrm{GVshade} - (\mathrm{NPV} + \mathrm{Soil})}{\mathrm{GVshade} + \mathrm{NPV} + \mathrm{Soil}} \tag{49.1}$$

where GVshade is the shade-normalized GV fraction given by,

$$\mathrm{GVshade} = \frac{\mathrm{GV}}{100 - \mathrm{Shade}} \tag{49.2}$$

NDFI values range from -1 to 1. For intact forests, NDFI shows high values (i.e., about 1) due to the combination of high GVshade (i.e., high GV and canopy shade) and low NPV and soil values. The NPV and soil fractions increase as forests are more degraded, lowering NDFI values relative to the intact forests. Deforested areas exhibit very low GV and shade and high NPV and soil, making it possible to distinguish them from degraded forests based on NDFI magnitude.

Recent studies compared NDFI with other spectral indices. NDFI generated more accurate results in deforestation detection (Schultz et al. 2016) and forest degradation (Bullock et al. 2018) in time-series analysis. One of the key components for its success is lowering unwanted noise and accounting for illumination variability through the shade normalization applied to the GV fraction.

49.2 Practicum

49.2.1 Section 1: Spectral Mixture Analysis Model

Let us first define the Landsat endmembers based on Souza et al. (2005). These endmembers were developed and tested in the Amazon. These endmembers work well in many other environments (see example applications for calculating NDFI in non-Amazonian tropical forest in Schultz et al. 2016 [Ethiopia and Viet Nam]; Kusbach et al. 2017 [central Europe]; Hirschmugl et al. 2013 [Cameroon and Central African Republic]). If you are working in a different region, assess how well they perform for your forest types. The ratio of these endmembers that makes up the spectral signature of each pixel gives a good indication of the plant health and composition for that area. When the ratio shifts over time towards one or more of the endmembers, we can quantify how the landscape is changing.

Below, we will create a new variable `endmembers` by copying the values from the code block below. The six numbers in square brackets define the pure reflectance values for the blue, green, red, SWIR1, and SWIR2 bands for each endmember material.

```
// SMA Model - Section 1

// Define the Landsat endmembers (source: Souza et al.
2005)
// They can be applied to Landsat 5, 7, 8, and potentially
9.
var endmembers = [
  [0.0119,0.0475,0.0169,0.625,0.2399,0.0675], // GV
  [0.1514,0.1597,0.1421,0.3053,0.7707,0.1975], // NPV
  [0.1799,0.2479,0.3158,0.5437,0.7707,0.6646], // Soil
  [0.4031,0.8714,0.79,0.8989,0.7002,0.6607] // Cloud
];
```

We will choose a single Landsat 5 image to work with for now, and select the bands we will need for the SMA and NDFI calculation.

Create a new variable `image` and assign it to the Landsat 5 image from the code block below. Select the visible, near infrared, and shortwave infrared bands. Then, center the map around the Landsat 5 image with a zoom scale of 10.

```
// Select a Landsat 5 scene on which to apply the SMA
model.
var image =
ee.Image('LANDSAT/LT05/C02/T1_L2/LT05_226068_19840411')
    .multiply(0.0000275).add(-0.2);

// Center the map on the image object.
Map.centerObject(image, 10);

// Define and select the Landsat bands to apply the SMA
model.
// use ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5',
'SR_B7'] for Landsat 5 and 7.
// use ['SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6',
'SR_B7'] for Landsat 8.
var bands = ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5',
'SR_B7'];
image = image.select(bands);
```

Next, we will need to create a couple of functions to use the endmembers and create the NDFI image. We will need to unmix the input Landsat image.

First, we will create a new function, `getSMAFractions` that takes two parameters: `image` and `endmembers`. To unmix the image, we first select the visible, near-infrared, and short wave infrared bands, and call the `unmix` function with the endmembers used as the argument. Note: The order in which the bands are selected matters. For each endmember (GV, NPV, soil, and cloud), each value in the array represents the pure value for each band being passed in. For example, if the selected bands were [‘SR_B2’, ‘SR_B3’, ‘SR_B4’, ‘SR_B5’, ‘SR_B6’, ‘SR_B7’] then the selected endmembers at position 0 would be:

- ‘SR_B2’ GV: 119
- ‘SR_B2’ NPV: 1514
- ‘SR_B2’ Soil: 1799
- ‘SR_B2’ Cloud: 4031.

```
// Unmixing image using Singular Value Decomposition.
var getSMAFractions = function(image, endmembers) {
    var unmixed = ee.Image(image)
        .select([0, 1, 2, 3, 4,
            5
        ]) // Use the visible, NIR, and SWIR bands only!
        .unmix(endmembers)
        .max(0) // Remove negative fractions, mostly Soil.
        .rename('GV', 'NPV', 'Soil', 'Cloud');
    return ee.Image(unmixed.copyProperties(image));
};
```

We will now write the function to calculate NDFI. We could write it out line by line for each time we want to perform SMA or calculate NDFI. But having the equation implemented as a function gives us many benefits, including reducing redundancy, allowing us to map the function over a collection of images, and reducing errors from typos or other little bugs that can find their way into our code.

We will use the fraction images obtained with the `getSMAFractions` function above to calculate shade, GVs, and NDFI using image expressions. This procedure will return a multiband image with the shade, GVs, and NDFI bands added to the input image.

First, we will create a new variable `sma` and pass in the image and endmembers as arguments. Then calculate the shade and GV shade-normalized (GVs) fractions from the SMA bands, and add the shade and GVs bands to the SMA image. We calculate NDFI using an expression implementing Eq. 49.1, and add the new band to the SMA image.

```
// Calculate GVS and NDFI and add them to image fractions.
// Run the SMA model passing the Landsat image and the
endmembers.
var sma = getSMAFractions(image, endmembers);

Map.addLayer(sma, {
    bands: ['NPV', 'GV', 'Soil'],
    min: 0,
    max: 0.45
}, 'sma');
```

```
// Calculate the Shade and GV shade-normalized (GVs)
fractions from the SMA bands.
var Shade = sma.reduce(ee.Reducer.sum())
    .subtract(1.0)
    .abs()
    .rename('Shade');

var GVs = sma.select('GV')
    .divide(Shade.subtract(1.0).abs())
    .rename('GVs');

// Add the new bands to the SMA image variable.
sma = sma.addBands([Shade, GVs]);

// Calculate the NDFI using image expression.
var NDFI = sma.expression(
    '(GVs - (NPV + Soil))  / (GVs + NPV + Soil)', {
        'GVs': sma.select('GVs'),
        'NPV': sma.select('NPV'),
        'Soil': sma.select('Soil')
    }).rename('NDFI');

// Add the NDFI band to the SMA image.
sma = sma.addBands(NDFI);
```

We will use a color palette that spans from white, to pink (i.e., bare land), to yellow, to green to visualize the NDFI image. Higher values of NDFI will be green, while lower values will span the colors of white, pink, and yellow. Copy the code block below into your Code Editor.

```
// Define NDFI color table.
var palettes = require(
    'projects/gee-edu/book:Part A - Applications/A3 -
Terrestrial Applications/A3.4 Forest Degradation and
Deforestation/modules/palettes'
);

var ndfiColors = palettes.ndfiColors;
```

Next, we can visualize all the hard work we have done unmixing each endmember and the NDFI bands (Fig. 49.1).

Create an image visualization object with bands 5, 4, and 3, chosen for visualization along with a min and max. Add the Landsat 5 image to the map using the image visualization object.

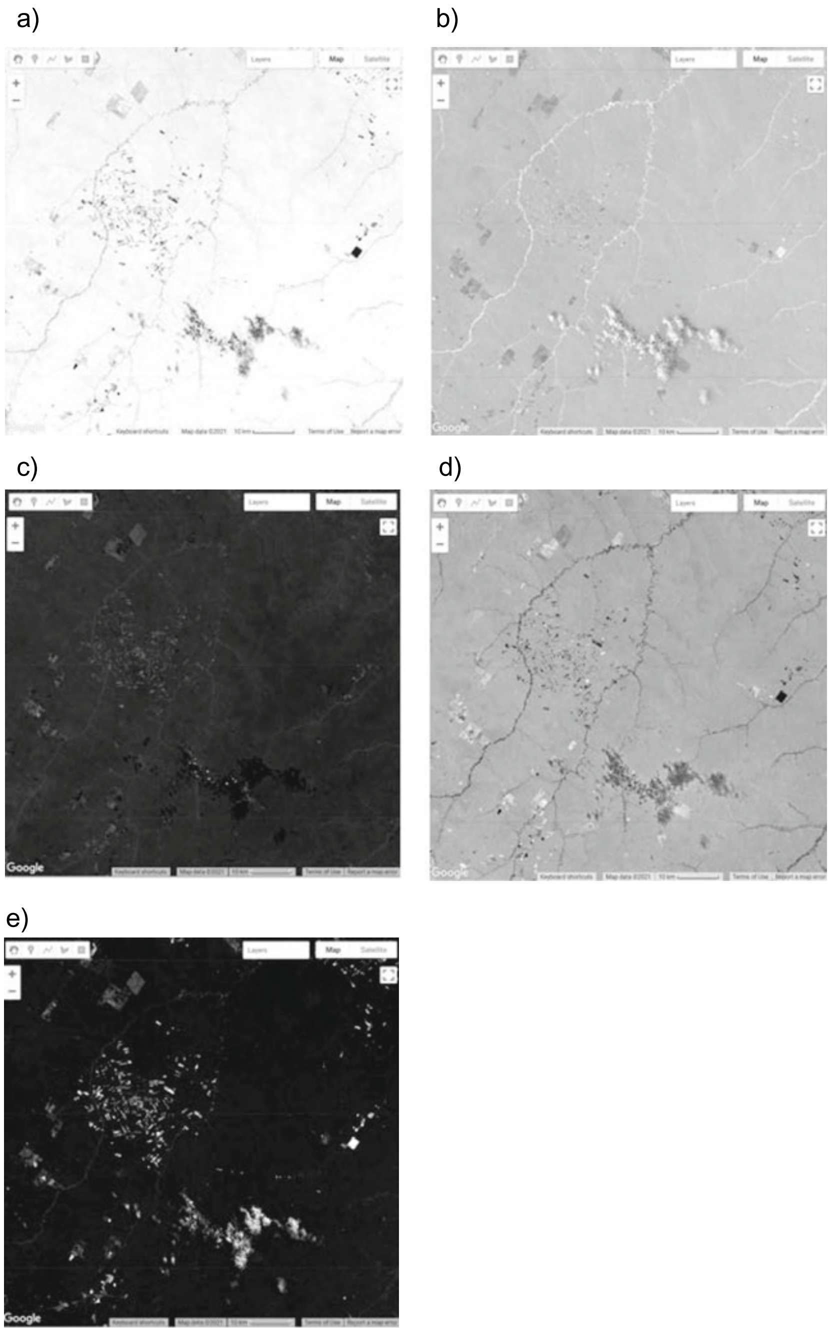


Fig. 49.1 Example maps of **a** green vegetation shade normalized fraction (more vegetation is whiter); **b** shade fraction (more shade is whiter); **c** non-photosynthetic vegetation fraction (more NPV is whiter); **d** green vegetation fraction (more vegetation is whiter); **e** soil fraction (more soil is whiter)

Now add each of the SMA bands and the NDFI band to the map. Note: Rather than defining the min, max, and bands for each of these in separate variables, we can pass in the object directly to the Map.addLayer function.

```
var imageVis = {
    'bands': ['SR_B5', 'SR_B4', 'SR_B3'],
    'min': 0,
    'max': 0.4
};

// Add the Landsat color composite to the map.
Map.addLayer(image, imageVis, 'Landsat 5 RGB-543', true);

// Add the fraction images to the map.
Map.addLayer(sma.select('Soil'), {
    min: 0,
    max: 0.2

}, 'Soil');
Map.addLayer(sma.select('GV'), {
    min: 0,
    max: 0.6
}, 'GV');
Map.addLayer(sma.select('NPV'), {
    min: 0,
    max: 0.2
}, 'NPV');
Map.addLayer(sma.select('Shade'), {
    min: 0,
    max: 0.8
}, 'Shade');
Map.addLayer(sma.select('GVs'), {
    min: 0,
    max: 0.9
}, 'GVs');
Map.addLayer(sma.select('NDFI'), {
    palette: ndfiColors
}, 'NDFI');
```

The last thing we will do in this section is to create a water and cloud mask. Water and clouds will have lower NDFI values. While water may not be too much of an issue, clouds will impact how we monitor forest degradation and loss, and thus we can simply mask them out. We can mask them using a thresholding method based on the values of our fraction images.

First, create a new function variable getWaterMask that takes an SMA image as the only argument.

Next, create a water mask using threshold values for the shade, GV, and soil bands, where shade is greater than or equal to 0.65; GV is less than or equal to 0.15; and soil is less than or equal to 0.05.

Now create a cloud mask by applying a threshold of 0.1 or greater to the Cloud band.

```
var getWaterMask = function(sma) {
    var waterMask = (sma.select('Shade').gte(0.65))
        .and(sma.select('GV').lte(0.15))
        .and(sma.select('Soil').lte(0.05));
    return waterMask.rename('Water');
};

// You can use the variable below to get the cloud mask.
var cloud = sma.select('Cloud').gte(0.1);
var water = getWaterMask(sma);
```

Next, we will combine the cloud and water masks using the max reducer. Since we want to mask both water and clouds, using the max works quite nicely here—as opposed to adding the images together—so we do not need to worry about overlaps.

Now add the cloud and water mask as a layer to the map.

Apply the cloud and water mask to the NDFI band using the `updateMask` function and invert the mask using the `not` function. Note: `updateMask` considers zeroes as invalid (i.e., to be masked) and ones as valid (i.e., to be kept). Since our original mask had values of 1 for cloud and water, we use the `not` function to invert the values.

```
var cloudWaterMask = cloud.max(water);
Map.addLayer(cloudWaterMask.selfMask(),
    {
        min: 1,
        max: 1,
        palette: 'blue'
    },
    'Cloud and water mask');

// Mask NDFI.
var maskedNDFI =
sma.select('NDFI').updateMask(cloudWaterMask.not());
Map.addLayer(maskedNDFI, {
    palette: ndfiColors
}, 'NDFI');
```

Code Checkpoint A34a. The book's repository contains a script that shows what your code should look like at this point.

49.2.2 Section 2: Deforestation and Forest Degradation Change Detection

To observe changes in the landscape over time, you need to create an NDFI image for two points in time and then calculate the difference between them. Previous studies have shown that images should not be more than one year apart because the forest degradation disturbance quickly disappears with tree foliage and understory vegetation growth. Changes in NDFI are a good indicator of forest change.

Now we will start using the functions we have built. First, we perform the SMA on a Landsat 5 image. Recall that the SMA function wants only the visible, near-infrared, and shortwave infrared bands. When using Landsat 5, those correspond to bands 1–5 and band 7.

First, create a variable for the Landsat 5 scene specified in the code block below and select the visible, near-infrared, and shortwave infrared bands. Use the `getSMAFractions` function with the Landsat image and the endmembers. Rename the output SMA bands as GV, NPV, Soil, and Cloud.

```
// Select two Landsat 5 scenes on which to apply the SMA
model.

// Select Landsat bands used for forest change detection.
var imageTime0 = ee.Image(
        'LANDSAT/LT05/C02/T1_L2/LT05_226068_20000509')
    .multiply(0.0000275).add(-0.2);
var bands = ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5',
'SR_B7'];
imageTime0 = imageTime0.select(bands);

// Run the SMA model.
var smaTime0 = getSMAFractions(imageTime0, endmembers);
```

Before we move on to calculate the NDFI, we will add the previous Landsat scene and the fractional images to the map to inspect them (example results in Fig. 49.2).

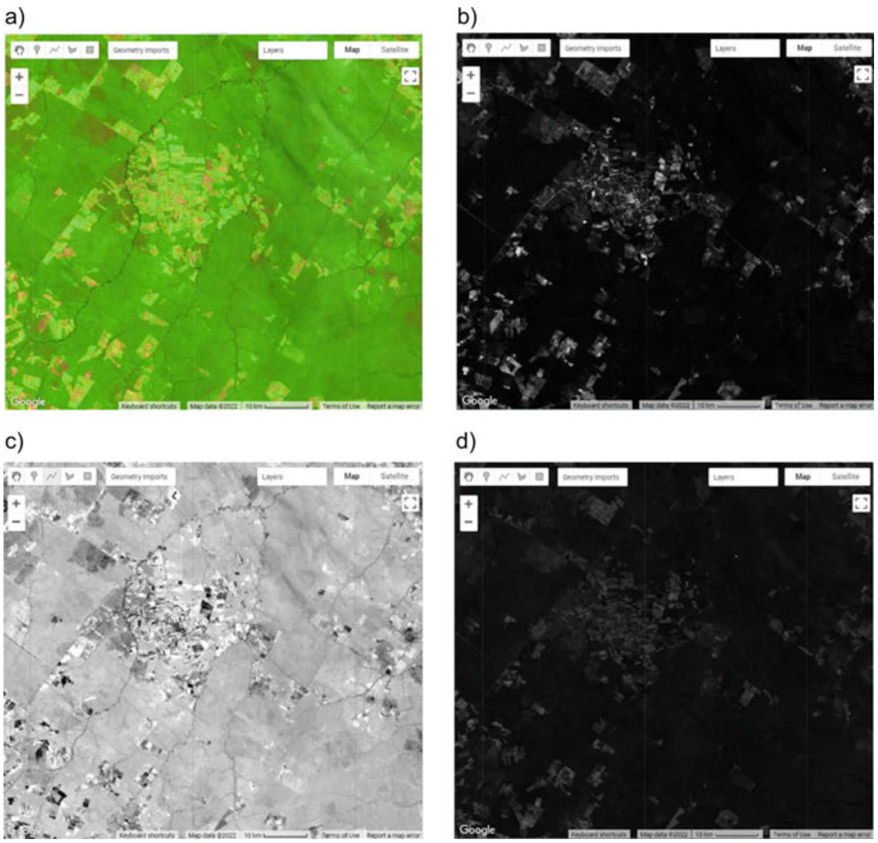


Fig. 49.2 Example maps of **a** RGB composite of bands 5–4–3 (green is vegetation and brown is barren ground); **b** soil fraction (more soil is whiter); **c** green vegetation fraction (more vegetation is whiter); **d** non-photosynthetic vegetation fraction (more NPV is whiter)

```
// Center the image object.
Map.centerObject(imageTime0, 10);

// Define the visualization parameters.
var imageVis = {
    'opacity': 1,
    'bands': ['SR_B5', 'SR_B4', 'SR_B3'],
    'min': 0,
    'max': 0.4,
    'gamma': 1
};

// Scale to the expected maximum fraction values.
var fractionVis = {
    'opacity': 1,
    'min': 0.0,
    'max': 0.5
};

// Add the Landsat color composite to the map.
Map.addLayer(imageTime0, imageVis, 'Landsat 5 RGB 543',
true);

// Add the fraction images to the map.
Map.addLayer(smaTime0.select('Soil'), fractionVis, 'Soil
Fraction');
Map.addLayer(smaTime0.select('GV'), fractionVis, 'GV
Fraction');
Map.addLayer(smaTime0.select('NPV'), fractionVis, 'NPV
Fraction');
```

Next, let's set up a function to systematically reproduce the work we did computing NDFI, since we will need it a couple more times. Hereafter, we will be able to call that function instead of needing to explicitly write out each step, thus simplifying our code and making it easier to change if we need to later.

```
function getNDFI(smaImage) {
    // Calculate the Shade and GV shade-normalized (GVs)
fractions
    // from the SMA bands.
    var Shade = smaImage.reduce(ee.Reducer.sum())
        .subtract(1.0)
        .abs()
        .rename('Shade');

    var GVs = smaImage.select('GV')
        .divide(Shade.subtract(1.0).abs())
        .rename('GVs');

    // Add the new bands to the SMA image variable.
    smaImage = smaImage.addBands([Shade, GVs]);

    var ndfi = smaImage.expression(
        '(GVs - (NPV + Soil))  / (GVs + NPV + Soil)', {
            'GVs': smaImage.select('GVs'),
            'NPV': smaImage.select('NPV'),
            'Soil': smaImage.select('Soil')
        }
    ).rename('NDFI');

    return ndfi;
}
```

Then, calculate NDFI for the earlier Landsat image's SMA bands (use `smaTime0` as an input) using the `getNDFI` function you wrote. Add this NDFI image to the map displayed in the `ndfiColors` palette (Fig. 49.3). This will serve as your calculated NDFI for your earlier point in time, the pre-change time (this time was defined as `imageTime0`).

```
// Create the initial NDFI image and add it to the map.
var ndfiTime0 = getNDFI(smaTime0);
Map.addLayer(ndfiTime0,
    {
        bands: ['NDFI'],
        min: -1,
        max: 1,
        palette: ndfiColors
    },
    'NDFI t0',
    false);
```

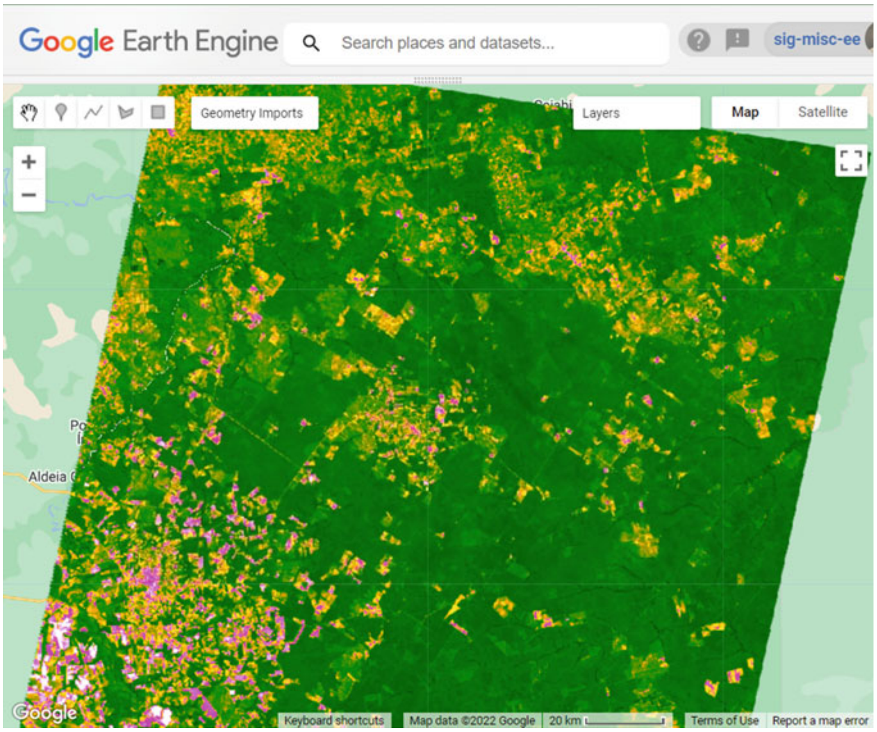


Fig. 49.3 NDFI image for the previous year obtained from the Landsat 5 (path/row 226/068) scene acquired on May 9, 2000. Orange colors indicate signs of forest disturbance associated with fires and selective logging. Pink and white colors are dry vegetation and bare soil in old deforested areas. Orange colors in pasturelands mean dry vegetation

Next, you will repeat this procedure to calculate the SMA fractions and NDFI of the second Landsat 5 image (`smaTime1`). You will utilize the same SMA method.

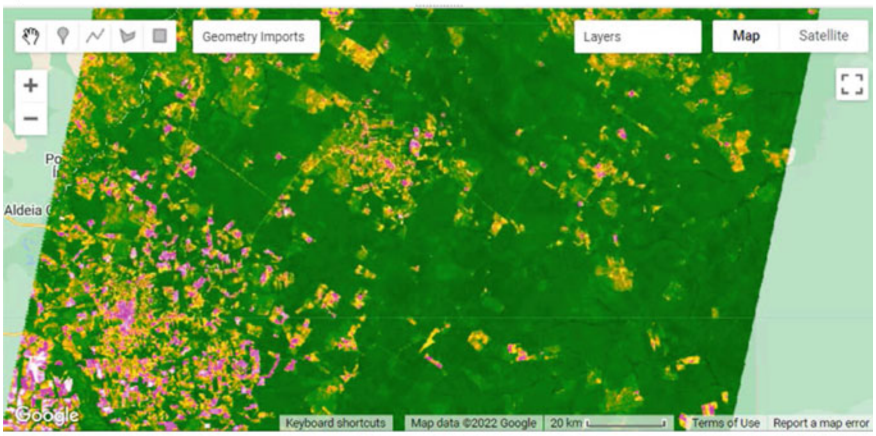
First, create a new variable (`imageTime1`), using the Landsat 5 scene from the code block below. Note that the band numbers for Landsat 8 are different from those for Landsat 5. In general, you should always make sure to check band names when working with multiple Landsat collections. Then, you will calculate the SMA fractions. The `getSMAFractions` function will rename the outputs to “GV”, “NPV”, “Soil”, and “Cloud”. Then, you will calculate NDFI for the new scene. Add an RGB composite and the NDFI to the map.

```
// Select a second Landsat 5 scene on which to apply the
SMA model.
var imageTime1 = ee.Image(
        'LANDSAT/LT05/C02/T1_L2/LT05_226068_20010629')
    .multiply(0.0000275).add(-0.2)
    .select(['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5',
'SR_B7']);
var smaTime1 = getSMAFractions(imageTime1, endmembers);

// Create the second NDFI image and add it to the map.
var ndfiTime1 = getNDFI(smaTime1);

Map.addLayer(imageTime1, imageVis, 'Landsat 5 t1 RGB-5',
true);
Map.addLayer(ndfiTime1,
    {
        bands: ['NDFI'],
        min: -1,
        max: 1,
        palette: ndfiColors
    },
    'NDFI_t1',
    false);
```

Before looking at changes between the two NDFI images, use the opacity slider in the **Layers** panel of the map to manually view the differences between the NDFI outputs, and then the RGB outputs. Where do you expect to see the greatest changes between the two?

An easier way to enhance changes using a pair of images is to build a temporal color composite. To construct a temporal color composite, we add the first image date to the R color channel and the second to the G and B channels. An example of RGB color composite (as first described in Chap. 2) is shown in Fig. 49.4, using the `NDFI_t0` (R) and `NDFI_t1` (G and B). Deforestation appears in bright red colors since we assigned the first NDFI image to the R color channel, indicating forest in the previous year, and removal in the following (i.e., G and B colors have low NDFI values due to forest removal). In contrast, the cyan colors in the NDFI temporal color composite indicate vegetation regrowth in the second year. The gradient of dark to gray colors suggests no change in NDFI between the two dates.

a)



b)

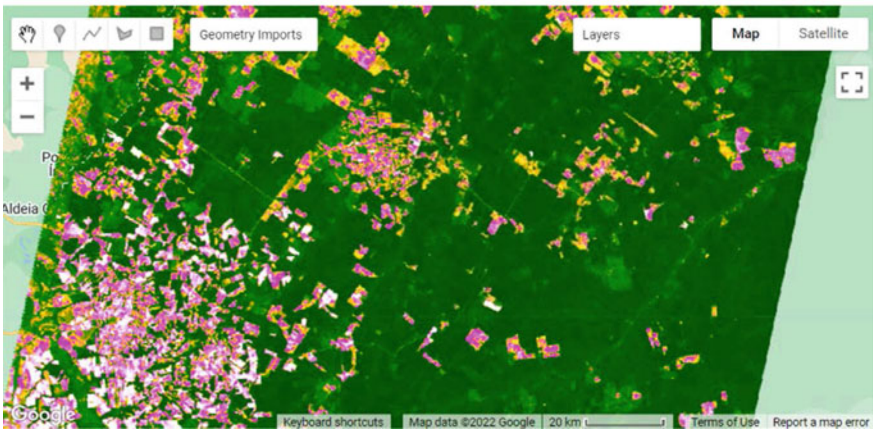
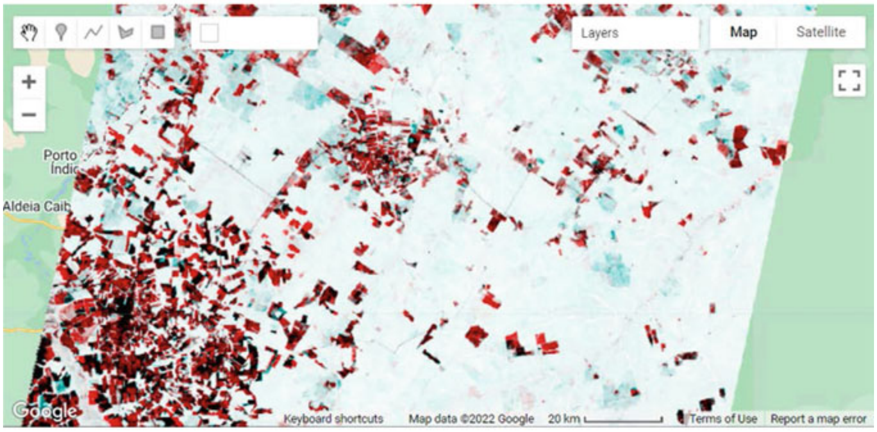


Fig. 49.4 Example of NDFI obtained for the first (**a**) and second (**b**) Landsat images, and a visualization of them using a temporal RGB color composite (**c**). Intense red colors are newly deforested areas, and light red are selectively logged forests. Cyan colors indicate vegetation regrowth in areas that were logged or burned a year ago or more. Example of NDFI difference histogram (**d**), with positive values indicating an increase in NDFI over time and negative values a decrease. Values at zero suggest no change

c)



d)

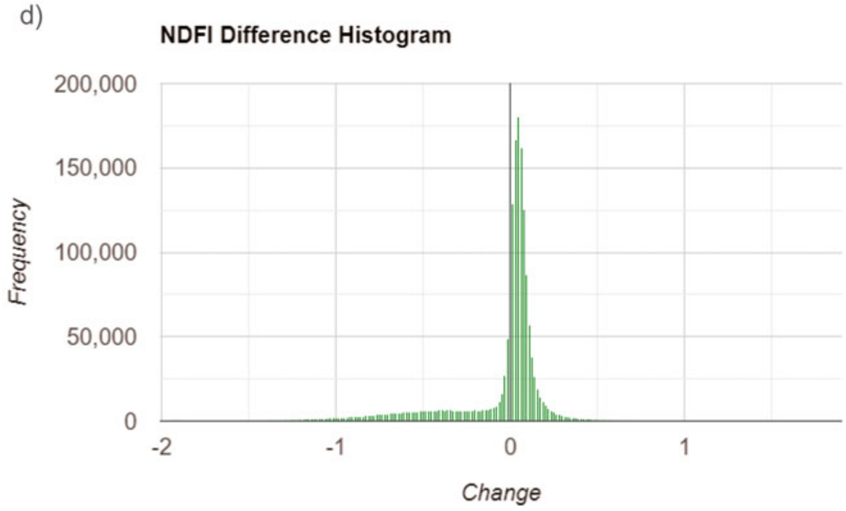


Fig. 49.4 (continued)

To find the change between our images, a simple difference method can be applied by subtracting the previous image from the current image. Then, we can apply an empirically defined threshold to classify the changes based on the inspection of the histogram and the NDFI temporal color composite. For more information on two-date differencing, see Chap. 16.

Next, we will combine the two NDFI images for time `t0` and `t1`. Create a new variable `ndfiChange` and subtract the current NDFI image (made using Landsat 5 imagery from 2001) from the previous NDFI image (made using Landsat 5 imagery from 2000). Note: The NDFI is calculated using the image expression method applied to the SMA fraction presented in the code above. We then combine the two NDFI bands from 2000 and 2001 in one variable to display the change over time.

```
// Combine the two NDFI images in a single variable.
var ndfi = ndfiTime0.select('NDFI')
    .addBands(ndfiTime1.select('NDFI'))
    .rename('NDFI_t0', 'NDFI_t1');

// Calculate the NDFI change.
var ndfiChange = ndfi.select('NDFI_t1')
    .subtract(ndfi.select('NDFI_t0'))
    .rename('NDFI Change');
```

Using the polygon drawing tool (as described in Chap. 6), draw a region that covers most of the `ndfiChange` image and rename it `region` in the variable import panel.

Next, make a histogram named `histNDFIChange` that bins the `ndfiChange` image using the region you just drew, and prints it to the **Console**. Optionally, click **Run** to view the histogram and identify some potential change thresholds. In your case the histogram may look slightly different.

```
var options = {
    title: 'NDFI Difference Histogram',
    fontSize: 20,
    hAxis: {
        title: 'Change'
    },
    vAxis: {
        title: 'Frequency'
    },
    series: {
        0: {
            color: 'green'
        }
    }
};

// Inspect the histogram of the NDFI change image to define
threshold
// values for classification. Make the histogram, set the
options.
var histNDFIChange = ui.Chart.image.histogram(
        ndfiChange.select('NDFI Change'), region, 30)
    .setSeriesNames(['NDFI Change'])
    .setOptions(options);

print(histNDFIChange);
```

Classify the difference image into new deforestation (red), forest degradation (orange), regrowth (cyan), and forest (green). The classification is based on slicing the NDFI difference image. Old deforested areas are detected using the NDFI first image date (`t0`).

Add the change classification and difference images to the map and click **Run**.

```
// Classify the NDFI difference image based on thresholds
// obtained from its histogram.
var changeClassification = ndfiChange.expression(
        '(b(0) >= -0.095 && b(0) <= 0.095) ? 1 :' +
        //  No forest change
        '(b(0) >= -0.250 && b(0) <= -0.095) ? 2 :' + //
Logging
        '(b(0) <= -0.250) ? 3 :' + // Deforestation
        '(b(0) >= 0.095) ? 4  : 0') // Vegetation regrowth
    .updateMask(ndfi.select('NDFI_t0').gt(
        0.60)); // mask out no forest

// Use a simple threshold to get forest in the first image
date.
var forest = ndfi.select('NDFI_t0').gt(0.60);
```

Finally, add code to add all the new layers to the map and click **Run**.

```
// Add layers to map
Map.addLayer(ndfi, {
    'bands': ['NDFI_t0', 'NDFI_t1', 'NDFI_t1']
}, 'NDFI Change');
Map.addLayer(ndfiChange, {}, 'NDFI Difference');
Map.addLayer(forest, {}, 'Forest t0 ');
Map.addLayer(changeClassification, {
        palette: ['000000', '1eaf0c', 'ffc239', 'ff422f',
            '74fff9']
    },
    'Change Classification');
```

Figure 49.5 shows an example of forest changes between two dates displayed using the cutoffs defined by the histogram. Categorizing the whole map into a simple system of no change, old deforestation, new deforestation, partial forest disturbance, and regrowth makes the forest use patterns in the region clearer. However, you must be careful with the thresholds you chose from the histogram when creating and interpreting the NDFI changes in this way, as those values can drastically alter your map. Consider when it might be more appropriate to use an RGB temporal color composite, and in what situations the classified map using the empirically defined thresholds would be better.

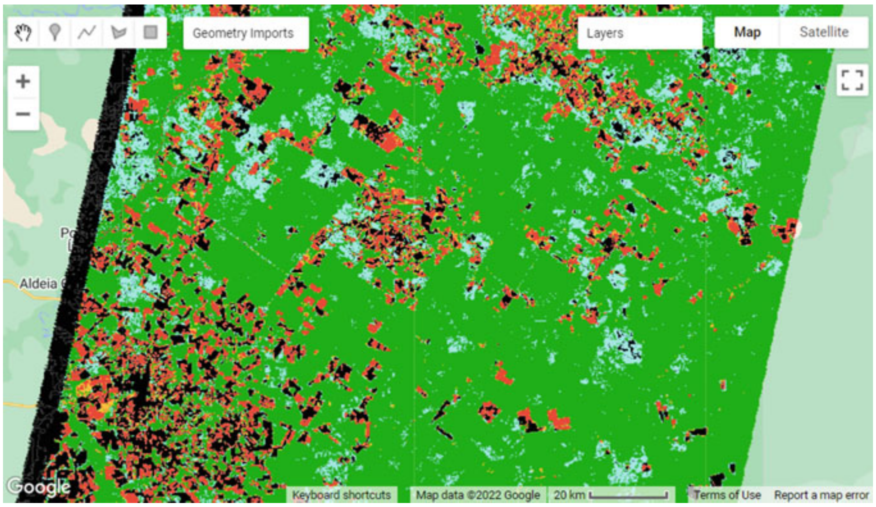


Fig. 49.5 Example of the change detection using two NDFI images. Green is remaining forest; black is old deforestation as detected in the first NDFI image; red highlights new deforestation; orange shows forest disturbance by logging or fires; and cyan is vegetation regrowth of forest since the previous date

Code Checkpoint A34b. The book's repository contains a script that shows what your code should look like at this point.

Save your script for your own future use, as outlined in Chap. 1. Then, refresh the page to begin with a new script for the next section.

49.2.3 Section 3: Deforestation and Forest Degradation Time Series Analysis

To assess deforestation and forest degradation with a time series, we can use a Google Earth Engine tool called CODED (Bullock et al. 2018, 2020; Bullock (2018). The algorithm is based on previous work in continuous land cover monitoring (Zhu and Woodcock 2014) and NDFI-based degradation mapping using spectral unmixing models (Souza et al. 2003, 2005). CODED has both a user interface application and an API, which can be accessed in the Earth Engine Code Editor. For this lesson, we will use the API.

Code Checkpoint A34c. The book's repository contains a script to use to begin this section. You will need to start with that script and script and paste code below into it. The checkpoint accesses the modules of the CODED API and support functions. Note: importing large modules can cause your browser to hang for a moment while they load.

Detecting degraded forest regions requires knowledge of the characteristics of the forest of interest when it is in its normal healthy state. Higher NDFI values, near 1, typically indicate a healthy forest, but the magnitude range of NDFI for a healthy forest is dependent on forest density, type of forest ecosystem, and the seasonality of the area. In order to determine the typical values of NDFI observed in each forest over time, CODED employs a training period. Within this period, a regression model, similar to the one initially introduced in Chap. 18, is fitted for NDFI values for each pixel. The regression model is composed of a constant for overall NDFI magnitude, a sine and cosine term encapsulating seasonal and intra-annual variability, and an RMSE to account for noise. In this way, CODED can find typical temporal patterns in the landscape, account for clouds and sensor noise, and better distinguish forests from non-forested areas.

The code below sets up the study area, accesses the Landsat collection to be used, and defines the study period. For this example, we will use the geometry of the previous Landsat scene as our study area. Then, we define a new variable `studyArea` and assign it to the image we used when first exploring NDFI, retrieving its geometry using the `geometry` method. Then, we use the `utils` module to call the `Inputs.getLandsat` function, which accesses the `ImageCollection`. We then filter the images to a start date and end date of interest. Note: CODED calibrates to find the typical variations in NDFI for the region by observing the NDFI patterns over time, so it is a good idea to filter the Landsat data so you have an extra six months to a year of data before the time period in which you are truly interested. For example, if you wanted to study forest change from 2000 to 2010, it would be good practice to use 1999 as the start date and 2010 as the end date. Paste the code below into your starter script you opened to begin this section.

```
// We will use the geometry of the image from the previous
section as
// the study area.
var studyArea = ee.Image(
        'LANDSAT/LT05/C02/T1_L2/LT05_226068_19840411')
    .geometry();

// Get cloud masked (Fmask) Landsat imagery.
var landsat = utils.Inputs.getLandsat()
    .filterBounds(studyArea)
    .filterDate('1984-01-01', '2021-01-01');
```

Getting our `ImageCollection` this way saves us quite a bit of coding. By default the returned collection will use every available Landsat mission, perform some simple cloud masking, and generate our image fractions of green vegetation (GV), soil, non-photosynthetic vegetation (NPV), shade-covered material, and NDFI, as well as additional indices.

First, make a new variable `gfwImage` and add the path to the Global Forest Change product from Hansen et al. (2013), which is used to create a forest mask.

Define a threshold of 40 for the percent canopy cover for the mask. Apply the threshold to the `treecover2000` band from the `gfwImage`. You could also choose to use a pre-prepared forest mask of your own instead of selecting one from the Global Forest Change product. This might be useful if you have a local high-quality forest mask, as it may slightly improve your results or provide consistency with other work you have done in the area.

```
// Make a forest mask
var gfwImage =
ee.Image('UMD/hansen/global_forest_change_2019_v1_7');

// Get areas of forest cover above the threshold
var treeCover = 40;
var forestMask = gfwImage.select('treecover2000')
    .gte(treeCover)
    .rename('landcover');
```

To identify degradation and deforestation, distinguished by whether the land cover remains forest or not after the event, we will use a prepared dataset of forest and non-forested areas. This data already has the predictor data for each feature, which will speed up the computation. Alternatively, refer back to Chap. 6 on classification for details on how to create your own unique forest and non-forested area dataset and ensure each feature has a `year` property with the year collected as an integer.

```
var samples = ee.FeatureCollection(
    'projects/gee-book/assets/A3-
4/sample_with_pred_hansen_2010');
```

There are many parameters that can be adjusted when running CODED, as summarized below.

- `minObservations`: The minimum number of consecutive observations required to label a disturbance event.
- `chiSquareProbability`: The chi-squared probability is a threshold that controls the sensitivity to change.
- `training`: A training dataset used to specify forest and non-forest. In this example, we will set it to the combination of the *forest* and *non-forest* training points using the `merge` function.
- `forestValue`: The integer value of forest in your training data.
- `startYear`, `endYear`: The start and end years to perform change detection.
- `classBands`: The bands used to train the coefficients.

- Note: `prepTraining` tells the algorithm to add the coefficients to your samples for training the classifier. This will initiate a task to export the prepared samples for later use if you wish. In this example, we will set it to false.

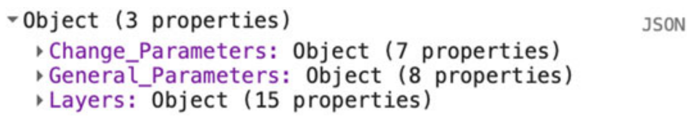
```
var minObservations = 4;
var chiSquareProbability = 0.97;
var training = samples;
var forestValue = 1;
var startYear = 1990;
var endYear = 2020;
var classBands = ['NDFI', 'GV', 'Shade', 'NPV', 'Soil'];
var prepTraining = false;
```

With the parameters defined, we now have everything needed to run CODED. CODED takes a single argument, which is stored as a JavaScript dictionary. Since we defined all the parameters as variables, it may seem redundant to put them into a dictionary, but having them as variables can be helpful when you are exploring functionality and adjusting parameter values frequently. Of course, entering the values directly into the dictionary would work as well.

Create a new dictionary and assign each parameter variable to a key of the same name.

```
//----------------- CODED parameters
var codedParams = {
    minObservations: minObservations,
    chiSquareProbability: chiSquareProbability,
    training: training,
    studyArea: studyArea,
    forestValue: forestValue,
    forestMask: forestMask,
    classBands: classBands,
    collection: landsat,
    startYear: startYear,
    endYear: endYear,
    prepTraining: prepTraining
};

// -------------- Run CODED
var results = api.ChangeDetection.coded(codedParams);
print(results);
```

Run CODED by clicking **Run**. This will set up the run, and issue the call to the `ChangeDetection.coded` function (Fig. 49.6).



```
▾Object (3 properties)                                 JSON
  ▸Change_Parameters: Object (7 properties)
  ▸General_Parameters: Object (8 properties)
  ▸Layers: Object (15 properties)
```

Fig. 49.6 The result of running the CODED change detection algorithm is an object with the general and change parameters used for the run and a layers object that has all the image outputs

Code Checkpoint A34d. The book's repository contains a script that shows what your code should look like at this point.

Next, you will relabel some of the results so that they are easier to understand and work with. You will rename the degradation layers to something more human readable, like 'degradation_1', 'degradation_2', etc. Rename the deforestation layers to something more human readable, like 'deforestation_1', 'deforestation_2', etc.

Set a variable for the mask layer. This is the same mask that was passed into CODED, so retrieving it in this method is not strictly necessary.

Set a variable for the change output that is the concatenation of the degradation and deforestation outputs. This is mostly for organizational purposes. Since change is more rare, self-masking removes all the non-change pixels, and casting to `Int32` helps keep all the bands in the same type, which you would need for exporting to a geoTIFF.

Set a variable `mag` to the minimum magnitude. There are multiple magnitude bands that correspond to the number of temporal segments that are retrieved when running CODED. These bands are reduced by the minimum since we want to find areas where the greatest negative change has occurred.

```
// Format the results for exporting.
var degradation = results.Layers.DatesOfDegradation
    .rename(['degradation_1', 'degradation_2',
        'degradation_3', 'degradation_4'
    ]);
var deforestation = results.Layers.DatesOfDeforestation
    .rename(['deforestation_1', 'deforestation_2',
        'deforestation_3', 'deforestation_4'
    ]);
var mask = results.Layers.mask.rename('mask');
var change = ee.Image.cat([degradation,
deforestation]).selfMask()
    .toInt32();
var mag = results.Layers.magnitude.reduce(ee.Reducer.min())
    .rename('magnitude');
```

Finally, we can combine all of this information into a stratified—or classified—output layer of forest, non-forest, degradation, and deforestation. We can define a function to take in each of the outputs and apply some logic to decide these categories. A new threshold we need to apply is the magnitude, `magThreshold`. This threshold will define the minimum amount of change we want to qualify as a degradation or deforestation event. This is a post-processing step in which we will stratify the results into categories of stable forest, stable non-forest, degradation, and deforestation.

Next, you will create a new function named `makeStrata` that takes an image and a threshold as the arguments. The base of the stratified image will be the mask which is remapped from [0, 1] to [2, 1]. This keeps the forest class at a value of 1 and updates the non-forest class to a value of 2.

Then, you will create a binary mask of the minimum threshold using the magnitude threshold parameter. Then, create a binary degradation image using all the degradation bands, and then multiply it by the magnitude mask. Similarly, you'll create a binary deforestation image using all the deforestation bands and then multiply it by the magnitude mask. Update the strata image using the `where` functions to first assign degradation to 3 and next to assign deforestation to 4. The `where` functions are applied sequentially. Deforestation needs to be updated last, because in this case, areas of degradation could overlap with areas that are also deforestation.

```
var makeStrata = function(img, magThreshold) {
    var strata = img.select('mask').remap([0, 1], [2, 1]);
    var mag = img.select('magnitude').lte(magThreshold);

    var deg =
img.select(['deg.*']).gt(0).reduce(ee.Reducer.max())
        .multiply(mag);
    var def =
img.select(['def.*']).gt(0).reduce(ee.Reducer.max())
        .multiply(mag);
    strata = strata.where(deg, 3).where(def, 4);

    return strata.clip(studyArea);
};
```

Concatenate the mask, change, and mag bands into a single image. Define a magnitude threshold of − 0.6. Apply the `makeStrata` function using the full output image and the magnitude threshold. Then, add code to export the strata to your assets.

```
var fullOutput = ee.Image.cat([mask, change, mag]);
var magnitudeThresh = -0.6;
var strata = makeStrata(ee.Image(fullOutput),
magnitudeThresh)
    .rename('strata');

Export.image.toAsset({
    image: strata,
    description: 'strata',
    region: studyArea,
    scale: 30,
    maxPixels: 1e13,
});
```

Click **Run**. The CODED algorithm is computationally heavy, so the results need to be exported before they can be viewed on the map in the Code Editor. Once the export has finished, you can add the layer to the map. We have created the asset for you and stored it in the book repository; you can access it with the code below:

```
var exportedStrata = ee.Image('projects/gee-book/assets/A3-
4/strata');
Map.addLayer(exportedStrata,
    {
        min: 1,
        max: 4,
        palette: 'green,black,yellow,red'
    },
    'strata');

Map.setCenter(-55.0828, -11.24, 11);
```

The resulting map should look something like the example in Fig. 49.7. With these stratified results, you can see the relative amounts and geographic distribution of forest that have been degraded or deforested in your time period of interest. What patterns do you observe? Does forest degradation or deforestation seem more prevalent?

Code Checkpoint A34e. The book's repository contains a script that shows what your code should look like at this point.

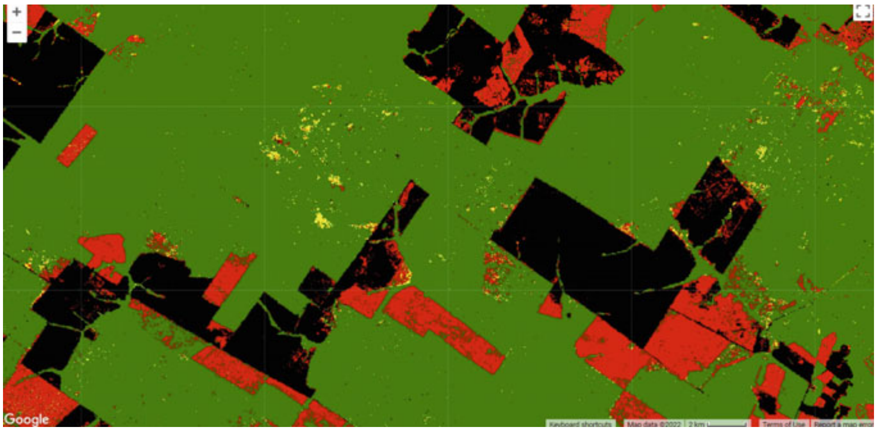


Fig. 49.7 Example of classified CODED results for 1990–2020. Non-forested areas are black, stable forest areas are green, deforestation is red, and degradation is yellow

49.3 Synthesis

Assignment 1. Study how adjusting each parameter affects your results.

a. When you decrease the chi-squared probability, do you see more or less deforestation, and more or less degradation?
b. When you decrease the magnitude threshold, do you see more or less deforestation, and more or less degradation?
c. When you decrease the number of observations, do you see more or less deforestation, and more or less degradation?

Assignment 2. Try adjusting the CODED parameters to see how they affect the detection of degradation and deforestation in a study area where you highly suspect from the imagery that these events have occurred. Observe in the map whether you are overestimating or underestimating the disturbed forest area with each parameter combination.

49.4 Conclusion

In this chapter, you learned about the spectral unmixing algorithm (SMA) and the Normalized Difference Fraction Index (NDFI) in order to map forest degradation and deforestation. We presented two examples of change detection applications: two-image differencing and an NDFI time-series approach using the CODED algorithm. NDFI is sensitive to subtle changes in forest composition, making it ideal for detection of forest degradation. CODED's use of a regression model fitting of

NDFI over a training period informs us more about what the NDFI magnitudes and variability of a forest should be in a healthy state, so changes from the norm can be more confidently labeled as forest disturbance events. However, your chosen CODED parameters have an important impact on your resulting map of forest loss and forest degradation.

References

Adams JB, Sabol DE, Kapos V et al (1995) Classification of multispectral images based on fractions of endmembers: application to land-cover change in the Brazilian Amazon. Remote Sens Environ 52:137–154. https://doi.org/10.1016/0034-4257(94)00098-8

Aryal RR, Wespestad C, Kennedy RE et al (2021) Lessons learned while implementing a time-series approach to forest canopy disturbance detection in Nepal. Remote Sens 13:2666. https://doi.org/10.3390/rs13142666

Asner GP, Keller M, Pereira R, Zweede JC (2002) Remote sensing of selective logging in Amazonia: assessing limitations based on detailed field observations, Landsat ETM+, and textural analysis. Remote Sens Environ 80:483–496. https://doi.org/10.1016/S0034-4257(01)00326-1

Asner GP, Keller M, Pereira R et al (2004) Canopy damage and recovery after selective logging in Amazonia: field and satellite studies. Ecol Appl 14:280–298. https://doi.org/10.1890/01-6019

Bullock E (2018) Background and motivation—CODED 0.2 documentation. https://coded.readthedocs.io/en/latest/background.html. Accessed 28 May 2021

Bullock E, Nolte C, Reboredo Segovia A (2018) Project impact assessment on deforestation and forest degradation: forest disturbance dataset, pp 1–44

Bullock EL, Woodcock CE, Souza C, Olofsson P (2020) Satellite-based estimates reveal widespread forest degradation in the Amazon. Glob Change Biol 26:2956–2969. https://doi.org/10.1111/gcb.15029

Cochrane MA (1998) Linear mixture model classification of burned forests in the Eastern Amazon. Int J Remote Sens 19:3433–3440. https://doi.org/10.1080/014311698214109

Hansen MC, Potapov PV, Moore R et al (2013) High-resolution global maps of 21st-century forest cover change. Science 342:850–853. https://doi.org/10.1126/science.1244693

Hirschmugl M, Steinegger M, Gallaun H, Schardt M (2013) Mapping forest degradation due to selective logging by means of time series analysis: case studies in Central Africa. Remote Sens 6:756–775. https://doi.org/10.3390/rs6010756

Kusbach A, Friedl M, Zouhar V et al (2017) Assessing forest classification in a landscape-level framework: an example from Central European forests. Forests 8:461. https://doi.org/10.3390/f8120461

Matricardi EAT, Skole DL, Cochrane MA et al (2007) Multi-temporal assessment of selective logging in the Brazilian Amazon using Landsat data. Int J Remote Sens 28:63–82. https://doi.org/10.1080/01431160600763014

Schultz M, Clevers JGPW, Carter S et al (2016) Performance of vegetation indices from Landsat time series in deforestation monitoring. Int J Appl Earth Obs Geoinf 52:318–327. https://doi.org/10.1016/j.jag.2016.06.020

Small C (2004) The Landsat ETM+ spectral mixing space. Remote Sens Environ 93:1–17. https://doi.org/10.1016/j.rse.2004.06.007

Souza CM Jr, Barreto P (2000) An alternative approach for detecting and monitoring selectively logged forests in the Amazon. Int J Remote Sens 21:173–179. https://doi.org/10.1080/014311600211064

Souza CM Jr, Firestone L, Silva LM, Roberts D (2003) Mapping forest degradation in the Eastern Amazon from SPOT 4 through spectral mixture models. Remote Sens Environ 87:494–506. https://doi.org/10.1016/j.rse.2002.08.002

Souza CM Jr, Roberts DA, Cochrane MA (2005) Combining spectral and spatial information to map canopy damage from selective logging and forest fires. Remote Sens Environ 98:329–343. https://doi.org/10.1016/j.rse.2005.07.013

Woodcock CE, Loveland TR, Herold M, Bauer ME (2020) Transitioning from change detection to monitoring with remote sensing: a paradigm shift. Remote Sens Environ 238:111558. https://doi.org/10.1016/j.rse.2019.111558

Zhu Z, Woodcock CE (2014) Continuous change detection and classification of land cover using all available Landsat data. Remote Sens Environ 144:152–171. https://doi.org/10.1016/j.rse.2014.01.011

50 Deforestation Viewed from Multiple Sensors

Xiaojing Tang

Overview
Combining data from multiple sensors is the best way to increase data density and hence detect change faster. The purpose of this chapter is to demonstrate a simple method of combining Landsat, Sentinel-2, and Sentinel-1 data for monitoring tropical forest disturbance. You will learn how to import, preprocess, and fuse optical and synthetic aperture radar (SAR) remote sensing data. You will also learn how to monitor change using time-series models.

Learning Outcomes

- Combining optical and SAR images for change detection.
- Fitting a time-series model to an `ImageCollection`.
- Using established time-series models to detect anomalies in new images.
- Performing change detection with a rolling monitoring window.
- Monitoring forest disturbance in near real time.

Helps if you know how to

- Understand the characteristics and preprocessing of Landsat and Sentinel images (Part I).
- Understand regressions in Earth Engine (Chap. 8).
- Work with array images (Chaps. 9 and 18).
- Understand the concept of spectral unmixing (Chap. 10).
- Write a function and `map` it over an `ImageCollection` (Chap. 12).

X. Tang (✉)
James Madison University, Harrisonburg, VA, USA
e-mail: tang3xx@jmu.edu

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_50

- Fit linear and nonlinear functions with regression in an `ImageCollection` time series (Chap. 18).
- Export and import results as Earth Engine assets (Chap. 22).

50.1 Introduction to Theory

Deforestation and forest degradation are large sources of carbon emissions and negatively impact biodiversity, food security, and human well-being. The ability to quickly and accurately detect forest disturbance events is essential for preventing future forest loss and mitigating the negative effects. Combining optical and radar data has the potential to achieve faster detection of forest disturbance than using an individual system. The main challenge is the methodological approach for fusing the different datasets. The Fusion Near Real-Time (FNRT) algorithm (Tang et al. 2023) is a monitoring algorithm for tropical forest disturbance that combines data from Landsat, Sentinel-2, and Sentinel-1. For each sensor system, the FNRT algorithm fits time-series models over data from a three-year training period prior to the one-year monitoring period. FNRT then produces a change score for each new observation collected during the monitoring period based on the residuals and the root-mean-square error (RMSE) of the time-series model. The change scores from all three different sensors are then combined to form one dense time series for change detection. In this chapter, we will learn how to run a simplified version of FNRT to monitor forest disturbance in 2020 for a test area located in the Brazilian Amazon.

50.2 Practicum

This lab is designed for advanced users of Earth Engine. We assume that you already know how to import and preprocess large quantities of Landsat, Sentinel-2, and Sentinel-1 data as image collections. The code for importing and preprocessing input data will be provided in the example script, but it will not be discussed in detail in this chapter. To learn more about FNRT, refer to Tang et al. (2023).

50.2.1 Section 1: Understand How FNRT Works

For this lab, we will use a graphical user interface to help us understand conceptually how FNRT combines data from different sensors and detects forest disturbance. Figure 50.1 shows the user interface of the app.

Code Checkpoint A35a. The book's repository contains information about accessing the app.

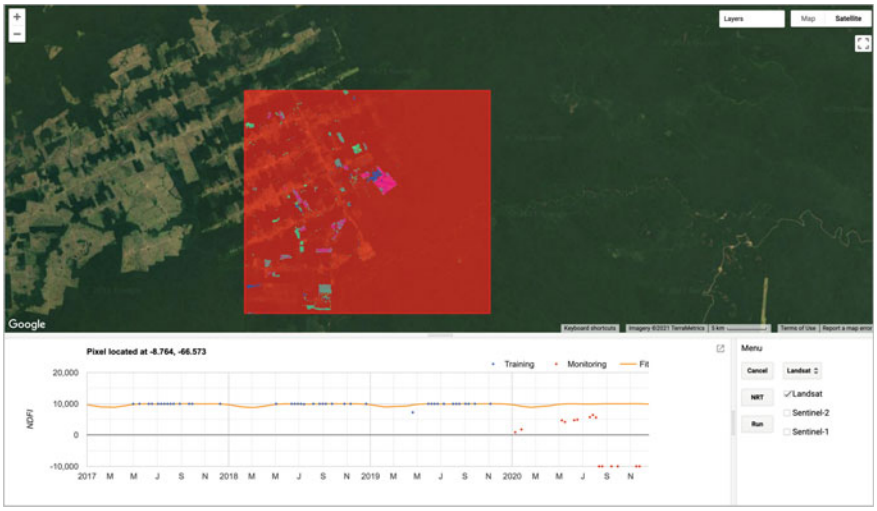


Fig. 50.1 User interface of FNRT app. The red box highlights a test site selected for demonstration of the algorithm. At the bottom left is a panel for showing plots of time-series data and model fit. The control menu is at the bottom right

There are three main tools that can be used to explore the FNRT algorithm. Note that only one plotting tool can be activated at a time, and each tool can be deactivated by clicking the button again:

1. The **Fit** button activates the "fit time-series model" tool, which allows users to click anywhere in the map area, load the time-series data, and fit a harmonic model for the pixel covering the clicked location, using data from the sensor selected in the widget just to the right of the **Fit** button.
2. The **Monitor** button activates the "near real-time monitoring" tool, which allows users to click anywhere in the map area and plot the results of FNRT for the pixel covering the clicked location, using data from all the sensors that were checked via the checkbox widgets to the right of the **Monitor** button.
3. The **Run** button runs FNRT for the entire test area using data from the sensors that are checked and loads the resulting forest disturbance map to the map area.

Let's start by looking at some examples of time series so we have a better understanding of the model. We first need to know where to look for changes. Let's try to have only "Landsat" checked, and click **Run**. This will run FNRT with only Landsat data and quickly produce a forest disturbance map for 2020 (Fig. 50.2a). Now, click **Fit** to activate the "Fit time-series model" tool, and then click on a pixel that shows forest disturbance activity according to the map. Wait a few seconds to let the app generate the time-series plot in the time-series panel. Once it is completed, you should see a plot similar to Fig. 50.2c.

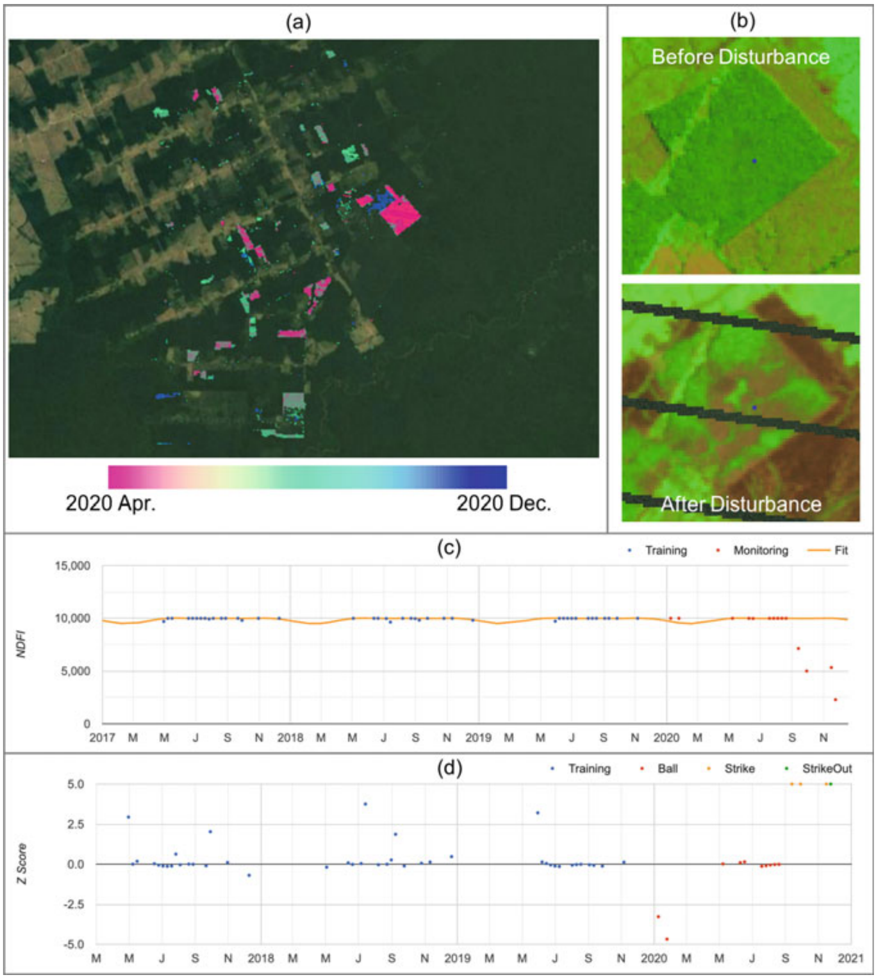


Fig. 50.2 **a** Forest disturbance map of 2020 for the test area created using Landsat data only; **b** Landsat images before and after the forest disturbance; **c** time series of Landsat NDFI and model fit; and **d** time series of change scores and change detection result

The model-fit plot (Fig. 50.2c) shows data from Landsat for the training period (blue) and monitoring period (red). It also shows a harmonic time-series model fit to the training data (orange line). The model fit extends into the monitoring period, which represents the model prediction. Notice that the satellite observations (red points) are very close to the model fit before the forest disturbance occurs. After the disturbance, the observations deviate away from the model fit. The algorithm monitors change by keeping track of differences between the observations and the model predictions.

Now click **Monitor** to activate the "near real-time monitoring" tool, and click the same pixel in the map. The app will then make a plot similar to (d) in Fig. 50.2. The near real-time monitoring plot (Fig. 50.2d) shows the change scores (*Z*-score) for the training and the monitoring period. A large change score indicates that something looks different in the satellite image. But sometimes that can be caused by a cloud or a cloud shadow that slipped through the masking process (e.g., the large change scores in the training period). Therefore, the monitoring algorithm looks for multiple large scores within a short monitoring window as an indicator of real forest disturbance.

The monitoring algorithm works through all the observations available during the monitoring period chronologically. Using baseball as an analogy, we flag each change score above a threshold as a "strike" and each change score below the threshold as a "ball." If the algorithm finds multiple strikes in a short monitoring window, a "strikeout" would then be called on the pixel and the date of the first strike would be recorded as the date of change (Fig. 50.2d).

Question 1. How many strikes were detected before a forest disturbance was confirmed (strikeout)?

When performing multi-sensor data fusion, FNRT would fit a separate time-series model and calculate change scores using data from each sensor (Fig. 50.3a–c). The change scores calculated using data from the three sensors are then combined, and change monitoring is applied to the combined scores (Fig. 50.4d).

The FNRT app also allows us to run change monitoring for the entire test area by clicking the **Run** button. A forest disturbance map showing the date of the disturbances is then added to the map. The users can use the checkbox widgets to choose which data stream is included in the process of making the map (e.g., Fig. 50.4). In general, optical data is more sensitive to forest disturbance than SAR data. Combining multiple data streams can help detect the disturbance earlier, compared to using data from each individual sensor.

Question 2. Can you find forest disturbances that were detected earlier through the use of data from all three sensors rather than using just Landsat data?

Now that we understand conceptually how FNRT works, let's create a simple script to implement it.

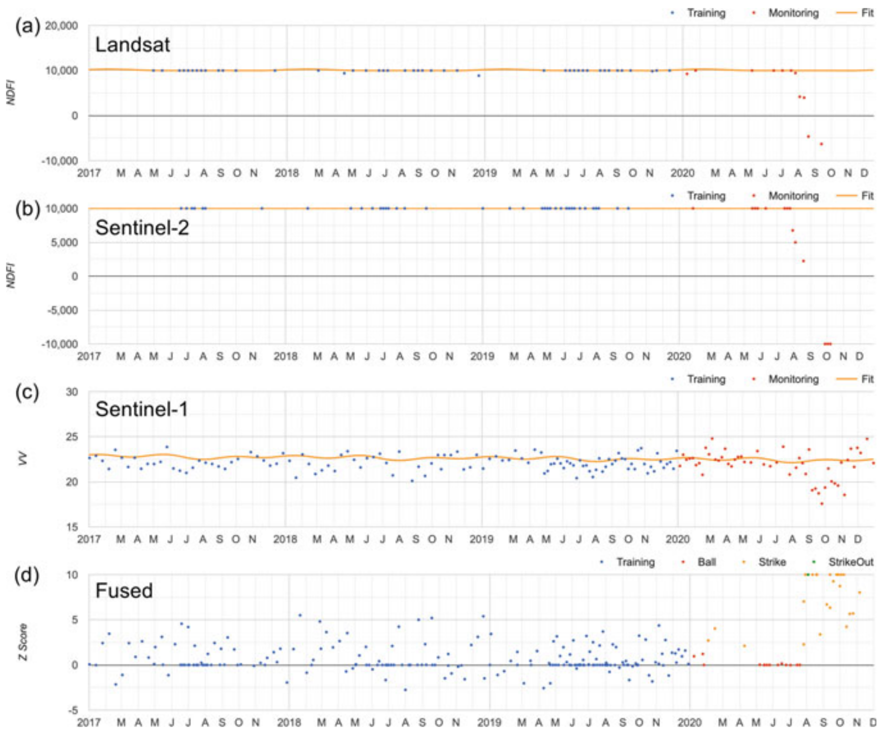


Fig. 50.3 Time-series data and model fit using **a** Landsat, **b** Sentinel-2, and **c** Sentinel-1 data; **d** shows the combined time series of change scores from all three sensors, and the change detection result

50.2.2 Section 2: Define Study Area and Model Parameters

50.2.2.1 Section 2.1: Study Area

There are several ways to define a study area: (1) import an existing `FeatureCollection` from an Earth Engine asset; (2) use the drawing tool of the Code Editor; and (3) create a polygon by entering the coordinates. Here, we select a box, approximately 20 km by 20 km, located in the Brazilian Amazon as our study area.

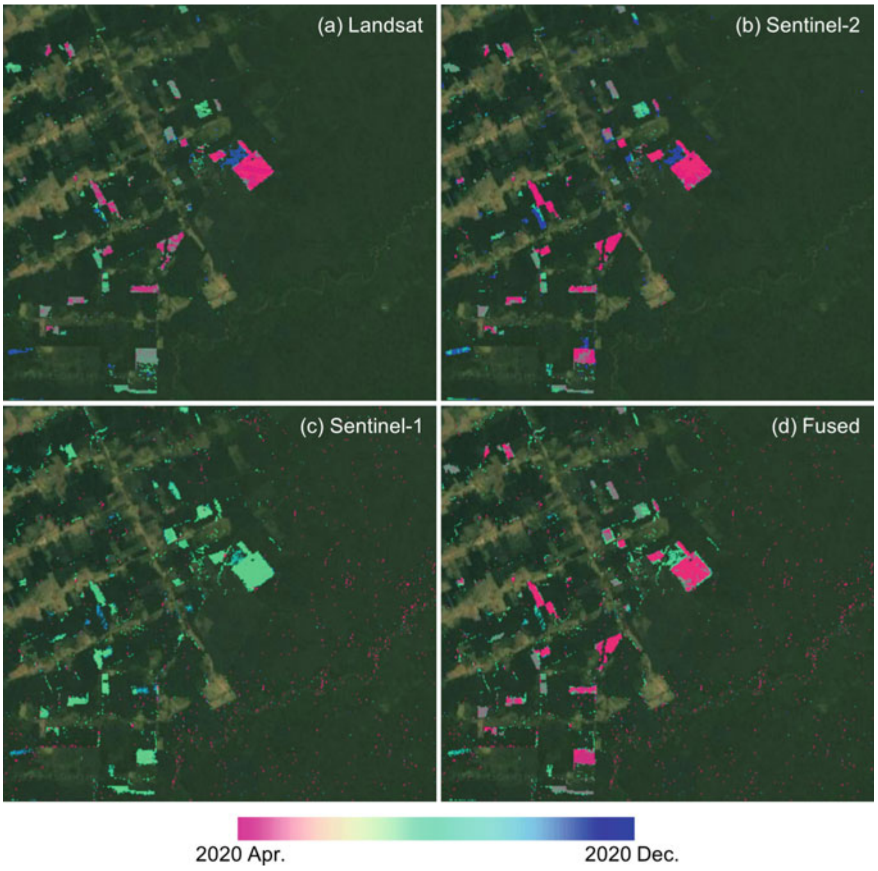


Fig. 50.4 Forest disturbance map made using data from **a** Landsat, **b** Sentinel-2, **c** Sentinel-1, and **d** all three sensors

```
var testArea = ee.Geometry.Polygon(
    [
        [
            [-66.73156878460787, -8.662236005089952],
            [-66.73156878460787, -8.916025640576244],
            [-66.44867083538912, -8.916025640576244],
            [-66.44867083538912, -8.662236005089952]
        ]
    ]);

Map.centerObject(testArea);
```

Note that because of the complexity of the FNRT algorithm, it requires quite a lot of computational resources and can result in a “User Memory Limit” error if applied to too large of an area. Divide the study area into multiple tiles if you are trying to monitor a large area.

50.2.2.2 Section 2.2: Model Parameters

A few model parameters need to be defined and can be tuned to optimize the algorithm for specific regions or monitoring conditions.

```
// Start and end of the training and monitoring period.
var trainPeriod = ee.Dictionary({
    'start': '2017-01-01',
    'end': '2020-01-01'
});
var monitorPeriod = ee.Dictionary({
    'start': '2020-01-01',
    'end': '2021-01-01'
});

// Near-real-time monitoring parameters.
var nrtParam = {
    z: 2,
    m: 5,
    n: 4
};

// Sensor specific parameters.
var lstParam = {
    band: 'NDFI',
    minRMSE: 0.05,
    strikeOnly: false
};
var s2Param = {
    band: 'NDFI',
    minRMSE: 0.05,
    strikeOnly: false
};
var s1Param = {
    band: 'VV',
    minRMSE: 0.01,
    strikeOnly: true
};
```

For demonstration of the algorithm, we define the year 2020 as our monitoring period (`monitorPeriod`). We then recommend using the three years prior to the monitoring period as the training period (`trainPeriod`) to establish baseline time-series models.

A few parameters can be used to control the sensitivity of the algorithm: a *z*-threshold (`z`) defines the minimum change score required to flag a possible change observation; an *m*-size (`m`) defines the size of the monitoring window, or how many consecutive observations we check for change; and an *n*-change (`n`) defines the number of possible change observations within the monitoring window required to confirm a forest disturbance.

For each sensor, we also need to select the band (`band`) to use for monitoring; a minimum RMSE (`minRMSE`) to prevent overly sensitive detection caused by unusually good model fit; and a strike-only flag (`strikeOnly`) to allow a data stream to contribute only to confirmation of a change.

Code Checkpoint A35b. The book's repository contains a script that shows what your code should look like at this point.

50.2.3 Section 3: Import and Preprocess Data

We need to import data from each sensor for the training period and the monitoring period as two separate image collections. Each image in the `ImageCollection` needs to be preprocessed. For optical data, we need to apply spectral unmixing and calculate the Normalized Difference Fraction Index (NDFI) (Souza et al. 2005). Since spectral unmixing would be applied to both Landsat and Sentinel-2 data, let's first implement a common "unmixing" function.

Several studies (e.g., Bullock et al. 2020; Chen et al. 2021) have shown that spectral unmixing and NDFI work very well in detecting tropical forest disturbance. Here, we use the same endmembers used in Souza et al. (2005) for the spectral unmixing and calculate NDFI based on Eqs. 4 and 5 in Souza et al. (2005).

```
var unmixing = function(col) {

    // Define endmembers and cloud fraction threshold.
    var gv = [500, 900, 400, 6100, 3000, 1000];
    var npv = [1400, 1700, 2200, 3000, 5500, 3000];
    var soil = [2000, 3000, 3400, 5800, 6000, 5800];
    var shade = [0, 0, 0, 0, 0, 0];
    var cloud = [9000, 9600, 8000, 7800, 7200, 6500];
    var cfThreshold = 0.05;

    return col.map(function(img) {
        // Select the spectral bands and perform unmixing
        var unmixed = img.select(['Blue', 'Green', 'Red',
                'NIR',
                'SWIR1', 'SWIR2'
            ])
            .unmix([gv, shade, npv, soil, cloud], true,
                true)
            .rename(['GV', 'Shade', 'NPV', 'Soil',
                'Cloud'
            ]);

        // Calculate Normalized Difference Fraction Index.+
        var NDFI = unmixed.expression(
            '10000 * ((GV / (1 - SHADE)) - (NPV + SOIL)) /
' +
            '((GV / (1 - SHADE)) + (NPV + SOIL))', {
                'GV': unmixed.select('GV'),
                'SHADE': unmixed.select('Shade'),
                'NPV': unmixed.select('NPV'),
                'SOIL': unmixed.select('Soil')
            }).rename('NDFI');

        // Mask cloudy pixel.
        var maskCloud = unmixed.select('Cloud').lt(
            cfThreshold);
        // Mask all shade pixel.
        var maskShade = unmixed.select('Shade').lt(1);
        // Mask pixel where NDFI cannot be calculated.
        var maskNDFI = unmixed.expression(
            '(GV / (1 - SHADE)) + (NPV + SOIL)', {
                'GV': unmixed.select('GV'),
                'SHADE': unmixed.select('Shade'),
                'NPV': unmixed.select('NPV'),
                'SOIL': unmixed.select('Soil')
            }).gt(0);
        // Scale fractions to 0-10000 and apply masks.
        return img
            .addBands(unmixed.select(['GV', 'Shade',
                    'NPV', 'Soil'
                ])
                .multiply(10000))
            .addBands(NDFI)
            .updateMask(maskCloud)
            .updateMask(maskNDFI)
            .updateMask(maskShade);
    });
};
```

For Landsat data, we need to load the Landsat 7 and Landsat 8 Surface Reflectance product as an `ImageCollection`. We can filter the collection based on our study area and time period. Each image in the `ImageCollection` is masked based on the QA band; then spectral unmixing is applied and NDFI calculated. The `loadLandsatData` function is provided for importing and preprocessing Landsat data.

Preprocessing of Sentinel-2 data is similar to that of Landsat data because they are both optical remote sensing data and are almost identical in many ways. We need to load the Sentinel-2 top-of-atmosphere product (surface reflectance data do not have long enough time series for our purpose) and the Sentinel-2 cloud probability product as image collections, and filter based on our study area and time period. We then join the two collections. Each image in the collection is then masked based on both the QA band and the cloud probability.

For Sentinel-1 data, we use the Sentinel-1 SAR GRD product. Preprocessing of Sentinel-1 data includes: (1) calculation of backscattering; (2) calculation of the ratio of VH and VV; (3) calculation of three-pixel spatial mean; (4) choice of the orbital pass with the most data for the region of interest; and (5) radiometric slope correction using volume model. Please refer to Mullissa et al. (2021) for more detail on the preprocessing of SAR data.

Here, we assume you have already learned how to load and preprocess large quantities of Landsat, Sentinel-2, and Sentinel-1 data. We will not discuss these steps in detail here in this lab. Instead, three functions (`loadLandsatData`, `loadS2Data`, and `loadS1Data`) are provided in a separate script with which we can import using the `require` function. Now, we can load all the required input data. We also need to apply the spectral unmixing and calculate NDFI for the Landsat and Sentinel-2 data.

```
var input = require(
    'projects/gee-edu/book:Part A - Applications/A3 -
Terrestrial Applications/A3.5 Deforestation Viewed from
Multiple Sensors/modules/Inputs'
);
var lstTraining = unmixing(input.loadLandsatData(testArea,
    trainPeriod));
var lstMonitoring =
unmixing(input.loadLandsatData(testArea,
    monitorPeriod));
var s2Training = unmixing(input.loadS2Data(testArea,
trainPeriod));
var s2Monitoring = unmixing(input.loadS2Data(testArea,
    monitorPeriod));
var s1Training = input.loadS1Data(testArea, trainPeriod);
var s1Monitoring = input.loadS1Data(testArea,
monitorPeriod);
```

In addition to the remote sensing data, we also need to create a forest mask to limit our monitoring to forested areas. Here, we will use the Hansen Global Forest Change dataset. We first use the "tree canopy cover" layer to find areas that had larger than 50% canopy cover in 2000. We then remove all the forest loss area during 2001–2019 and add in all the forest gain from 2000 to 2012 to create a mask of forested area by the beginning of our monitoring period.

```
var hansen =
ee.Image('UMD/hansen/global_forest_change_2020_v1_8')
    .unmask();
var forestMask = hansen.select('treecover2000')
    .gt(50)
    .add(hansen.select('gain'))
    .subtract(hansen.select('loss'))
    .add(hansen.select('lossyear')
        .eq(20))
    .gt(0)
    .clip(testArea);
```

Before we move on to the next section, let's do some simple checking. Print out the image collections and check whether the correct data is loaded. You can also load the forest mask and visually examine the quality of the mask.

```
var maskVis = {
    min: 0,
    max: 1,
    palette: ['blue', 'green']
};
Map.addLayer(forestMask, maskVis, 'Forest Mask');
print('lstTraining', lstTraining);
print('lstMonitoring', lstMonitoring);
print('s2Training', s2Training);
print('s2Monitoring', s2Monitoring);
print('s1Training', s1Training);
print('s1Monitoring', s1Monitoring);
```

Question 3. How many images are available from each sensor for our monitoring period? Are you surprised by the numbers? (Note that one sensor may have more data because the study area happened to be in the side-lap zone where two orbit passes overlap.) What bands are included in the data of each sensor?

Code Checkpoint A35c. The book's repository contains a script that shows what your code should look like at this point.

50.2.4 Section 4: Establish Baseline Time-Series Model

Before we can monitor change in near real time, we need to establish baseline time-series models using the data of the training period, so that we know what to expect from new observations in the monitoring period. Due to the different physical nature of optical and radar data, it is easier to fit a separate time-series model to data from each sensor. Here, we use a harmonic time-series model similar to the one used in the Continuous Change Detection and Classification (CCDC) algorithm (Zhu and Woodcock 2014). Because the timestamps of all the input images are stored in the unit of milliseconds while the harmonic model prefers a unit of years, we first need to define a function to convert any date object into the unit of fractional year (e.g., 2015-03-01 to 2015.1612).

```
var toFracYear = function(date) {
    var year = date.get('year');
    var fYear = date.difference(
        ee.Date.fromYMD(year, 1, 1), 'year');
    return year.add(fYear);
};
```

Here, we define a function (fitHarmonicMode) to fit a harmonic model to an ImageCollection of time-series data. We first construct the dependent variables according to Eq. 1 in Zhu and Woodcock (2014) and add them to each of the input images as new bands (using addDependents function). Then, we use a reducer (ee.Reducer.robustLinearRegression) to fit the model to the data using robust linear regression. The function returns the model coefficients and the model RMSE.

```
var fitHarmonicModel = function(col, band) {
    // Function to add dependent variables to an image.
    var addDependents = function(img) {
        // Transform time variable to fractional year.
        var t = ee.Number(toFracYear(
            ee.Date(img.get('system:time_start')), 1));
        var omega = 2.0 * Math.PI;
        // Construct dependent variables image.
        var dependents = ee.Image.constant([
                1, t,
                t.multiply(omega).cos(),
                t.multiply(omega).sin(),
                t.multiply(omega * 2).cos(),
                t.multiply(omega * 2).sin(),
                t.multiply(omega * 3).cos(),
                t.multiply(omega * 3).sin()
            ])
            .float()
            .rename(['INTP', 'SLP', 'COS', 'SIN',
                'COS2', 'SIN2', 'COS3', 'SIN3'
            ]);
        return img.addBands(dependents);
    };
    // Function to add dependent variable images to all
images.
    var prepareData = function(col, band) {
        return ee.ImageCollection(col.map(function(img) {
            return addDependents(img.select(band))
                .select(['INTP', 'SLP', 'COS',
                    'SIN',
                    'COS2', 'SIN2', 'COS3',
                    'SIN3',
                    band
                ])
                .updateMask(img.select(band)
                    .mask());
        }));
    };

    var col2 = prepareData(col, band);
    // Fit model to data using robust linear regression.
    var ccd = col2
        .reduce(ee.Reducer.robustLinearRegression(8, 1), 4)
        .rename([band + '_coefs', band + '_rmse']);

    // Return model coefficients and model rmse.
    return ccd.select(band + '_coefs').arrayTranspose()
        .addBands(ccd.select(band + '_rmse'));
};
```

With this function (`fitHarmonicModel`), we can fit the harmonic time-series model to the training data of all three sensors. It is always a good practice to also add some metadata to the results for future reference. The model fitting process will take a few minutes and quite a lot of computer memory. Therefore, it is recommended that we run them as tasks and save the results as assets before we proceed to the next step.

```
// Fit harmonic models to training data of all sensors.
var lstModel = fitHarmonicModel(lstTraining, lstParam.band)
    .set({
        region: 'test',
        sensor: 'Landsat'
    });
var s2Model = fitHarmonicModel(s2Training, s2Param.band)
    .set({
        region: 'test',
        sensor: 'Sentinel-2'
    });
var s1Model = fitHarmonicModel(s1Training, s2Param.band)
    .set({
        region: 'test',
        sensor: 'Sentinel-1'
    });

// Define function to save the results.
var saveModel = function(model, prefix) {
    Export.image.toAsset({
        image: model,
        scale: 30,
        assetId: prefix + '_CCD',
        description: 'Save_' + prefix + '_CCD',
        region: testArea,
        maxPixels: 1e13,
        pyramidingPolicy: {
            '.default': 'sample'
        }
    });
};

// Run the saving function.
saveModel(lstModel, 'LST');
saveModel(s2Model, 'S2');
saveModel(s1Model, 'S1');
```

Run the script. The model fitting results in export tasks to create assets. Instead of exporting the results for this exercise, however, you should proceed to the next section. In that section, we will continue with a precomputed asset that is the same as what the above tasks would create.

Code Checkpoint A35d. The book's repository contains a script that shows what your code should look like at this point.

50.2.5 Section 5: Create Predicted Values for the Monitoring Period

We will now start exploring the pre-exported results mentioned in the previous section. Place this code below the checkpoint of the previous section:

```
var models = ee.ImageCollection('projects/gee-
book/assets/A3-5/ccd');
var lstModel = models
    .filterMetadata('sensor', 'equals', 'Landsat').first();
var s2Model = models
    .filterMetadata('sensor', 'equals', 'Sentinel-
2').first();
var s1Model = models
    .filterMetadata('sensor', 'equals', 'Sentinel-
1').first();
```

In this section, we will use the time-series models that we produced in Sect. 50.2.4 to create a predicted image (also commonly referred to as a synthetic image; see Zhu et al. 2015) for any given date. The predicted image can give us a good estimate of the reflectance (or index such as NDFI) value of a place assuming no change has occurred. We can then compare the predicted image to the image collected by the satellite on the same date. Areas that look very different would likely be areas that have changed. Note that all the coefficients of the harmonic model were saved as a single-band array. So the very first step here is to define a function `dearrayModel` to convert the array image into a multiband image with each coefficient in one band.

```
var dearrayModel = function(model, band) {
    band = band + '_';

    // Function to extract a non-harmonic coefficients.
    var genCoefImg = function(model, band, coef) {
        var zeros = ee.Array(0).repeat(0, 1);
        var coefImg = model.select(band + coef)
            .arrayCat(zeros, 0).float()
            .arraySlice(0, 0, 1);
        return ee.Image(coefImg
            .arrayFlatten([
                [ee.String('S1_')
                    .cat(band).cat(coef)
                ]
            ]));
    };

    // Function to extract harmonic coefficients.
    var genHarmImg = function(model, band) {
        var harms = ['INTP', 'SLP', 'COS', 'SIN',
            'COS2', 'SIN2', 'COS3', 'SIN3'
        ];
        var zeros = ee.Image(ee.Array([
                ee.List.repeat(0, harms.length)
            ]))
            .arrayRepeat(0, 1);
        var coefImg = model.select(band + 'coefs')
            .arrayCat(zeros, 0).float()
            .arraySlice(0, 0, 1);
        return ee.Image(coefImg
            .arrayFlatten([
                [ee.String(band).cat('coef')], harms
            ]));
    };

    // Extract harmonic coefficients and rmse.
    var rmse = genCoefImg(model, band, 'rmse');
    var coef = genHarmImg(model, band);
    return ee.Image.cat(rmse, coef);
};
```

Second, we need to define a function that creates a predicted image for all the dates for which a real image is available during the monitoring period.

```
var createPredImg = function(modelImg, img, band, sensor) {
    // Reformat date.
    var date =
toFracYear(ee.Date(img.get('system:time_start')));
    var dateString = ee.Date(img.get('system:time_start'))
        .format('yyyyMMdd');
    // List of coefficients .
    var coefs = ['INTP', 'SLP', 'COS', 'SIN',
        'COS2', 'SIN2', 'COS3', 'SIN3'
    ];
    // Get coefficients images from model image.
    var coef = ee.Image(coefs.map(function(coef) {
        return modelImg.select(".*".concat(coef));
    })).rename(coefs);
    var t = ee.Number(date);
    var omega = 2.0 * Math.PI;
    // Construct dependent variables.
    var pred = ee.Image.constant([
            1, t,
            t.multiply(omega).cos(),
            t.multiply(omega).sin(),
            t.multiply(omega * 2).cos(),
            t.multiply(omega * 2).sin(),
            t.multiply(omega * 3).cos(),
            t.multiply(omega * 3).sin()
        ])
        .float();
    // Matrix multiply dependent variables with
coefficients.
    return pred.multiply(coef).reduce('sum')
        // Add original image and rename bands.
        .addBands(img, [band]).rename(['predicted', band])
        // Preserve some metadata.
        .set({
            'sensor': sensor,
            'system:time_start':
img.get('system:time_start'),
            'dateString': dateString
        });
};
```

The `createPredImg` function calculates a predicted image based on the date of the real image and the model coefficients. It returns a new image with both the predicted image and the real image.

We can then apply the `createPredImg` function to each image in the image collections of the monitoring data. We can define another function to `map` over an `ImageCollection` to apply `createPredImg` to all images in the collection.

```
var addPredicted = function(data, modelImg, band, sensor) {
    return ee.ImageCollection(data.map(function(img) {
        return createPredImg(modelImg, img, band,
            sensor);
    }));
};
```

Let's apply `addPredicted` to monitoring data of all three sensors. Note this process is still sensor-specific. Let's print out the result for Landsat to examine the structure of the output.

```
// Convert models to non-array images.
var lstModelImg = dearrayModel(lstModel, lstParam.band);
var s2ModelImg = dearrayModel(s2Model, s2Param.band);
var s1ModelImg = dearrayModel(s1Model, s1Param.band);

// Add predicted image to each real image.
var lstPredicted = addPredicted(lstMonitoring, lstModelImg,
    lstParam.band, 'Landsat');
var s2Predicted = addPredicted(s2Monitoring, s2ModelImg,
    s2Param.band, 'Sentinel-2');
var s1Predicted = addPredicted(s1Monitoring, s1ModelImg,
    s1Param.band, 'Sentinel-1');

print('lstPredicted', lstPredicted);
```

Question 4. How many images are in the `ImageCollection` of the predicted images? Is it the same number as the original `ImageCollection` of the monitoring data?

Code Checkpoint A35e. The book's repository contains a script that shows what your code should look like at this point.

50.2.6 Section 6: Calculate Change Scores

Assuming a good model fit, the predicted image would be very close to the real image of the same date for areas with no change. So what we need to do now is examine each pair of real and predicted images and look for areas where they differ from each other. And if we merge the three data streams, we would be able to significantly increase the data density and potentially detect changes faster. But before we can fuse the three data streams together, we need to produce a normalized measurement of the likelihood of change.

The FNRT algorithm calculates a change score (Tang et al. 2019) very similar to a *z*-score. First, it calculates the residuals defined as the difference between the predicted value and the observed value. The residuals are then scaled with the RMSE of the time-series model to get the change score. If a change score is above the preset threshold (`z` in the parameters), then it is flagged as "possible change" (or a "strike" in baseball terminology). The implementation of this part is quite simple:

```
// Function to calculate residuals.
var addResiduals = function(data, band) {
    return ee.ImageCollection(data.map(function(img) {
        return img.select('predicted')
            // Restrict predicted value to be under 10000
            .where(img.select('predicted').gt(10000),
                10000)
            // Calculate the residual
            .subtract(img.select(band))
            .rename('residual')
            // Save some metadata
            .set({
                'sensor': img.get('sensor'),
                'system:time_start': img.get(
                    'system:time_start'),
                'dateString': img.get(
                    'dateString')
            });
    }));
};

// Function to calculate change score and flag change.
var addChangeScores = function(data, rmse, minRMSE,
    threshold, strikeOnly) {
    // If strikeOnly then we need a mask for balls.
    var mask = ee.Image(0);
    if (strikeOnly) {
        mask = ee.Image(1);
    }
    return ee.ImageCollection(data.map(function(img) {
        // Calculate change score
        var z = img.divide(rmse.max(minRMSE));
        // Check if score is above threshold
        var strike = z.multiply(z.gt(threshold));
        // Create the output image.
        var zStack = ee.Image.cat(z, strike).rename([
                'z', 'strike'
            ])
            .set({
                'sensor': img.get('sensor'),
                'system:time_start': img.get(
                    'system:time_start')
            });
        // Mask balls if strikeOnly.
        return zStack.updateMask(strike.gt(0).or(
            mask));
    }));
};
```

Note that here, we restrict the predicted value to be under 10,000 because that is the theoretical maximum for NDFI.

We also need to have a minimum RMSE (`minRMSE`) when scaling the residuals. This is because when using indices such as NDFI, it is sometimes possible to have perfect model fits resulting in very low or even zero RMSE, which would then cause the model to be overly sensitive to subtle variations in the monitoring data. The `minRMSE` is calculated for each pixel as a percentage of the average values of all observations in the training period.

For this particular implementation of FNRT, the determination of a strike is directional, meaning that we are looking for changes only in one direction. This is because we already expect that a forest disturbance will cause a decrease in NDFI or VV backscattering. Modification would be required if monitoring in another direction (e.g., if using SWIR reflectance) or bidirectional monitoring is desired. Let’s apply the two new functions to our three data streams:

```
// Add residuals to collection of predicted images.
var lstResiduals = addResiduals(lstPredicted,
lstParam.band);
var s2Residuals = addResiduals(s2Predicted, s2Param.band);
var s1Residuals = addResiduals(s1Predicted, s1Param.band);

// Add change score to residuals.
var lstScores = addChangeScores(
    lstResiduals, lstModelImg.select('.*rmse'),
    lstPredicted.select(lstParam.band).mean()
    .abs().multiply(lstParam.minRMSE),
    nrtParam.z, lstParam.strikeOnly);
var s2Scores = addChangeScores(
    s2Residuals, s2ModelImg.select('.*rmse'),
    s2Predicted.select(s2Param.band).mean()
    .abs().multiply(s2Param.minRMSE),
    nrtParam.z, s2Param.strikeOnly);
var s1Scores = addChangeScores(
    s1Residuals, s1ModelImg.select('.*rmse'),
    s1Predicted.select(s1Param.band).mean()
    .abs().multiply(s1Param.minRMSE),
    nrtParam.z, s1Param.strikeOnly);

print('lstScores', lstScores);
```

Code Checkpoint A35f. The book's repository contains a script that shows what your code should look like at this point.

Question 5. What bands are included in the change scores images?

50.2.7 Section 7: Multisensor Data Fusion and Change Detection

Now, we are finally ready to fuse the three data streams together. This step is quite straightforward: we just need to `merge` the three image collections of the change scores, and then `sort` them by their timestamps.

```
var fused = lstScores.merge(s2Scores).merge(s1Scores)
    .sort('system:time_start');
```

The change monitoring function uses a binary array to keep track of the number of strikes within a monitoring window. The size of the monitoring window is defined by the model parameters (`nrtParam.m`). The default monitoring window size is five.

We first initiate the monitoring window with a binary array of 0 with a size of five bits (00000) for each pixel. When we iterate through all the change score images chronologically, we shift the binary array left for those pixels that were not masked in the current change score image (00000 → 0000). We then append the current change flag (strike or not) to the shifted binary array (0000 → 00001). The process continues, as we iterate through all the change scores images (e.g., 00001 → 00011 → 00110 → 01101 → 11011). Every time a new value is appended to the binary array, we also check how many strikes (1s) are there. If the number of strikes exceeds the required number (`nrtParam.n`) to flag a change, then a change is flagged and the date of the change recorded.

```
var monitorChange = function(changeScores, nrtParam) {
    // Initialize an empty image.
    var zeros = ee.Image(0).addBands(ee.Image(0))
        .rename(['change', 'date']);
    // Determine shift size based on size of monitoring
window.
    var shift = Math.pow(2, nrtParam.m - 1) - 1;
    // Function to monitor.
    var monitor = function(img, result) {
        // Retrieve change image at last step.
        var change = ee.Image(result).select('change');
        // Retrieve change date image at last step.
        var date = ee.Image(result).select('date');
        // Create a shift image to shift the change binary
array
        // left for one space so that new one can be
appended.
        var shiftImg = img.select('z').mask().eq(0)
            .multiply(shift + 1).add(shift);
        change = change.bitwiseAnd(shiftImg)
            .multiply(shiftImg.eq(shift).add(1))
            .add(img.select('strike').unmask().gt(0));
        // Check if there are enough strike in the current
        // monitoring window to flag a change.
        date = date.add(change.bitCount().gte(nrtParam.n)
            // Ignore pixels where change already detected.
            .multiply(date.eq(0))
            // Record change date where change is flagged.
            .multiply(ee.Number(toFracYear(
                ee.Date(img.get(
                    'system:time_start')), 1))));
        // Combine change and date layer for next
iteration.
        return (change.addBands(date));
    };

    // Iterate through the time series and look for change.
    return ee.Image(changeScores.iterate(monitor, zeros))
        // Select change date layer and selfmask.
        .select('date').rename('Alerts').selfMask();
};
```

Now, we apply the monitoring function (`monitorChange`) to the fused data (`fused`) using the default model parameters (`nrtParam`). We can then add the result, the forest disturbance map, to the map panel to visually examine it.

```
var alerts = monitorChange(fused,
nrtParam).updateMask(forestMask);
print('alerts', alerts);

// Define a visualization parameter.
var altVisParam = {
    min: 2020.4,
    max: 2021,
    palette: ['FF0080', 'EC1280', 'DA2480', 'C83680',
'B64880',
        'A35B80', '916D80', '7F7F80', '6D9180', '5BA380',
        '48B680', '36C880', '24DA80', '12EC80', '00FF80',
        '00EB89', '00D793', '00C49D', '00B0A7', '009CB0',
        '0089BA', '0075C4', '0062CE', '004ED7', '003AE1',
        '0027EB', '0013F5', '0000FF'
    ]
};
Map.centerObject(testArea, 10);
Map.addLayer(alerts, altVisParam, 'Forest Disturbance Map
(2020)');
Map.setOptions('SATELLITE');
```

Code Checkpoint A35g. The book's repository contains a script that shows what your code should look like at this point.

If we run the whole script now, we can create a map of forest disturbance that occurred in 2020 for our test area. The visualization of the map shows us the timing of all the forest disturbance detected by the FNRT algorithm (Fig. 50.5). You will notice a few large disturbance events. There may also be some tiny speckles, which are likely errors of commission. Try adjusting the monitoring parameters (`nrtParam`) to see if you can improve the map. Another common practice is to apply a spatial filter to filter out scattered single pixels of change (Bullock et al. 2020).

Question 6. Can you run the change detection without Sentinel-1 input or with just Landsat input? Was forest disturbance detected faster when we used data from all three sensors?

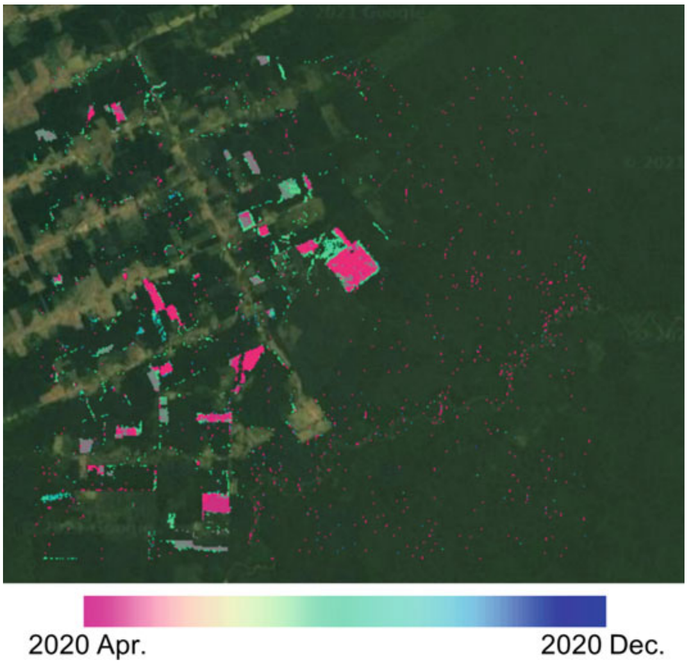


Fig. 50.5 Map of forest disturbance occurring in 2020 for a selected test area located on the frontline of deforestation in the Brazilian Amazon

50.3 Synthesis

Now that you have learned how to implement the simplified version of FNRT for the test area that we selected, let's try to use it to do some real monitoring of tropical forest disturbance.

Assignment 1. Modify the code and apply it in a different area of your choice. Note that you will need to keep the area small in order to be able to run everything easily in the Code Editor.

Assignment 2. Try to monitor more recent changes by changing the monitoring period to 2021. Note you would need to change the training period to 2018–2020, and also update the forest mask.

Assignment 3. If you want to try a more challenging task, you can integrate your results into the FNRT app that we used in Sect. 50.2.1.

50.4 Conclusion

In this chapter, we experimented with a simple way to combine data from three different sensors (Landsat, Sentinel-2, and Sentinel-1) for the purpose of detecting tropical forest disturbance as early as possible. We started with loading and preprocessing the input data. We learned how to fit a harmonic model to time-series data, and how to make predicted observations based on a model fit. We also learned how to look for changes with a moving monitoring window over a time series of change scores. The full FNRT algorithm is more complex than the "lite" version that we implemented in this lab. If interested, please refer to Tang et al. (2023) for more details.

References

Bullock EL, Woodcock CE, Olofsson P (2020) Monitoring tropical forest degradation using spectral unmixing and Landsat time series analysis. Remote Sens Environ 238:110968. https://doi.org/10.1016/j.rse.2018.11.011

Chen S, Woodcock CE, Bullock EL et al (2021) Monitoring temperate forest degradation on Google Earth Engine using Landsat time series analysis. Remote Sens Environ 265:112648. https://doi.org/10.1016/j.rse.2021.112648

Mullissa A, Vollrath A, Odongo-Braun C et al (2021) Sentinel-1 SAR backscatter analysis ready data preparation in Google Earth Engine. Remote Sens 13:1954. https://doi.org/10.3390/rs13101954

Souza CM Jr, Roberts DA, Cochrane MA (2005) Combining spectral and spatial information to map canopy damage from selective logging and forest fires. Remote Sens Environ 98:329–343. https://doi.org/10.1016/j.rse.2005.07.013

Tang X, Bullock EL, Olofsson P et al (2019) Near real-time monitoring of tropical forest disturbance: new algorithms and assessment framework. Remote Sens Environ 224:202–218. https://doi.org/10.1016/j.rse.2019.02.003

Tang X, Bratley KH, Cho K et al (2023) Near real-time monitoring of tropical forest disturbance by fusion of Landsat, Sentinel-2, and Sentinel-1 data. Remote Sens Environ, 294:113626. https://doi.org/10.1016/j.rse.2023.113626

Zhu Z, Woodcock CE (2014) Continuous change detection and classification of land cover using all available Landsat data. Remote Sens Environ 144:152–171. https://doi.org/10.1016/j.rse.2014.01.011

Zhu Z, Woodcock CE, Holden C, Yang Z (2015) Generating synthetic Landsat images based on all available Landsat data: predicting Landsat surface reflectance at any given time. Remote Sens Environ 162:67–83. https://doi.org/10.1016/j.rse.2015.02.009

Working with GPS and Weather Data

51

Peder Engelstad, Daniel Carver, and Nicholas E. Young

Overview

The purpose of this chapter is to demonstrate how to use Google Earth Engine as a means of associating remotely sensed data (weather observations) with open-source GPS point locations. These methods will provide a quick and easy way to access and analyze large amounts of information relative to your own research and efficiently move your data outside Earth Engine.

Learning Outcomes

- Pairing values from remotely sensed data with uploaded data.
- Exporting features from Earth Engine.

Helps if you know how to

- Import images and image collections, filter, and visualize (Part I).
- Exporting calculated data to tables with tasks (Chap. 22).

P. Engelstad (✉) · N. E. Young
Natural Resource Ecology Laboratory, Colorado State University, Fort Collins, CO, USA
e-mail: peder.engelstad@colostate.edu

N. E. Young
e-mail: nicholas.young@colostate.edu

D. Carver
Geospatial Centroid, Colorado State University, Fort Collins, CO, USA
e-mail: carverd@colostate.edu

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_51

51.1 Introduction to Theory

Knowing how animals interact with their environment is critical to understanding how to manage them. The choices animals make are influenced by basic survival needs (e.g., food, shelter, water) and dynamic factors such as local weather conditions. Without direct observation, it can be difficult to understand the relationship between animal movement and weather conditions.

In this chapter, we will explore information retrieved from a GPS collar worn by a cougar and explore its relationship to daily temperature estimates from the Daymet climatological dataset available in Google Earth Engine. This will require us to bring an asset into Earth Engine, connect the weather values to the point locations, and bring those value-added data back out of Earth Engine for further analysis.

51.2 Practicum

51.2.1 Section 1: GPS Location Data

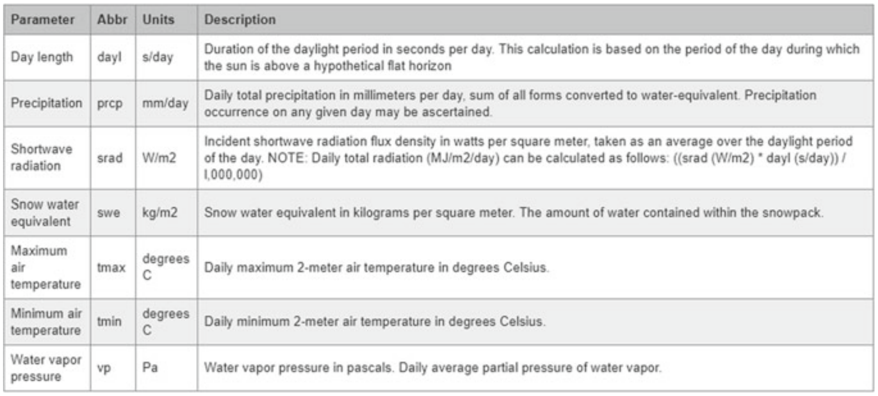
Using GPS collars, Mahoney and Young (2017) tracked the movement of 2 cougars and 16 coyotes in Central Utah. These data were used to understand some of the behavioral patterns of the individuals. These data have been freely shared with the broader research community and the public through Movebank, an online repository for animal movement datasets from across the globe. While some Movebank datasets list only the contact information of the authors, others (like those from the Mahoney study) allow you to display and download the information via an interactive web map.

We will pair the data on cougar movement with Daymet weather data. According to the Daymet website, the dataset "provides gridded estimates of daily weather parameters. Seven surface weather parameters are available at a daily time step, 1 km × 1 km spatial resolution, with a North American spatial extent allowing a rich resource of daily surface meteorology."

With data for every day at a 1 km^2 spatial resolution, the Daymet data are a great resource for the temporal and spatial scales at which a cougar would interact with the landscape. There are seven measured values in total, allowing us to check multiple aspects of the weather to assess how it may be affecting behavior (Fig. 51.1).

51.2.2 Section 2: Bringing Data into Earth Engine

In this chapter, we will discuss how to import assets into Earth Engine, extract values from a dataset, and export those values out of Earth Engine. The processes by which you can bring data into Earth Engine change frequently, so it is best to go directly to the documentation to view the latest updates.



Parameter	Abbr	Units	Description
Day length	dayl	s/day	Duration of the daylight period in seconds per day. This calculation is based on the period of the day during which the sun is above a hypothetical flat horizon
Precipitation	prcp	mm/day	Daily total precipitation in millimeters per day, sum of all forms converted to water-equivalent. Precipitation occurrence on any given day may be ascertained.
Shortwave radiation	srad	W/m2	Incident shortwave radiation flux density in watts per square meter, taken as an average over the daylight period of the day. NOTE: Daily total radiation (MJ/m2/day) can be calculated as follows: ((srad (W/m2) * dayl (s/day)) / l,000,000)
Snow water equivalent	swe	kg/m2	Snow water equivalent in kilograms per square meter. The amount of water contained within the snowpack.
Maximum air temperature	tmax	degrees C	Daily maximum 2-meter air temperature in degrees Celsius.
Minimum air temperature	tmin	degrees C	Daily minimum 2-meter air temperature in degrees Celsius.
Water vapor pressure	vp	Pa	Water vapor pressure in pascals. Daily average partial pressure of water vapor.

Fig. 51.1 Metadata for Daymet imagery within Earth Engine

51.2.2.1 Section 2.1: Bringing in an Asset

Using the following code, start a fresh script and import Mahoney's cougar movement data from Movebank (which has already been uploaded for you in the assets of this book). Here, we will focus on a single animal (cougar ID F53). Because the original dataset was in CSV format, it has been converted into a shapefile outside of GEE. During this process, it is important to note that data have been projected to the WGS 1984 (EPSG: 4326) coordinate reference system. Projecting to EPSG: 4326 is recommended in Earth Engine to minimize the reprojection errors during the uploading process.

```
// Import the data and add it to the map and print.
var cougarF53 = ee.FeatureCollection(
    'projects/gee-book/assets/A3-6/cougarF53');

Map.centerObject(cougarF53, 10);

Map.addLayer(cougarF53, {}, 'cougar presence data');

print(cougarF53, 'cougar data');
```

You can use the **Inspector** tool to look at the attribute data associated with the new asset. With the points visualized, make a geometry feature that encompasses our area of interest by selecting the square geometry tool and drawing a box that encompasses the points (Fig. 51.2). We will use the geometry feature to filter our climate data.

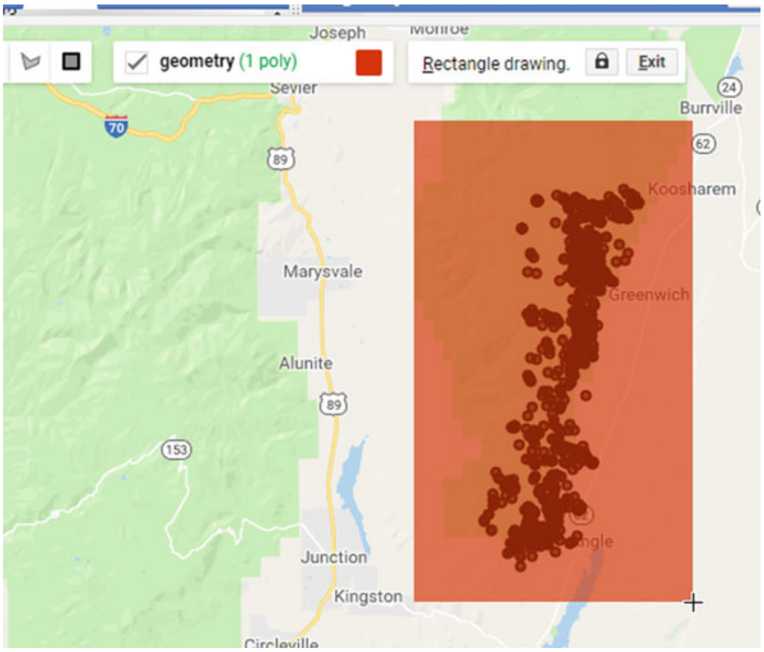


Fig. 51.2 Draw a geometry feature around the points to spatially filter the climate data

51.2.2.2 Section 2.2: Defining Weather Variables

In this chapter, we are using Earth Engine as a means of associating remotely sensed data (i.e., our rasters) with our point locations. While the process is conceptually straightforward, it does take some work to accomplish. With our points loaded, the next step is to import the Daymet weather variables.

We are using the NASA-derived dataset Daymet V4 due to its 1 km^2 spatial resolution and the fact that it measures the environmental variables we think might be related to cougar movement or behavior. We will import these variables by calling the unique ID of the dataset, filtering it to our bounding box geometry and the dates during which the data were collected.

```
// Call in image collection and filter.
var Daymet = ee.ImageCollection('NASA/ORNL/DAYMET_V4')
    .filterDate('2014-02-11', '2014-11-02')
    .filterBounds(geometry)
    .map(function(image) {
        return image.clip(geometry);
    });

print(Daymet, 'Daymet');
```

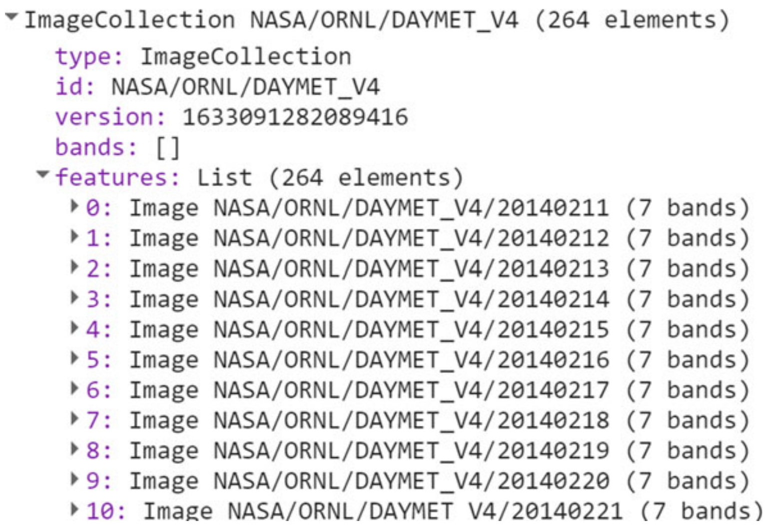

```
▾ImageCollection NASA/ORNL/DAYMET_V4 (264 elements)
    type: ImageCollection
    id: NASA/ORNL/DAYMET_V4
    version: 1633091282089416
    bands: []
   ▾features: List (264 elements)
      ▸0: Image NASA/ORNL/DAYMET_V4/20140211 (7 bands)
      ▸1: Image NASA/ORNL/DAYMET_V4/20140212 (7 bands)
      ▸2: Image NASA/ORNL/DAYMET_V4/20140213 (7 bands)
      ▸3: Image NASA/ORNL/DAYMET_V4/20140214 (7 bands)
      ▸4: Image NASA/ORNL/DAYMET_V4/20140215 (7 bands)
      ▸5: Image NASA/ORNL/DAYMET_V4/20140216 (7 bands)
      ▸6: Image NASA/ORNL/DAYMET_V4/20140217 (7 bands)
      ▸7: Image NASA/ORNL/DAYMET_V4/20140218 (7 bands)
      ▸8: Image NASA/ORNL/DAYMET_V4/20140219 (7 bands)
      ▸9: Image NASA/ORNL/DAYMET_V4/20140220 (7 bands)
      ▸10: Image NASA/ORNL/DAYMET_V4/20140221 (7 bands)
```

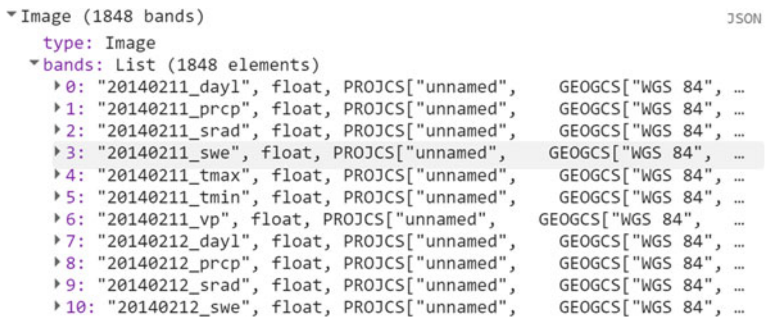
Fig. 51.3 View of the structure of the Daymet V4 data from the print statement

From the print statement (Fig. 51.3), we can see that this is an `ImageCollection` with 264 images (though your total number of images may be different, as the dataset changes over time). Each image has seven bands relating to specific weather measurements (see Fig. 51.1). Now that both datasets are loaded, we will associate the cougar occurrences data with the weather data.

51.2.2.3 Section 2.3: Extracting Values

With our points and imagery loaded, we can call a function to extract values from the underlying raster based on the known locations of the cougar. We will do this using the `ee.Image.sampleRegions` function. Search for the `ee.Image.sampleRegions` function under the **Docs** tab to familiarize yourself with the parameters it requires.

If we tried to call this function on the Daymet `ImageCollection`, we would get an error, because `ee.Image.sampleRegions` is a function of an image. To get around this, we will convert the Daymet `ImageCollection` into a multiband image (Fig. 51.4). Each of the seven measurements for each day will become a specific band in our multiband image. This process will help us in the end, because each band is defined by the date it was collected and the variable it shows. We can use this information to determine which data connects to the positions of the cougar on a specific day.



```
▾Image (1848 bands)                                                      JSON
   type: Image
  ▾bands: List (1848 elements)
    ▸0: "20140211_dayl", float, PROJCS["unnamed",    GEOGCS["WGS 84", …
    ▸1: "20140211_prcp", float, PROJCS["unnamed",    GEOGCS["WGS 84", …
    ▸2: "20140211_srad", float, PROJCS["unnamed",    GEOGCS["WGS 84", …
    ▸3: "20140211_swe", float, PROJCS["unnamed",    GEOGCS["WGS 84",  …
    ▸4: "20140211_tmax", float, PROJCS["unnamed",    GEOGCS["WGS 84", …
    ▸5: "20140211_tmin", float, PROJCS["unnamed",    GEOGCS["WGS 84", …
    ▸6: "20140211_vp", float, PROJCS["unnamed",    GEOGCS["WGS 84",   …
    ▸7: "20140212_dayl", float, PROJCS["unnamed",    GEOGCS["WGS 84", …
    ▸8: "20140212_prcp", float, PROJCS["unnamed",    GEOGCS["WGS 84", …
    ▸9: "20140212_srad", float, PROJCS["unnamed",    GEOGCS["WGS 84", …
    ▸10: "20140212_swe", float, PROJCS["unnamed",    GEOGCS["WGS 84", …
```

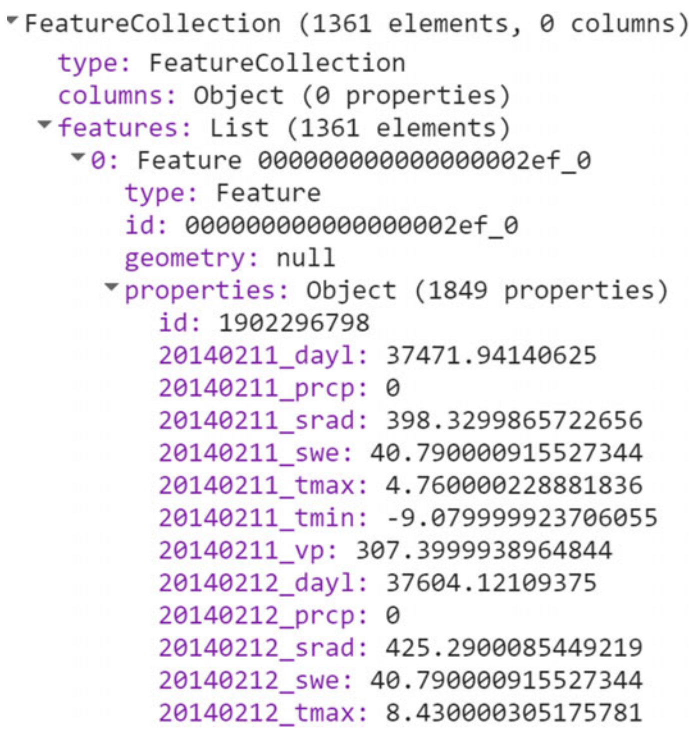
Fig. 51.4 Print statement showing the result of converting the Daymet `ImageCollection` into a multiband image

It is important to note that, with many images in the `ImageCollection`, we are going to create a single image with a large number of bands. Because Earth Engine is good at data manipulations, it can handle this type of request.

```
// Convert to a multiband image.
var DaymetImage = Daymet.toBands();

print(DaymetImage, 'DaymetImage');
```

Now that we have a single multiband image, we can use the `sampleRegions` function to extract information across multiple dates at each of our GPS point locations (Fig. 51.5). There are three parameters of this function that should be considered:

- **Collection**: This is the vector dataset that the sampled data will be associated with.
- **Properties**: This defines which columns of the vector dataset will remain. In this case, we want to keep the ID column, because that is what we will use to join this dataset back to the original outside of Earth Engine.
- **Scale**: This refers to the spatial resolution of the dataset. The scale parameter should always match the spatial resolution of your raster data. If you are not sure what the resolution of a raster is, you can search for the dataset using the search bar and locate that information in the documentation. (If you want to get the number for use in code, the `nominalScale` function, as described in Chap. 4, can extract it from an image.)



```
▾FeatureCollection (1361 elements, 0 columns)
   type: FeatureCollection
   columns: Object (0 properties)
  ▾features: List (1361 elements)
    ▾0: Feature 00000000000000002ef_0
       type: Feature
       id: 00000000000000002ef_0
       geometry: null
      ▾properties: Object (1849 properties)
         id: 1902296798
         20140211_dayl: 37471.94140625
         20140211_prcp: 0
         20140211_srad: 398.3299865722656
         20140211_swe: 40.790000915527344
         20140211_tmax: 4.760000228881836
         20140211_tmin: -9.079999923706055
         20140211_vp: 307.3999938964844
         20140212_dayl: 37604.12109375
         20140212_prcp: 0
         20140212_srad: 425.2900085449219
         20140212_swe: 40.790000915527344
         20140212_tmax: 8.430000305175781
```

Fig. 51.5 Print statement from the `sampleRegions` function showing that our GPS point locations now have weather measurements associated with them

```
// Call the sample regions function.
var samples = DaymetImage.sampleRegions({
    collection: cougarF53,
    properties: ['id'],
    scale: 1000
});

print(samples, 'samples');
```

Code Checkpoint A36a. The book's repository contains a script that shows what your code should look like at this point.

51.2.2.4 Section 2.4: Exporting

Exporting Points

We now have a series of daily weather data associated with the known locations of the cougar known as F53. While we could work more with these data in Earth

Engine, it will be easy to bring them into R, Python, or Excel for further analysis. There are a few options to define where the exported data will be created. Generally speaking, saving the data to a Google Drive account provides an easy way to access the data with another program. Here, we will use a dictionary, denoted by the curly brackets `{}`, to define the parameters of the `Export.table.toDrive` function.

We would have preferred to export to create a shapefile, but a shapefile can contain only 255 columns. The `samples` variable has 1361 columns, so we will export the data as a CSV file.

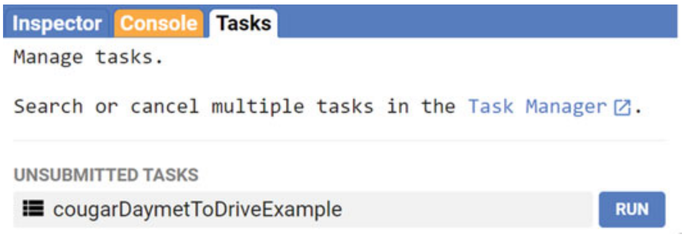
```
// Export value added data to your Google Drive.
Export.table.toDrive({
    collection: samples,
    description: 'cougarDaymetToDriveExample',
    fileFormat: 'csv'
});
```

When you export something, the **Tasks** panel will light up. To run the task, click the **Run** button (Fig. 51.6).

When you click the **Run** button, the pop-up shown in Fig. 51.7 will appear. This allows you to edit the details of the export.

Exporting a Raster

While working with these spatial data, you may have realized that a raster showing the median values over the time period when data were collected on the cougar could be useful information to have. To do this, we will apply a `median` reducer function to the Daymet `ImageCollection` to generate a median value for each parameter in each cell. As with the tabular data, we will export this multiband image to Google Drive. Once we convert the `ImageCollection` to an image using the `median` function, we can clip it to the geometry feature object. This feature will be exported as a multiband raster.



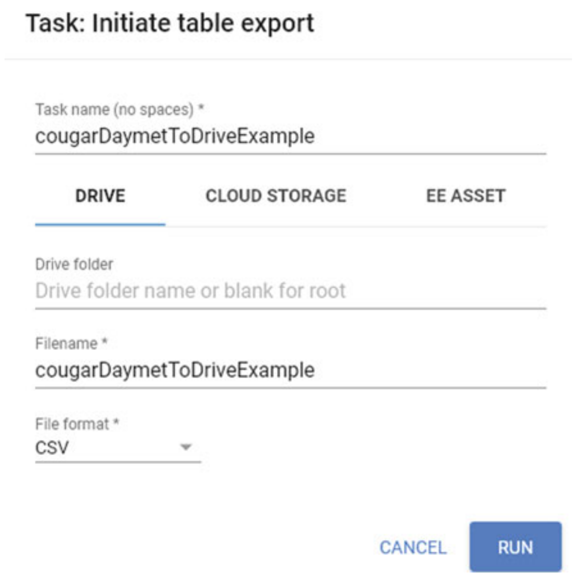
Manage tasks.

Search or cancel multiple tasks in the Task Manager.

UNSUBMITTED TASKS

cougarDaymetToDriveExample

Fig. 51.6 Example of the **Tasks** bar once a script with an export function has been run


Task: Initiate table export

Task name (no spaces) *
cougarDaymetToDriveExample

Drive folder
Drive folder name or blank for root

Filename *
cougarDaymetToDriveExample

File format *
CSV

Fig. 51.7 Example of the user-defined parameters present when exporting a feature from Earth Engine

```
// Apply a median reducer to the dataset.
var daymet1 = Daymet
    .median()
    .clip(geometry);

print(daymet1);

// Export the image to drive.
Export.image.toDrive({
    image: daymet1,
    description: 'MedianValueForStudyArea',
    scale: 1000,
    region: geometry,
    maxPixels: 1e9
});
```

In Earth Engine, there are many options for exporting images. One of the most important options when exporting data is the `maxPixels` parameter. Generally speaking, Earth Engine will not allow you to export a raster with more than 10^9 pixels. With the `maxPixels` parameter, you can bump this up to around 10^{12} pixels per image. If you are exporting data for an area larger than 10^{12} pixels, you will need to be creative about how to get the information out of Earth Engine.

Sometimes, this involves splitting the image into smaller pieces (as described in Chap. 29) or even reevaluating the usefulness of such a large image outside of Earth Engine.

Code Checkpoint A36b. The book's repository contains a script that shows what your code should look like at this point.

Question 1. With the `maxPixels` count set to 1e9, what square area would you expect to be able to export for imagery with 1000, 250, 30, 10 m cell size? Determine how this area measure changes when using 10^{12} pixels for one of the cell sizes to determine the percent increase.

Question 2. We selected all available variables from the Daymet collection. What specific bands from Daymet might be most relevant to cougars? How would you edit the code to select just one or a few bands?

Question 3. When we extracted the raster data, we set the scale parameter to 1000 to match the known spatial resolution of the Daymet dataset. Would it be reasonable to change that scale parameter to 500 to get final resolution data? What does Earth Engine tell you when you attempt to change the scale of an export?

51.3 Synthesis

Assignment 1. Utilize the chapter's script with a different dataset to process tracking information from a different animal from Movebank or another occurrence database of your choice (e.g., GBIF.org). Evaluate whether the temperature ranges between the two animals are different and make a hypothesis about why this might be.

51.4 Conclusion

While Google Earth Engine can be used for planetary-scale analyses, it is also an effective resource for quickly accessing and analyzing large amounts of information across time at a scale relative to your own data. The method presented in this chapter is a great way to add value to your own research. Here, we worked with weather data, but that is by no means the only option. You can connect your data to many other datasets within Earth Engine. It is up to you to explore and find what is relevant to your work.

Reference

Mahoney PJ, Young JK (2017) Uncovering behavioural states from animal activity and site fidelity patterns. Methods Ecol Evol 8:174–183. https://doi.org/10.1111/2041-210X.12658

Creating Presence and Absence Points

52

Peder Engelstad, Daniel Carver, and Nicholas E. Young

Overview

The purpose of this chapter is to demonstrate a method to generate your own presence and absence data and distribute those samples using specific ecological characteristics found in remotely sensed imagery. You will see that even when field data is unavailable, you can still digitally sample a landscape and gather information on current or past ecological conditions.

Learning Outcomes

- Generating presence and absence data manually using high-resolution imagery.
- Generating randomly distributed points automatically within a feature class layer to use as field sampling locations.
- Filtering your points to refine your sampling locations.

P. Engelstad (✉) · N. E. Young
Natural Resource Ecology Laboratory, Colorado State University, Fort Collins, CO 80523-1499, USA
e-mail: peder.engelstad@colostate.edu

N. E. Young
e-mail: nicholas.young@colostate.edu

D. Carver
Geospatial Centroid, Colorado State University, Fort Collins, CO 80523-1499, USA
e-mail: carverd@colostate.edu

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_52

Helps if you know how to

- Import images and image collections, filter, and visualize (Part I).
- Visualize images with a variety of false color band combinations (Chap. 2).
- Perform basic image analysis: select bands, compute indices, create masks (Part II).
- Use `normalizedDifference` to calculate vegetation indices (Chap. 5).
- Use drawing tools to create points, lines, and polygons (Chap. 6).
- Filter a `FeatureCollection` to obtain a subset (Chaps. 22, and 23).

52.1 Introduction to Theory

Herbivore grazing can have a negative effect on aspen regeneration in some areas, as aspens tend to grow in large monotypic stands (Halofsky and Ripple 2008). Excluding elk, deer, and cow grazing from an area has observable effects on aspen regrowth. In a hypothetical study, we may be interested in understanding the effect of herbivory from these species on various ecological aspects of aspen stands. But how can we monitor aspen stands without setting foot in the field? In this chapter, we will use multiple datasets and high-resolution resolution imagery (1 m^2) to generate sampling locations for this theoretical field survey. We will also build a presence/absence dataset that could be used to train a spatial model of aspen coverage.

52.2 Practicum

The National Land Cover Database (NLCD) is a Landsat-derived land cover database with 30 m spatial resolution, available at multiple time periods. Data from 1992 are primarily based on an unsupervised classification of Landsat data. The other years rely on a decision tree classification to identify the land cover classes.

The National Agriculture Imagery Program (NAIP) acquires aerial imagery during the agricultural growing seasons across the continental United States. NAIP projects are contracted each year based upon available funding and the Farm Service Agency imagery acquisition cycle. NAIP imagery is acquired at a one-meter ground sample distance with a horizontal accuracy that matches within six meters of photo-identifiable ground control points. NAIP imagery is often collected in four bands: blue, green, red, and near infrared. The near infrared band is helpful in distinguishing between different types of vegetation.

The USGS National Elevation Dataset (NED) 1/3 arc-second is an elevation dataset produced by the USGS. This coast-to-coast dataset is available at 0.33 arc-second spatial resolution (30 m) across the lower 48 states, parts of Alaska, Hawaii, and US Territories.

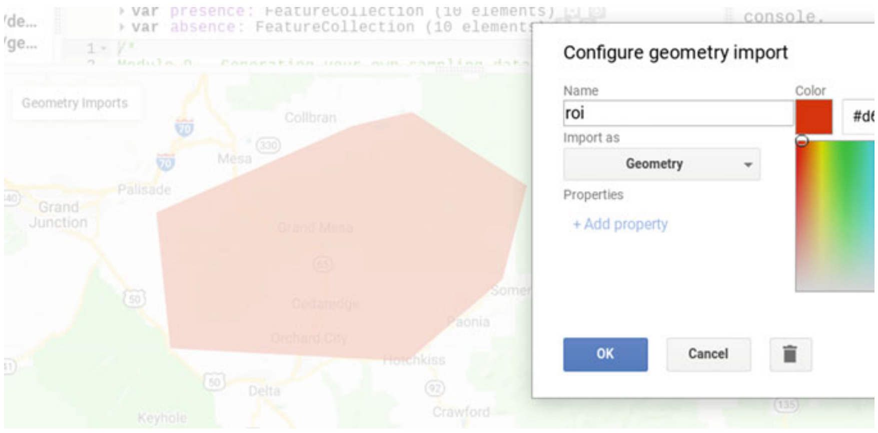


Fig. 52.1 Example of the general region of interest to use for this module

52.2.1 Section 1: Developing Your Own Sampling Locations

We will start by developing potential field sampling locations based on the relative physical and ecological conditions.

52.2.1.1 Section 1.1: Region of Interest

The geographic region for this chapter is the Grand Mesa in western Colorado. Considered to be the largest mesa in the world by area, the Grand Mesa rises from 4000 feet near Grand Junction, Colorado, to over 10,000 feet at its highest point. Historically, the Grand Mesa was glaciated, leaving its top flattened and spotted with lakes. While heavily forested at higher elevation, there are distinct transitions between shrubland, aspen, and conifer forest along the steep slopes. As one of the westernmost extensions of the montane environment in Colorado, the Grand Mesa represents important habitat for many ungulate species.

Our first step is to open a new script in Earth Engine. First, create a region of interest that encompasses the Grand Mesa (you can search for it by name in the search bar at the top). Do so using the geometry tool. Once you create the feature, rename it `roi` (Fig. 52.1).

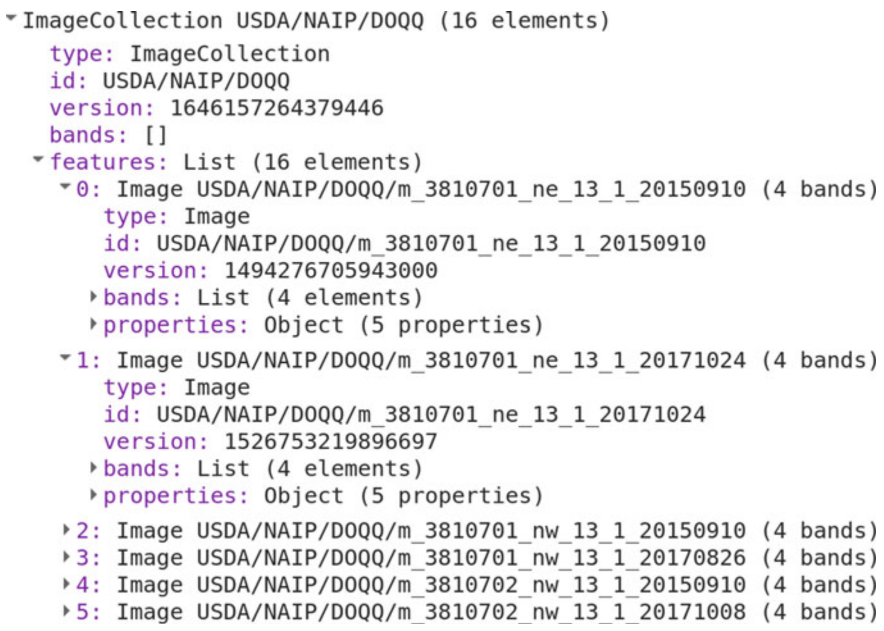
52.2.1.2 Section 1.2: Working with NAIP

We will rely on NAIP imagery for multiple steps in this process. NAIP imagery is not collected every year, so it makes sense to load in multiple years to determine what time frame is available. We will use a print statement to see what years are available for our area from 2015 and 2017. Copy the following code to start your script.

```
// Call in NAIP imagery as an image collection.
var naip = ee.ImageCollection('USDA/NAIP/DOQQ')
    .filterBounds(roi)
    .filterDate('2015-01-01', '2017-12-31');

Map.centerObject(naip);

print(naip);
```

When you run the script you will see a `print` statement similar to the one shown in Fig. 52.2.

The `print` statement shows us that imagery is available for both 2015 and 2017, yet it is difficult to determine the extent of coverage present without visualizing the data. For now, we will add both collections to the map to see what the image availability looks like. Add the following code to your existing script.



```
▾ImageCollection USDA/NAIP/DOQQ (16 elements)
   type: ImageCollection
   id: USDA/NAIP/DOQQ
   version: 1646157264379446
   bands: []
  ▾features: List (16 elements)
    ▾0: Image USDA/NAIP/DOQQ/m_3810701_ne_13_1_20150910 (4 bands)
        type: Image
        id: USDA/NAIP/DOQQ/m_3810701_ne_13_1_20150910
        version: 1494276705943000
       ▸bands: List (4 elements)
       ▸properties: Object (5 properties)
    ▾1: Image USDA/NAIP/DOQQ/m_3810701_ne_13_1_20171024 (4 bands)
        type: Image
        id: USDA/NAIP/DOQQ/m_3810701_ne_13_1_20171024
        version: 1526753219896697
       ▸bands: List (4 elements)
       ▸properties: Object (5 properties)
    ▸2: Image USDA/NAIP/DOQQ/m_3810701_nw_13_1_20150910 (4 bands)
    ▸3: Image USDA/NAIP/DOQQ/m_3810701_nw_13_1_20170826 (4 bands)
    ▸4: Image USDA/NAIP/DOQQ/m_3810702_nw_13_1_20150910 (4 bands)
    ▸5: Image USDA/NAIP/DOQQ/m_3810702_nw_13_1_20171008 (4 bands)
```

Fig. 52.2 Print statement identifying the years of available imagery for this region of interest

```
// Filter the data based on date.
var naip2017 = naip
    .filterDate('2017-01-01', '2017-12-31');

var naip2015 = naip
    .filterDate('2015-01-01', '2015-12-31');

// Define viewing parameters for multi band images.
var visParamsFalse = {
    bands: ['N', 'R', 'G']
};
var visParamsTrue = {
    bands: ['R', 'G', 'B']
};

// Add both sets of NAIP imagery to the map to compare
coverage.
Map.addLayer(naip2015, visParamsTrue, '2015_true', false);
Map.addLayer(naip2017, visParamsTrue, '2017_true', false);
```

Compared to the 2015 imagery (Fig. 52.3), the 2017 imagery (Fig. 52.4) has been captured later in the season and we can start to see some yellow in the aspen. A challenge with this image is that there were some distinct time lags between neighboring flight paths during the initial image collection. Notice the vertical band of green in the 2017 image. While it is usually best practice to use images that closely match the time that you will be in the field, in this case, the consistency of the 2015 imagery outweighs those temporal concerns.

We will also add a false color image to the 2015 data because this band combination visualization is very helpful for distinguishing between deciduous and coniferous forests. We do this by adding the `naip2015` object with a different set of visualization parameters to the map. Add the following code to your existing script.

```
// Add 2015 false color imagery.
Map.addLayer(naip2015, visParamsFalse, '2015_false',
false);
```

52.2.1.3 Section 1.3: Aspen Exclosures

In our hypothetical study, let us imagine land managers have established some aspen exclosures on the southern extent of the Grand Mesa. Let us also say that the land managers did not have specific shapefiles of the exclosures but did have GPS locations of the four corners. We will use these data to add the exclosures

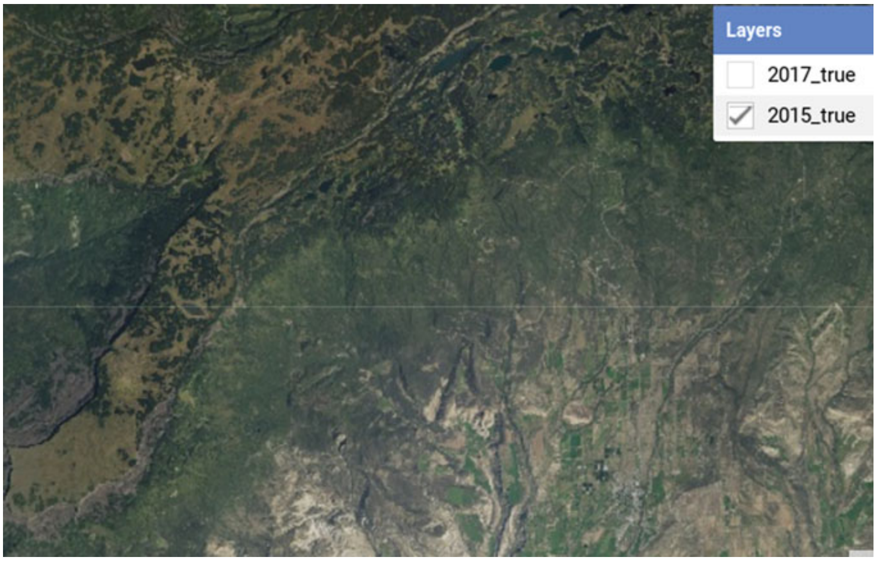


Fig. 52.3 NAIP imagery from 2015 has captured the aspen forest across the extent of our study area

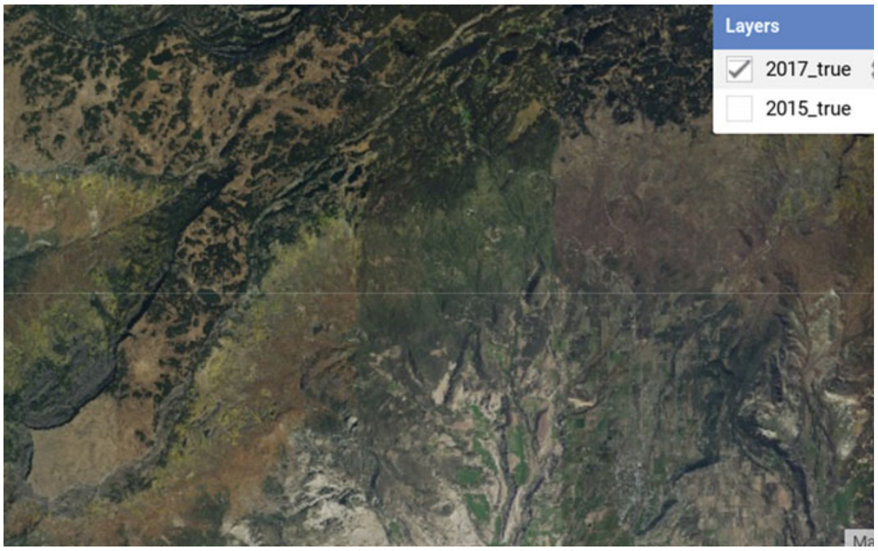


Fig. 52.4 NAIP imagery from 2017 has captured the yellowing of the aspen forest but shows a distinct difference in vegetation cover due to variability in the time at which the given images were captured

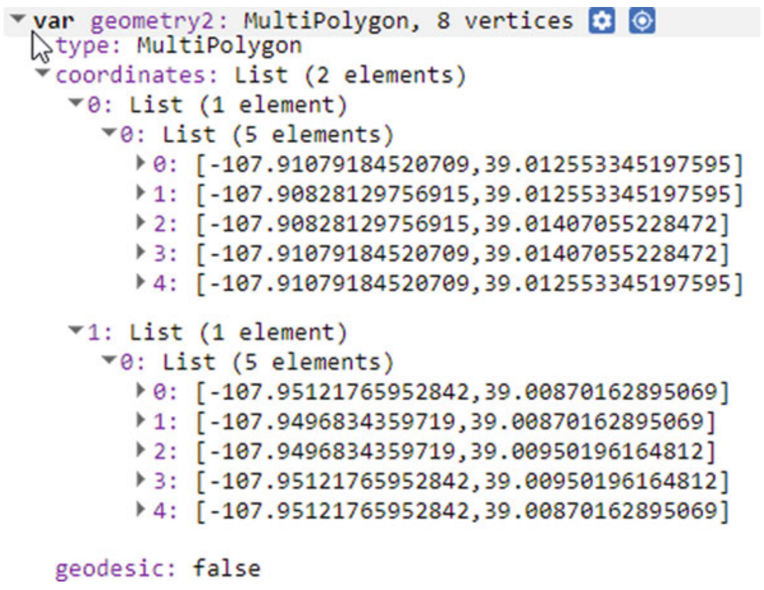
to the map by creating a geometry feature within our script. In this case we are creating an `ee.Geometry.MultiPolygon` feature. Add the following code to your existing script. If your `roi` object does not encompass the exclosures, edit its bounds or redraw it.

```
// Creating a geometry feature.
var exclosure = ee.Geometry.MultiPolygon([
    [
        [-107.91079184, 39.012553345],
        [-107.90828129, 39.012553345],
        [-107.90828129, 39.014070552],
        [-107.91079184, 39.014070552],
        [-107.91079184, 39.012553345]
    ],
    [
        [-107.9512176, 39.00870162],
        [-107.9496834, 39.00870162],
        [-107.9496834, 39.00950196],
        [-107.95121765, 39.00950196],
        [-107.95121765, 39.00870162]
    ]
]);

print(exclosure);

Map.addLayer(exclosure, {}, 'exclosures');
```

The structure of the `ee.Geometry.MultiPolygon` feature is a bit complex, but it is effectively a set of nested lists. There are three tiers of lists present:

```
// Nested list example.
['tier 1' ['tier 2' ['tier 3']]]
```

- **Tier 1**: A single list that holds all the data. The `ee.Geometry.MultiPolygon` function requires the input to be a list.
- **Tier 2**: A list for each polygon, containing a unique set of coordinates.
- **Tier 3**: A list for each x,y coordinate pair. Each polygon is composed of a series of x,y points with a point that overlaps exactly with the first coordinate pair. This last coordinate pair is the same as the first and is essential to completing the feature.



```
▼ var geometry2: MultiPolygon, 8 vertices
   type: MultiPolygon
  ▼coordinates: List (2 elements)
    ▼0: List (1 element)
      ▼0: List (5 elements)
        ▶0: [-107.91079184520709,39.012553345197595]
        ▶1: [-107.9082812975691‌5,39.012553345197595]
        ▶2: [-107.90828129756915,39.01407055228472]
        ▶3: [-107.91079184520709,39.01407055228472]
        ▶4: [-107.91079184520709,39.012553345197595]

    ▼1: List (1 element)
      ▼0: List (5 elements)
        ▶0: [-107.9512176595284‌2,39.00870162895069]
        ▶1: [-107.9496834359719,39.00870162895069]
        ▶2: [-107.9496834359719,39.00950196164812]
        ▶3: [-107.95121765952842,39.00950196164812]
        ▶4: [-107.95121765952842,39.00870162895069]

   geodesic: false
```

Fig. 52.5 View of the exclosure object coordinate details from the print statement

If you are trying to make a geometry feature and are having trouble, you can always create one with the draw shape tool and look at the values in a print statement as a template (Fig. 52.5).

52.2.1.4 Section 1.4: Determining Similar Areas for Sampling

Now that we have our aspen exclosures loaded, we are going to bring in some additional layers to help quantify the landscape characteristics of the exclosures. We will use those values to find similar areas nearby to use as sampling sites outside of the exclosures. By keeping the environmental conditions similar, we can make stronger statements about the comparative effects of herbivory on aspen stands.

We will use three datasets to describe the conditions within the sampled areas:

1. NED**:** Select areas in a similar elevation range. Elevation is correlated with many environmental conditions, so we are using it as a proxy for features such as temperature, precipitation, and solar radiation.
2. NAIP**:** Calculate an NDVI index to get a measure of vegetation productivity.
3. NLCD**:** Select the deciduous forest class as a way to limit the location of sampling sites.

52.2.1.5 Section 1.5: Loading in the Data

First, we will call the NED by its unique ID. We can gather these details by searching for the feature and reading through the metadata. Add the following code to your existing script.

```
// Load in elevation dataset; clip it to general area.
var elev = ee.Image('USGS/NED')
    .clip(roi);

Map.addLayer(elev, {
    min: 1500,
    max: 3300
}, 'elevation', false);
```

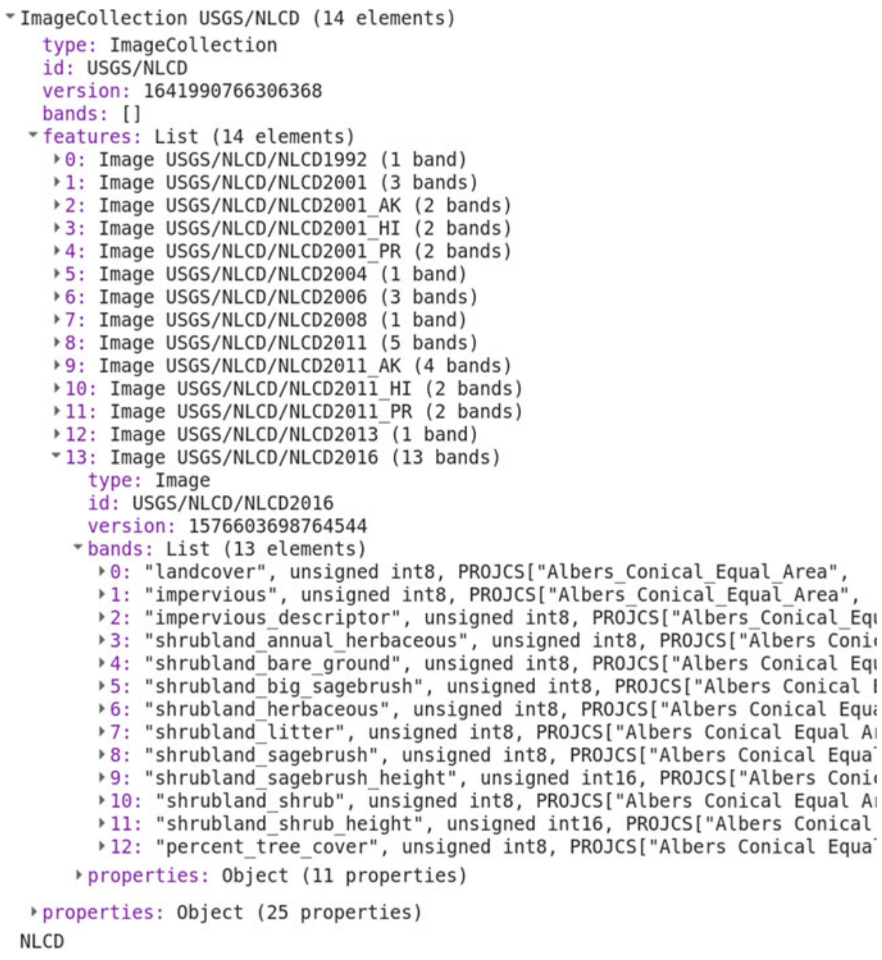
We already have NAIP imagery loaded but we need to convert it to an image and calculate NDVI. We have already filtered our NAIP imagery to a single year, so there is only one image per area. We could apply a reducer to convert the `ImageCollection` to an image, but a reducer is not necessary for a single layer. A more logical option for this is to apply the `mosaic` function to convert the `ImageCollection` to an image. Add the following code to your existing script.

```
// Apply mosaic, clip, then calculate NDVI.
var ndvi = naip2015
    .mosaic()
    .clip(roi)
    .normalizedDifference(['N', 'R'])
    .rename('ndvi');

Map.addLayer(ndvi, {
    min: -0.8,
    max: 0.8
}, 'NDVI', false);

print(ndvi, 'ndvi');
```

The last dataset we are bringing in is the NLCD. Add the following code to your existing script.

```
// Add National Land Cover Database (NLCD).
var dataset = ee.ImageCollection('USGS/NLCD');

print(dataset, 'NLCD');
```

From the print statement (Fig. 52.6), we can see that some of the images have a band called `'landcover'`. Rather than trying to pull it out from the feature collection, we will call the 2016 NLCD image directly by its unique ID and select the `'landcover'` band.



```
▾ImageCollection USGS/NLCD (14 elements)
   type: ImageCollection
   id: USGS/NLCD
   version: 1641990766306368
   bands: []
  ▾features: List (14 elements)
    ▸0: Image USGS/NLCD/NLCD1992 (1 band)
    ▸1: Image USGS/NLCD/NLCD2001 (3 bands)
    ▸2: Image USGS/NLCD/NLCD2001_AK (2 bands)
    ▸3: Image USGS/NLCD/NLCD2001_HI (2 bands)
    ▸4: Image USGS/NLCD/NLCD2001_PR (2 bands)
    ▸5: Image USGS/NLCD/NLCD2004 (1 band)
    ▸6: Image USGS/NLCD/NLCD2006 (3 bands)
    ▸7: Image USGS/NLCD/NLCD2008 (1 band)
    ▸8: Image USGS/NLCD/NLCD2011 (5 bands)
    ▸9: Image USGS/NLCD/NLCD2011_AK (4 bands)
    ▸10: Image USGS/NLCD/NLCD2011_HI (2 bands)
    ▸11: Image USGS/NLCD/NLCD2011_PR (2 bands)
    ▸12: Image USGS/NLCD/NLCD2013 (1 band)
    ▾13: Image USGS/NLCD/NLCD2016 (13 bands)
       type: Image
       id: USGS/NLCD/NLCD2016
       version: 1576603698764544
      ▾bands: List (13 elements)
        ▸0: "landcover", unsigned int8, PROJCS["Albers_Conical_Equal_Area",
        ▸1: "impervious", unsigned int8, PROJCS["Albers_Conical_Equal_Area",
        ▸2: "impervious_descriptor", unsigned int8, PROJCS["Albers_Conical_Equ
        ▸3: "shrubland_annual_herbaceous", unsigned int8, PROJCS["Albers Coni
        ▸4: "shrubland_bare_ground", unsigned int8, PROJCS["Albers Conical Equ
        ▸5: "shrubland_big_sagebrush", unsigned int8, PROJCS["Albers Conical
        ▸6: "shrubland_herbaceous", unsigned int8, PROJCS["Albers Conical Equa
        ▸7: "shrubland_litter", unsigned int8, PROJCS["Albers Conical Equal A
        ▸8: "shrubland_sagebrush", unsigned int8, PROJCS["Albers Conical Equa
        ▸9: "shrubland_sagebrush_height", unsigned int16, PROJCS["Albers Coni
        ▸10: "shrubland_shrub", unsigned int8, PROJCS["Albers Conical Equal A
        ▸11: "shrubland_shrub_height", unsigned int16, PROJCS["Albers Conical
        ▸12: "percent_tree_cover", unsigned int8, PROJCS["Albers Conical Equa
      ▸properties: Object (11 properties)
  ▸properties: Object (25 properties)
 NLCD
```

Fig. 52.6 Print statement showing the structure of the NLCD `ImageCollection`

```
// Load the selected NLCD image.
var landcover = ee.Image('USGS/NLCD/NLCD2016')
    .select('landcover')
    .clip(roi);

Map.addLayer(landcover, {}, 'Landcover', false);
```

Question 1. Investigate the other available years in the NLCD collection. What parts of the study area have land cover classes that change? What might be driving that change?

52.2.1.6 Section 1.6: Generating Random Points

With our three datasets loaded, we can now generate a series of potential survey sites. We will do this by generating random points within a given area (Fig. 52.7). We want these sites to be accessible, near the two exclosures, and within the public land boundary. We will create another geometry feature that we will use to contain the randomly generated points. Hover over the **Geometry imports** box and click + **new layer**. Be sure to name this second geometry feature `sampleArea`.

With the geometry feature in place, we can create points using the `randomPoints` function. Add the following code to your existing script.

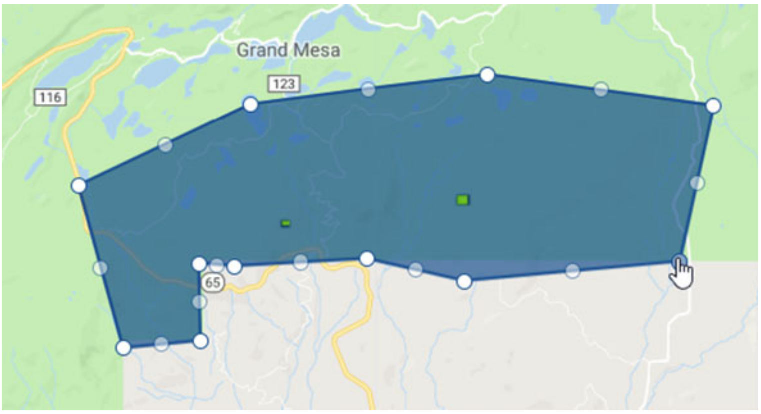


Fig. 52.7 An example of what your second sample area geometry feature might look like

```
// Generate random points within the sample area.
var points = ee.FeatureCollection.randomPoints({
    region: sampleArea,
    points: 1000,
    seed: 1234
});

print(points, 'points');

Map.addLayer(points, {}, 'Points', false);
```

Here, we are using a dictionary to define the parameters of the function. The `region` parameter is the area in which the points are created. In our case, we are going to set this to `sampleArea` polygon. The `points` parameter specifies the total number of points to generate. The `seed` parameter is used to indicate specific random values. Think of this as a unique ID for a set of random values. The seed number (1234 in this example) refers to an existing random list of values. Setting the seed is very helpful because you are still using random values, but the code will place the points in the same locations every time until the seed is changed.

52.2.1.7 Section 1.7: Extracting Values to Points

To associate the landscape characteristics with the point locations, we are going to call the `ee.Image.sampleRegions` function. This function requires a single image. Rather than calling three times on our three unique images (elevation, NDVI, and NLCD), we are going to add all those images together to create a multiband image so we can call this function a single time. Add the following code to your existing script.

```
// Add bands of elevation and NAIP.
var ndviElev = ndvi
    .addBands(elev)
    .addBands(landcover);

print(ndviElev, 'Multi band image');
```

With the multiband image set, we will call the `sampleRegions` function. Add the following code to your existing script.

```
// Extract values to points.
var samples = ndviElev.sampleRegions({
    collection: points,
    scale: 30,
    geometries: true
});

print(samples, 'samples');

Map.addLayer(samples, {}, 'samples', false);
```

Within this function, the `collection` parameter is set to the feature collection where the extracted values will be added. In this example, it is a point dataset. The `scale` parameter refers to the spatial resolution (pixel size) of the data. The `geometries` parameter indicates whether or not you want to maintain the x,y coordinate pairs for each element in the collection. The default is false, but we set it to true here because we eventually want to show these points on the map, as they will represent suitable sampling site locations.

Segue on Scale

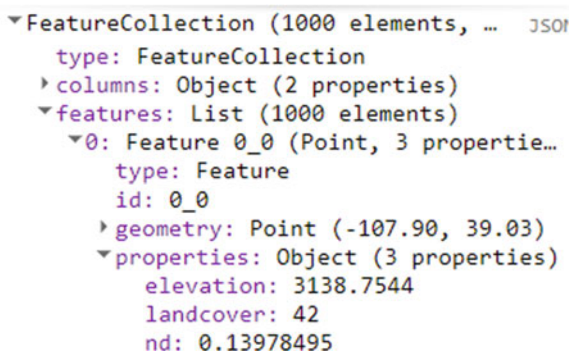
The idea of *scale* is very important in remote sensing, but you will often encounter multiple definitions of it. *Map scale*, the relationship between a measured distance on a map and the actual distance on the landscape, is the most common in remote sensing. The image *extent* can also be referred to as the *scale of the image* or the *spatial scale*. Confusing, right?

In the case of the `scale` argument in the function above, it is referring to yet another definition of that word—the pixel size of the image. In our example, the multiband image has two bands with a pixel size of 30 m, and one with a pixel size of 1 m. It is best practice to use the largest pixel size when working with data having different spatial resolutions. Here, this means you are upscaling the 1 m image to 30 m. This means a potential loss of precision in your data. However, you can be confident that the number in that upscaled cell is representative of the mean value across all the cells. If you go in the opposite direction and downscale an image, you effectively make up the data to fill the gap. Because NAIP imagery has a 1 m pixel size, it contains 30 × 30 more pixels than a single 30 m pixel. Take a look at the examples below to help visualize these processes.

Upscaling takes the available data and finds the mean value.

3	7	8
4	2	2
1	3	2

Question 2. Calculate the mean for the matrix above. How would this value change your understanding of the pixel? Is it still representative of the overall "story" the data is telling, or would a different reducer (i.e., median) be more appropriate?



```
▾FeatureCollection (1000 elements, …   JSON
   type: FeatureCollection
  ▸columns: Object (2 properties)
  ▾features: List (1000 elements)
    ▾0: Feature 0_0 (Point, 3 propertie…
       type: Feature
       id: 0_0
      ▸geometry: Point (-107.90, 39.03)
      ▾properties: Object (3 properties)
         elevation: 3138.7544
         landcover: 42
         nd: 0.13978495
```

Fig. 52.8 Result of our sampling function has given us 1000 potential sites to choose from. We will limit this pool by comparing the measurable data we have at these sites to the mean values of those data from our exclosures

Downscaling takes a single value and places the value for all locations in the grid.

7	7	7
7	7	7
7	7	7

From our print statement (Fig. 52.8), we can see that each of our 1000 point locations has three properties: elevation, land cover, and NDVI. We want to use these values to filter out sites that do not match the conditions of the exclosures. The land cover data are categorical, so it is easy to filter. We know from the metadata on NLCD that the land cover class for deciduous forest is a single value (41).

We will use the `filter` function to select all sites that are within aspen forests. Add the following code to your existing script.

```
// Filter metadata for sites in the NLCD deciduous forest
layer.
var aspenSites =
samples.filter(ee.Filter.equals('landcover', 41));

print(aspenSites, 'Sites');
```

From the print statement, we can see this reduces the total number of potential sites (1000) by about 25%, depending on your study area shape. However, to get down to roughly 10 potential new monitoring sites, there is still a lot of trimming to do.

Filtering based on elevation and NDVI is a bit trickier because both of these variables are continuous data. You want to find sites that have similar values to those in the exclosures, but they do not need to be exactly the same. For this example, we will say that if a value is within ± 10% of the mean value found within the exclosure, we will group it as similar. There are a couple of things that need to be calculated before we can filter our potential sampling sites:

1. Mean values within exclosures.
2. 10% above and below the mean.

Question 3. The above use of ± 10% is somewhat arbitrary. How might the range change given the inclusion of additional exclosures in our study area? What other methods could be used to generate this range?

We will work with the NDVI image first and then apply this process to the elevation dataset.

- Step 1: To find the mean value, we are going to apply a mean reducer over the area that is inside of the exclosure. Add the following code to your existing script.

```
// Set the NDVI range.
var ndvi1 = ndvi
    .reduceRegion({
        reducer: ee.Reducer.mean(),
        geometry: exclosure,
        scale: 1,
        crs: 'EPSG:4326'
    });

print(ndvi1, 'Mean NDVI');
```

The `reduceRegion` function takes in an image and returns a dictionary where the key is the name of the band, and the value is the output of the reducer.

- Step 2: Determine the acceptable variability around the mean.

We are relying on simple mathematical functions to find the ± 10% values. Add the following code to your existing script.

```
// Generate a range of acceptable NDVI values.
var ndviNumber = ee.Number(ndvi1.get('ndvi'));
var ndviBuffer = ndviNumber.multiply(0.1);
var ndviRange = [
    ndviNumber.subtract(ndviBuffer),
    ndviNumber.add(ndviBuffer)
];

print(ndviRange, 'NDVI Range');
```

The first step in this process is all about understanding data types. The `ndvi1` object is a dictionary so we call the `get` function to pull a value based on a known key. We then convert that value to an `ee.Number` type object so we can apply the math functions. Our output is a list with the minimum and maximum values ($\pm 10\%$ of the mean NDVI values).

52.2.1.8 Section 1.8: Create Your Own Function

We now have our range for NDVI, but we also need to apply the same process to elevation. In this case, we are only applying this process twice, but it seems like a useful chunk of code that might be handy down the road. Rather than just copying and pasting the code again and again, we are going to create a function with flexible parameters so we can apply this useful bit of code efficiently.

A function has the following requirements: parameters, statements, and a return value. Here is an example of the structure of a function from the official Earth Engine documentation. The following pseudocode demonstrates the syntax and structure of a function in JavaScript. *Do not add this code to your script.*

```
var myFunction = function(parameter1, parameter2,
parameter3) {
    statement;
    statement;
    var value = statement;
    return value;
};
```

If we apply this structure to our goal of reducing the elevation by area (like we did for NDVI with the code we created above), we would need to consider the following:

- Parameters: an image, a geometry object, and `pixelSize`.
- Statements: `reduceRegion` function.
- A return value: output of the `reduceRegion` function.

When using functions, it is important to use informative names within your parameters that give some indication of the data type that is required. If we want our function to be reproducible, we can provide some more information as a longer comment when we define the function. Add the following code to your existing script.

```
/*
This function is used to determine the mean value of an
image within a given area.
image: an image with a single band of ordinal or interval
level data
geom: geometry feature that overlaps with the image
pixelSize: a number that defines the cell size of the image
Returns a dictionary with the median values of the band,
the key is the band name.
*/
var reduceRegionFunction = function(image, geom, pixelSize)
{
    var dict = image.reduceRegion({
        reducer: ee.Reducer.mean(),
        geometry: geom,
        scale: pixelSize,
        crs: 'EPSG:4326'
    });
    return (dict);
};
```

Let us check our function definition by verifying that it gives the same answer for the NDVI range as when we did it step by step:

```
// Call function on the NDVI dataset to compare.
var ndvi_test = reduceRegionFunction(ndvi, exclosure, 1);

print(ndvi_test, 'ndvi_test');
```

This is a very clean method for coding when you need to apply a process multiple times. The function has been defined and now we can call it on the elevation dataset. Add the following code to your existing script.

```
// Call function on elevation dataset.
var elev1 = reduceRegionFunction(elev, exclosure, 30);

print(elev1, 'elev1');
```

We will define a second function to determine ± 10% around a mean value.

- Parameters: an image, band name, proportion.
- Statements: multiple steps.
- A return value: list.

Add the following code to your existing script.

```
/*
Generate a range of acceptable values.
dictionary: a dictionary object
key: key to the value of interest, must be a string
proportion: a percentile to define the range of the values
around the mean
Returns a list with a min and max value for the given
range.
*/
var effectiveRange = function(dictionary, key, proportion)
{
    var number = ee.Number(dictionary.get(key));
    var buffer = number.multiply(proportion);
    var range = [
        number.subtract(buffer),
        number.add(buffer)
    ];
    return (range);
};
```

We will call the `effectiveRange` function on the output of the `reduceRegionFunction` function. Add the following code to your existing script.

```
// Call function on elevation data.
var elevRange = effectiveRange(elev1, 'elevation', 0.1);

print(elevRange);
```

Now that we have an effective range for both our NDVI and elevation values, we can apply an additional set of filters to thin the list of potential sample sites. We can do this by chaining multiple `ee.Filter` calls together. Add the following code to your existing script.

```
// Apply multiple filters to get at potential locations.
var combinedFilter = ee.Filter.and(
    ee.Filter.greaterThan('ndvi', ndviRange[0]),
    ee.Filter.lessThan('ndvi', ndviRange[1]),
    ee.Filter.greaterThan('elevation', elevRange[0]),
    ee.Filter.lessThan('elevation', elevRange[1])
);

var aspenSites2 = aspenSites.filter(combinedFilter);

print(aspenSites2, 'aspenSites2');

Map.addLayer(aspenSites2, {}, 'aspenSites2', false);
```

Depending on how you have drawn your study area, this filtering process should reduce the 1000 original sites by about 90%. From the ~ 100 remaining sites, we can manually select 10 either by something we already know about the landscape features of our study area, or by using a more restrictive range in our function (e.g., ± 5%). This approach to site selection can be a good first step in ensuring you are sampling similar conditions in the field.

Code Checkpoint A37a. The book's repository contains a script that shows what your code should look like at this point.

52.2.2 Section 2: Generating Your Own Training Dataset

As you have been examining this landscape, you may have noticed some misclassifications within the NLCD land cover layer (e.g., forests in non-forested areas). Some misclassifications are expected in any land cover dataset. While the NLCD is trained to produce classifications of specific land cover assemblages across the United States, the aspen forest class that we are examining is included within a much larger grouping ("Deciduous forest"). Also, the particular NLCD image we selected shows land cover as it was detected in 2011. While forests are fairly stable over time, we can expect that some level of change has occurred. This might get you thinking about the possibility of generating your own aspen land cover product based on a remote sensing model for this specific region. There is a lot that goes into this process that we are not going to cover here. However, we are going to take the first step, which is generating our own presence/absence training dataset.

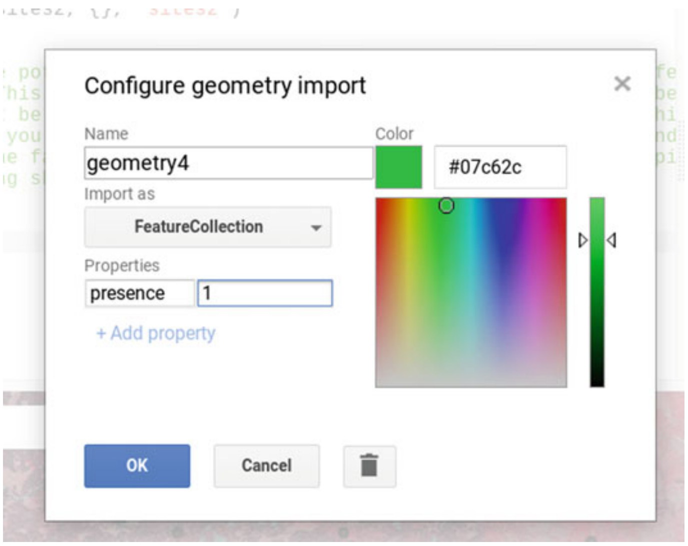
52.2.2.1 Section 2.1: Ocular Sampling

Generating your own training data relies on the assumption that you can confidently identify your species of interest using high-resolution imagery. We are using NAIP for this process because it is freely available and has a known collection date, allowing us to mark areas of aspen forest as "presence" and areas without aspens as "absence." These data could then be used later to train a model of aspen occurrence on the landscape.

52.2.2.2 Section 2.2: Adding Presence and Absence Points

First, we will need to create specific layers that will hold our new sampling points. Adding in presence and absence layers is a straightforward process accomplished by manually creating and placing geometry features on representative locations on the map. Hover over the **Geometry imports** box and click **+ new layer (Fig. 52.9)**. A geometry feature named "geometry" will be added. Select the gear icon next to **geometry** and a pop-up will open. Change the **Import as** type to **FeatureCollection**, then press the **+ Add property** button. Fill in the **Properties** values with "presence" and "1" and press **OK** to save your feature.

Change the feature collection name to `presence` and select a color you enjoy. Repeat this process to create an `absence` feature collection where the added property values are presence and "0". We will use the binary value in the presence


Configure geometry import

Name: geometry4

Color: #07c62c

Import as: FeatureCollection

Properties: presence 1

Fig. 52.9 Example of changing the parameters for the creation of the presence geometry feature

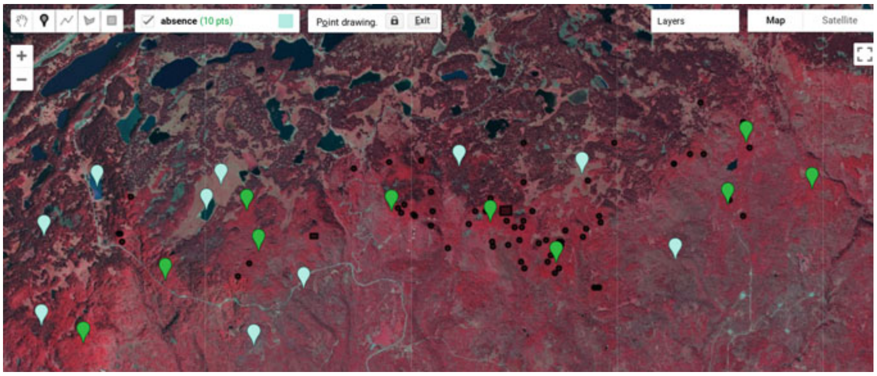


Fig. 52.10 Examples of presence and absence locations on the NAIP imagery created using the marker tool. Do your best to select locations that look correct to you

column of both datasets to define what that location is referring to: 1 = Yes, this is aspen; 0 = No, this is not aspen.

Once the feature collections are created, we select the specific feature collection (`presence` or `absence`) and use the marker tool to drop points on the imagery. The sampling methodology you use will depend on your study. In this example, green presence points represent aspen forest, and blue points are not aspen (absence).

Use the 2015 false color imagery in combination with the NDVI or NLCD to distinguish aspen from other land cover types. Aspen stands are brighter red than most other vegetation types and tend to have a more complex texture than herbaceous vegetation in the imagery. Drop some points in what you perceive to be aspen forest (Fig. 52.10).

Feel free to sample as many locations as you would like. Again, the quality of this data will depend on your ability to differentiate the multiple land cover classes present.

52.2.2.3 Section 2.3: Exporting Points

Currently our point locations are stored in two different features classes. We will merge these features into one feature class before exporting the data. We can merge the layers straightforwardly because they share the same data type (point geometry feature) and the same attribute data ("presence" with a numeric data value). Add the following code to your existing script.

```
// Merge presence and absence datasets.
var samples = presence.merge(absence);

print(samples, 'Samples');
```

Now that the sampling features classes are merged, we will export the features to our Google Drive. When you run the code below, the **Tasks** tab in the upper right-hand panel will light up. Earth Engine does not run tasks without you directing it to execute the task from the **Tasks** tab. Add the following code to your existing script and run it to export your completed dataset.

```
Export.table.toDrive({
    collection: samples,
    description: 'presenceAbsencePointsForForest',
    fileFormat: 'csv'
});
```

Code Checkpoint A37b. The book's repository contains a script that shows what your code should look like at this point.

52.3 Synthesis

Assignment 1. Compare samples of NDVI ranges for aspen presence versus absence. Are the ranges significantly different? Create samples using the code presented here but for a coniferous tree of your choice. How different are these values from the values of deciduous forest?

52.4 Conclusion

In this module, we identified aspen locations with similar environmental characteristics and generated our own sampling data from those locations. Both processes are simple in concept but can be somewhat complex to implement without access to all your data in a single place. In both cases, we are generating value-added products that are informed by remote sensing but are not inherently remote sensing processes. This ability to be creative regarding how you use remotely sensed data is part of the beauty of the Earth Engine platform.

Reference

Halofsky J, Ripple W (2008) Linkages between wolf presence and aspen recruitment in the Gallatin elk winter range of southwestern Montana, USA. Forestry 81:195–207. https://doi.org/10.1093/forestry/cpm044

Detecting Land Cover Change in Rangelands

53

Ginger Allington and Natalie Kreitzer

Overview
The purpose of this chapter is to familiarize you with the unique challenges of detecting land cover change in arid rangeland systems, and to introduce you to an approach for classifying such change that provides us with a better understanding of these systems. You will learn how to extract meaningful data about changes in vegetation cover from satellite imagery, and how to create a classification based on trajectories over time.

Learning Outcomes

- Visualizing and explaining the challenges of utilizing established land cover data products in arid rangelands.
- Applying a temporal segmentation algorithm to a time series of information about vegetation productivity.
- Classifying pixels based on similarities in their temporal trajectories.
- Extracting and visualizing data on the new trajectory classes.
- Comparing trajectory classes to information from traditional land cover data.

Helps if you know how to

- Import images and image collections, filter, and visualize (Part I).
- Perform pixel-based supervised or unsupervised classification (Chap. 6).

G. Allington (✉)
Department of Natural Resources and the Environment, Cornell University, Ithaca, NY, USA
e-mail: gra38@cornell.edu

N. Kreitzer
George Washington University, Washington, DC, USA

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_53

- Use expressions to perform calculations on image bands (Chap. 9).
- Write a function and `map` it over an `ImageCollection` (Chap. 12).
- Interpret the outputs from the LandTrendr algorithm implementation in Earth Engine (Chap. 17).
- Write a function and `map` it over a `FeatureCollection` (Chaps. 23 and 24).
- Use the `require` function to load code from existing modules (Chap. 28).

53.1 Introduction to Theory

Arid and semi-arid rangelands cover approximately 41% of the global land surface (Asner et al. 2004) and provide livelihoods for 38% of the human population (Millennium Ecosystem Assessment 2005), and forage for three-quarters of the world's livestock (Derner et al. 2017). Rangelands are located in regions of the world that are experiencing some of the most rapid changes in climate (Huang et al. 2016, Melillo et al. 2014), which can affect ecosystem productivity and resilience (Briske 2017). Land conversion to agriculture (Lambin and Meyfroidt 2011), urbanization and development (Fan et al. 2016; Sleeter et al. 2013), and afforestation (Cao et al. 2011) are increasing in many rangeland regions globally. Uncertainties about socioeconomic and sociopolitical forces such as land tenure security (Campbell et al. 2005; Li et al. 2007; Liu et al. 2015; Reid et al. 2000), rural out-migrations and urbanization (Lang et al. 2016), and changes in human demography and dietary preferences (Alexandratos and Bruinsma 2012) limit our ability to predict the future sustainability of rangeland systems. In order to understand the causes and consequences of land cover change in rangelands, we must first classify and quantify the change.

There are several readily available datasets commonly used to assess changes in land cover over large regions. The three most prominent global inventories are the annual MODIS 12Q land cover product (500 m) (Friedl et al. 2010), the 300 m GlobCover data from the European Space Agency (ESA) (Arino et al. 2008; Bontemps et al. 2011), and the new 10 m WorldCover data, also from the ESA (Zanaga et al. 2021), the latter two of which are available for single nominal dates. National-level data products typically are available at finer spatial resolutions and tend to use more categories to classify the surface (e.g., NLCD in the US, Homer et al. 2015)—though they are not available for all countries. While these global and national land cover data products have been useful for documenting a number of important surface phenomena such as forest loss, urbanization, and habitat conversion in a variety of ecosystems (Schneider et al. 2015, Prestele et al. 2016), they generally have poor accuracy in arid portions of the globe (Friedl et al. 2002; Ganguly et al. 2010; García-Mora et al. 2012).

Additionally, traditional methods for change detection are derived from categorical land cover classifications, which can identify change only when a pixel

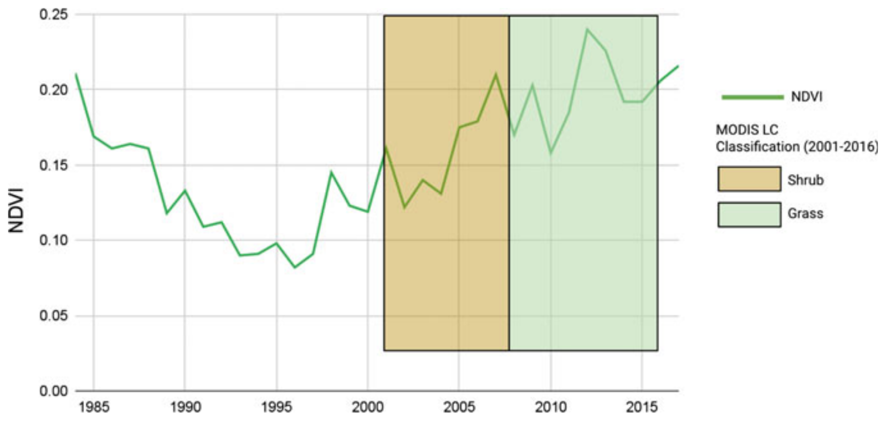


Fig. 53.1 Time series of NDVI in Naiman Banner, Inner Mongolia. Line represents the median value calculated over the region. Overlaid colored bars represent the corresponding land cover classes for the years 2001–2016, according to the MODIS-derived MCD12Q1 data

crosses a threshold to a new state (e.g., from Grassland to Barren), and therefore can identify change only after it has occurred, not when it begins. In systems with long time lags in vegetation response (such as has been shown for recovery in arid grasslands), this can create uncertainty about the efficacy of environmental policies and the response of landscapes to large-scale management changes (Fig. 53.1).

As an alternative to classification-based change detection, new methods have emerged that identify unique features in time series of remotely sensed data to pinpoint disturbance events such as logging, wildfires, and flooding. These time series-based methods, such as CCDC (see Chap. 19) and LandTrendr (see Chap. 17), are typically calibrated and executed to detect abrupt changes in spectral reflectance or derived index values associated with those disturbances (Kennedy et al. 2010, 2014; Zhu and Woodcock 2014). This is extremely useful for detecting change events like deforestation or the defoliation resulting from an insect outbreak where the disturbance is punctuated in time and where the difference in spectral index is significant, and regreening occurs on the order of years (Fig. 53.2, forested pixel).

For these reasons, contemporary classification approaches are extremely limited in their ability to provide reliable information about landscape change and dynamics in rangeland systems. Current pressures from climate change, land use intensification, and urban expansion create an urgent need for methods to more accurately map and track land cover status in a way that is useful for arid-systems research, land use change detection, and the monitoring and management of rangelands.

Forest land cover changes are stark in terms of the change in indices such as NDVI and NBR (Kennedy et al. 2016). When a forest is disturbed, the NDVI of a image pixel declines by a large magnitude nearly immediately (Fig. 53.2, forest pixel). It is relatively straightforward to code a change-detection algorithm to

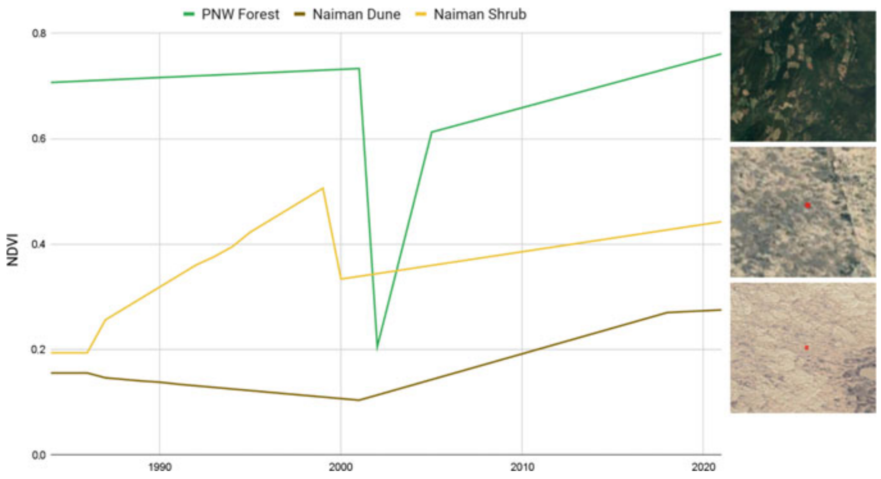


Fig. 53.2 Comparison of predicted NDVI for representative pixels from three regions: a forested area in southern British Columbia and two regions in Naiman Banner, Inner Mongolia, one dominated by woody shrubs and perennial forbs, the other by annual grasses and forbs

detect these large, abrupt transitions. Additionally, the visual change is stark, so a visual interpreter can easily create a training dataset without ancillary information to identify conversion from forest to burned, cleared, or developed.

In contrast, vegetation degradation and recovery in rangelands are typically characterized by more gradual changes over longer periods of time. Unlike with forest change, a trained interpreter using only satellite imagery cannot visually identify a rangeland land cover class transition until years after it has begun, if at all.

However, there is potential to utilize the information generated from temporal segmentation to characterize other aspects of change. Here, we will employ the LandTrendr time series segmentation algorithm (as described in Chap. 17) to derive information about how pixels are changing over time, and use that to generate a new land cover classification for a region of northern China.

To truly understand how rangelands are changing in space and time, and to provide adequate monitoring to support sustainable rangeland management, we need tools to help us capture and quantify those changes at landscape scales. Further, we need new ways of measuring changes within land cover types and interpreting those changes in ways that reflect rangeland dynamics and ecological processes. This necessitates rethinking the framework we use to categorize these lands in the first place. Approaches making use of greater temporal information on forest (Healey et al. 2018) and wetland systems (Dronova et al. 2015) show promise for deployment on rangeland systems.

We present a hybrid approach to rangeland classification that categorizes observable dynamic patterns in vegetation cover to classify pixels based on similarities in trajectory history. These new classes reveal much more meaningful information about the current status of a given pixel than a class based on cover type, and also yield information about the potential of a given pixel to respond to stressors in the future.

53.2 Practicum

53.2.1 Section 1: Inspecting Information About the Study Area

In this section, we will load and explore data sets that illustrate the focal study area, and form the basis for the analysis.

53.2.1.1 Section 1.1: Inspect the Study Area

Naiman Banner (Fig. 53.3) is located in the southeastern portion of the Inner Mongolia Autonomous Region of northern China. Historically, this region was occupied by ethnic Mongolian pastoralists, and the primary vegetation was perennial grasses with some shrub cover. The region has undergone significant intensification of land uses over the past 60 years (John et al. 2009). Heavy grazing and conversion to crops have removed vegetation, exposing soils to erosion and resulting in extensive desertification. Since the early 1990s, a series of environmental policies and restoration programs, including grazing restriction and afforestation, have been implemented to halt the spread of desertification and promote revegetation of the rangelands. At the same time, cropland has continued to expand, and the use of irrigation has eliminated almost all of the surface water sources and severely lowered the groundwater table.

In this exercise, we will focus on a portion of central Naiman Banner that spans from the extensive Horqin Sandy Lands in the west to the dense agricultural area in the east.

Our first step is to load a shapefile for our area of interest (AOI) and explore the landscape using the default satellite basemap.

```
// Load the shapefile asset for the AOI as a Feature
Collection
var aoi = ee.FeatureCollection(
    'projects/gee-book/assets/A3-8/GEE_Ch_AOI');
Map.centerObject(aoi, 11);
Map.addLayer(aoi, {}, 'Subset of Naiman Banner');
```

Switch the basemap to **Satellite** (Fig. 53.4). Turn the layer with the AOI boundary on and off and pan around the study area. Inspect the difference in land uses and cover types between the far western and eastern edges of the AOI. Mark them with pointer markers so you can revisit them later.

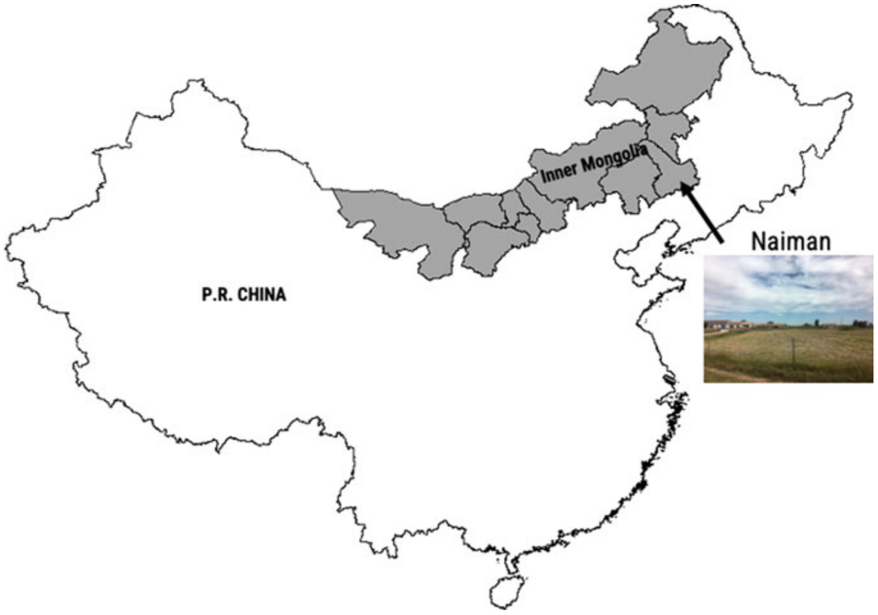


Fig. 53.3 Location of the Horqin Sandy Lands within Naiman Banner, Inner Mongolia, People's Republic of China

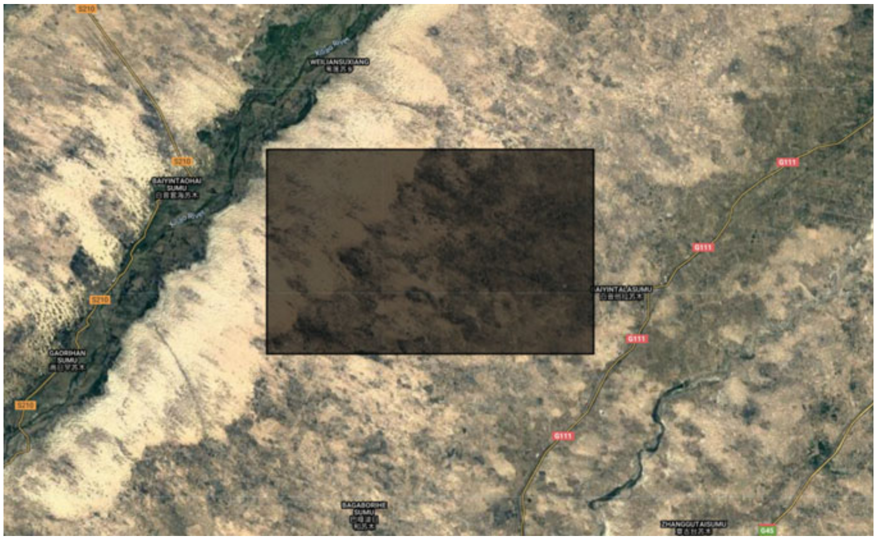


Fig. 53.4 Location of the study region, within the Horqin Sandy Lands

Question 1. List four potential land use or land cover classes that you observe in the image.

53.2.1.2 Section 1.2: Inspect Existing Land Use Data

Next, we will explore two different sources of land cover data to get a sense of how this landscape is typically categorized by these kinds of classified products.

First, we will explore the MODIS MCD12Q1 Land Cover Type dataset. These global layers are available annually from 2001 to the present at a resolution of 500 m. We will filter and visualize data for 2001, 2009, and 2016 in order to compare how MODIS captures change over this period.

Next, we will define and execute several different functions that we will need to use in this lesson. These functions will help us to execute the same logic across multiple images within different image collections.

```
// Filter the MODIS Collection
var MODIS_LC =
ee.ImageCollection('MODIS/006/MCD12Q1').select(
    'LC_Type1');

// Function to clip an image from the collection and set
the year
var clipCol = function(img) {
    var date = ee.String(img.get('system:index'));
    date = date.slice(0, 4);
    return img.select('LC_Type1').clip(aoi) // .clip(aoi)
        .set('year', date);
};
```

```
// Generate images for diff years you want to compare
var modis01 = MODIS_LC.filterDate('2001-01-01', '2002-01-
01').map(
    clipCol);
var modis09 = MODIS_LC.filterDate('2009-01-01', '2010-01-
01').map(
    clipCol);
var modis16 = MODIS_LC.filterDate('2016-01-01', '2017-01-
01').map(
    clipCol);
// Create an Image for each of the years
var modis01 = modis01.first();
var modis09 = modis09.first();
var modis16 = modis16.first();
```

Now that we have loaded all three datasets, let's take a look at them and see how they compare. The `randomVisualizer` code below assigns random colors to each class in each layer.

```
Map.addLayer(modis01.randomVisualizer(), {}, 'modis 2001',
false);
Map.addLayer(modis09.randomVisualizer(), {}, 'modis 2009',
false);
Map.addLayer(modis16.randomVisualizer(), {}, 'modis 2016',
false);
```

You should end up with something that looks like Fig. 53.5. (The colors assigned to classes in your map may differ.)

Use the slider bar in the **Layer** menu to turn the data on and off to compare classifications across the three years. Use the **Inspector tool** to select a few pixels around the AOI to pull information on the specific classes. We have provided an information from an example pixel in Fig. 53.6. Compare the pixel values for the band 'LC_Type1' reported in your **Inspector** to the land cover codes and descriptions listed in Table 53.1.

Question 2. What are the land cover types identified in this region, as classified by the MODIS data? Be sure to check all three time periods.

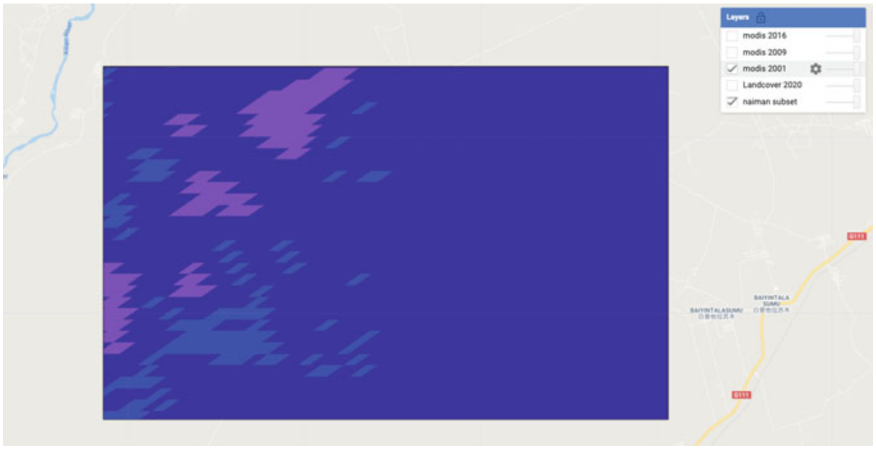


Fig. 53.5 Land cover classes identified in 2001 from MODIS MCD12Q1

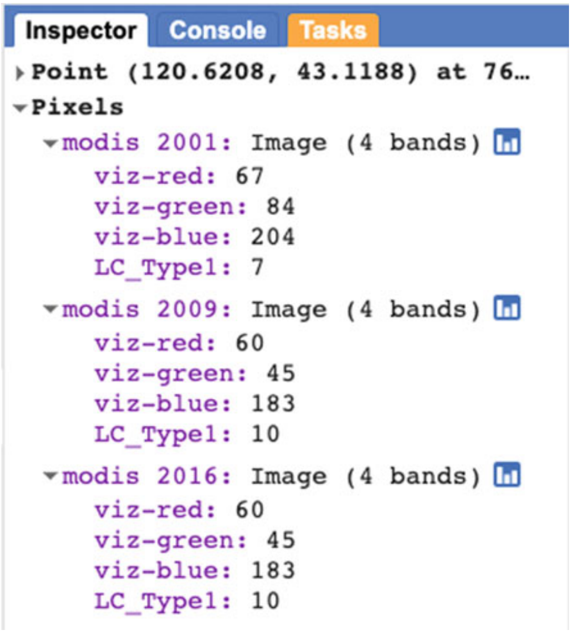
Table 53.1 Classes in the MODIS MCD12Q1 land cover dataset

Value	Description
1	Evergreen Needleleaf Forests: dominated by evergreen conifer trees (canopy >2 m). Tree cover >60%
2	Evergreen Broadleaf Forests: dominated by evergreen broadleaf and palmate trees (canopy >2 m). Tree cover >60%
3	Deciduous Needleleaf Forests: dominated by deciduous needleleaf (larch) trees (canopy >2 m). Tree cover >60%
4	Deciduous Broadleaf Forests: dominated by deciduous broadleaf trees (canopy >2 m) Tree cover >60%
5	Mixed Forests: dominated by neither deciduous nor evergreen (40–60% of each) tree type (canony >2 m) Tree
6	Closed Shrublands: dominated by woody perennials (1–2 m height) >60% cover
7	Open Shrublands: dominated by woody perennials (1–2 m height) 10–60% cover
8	Woody Savannas: tree cover 30–60% (canopy >2 m)
9	Savannas: tree cover 10–30% (canopy >2m)
10	Grasslands: dominated by herbaceous annuals (<2 m)
11	Permanent Wetlands: permanently inundated lands with 30–60% water cover and >10% vegetated cover
12	Croplands: at least 60% of area is cultivated cropland
13	Urban and Built-up Lands: at least 30% impervious surface area including building materials, asphalt and vehicles
14	Cropland/Natural Vegetation Mosaics: mosaics of small-scale cultivation 40–60% with natural tree, shrub, or herbaceous vegetation
15	Permanent Snow and Ice: at least 60% of area is covered by snow and ice for at least 10 months of the year
16	Barren: at least 60% of area is non-vegetated barren (sand, rock, soil) areas with less than 10% vegetation
17	Water Bodies: at least 60% of area is covered by permanent water bodies

Next, we will add the WorldCover dataset from the ESA. This dataset was generated for 2020 only, and has a resolution of 10 m.

```
// Add and clip the WorldCover data
var wCov =
ee.ImageCollection('ESA/WorldCover/v100').first();
var landcover20 = wCov.clip(aoi);
Map.addLayer(landcover20, {}, 'Landcover 2020');
```

The WorldCover dataset includes palette information in the `'Map'` band, so you should end up with something that looks like Fig. 53.7 (Table 53.2). (The colors assigned to classes in your map may differ.)



```
▸Point (120.6208, 43.1188) at 76…
▾Pixels
  ▾modis 2001: Image (4 bands)
     viz-red: 67
     viz-green: 84
     viz-blue: 204
     LC_Type1: 7
  ▾modis 2009: Image (4 bands)
     viz-red: 60
     viz-green: 45
     viz-blue: 183
     LC_Type1: 10
  ▾modis 2016: Image (4 bands)
     viz-red: 60
     viz-green: 45
     viz-blue: 183
     LC_Type1: 10
```

Fig. 53.6 Land cover classes assigned to a single pixel in the study area in 2001, 2009, and 2016 from MODIS MCD12Q1

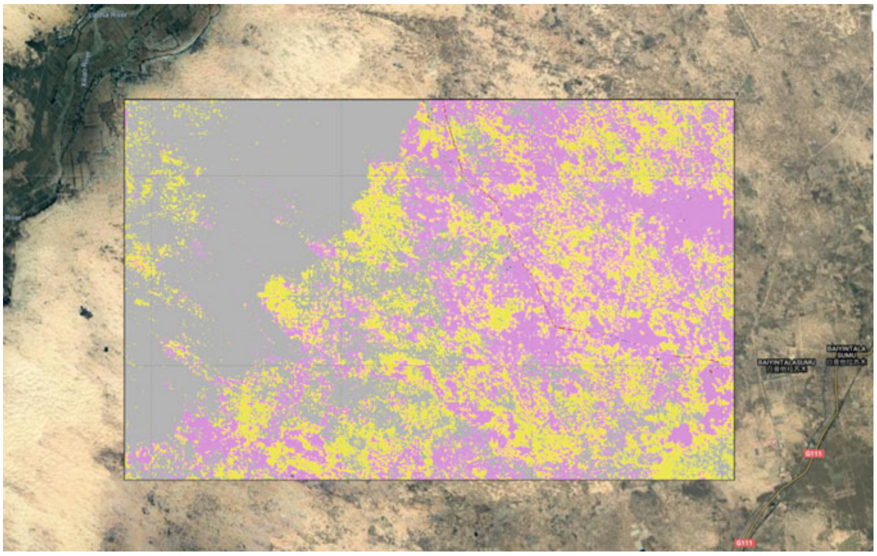


Fig. 53.7 Land cover classes identified in 2020 in the ESA WorldCover dataset

Table 53.2 Subset of the 11 land cover classes that appear within the study area in the ESA WorldCover dataset

Value	Description
10	Trees
30	Grassland
40	Cropland
50	Built-up
60	Barren/sparse vegetation

Code Checkpoint A38a. The book's repository contains a script that shows what your code should look like at this point.

Question 3. Spend a few minutes turning the different land cover layers on and off and using the Inspector to identify the classes for different pixels. How do the MODIS and WorldCover datasets differ? In what ways are they similar?

Question 4. Return to the points that you flagged in Sect. 53.2.1. Do your estimations of land cover correspond to the classifications from the two different data sources?

Question 5. Qualitatively compare change over time according to the MODIS data. What do you observe from these data? Approximately how much of the AOI has changed between 2001 and 2020, according to these data? Where are the regions of changing classification located? What regions seem to be fairly stable?

53.2.2 Section 2: Compile the Time Series of Vegetation Cover

The normalized difference vegetation index (NDVI) is an index of vegetative productivity derived from the relationship between the red (approx. 650 nm) spectra and the near-infrared (750–2500 nm) spectra. NDVI is a consistently good proxy for productivity in this region (de Beurs et al. 2015). It is also highly correlated with precipitation (John et al. 2009). This relationship has the potential to introduce short-term responses of increased vegetation productivity that could be falsely identified as land cover change (Fig. 53. 8a, b). In order to account for this effect, we need to remove the main effect of precipitation on NDVI for a given year so that we can assess changes in greenness outside of that variability. To do this, we will derive individual regression models for each pixel (as detailed in Chap. 18) for the relationship between total annual water-year precipitation and maximum greenness in each year. We will then predict greenness for each pixel in each year based on observed precipitation, and use the residual values of greenness from the predicted model as an input to LandTrendr. We can think of the residuals as the remaining annual primary productivity, after we have removed the effect of precipitation (Fig. 53. 8c).

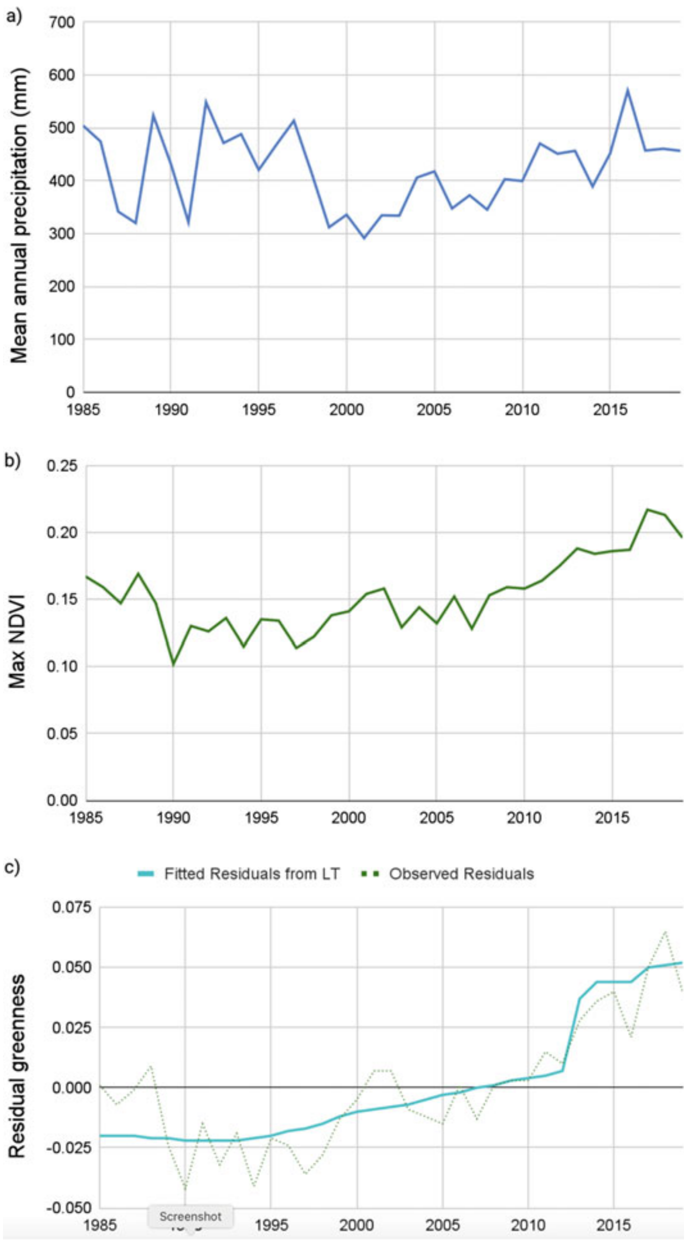
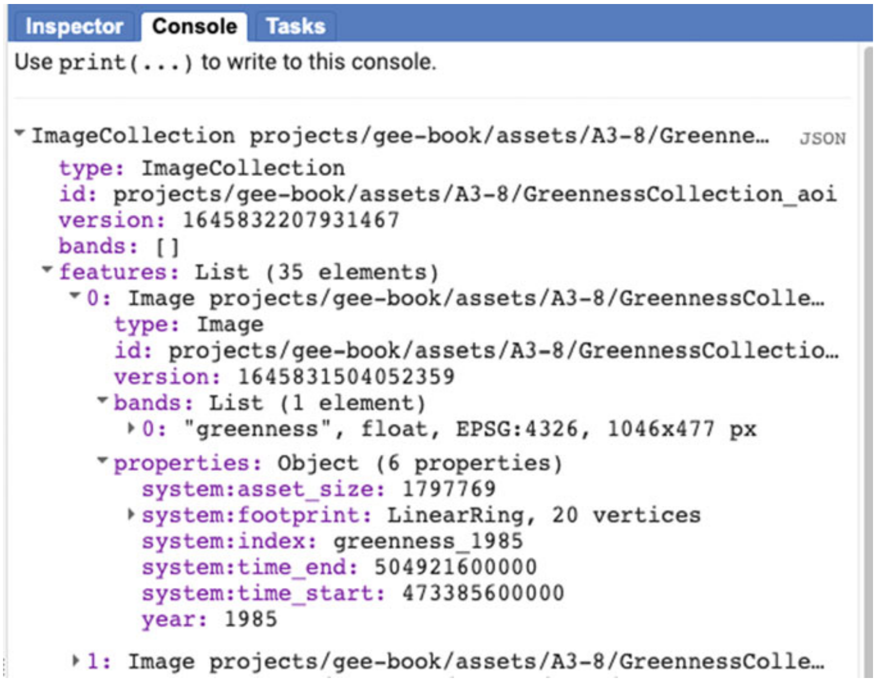


Fig. 53.8 Comparison of the variation over time in, **a** mean annual precipitation; **b** maximum NDVI for the AOI; **c** residual greenness from a regression of precipitation and NDVI



```
Use print(...) to write to this console.

▾ImageCollection projects/gee-book/assets/A3-8/Greenne…
   type: ImageCollection
   id: projects/gee-book/assets/A3-8/GreennessCollection_aoi
   version: 1645832207931467
   bands: []
  ▾features: List (35 elements)
    ▾0: Image projects/gee-book/assets/A3-8/GreennessColle…
        type: Image
        id: projects/gee-book/assets/A3-8/GreennessCollectio…
        version: 1645831504052359
      ▾bands: List (1 element)
        ▸0: "greenness", float, EPSG:4326, 1046x477 px
      ▾properties: Object (6 properties)
          system:asset_size: 1797769
        ▸system:footprint: LinearRing, 20 vertices
          system:index: greenness_1985
          system:time_end: 504921600000
          system:time_start: 473385600000
          year: 1985
    ▸1: Image projects/gee-book/assets/A3-8/GreennessColle…
```

Fig. 53.9 Properties of the pre-generated Image Collection of maximum annual greenness values, generated from NDVI

Before we go any further, let's add two pre-generated image collection assets for greenness and precipitation (`greennessColl` and `precipColl`) and explore them. Print each of them to the **Console** to inspect the contents (Fig. 53.9).

```
var greennessColl = ee.ImageCollection(
    'projects/gee-book/assets/A3-
8/GreennessCollection_aoi');
var precipColl = ee.ImageCollection(
    'projects/gee-book/assets/A3-8/PrecipCollection');
print(greennessColl, 'Greenness Image Collection');
print(precipColl, 'Precip Image Collection');
```

We saw in the plots above (Figs. 53.1 and 53.2) how NDVI can vary significantly over time in this region. We can also select out a few years to visualize how NDVI ("greenness") varies spatially as well.

```
var greennessParams = {
    bands: ['greenness'],
    max: 0.5,
    min: 0.06,
    opacity: 1,
    palette: ['e70808', 'ffffff', '1de22c']
};

var greenness1985 = greennessColl.filterDate('1985-01-01',
    '1986-01-01').select('greenness');
var greenness1999 = greennessColl.filterDate('1999-01-01',
    '2000-01-01').select('greenness');
```

```
print(greenness1999);
var greenness2019 = greennessColl.filterDate('2019-01-01',
    '2020-01-01').select('greenness');

Map.addLayer(greenness1985, greennessParams, 'Greenness
1985', false);
Map.addLayer(greenness1999, greennessParams, 'Greenness
1999', false);
Map.addLayer(greenness2019, greennessParams, 'Greenness
2019', false);
```

Turn the layers for the different years on and off to compare the range and spatial distribution of NDVI (Fig. 53.10).

Question 6. What do you observe about the similarities and differences in the distribution of NDVI across the selected years?

Combining the greenness and precipitation collections and calculating the model to generate residuals is a relatively long process. To speed things along, we will employ a function that has been defined in a script module called `residFunctions`.

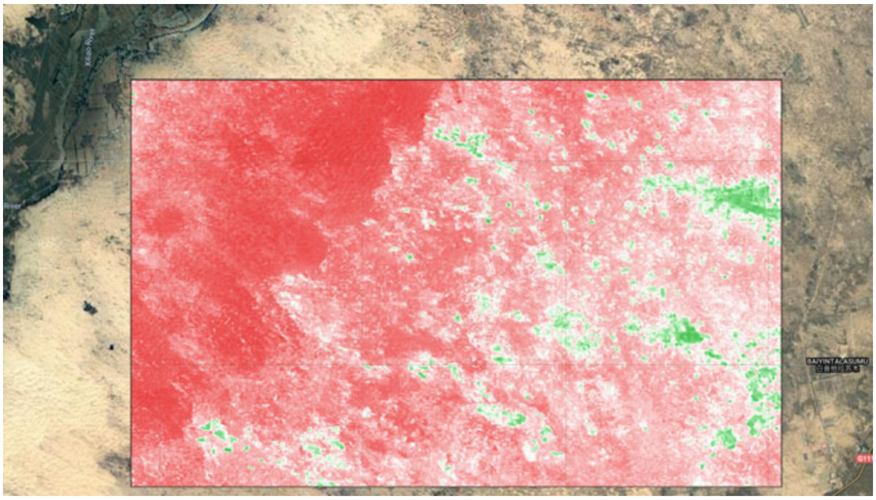
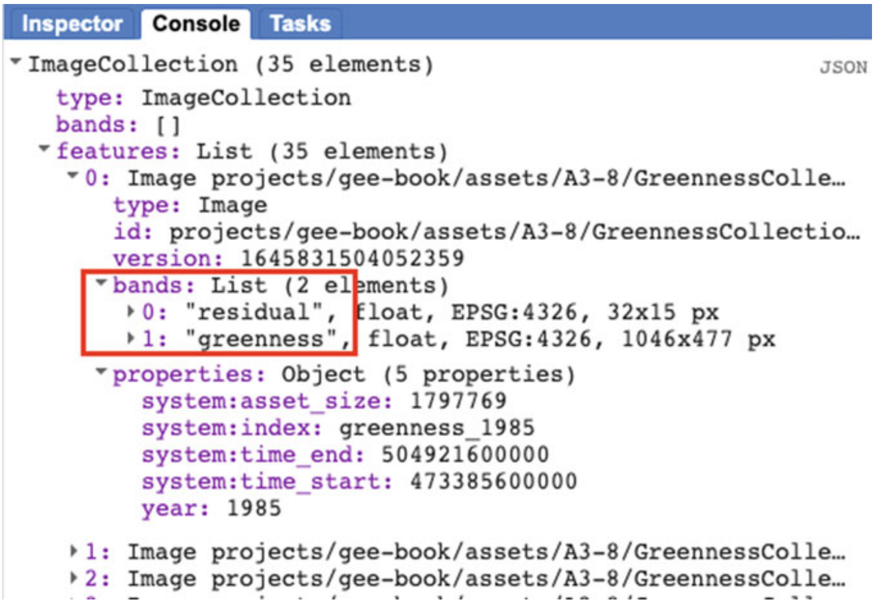


Fig. 53.10 Observed greenness (NDVI) in the study region in 2019. The values range from a minimum of 0.07 to a maximum of 0.5

```
// Load a function that will combine the Precipitation and
Greenness collections, run a regression, then predict NDVI
and calculate the residuals.

// Load the module
var residFunctions = require(
    'projects/gee-edu/book:Part A - Applications/A3 -
Terrestrial Applications/A3.8 Detecting Land Cover Change
in Rangelands/modules/calcResid'
);

// Call the function we want that is in that module
// It requires three input parameters:
// the greenness collection, the precipitation collection
and the aoi
var residualColl =
(residFunctions.createResidColl(greennessColl,
    precipColl, aoi));

// Now inspect what you have generated:
print('Module output of residuals', residualColl);
```

Print the resulting `ImageCollection` and inspect the bands and properties in the **Console** (Fig. 53.11).



Inspector Console Tasks

```
▾ImageCollection (35 elements)                                JSON
   type: ImageCollection
   bands: []
  ▾features: List (35 elements)
    ▾0: Image projects/gee-book/assets/A3-8/GreennessColle…
       type: Image
       id: projects/gee-book/assets/A3-8/GreennessCollectio…
       version: 1645831504052359
      ▾bands: List (2 elements)
        ▸0: "residual", float, EPSG:4326, 32x15 px
        ▸1: "greenness", float, EPSG:4326, 1046x477 px
      ▾properties: Object (5 properties)
         system:asset_size: 1797769
         system:index: greenness_1985
         system:time_end: 504921600000
         system:time_start: 473385600000
         year: 1985

    ▸1: Image projects/gee-book/assets/A3-8/GreennessColle…
    ▸2: Image projects/gee-book/assets/A3-8/GreennessColle…
```

Fig. 53.11 Each image in the residualColl image collection contains two bands, residual and greenness. The residual band is what we will pass to LandTrendr

Next, you will filter the `residualColl` collection to map the same years we explored for the observed NDVI (greenness). The code chunk below will pull the image for 1985 (the first year). Use `filterDate` to select the other years you want to view. Add each to the map as new layers.

```
var resids = residualColl.first();
var res1 = resids.select(['residual']);
print(res1.getInfo(), 'residual image');
Map.addLayer(res1, {
    min: -0.2,
    max: 0.2,
    palette: ['red', 'white', 'green']
}, 'residuals 1985', false);
```

Map and compare the residual greenness for a few different years (Fig. 53.12).

Code Checkpoint A38b. The book's repository contains a script that shows what your code should look like at this point.

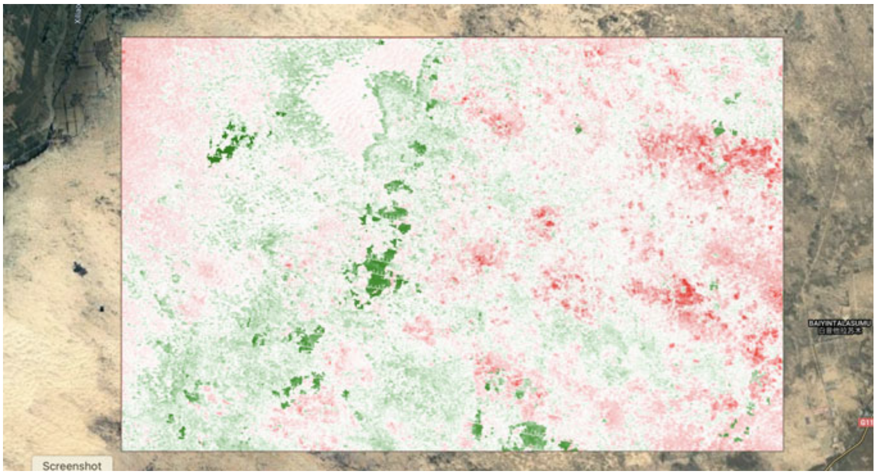


Fig. 53.12 Residual greenness in 1985, after removing the effect of precipitation. Models were fit on a per-pixel basis

Question 7. Compare the layers for residual greenness to the observed greenness values from Question 6. Why are we passing residuals to LandTrendr rather than the observed greenness (NDVI) values?

53.2.3 Section 3: Time Series Segmentation

Now you are ready to apply the LandTrendr time series segmentation algorithm to your Image Collection of residual greeness for the years 1985–2019. This will generate information for each pixel in the study area about how it has changed over time. In order to execute the code, you need to first define pertinent parameters in a dictionary, which you will provide to the LandTrendr algorithm along with your data. Chap. 17 gives an explanation of the LandTrendr algorithm and its parameters. That chapter showed a graphical interface for interacting with LandTrendr; below, we utilize JavaScript functions to execute LandTrendr directly in the code editor.

```
//---- DEFINE RUN PARAMETERS---//
// LandTrendr run parameters
var runParams = {
    maxSegments: 6,
    spikeThreshold: 0.9, //
    vertexCountOvershoot: 3,
    preventOneYearRecovery: true,
    recoveryThreshold: 0.25, //
    pvalThreshold: 0.05, //
    bestModelProportion: 0.75,
    minObservationsNeeded: 10 //
};
```

Follow the next steps to combine the dictionary of parameter settings with the image collection and apply LandTrendr with the API functionality `ee.Algorithms.TemporalSegmentation.LandTrendr`. Then print and explore the output.

```
// Append the image collection to the LandTrendr run
parameter dictionary
var srCollection = residualColl;
runParams.timeSeries = srCollection;

// Run LandTrendr
var lt =
ee.Algorithms.TemporalSegmentation.LandTrendr(runParams);
// Explore the output from running LT
var ltlt = lt.select('LandTrendr');
print(ltlt);
```

The implementation of the LandTrendr Algorithm in GEE generates a multidimensional array containing subarrays for the observation year, observed residuals, fitted residuals, and a Boolean vertex layer that tells the user if a change in pixel trajectory occurred in a given observation year. In order to analyze the outputs year over year, we first must slice out the subarrays of the output multidimensional array, and transform them into image collections. We will not go into detail about slicing arrays in this lesson. For your purposes here, you can just follow along with the provided code. If you wish to learn more about array indexing and slicing, you can refer back to Chap. 18).

```
//---- SLICING OUT DATA -----------------//

// Select the LandTrendr band.
var ltlt = lt.select('LandTrendr');
// Observation Year.
var years = ltlt.arraySlice(0, 0, 1);
// Slice out observed Residual value.
var observed = ltlt.arraySlice(0, 1, 2);
// Slice out fitted Residual values (predicted residual
from final LT model).
var fitted = ltlt.arraySlice(0, 2, 3);
// Slice out the 'Is Vertex' row - yes(1)/no(0).
var vertexMask = ltlt.arraySlice(0, 3, 4);
// Use the 'Is Vertex' row as a mask for all rows.
var vertices = ltlt.arrayMask(vertexMask);
```

Next, we will extract fitted residual values for each pixel in each year from the array slice and convert them to an image with one band per year. First, we need to define a few parameters that we will need to call in future steps.

```
// Define a few params we'll need next:
var startYear_Num = 1985;
var endYear_Num = 2019;
var numYears = endYear_Num - startYear_Num;
var startMonth = '-01-01';
var endMonth = '-12-31';
```

And now we can use the following code block to create a multi-band Image of the residual greeness predicted by LandTrendr for each pixel, with one band per year.

```
// Extract fitted residual value per year, per pixel and
aggregate into an Image with one band per year
var years = [];
for (var i = startYear_Num; i <= endYear_Num; ++i)
years.push(i
    .toString());
var fittedStack = fitted.arrayFlatten([
    ['fittedResidual'], years
]).toFloat();
print(fittedStack, 'fitted stack');
```

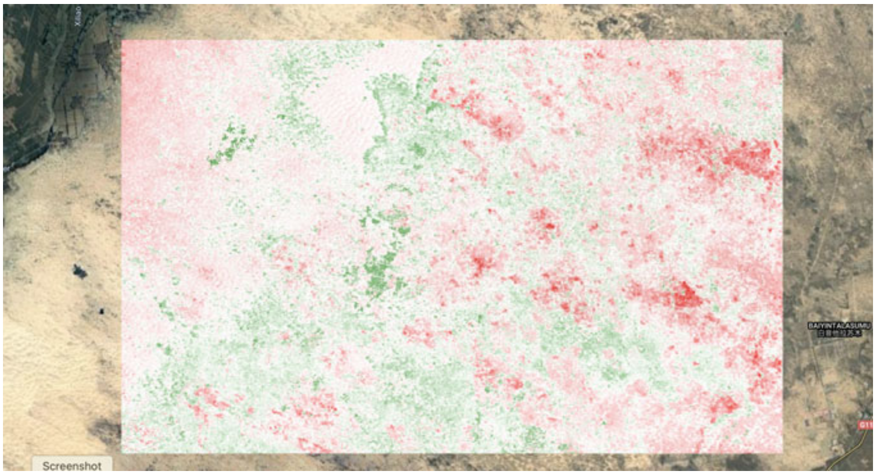


Fig. 53.13 Fitted values for residual greenness in 1985, as predicted by the model fit via LandTrendr

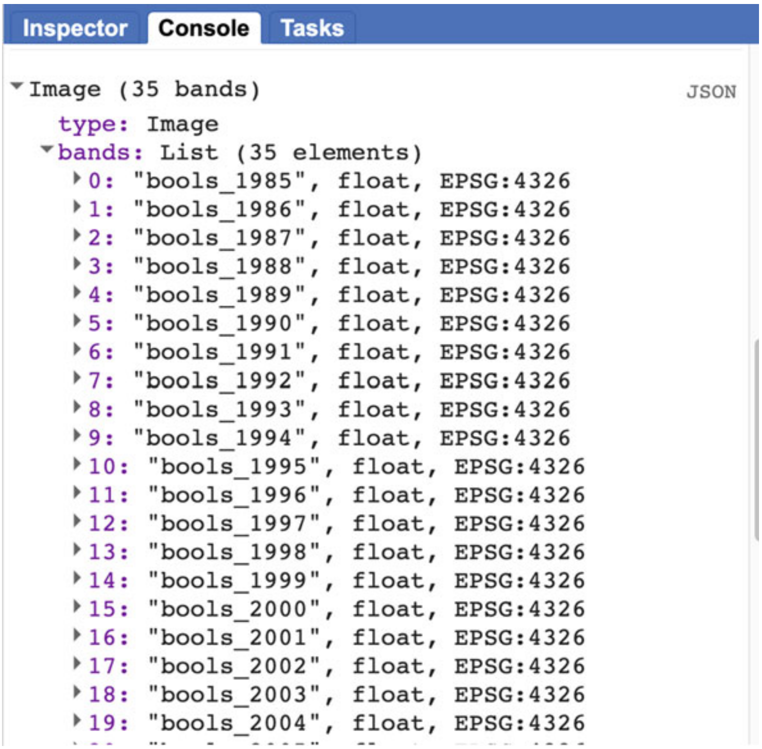
Add the fitted residuals for 1985 as a layer and compare them to the observed values you mapped previously. Notice how the range of values is more constrained than the residuals of observed values shown in Fig. 53.12. This is because the values are predicted from the best-fit model for the series, and thus do not contain the outliers and extreme values we might capture in the observed data.

```
Map.addLayer(fittedStack, {
    bands: ['fittedResidual_1985'],
    min: -0.2,
    max: 0.2,
    palette: ['red', 'white', 'green']
}, 'Fitted Residuals 1985');
```

One of the very useful outputs from LandTrendr is information on whether the algorithm assigned a vertex to a given pixel in a given year. We can generate a raster with a band for each year, which indicates whether or not a pixel had a vertex identified in that year with a Boolean no/yes (0/1). We can use that information to assess how much of the AOI changed in a given year by assessing the prevalence of pixels with a value of 1 (indicating that a vertex was identified). To do this, we need to slice the Boolean information out of the LandTrendr output array, and then assign a year to each band.

```
// Extract Boolean 'Is Vertex?' value per year, per pixel
and aggregate into image w/ Boolean band per year
var years = [];
for (var i = startYear_Num; i <= endYear_Num; ++i)
years.push(i
    .toString());

var vertexStack = vertexMask.arrayFlatten([
    ['bools'], years
]).toFloat();

print(vertexStack.getInfo(), 'vertex Stack');
```

If you print the resulting image, you should see something like Fig. 53.14 in your **Console**. You'll notice that again we have a band for each year, but this time each band is a binary raster, where a value of one (1) indicates that the pixel had a vertex in that year.



```
▾Image (35 bands)
   type: Image
  ▾bands: List (35 elements)
    ▸0: "bools_1985", float, EPSG:4326
    ▸1: "bools_1986", float, EPSG:4326
    ▸2: "bools_1987", float, EPSG:4326
    ▸3: "bools_1988", float, EPSG:4326
    ▸4: "bools_1989", float, EPSG:4326
    ▸5: "bools_1990", float, EPSG:4326
    ▸6: "bools_1991", float, EPSG:4326
    ▸7: "bools_1992", float, EPSG:4326
    ▸8: "bools_1993", float, EPSG:4326
    ▸9: "bools_1994", float, EPSG:4326
    ▸10: "bools_1995", float, EPSG:4326
    ▸11: "bools_1996", float, EPSG:4326
    ▸12: "bools_1997", float, EPSG:4326
    ▸13: "bools_1998", float, EPSG:4326
    ▸14: "bools_1999", float, EPSG:4326
    ▸15: "bools_2000", float, EPSG:4326
    ▸16: "bools_2001", float, EPSG:4326
    ▸17: "bools_2002", float, EPSG:4326
    ▸18: "bools_2003", float, EPSG:4326
    ▸19: "bools_2004", float, EPSG:4326
```

Fig. 53.14 This is how the Boolean value binary rasters should appear in your **Console** once they are sliced out of the LandTrendr. Notice that the observation year has been appended to the name of each raster

In this next step, we will inspect a plot of the mean value of the Boolean (vertex) layers for each year in order to identify which years had the most change. We will estimate the proportion of the AOI that has a vertex identified in each year by mapping a Reducer over a collection to calculate the mean pixel values for each year. A mean of zero would indicate that no vertices were identified in that year, and a mean of one would indicate that all the pixels changed. Of course, neither of these values are likely; most years will have a relatively small number of pixels that change, and thus the value will be low, but not zero.

Charting functions in Earth Engine requires an `ImageCollection`. In the interest of time, we will load this for you as a new asset. If you would like to see an example script for transforming a multi-band image to an `ImageCollection`.

```
// Load an Asset that has the Booleans converted to
Collection
var booleanColl = ee.ImageCollection(
    'projects/gee-book/assets/A3-8/BooleanCollection');
```

Now that we have our Boolean bands in an `ImageCollection` with one image per year, we can run a Reducer over it and create a chart of our results.

```
var chartBooleanMean = ui.Chart.image
    .series({
        imageCollection: booleanColl.select('bools'),
        region: aoi,
        reducer: ee.Reducer.mean(),
        scale: 60,
        xProperty: 'system:time_start'
    })
    .setChartType('ScatterChart')
    .setOptions({
        title: 'Naiman Boolean Mean Per Year',
        vAxis: {
            title: 'Boolean Mean Per Year'
        },
        lineWidth: 1
    });

print(chartBooleanMean);
```

You should end up with a plot in the **Console** that looks something like Fig. 53.15. The largest amount of change in this landscape occurred in 1997, when approximately 20% of the pixels changed trajectory in some way.

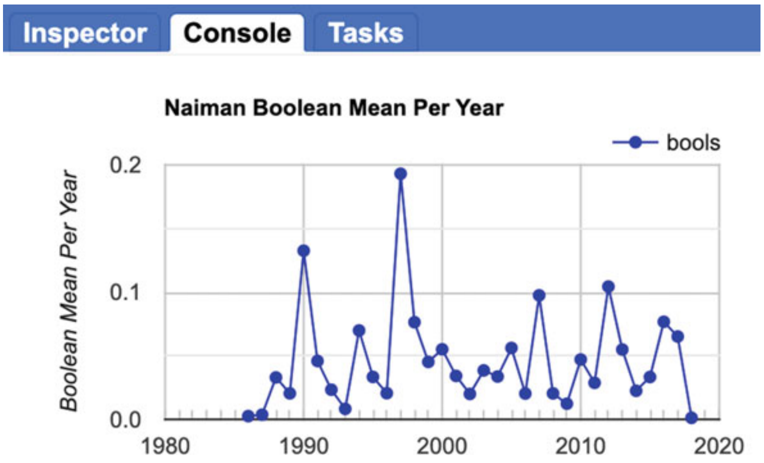


Fig. 53.15 Average pixel value across the study area in each year. A value of one would indicate that all pixels in the image had a vertex identified in that year

Question 9. Which years had the greatest number of pixels with a vertex (indicating a change in the trajectory of the timeseries)? List the four years with the highest proportion of change.

Now that you have identified those years, let's inspect the spatial patterns and locations of those pixels for the major change years. For this, we can use our `vertexStack` image. Set the visualization parameters using the code provided below, then edit the specific year in the `bands` setting to match the year you want to visualize. Figure 53.16 displays the pixels that changed in 1997.

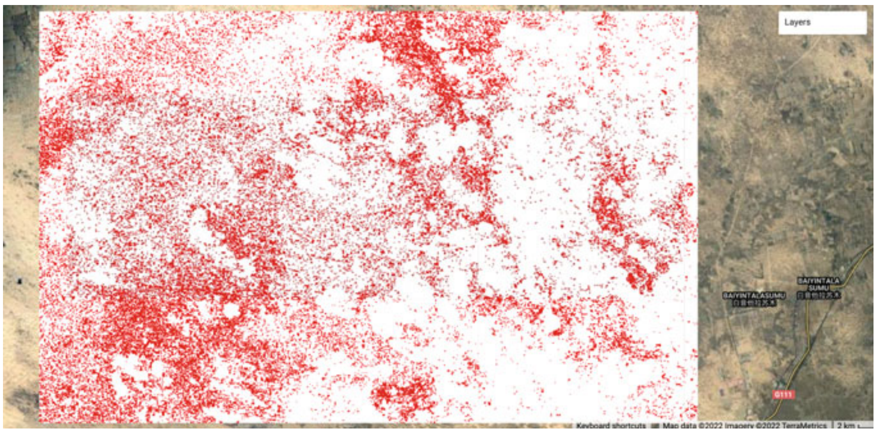


Fig. 53.16 Location of pixels with a vertex identified in 1997

```
// Plot individual years to see the spatial patterns in the
vertices.
var boolParams = {
    // change this for the year you want to view
    bands: 'bools_1997',
    min: 0,
    // no vertex
    max: 1,
    // vertex identified by LT for that year
    palette: ['white', 'red']
};

Map.addLayer(vertexStack, boolParams, 'vertex 1997',
false);
// this visualizes all pixels with a vertex in that year.
```

Run the script several times, changing the year in the bands parameter to match the top change years you identified in the previous step. Take a screenshot of each one and store it so that you can compare them.

Question 10. Do you notice any differences in the spatial patterns of pixels across the top four change years, either in where they are located or their relative patch sizes or aggregation?

Code Checkpoint A38c. The book's repository contains a script that shows what your code should look like at this point.

53.2.4 Section 4: Classify Pixels Based on Similarities in Time Series Trajectories

In the `vertexStack` object that we created, we have information about every time there was a change in trajectory of the time series of residuals for every pixel in the AOI. In this next step, we are going to look for patterns or similarities in those trajectories to characterize different trajectory archetypes, representing unique pixel histories, as the basis for a classification.

In the visualizations above, we focused on the timing and spatial patterns for the top four years of change. If we extract the time series of residual greenness for every pixel in Fig. 53.16, we get something like the plots shown in Fig. 53.17. If we extract the time series of all pixels that changed in 1990, we get something like Fig. 53.17a. If we compare these plots back to our screenshots of the locations of the vertex pixels in those years, we can see that there are distinct patterns both in the spatial location and the shape of the time series between pixels that had a vertex in 1990 and those that changed in 1997 (Fig. 53.17b), or even 2012 (Fig. 53.17c).

Notice the differences in the trajectory of greenness prior to the vertex across the three plots, and that the scale of the *y*-axis varies across the plots as well.

The plots shown in Fig. 53.17 show the median value for each year across all pixels that had a vertex in that year. But it is also possible that a single pixel has more than one vertex in its history (i.e., that it changed trajectory more than once, such as in Fig. 53.2), so there may be some overlap between pixels with a vertex in 1990 and those with a vertex in 1997, or in other years. And it is possible that groups of pixels are changing in similar ways, due to similarities in management,

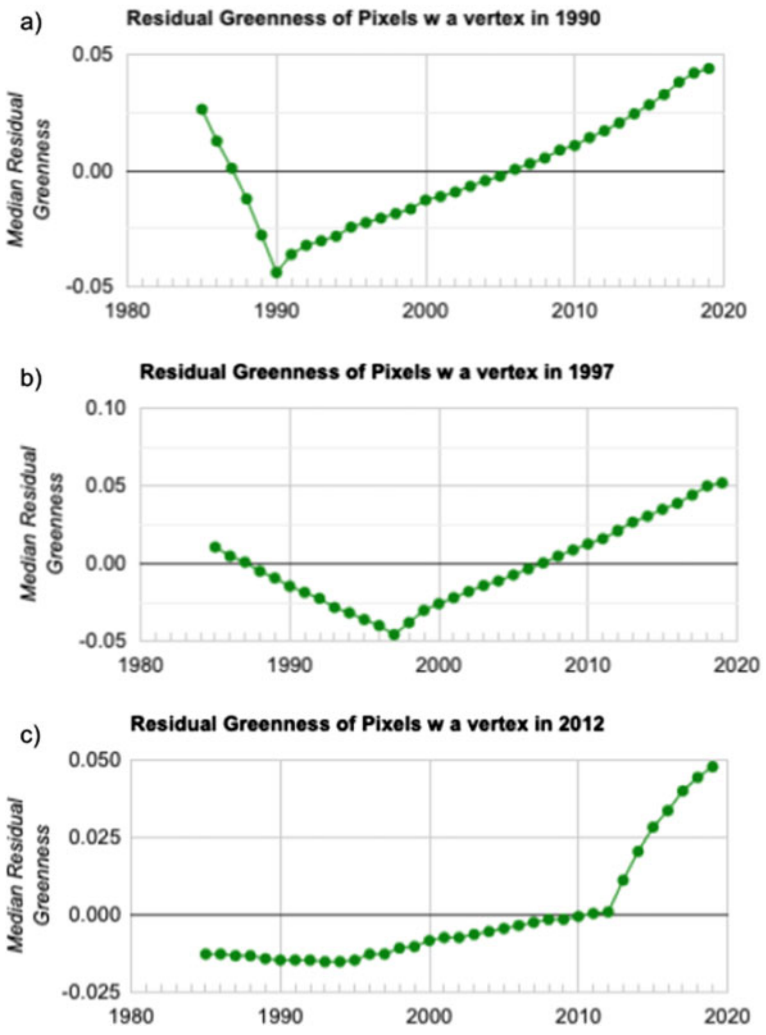


Fig. 53.17 Median residual greenness values for all pixels in the AOI with a vertex identified by LandTrendr in, **a** 1990; **b** 1997; and **c** 2012

soils or other factors. We are going to use all of the information on the changes across all years to characterise the different types of change that have occurred in our study area.

We are going to employ an unsupervised classification approach to clustering these data, as we do not have ground truth or pre-classified training data that we are trying to replicate. Rather, we are interested in finding out what kinds of inherent patterns might exist across the pixels in our AOI, based on similarities in the shape of their trajectories. Different trajectories in the time series of greenness might be due to things like starting conditions, as well as different kinds of management or environmental stress. If you need to review the basics of classification, you can review Chap. 6.

We will start by creating some naive training data from our multiband image of Boolean layers.

```
// Create training data.
var training = vertexStack.sample({
    region: aoi,
    scale: 60,
    numPixels: 5000
});
```

Now, set the maximum number of allowable clusters to 10, train the clusterer, and apply it to the vertex data.

```
var maxclus = 10;

// Instantiate the clusterer and train it.
var trained_clusterer =
ee.Clusterer.wekaKMeans(maxclus).train(
    training);

// Cluster the input using the trained clusterer
var cluster_result =
vertexStack.cluster(trained_clusterer);
```

The default indexing in Java starts at zero, so the first class assigned by the clusterer is labeled with the value 0. This can pose problems if you want to mask out the results to view only one cluster at a time, so we will quickly remap the 0 value to be a 10. Then, we will add the results as a new layer to quickly visualize the classified raster (Fig. 53.18).

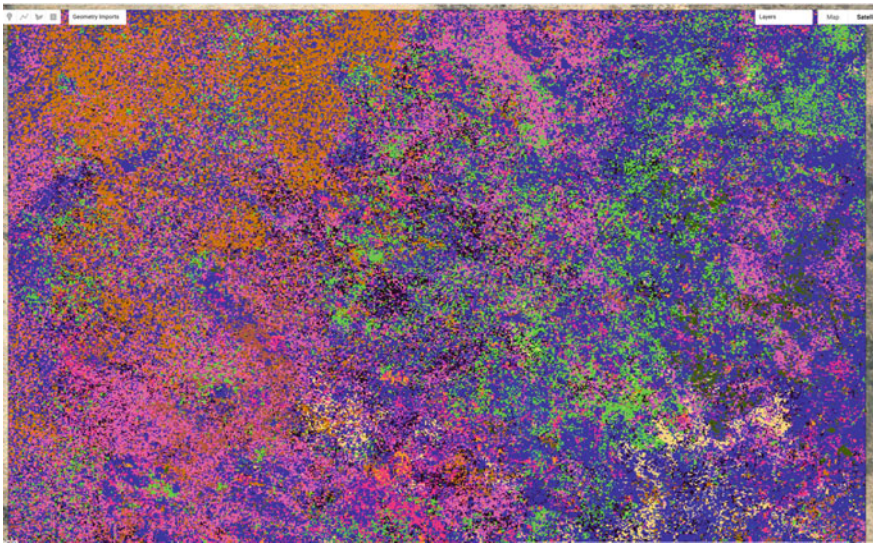


Fig. 53.18 Random-color visualization of unsupervised clusters

```
// Remap result_totalChange so that class 0 is class 10
cluster_result = cluster_result.remap(
        [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
        [10, 1, 2, 3, 4, 5, 6, 7, 8, 9])
    .toFloat()
    .rename('cluster');
Map.addLayer(cluster_result.randomVisualizer(), {}, maxclus
.toString() + '_clusters');
```

Turn on the satellite basemap and move the opacity slider for the classified raster layer `10_clusters`. (The colors assigned to classes in your map may differ.) Some classes seem to align somewhat with observable features in the landscape, but many of them do not. This indicates that there is some similarity in the history of these pixels that is not immediately obvious from their current cover.

Code Checkpoint A38d. The book's repository contains a script that shows what your code should look like at this point.

53.2.5 Section 5: Explore the Characteristics of the New Classes

We have now generated a classified raster based on similarities in the trajectory of the greenness in each pixel. In order to understand what these classes actually mean and what particular trajectories these classes represent, we will create summaries of the median greenness of the pixels in each class and compare them.

We will use the `ImageCollection` of observed greenness that we used in Sect. 53.2.2, and also a collection of the fitted residuals generated by LandTrendr in Sect. 53.2.3. Our first step will be to add a band with the cluster number to those collections.

```
// GOAL: Find Median Greenness for each cluster per year in
the image
// define a function to add the cluster number band to each
Image in the collection
var addClusters = function(img) {
    return img.addBands(cluster_result);
};

// Add the cluster band
var ObvGreen_wClusters = greennessColl.map(addClusters);
```

Next, we need to select and mask out the class we are interested in exploring. We'll just start with the first class.

```
//---Select and mask pixels by cluster number
var cluster_num = 1; // change this to the class of
interest

// Mask all pixels but the selected cluster number
// Define a function so we can map it over the entire
collection
var maskSelCluster = function(img) {
    var selCluster = img.select('cluster').eq(cluster_num);
    return img.mask(selCluster);
};
```

```
// map the function over the entire collection
var selClusterColl =
ObvGreen_wClusters.map(maskSelCluster);

// Use the following to visualize the location of the focal
class:
Map.addLayer(selClusterColl.select('cluster').first(), {
    palette: 'green'
}, 'Cluster ' + cluster_num.toString());
```

Next, you will utilize the `ui.Chart.image` functionality, combined with a Reducer, to plot the median value of observed greenness for each year for the focal class (Fig. 53.19).

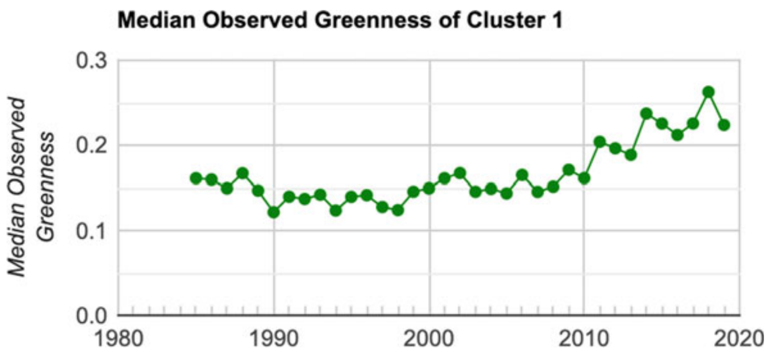


Fig. 53.19 Median value of the observed greenness value for all pixels in the AOI that are identified in Cluster 1

```
var chartClusterMedian = ui.Chart.image.seriesByRegion({
        imageCollection: selClusterColl,
        regions: aoi,
        reducer: ee.Reducer.median(),
        band: 'greenness',
        scale: 90,
        xProperty: 'system:time_start',
        seriesProperty: 'label'
    })
    .setChartType('ScatterChart')
    .setOptions({
        title: 'Median Observed Greenness of Cluster ' +
            cluster_num.toString(),
        vAxis: {
            title: 'Median Observed Greenness'
        },
        lineWidth: 1,
        pointSize: 4,
        series: {
            0: {
                color: 'green'
            },
        }
    });

print(chartClusterMedian);
```

Now we will do the same, but for the fitted residual greenness values predicted from LandTrendr.

```
var fittedresidColl = ee.ImageCollection(
    'projects/gee-book/assets/A3-8/FR_Collection');
// add the cluster number band to each (function defined
above, just use again here)
var fittedresid_wClusters =
fittedresidColl.map(addClusters);

//Mask all pixels but the selected cluster number
// again, function defined above, just call it here
var selFRClusterColl =
fittedresid_wClusters.map(maskSelCluster);

Map.addLayer(selFRClusterColl.select('cluster').first(), {
    palette: ['white', 'blue']
}, 'Cluster ' + cluster_num.toString());

//Chart Median Fitted Residual Values by cluster

var chartClusterMedian = ui.Chart.image.seriesByRegion({
        imageCollection: selFRClusterColl,
        regions: aoi,
        reducer: ee.Reducer.median(),
        band: 'FR',
        scale: 90,
        xProperty: 'system:time_start',
        seriesProperty: 'label'
    }).setChartType('ScatterChart')
    .setOptions({
        title: 'Median Fitted Residual Greenness of Cluster
' +
            cluster_num.toString(),
        vAxis: {
            title: 'Median Residual Greenness'
        },
        lineWidth: 1,
        pointSize: 4,
        series: {
            0: {
                color: 'red'
            },
        }
    });
print(chartClusterMedian);
```

From the **Console**, expand each of the two plots to a new tab, then download the data as a.csv file and rename it something like "observed_green_Class1.csv." **Repeat the steps above for each of the classes.**

Figure 53.20 provides an example of the outputs for a few of the classes that you have generated. When viewed side by side, it is easier to see how pixels in some parts of the AOI have experienced different trajectories of vegetation cover over time and that the exact time and nature of the shifts in their trajectories are also different. This may be due to differences in underlying vegetation, land use, and management activity.

In a final step, we will compare the classes you generated by clustering the time series data to how the same locations are classified over time by the MODIS land cover data. We will do this by picking a few points within the AOI and plotting the land cover type number for each year of the MODIS data (see Table 53.1 to link the pixel value to the descriptions of the class types).

Use the Inspector tool to select a pixel in a region that interests you. Copy the code chunk below to your script. In the Inspector window, expand the information by clicking on the triangle next to Point and copy the longitude and latitude values, and replace the values in the code chunk below with your values.

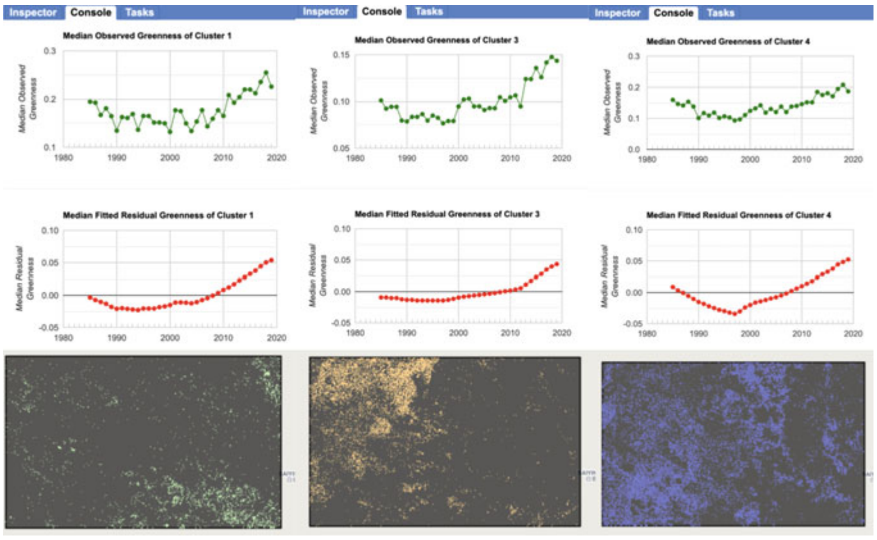

Inspector
Console
Tasks
Median Observed Greenness of Cluster 1
Median Observed Greenness of Cluster 3
Median Observed Greenness of Cluster 4
Median Observed Greenness
Median Fitted Residual Greenness of Cluster 1
Median Fitted Residual Greenness of Cluster 3
Median Fitted Residual Greenness of Cluster 4
Median Residual Greenness
1980
1990
2000
2010
2020

Fig. 53.20 Example outputs for three of the clusters identified in the data. The top row is the median greenness value for all pixels in the cluster. The middle row is the residual greenness value predicted by the fitted LandTrendr model. The bottom row is the location of all pixels in the AOI within that class; cluster 1 is more common in the southeast, cluster 3 is aggregated in the northwestern region, and cluster 4 has patches throughout the AOI

```
// Generate a point geometry.
var expt = ee.Geometry.Point(
    [120.52062120781073, 43.10938146169287]);
// Convert to a Feature.
var point = ee.Feature(expt, {});
```

The last line in the code chunk above converts the point to a feature. Now you can run a Reducer over that point to extract values and plot them to a chart (Fig. 53.21).

```
// Create a time series chart of MODIS Classification:
var chart_LC = ui.Chart.image.seriesByRegion(
        MODIS_LC, point, ee.Reducer.mean(), 'LC_Type1', 30,
        'system:time_start', 'label')
    .setChartType('ScatterChart')
    .setOptions({
        title: 'LC of Selected Pixels',
        vAxis: {
            title: 'MODIS landcover'
        },
        lineWidth: 1,
        pointSize: 4
    });

print(chart_LC);
```

Repeat the steps above for a few points of interest to you and save screenshots of the plots.

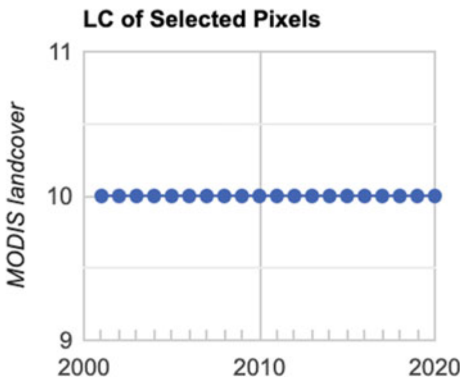


Fig. 53.21 Example output of the MODIS MCD12Q1 landcover class values for a single point in the AOI. Class 10 is "Grasslands: dominated by herbaceous annuals (< 2 m)"

Code Checkpoint A38e. The book's repository contains a script that shows what your code should look like at this point.

53.3 Synthesis

Assignment 1. Combine all of your downloaded data files to create a single plot (in Excel or Google Sheets, for example) of the trajectory of median observed greenness for each of the classes. Compare each of these to the spatial patterns of the different classes (as in Fig. 53.17 and in your Map view), and to the underlying satellite image view. Pick three of the classes and describe the general trend in the time series (how has greenness changed over time?) and the spatial distribution of the pixels in that class.

We implemented this by specifying that the clusterer should look for 10 classes in the data. In a true implementation, we would want to explore the outputs across a range of cluster numbers. We may have forced the algorithm to split the data into too many (or too few) groups. Based on your inspection of the timing of the vertices and the spatial distribution of the final classes, are there any that you think could be grouped together in a final classification? What would you estimate to be the final number of classes in these data?

How much variation over time has there been for the different points according to the MODIS data? How does this compare to the variation in greenness represented in the final class that you generated?

53.4 Conclusion

In this module, you explored a new approach to classifying land cover that is based on the temporal trajectory of individual pixels. Earth Engine is a valuable tool for this analysis because you are able to access the historical archive of imagery and climate data, as well as the computational tools needed to process, analyze, and classify these data. You learned how to create new derived datasets to use both as inputs to the analysis and as a final classified product. By comparing the temporal trajectories of the new classes against traditional land cover data, you learned how to distinguish the pros and cons of existing datasets for meeting land cover mapping objectives. Now that you understand the basics of the challenges of detecting land cover change in rangelands and have explored a new approach to classifying different trajectories, you can apply this approach to your own areas of interest to better understand the history of response.

References

Alexandratos N, Bruinsma J (2012) World agriculture towards 2030/2050: the 2012 revision. In: ESA Work Paper, vol 12, p 146. https://doi.org/10.22004/ag.econ.288998

Arino O, Bicheron P, Achard F et al (2008) GlobCover: The most detailed portrait of Earth. Eur Space Agency Bull 2008:24–31

Asner GP, Elmore AJ, Olander LP et al (2004) Grazing systems, ecosystem responses, and global change. Ann Rev Environ Resour 29:261–299. https://doi.org/10.1146/annurev.energy.29.062403.102142

Bontemps S, Defourny P, Van Bogaert E et al (2011) GLOBCOVER 2009: Products description and validation report. ESA Bull 136:53

Briske DD (2017) Rangeland systems: processes, management and challenges. Springer Nature

Campbell DJ, Lusch DP, Smucker TA, Wangui EE (2005) Multiple methods in the study of driving forces of land use and land cover change: a case study of SE Kajiado District, Kenya. Hum Ecol 33:763–794. https://doi.org/10.1007/s10745-005-8210-y

Cao S, Sun G, Zhang Z et al (2011) Greening China naturally. Ambio 40:828–831. https://doi.org/10.1007/s13280-011-0150-8

Derner JD, Hunt L, Ritten J et al (2017) Livestock production systems. Rangeland systems. Springer, Cham, pp 347–372

Fan P, Chen J, John R (2016) Urbanization and environmental change during the economic transition on the Mongolian Plateau: Hohhot and Ulaanbaatar. Environ Res 144:96–112. https://doi.org/10.1016/j.envres.2015.09.020

Friedl MA, McIver DK, Hodges JCF et al (2002) Global land cover mapping from MODIS: algorithms and early results. Remote Sens Environ 83:287–302. https://doi.org/10.1016/S0034-4257(02)00078-0

Friedl MA, Sulla-Menashe D, Tan B et al (2010) MODIS collection 5 global land cover: algorithm refinements and characterization of new datasets. Remote Sens Environ 114:168–182. https://doi.org/10.1016/j.rse.2009.08.016

Ganguly S, Friedl MA, Tan B et al (2010) Land surface phenology from MODIS: characterization of the collection 5 global land cover dynamics product. Remote Sens Environ 114:1805–1816. https://doi.org/10.1016/j.rse.2010.04.005

García-Mora TJ, Mas JF, Hinkley EA (2012) Land cover mapping applications with MODIS: a literature review. Int J Digit Earth 5:63–87. https://doi.org/10.1080/17538947.2011.565080

Healey SP, Cohen WB, Yang Z et al (2018) Mapping forest change using stacked generalization: an ensemble approach. Remote Sens Environ 204:717–728. https://doi.org/10.1016/j.rse.2017.09.029

Homer C, Dewitz J, Yang L et al (2015) Completion of the 2011 national land cover database for the conterminous United States—representing a decade of land cover change information. Photogramm Eng Remote Sensing 81:345–354

Huang J, Yu H, Guan X et al (2016) Accelerated dryland expansion under climate change. Nat Clim Chang 6:166–171. https://doi.org/10.1038/nclimate2837

John R, Chen J, Lu N, Wilske B (2009) Land cover/land use change in semi-arid Inner Mongolia: 1992–2004. Environ Res Lett 4:45010. https://doi.org/10.1088/1748-9326/4/4/045010

Kennedy RE, Yang Z, Cohen WB (2010) Detecting trends in forest disturbance and recovery using yearly Landsat time series: 1. LandTrendr—temporal segmentation algorithms. Remote Sens Environ 114:2897–2910. https://doi.org/10.1016/j.rse.2010.07.008

Kennedy RE, Andréfouët S, Cohen WB et al (2014) Bringing an ecological view of change to Landsat-based remote sensing. Front Ecol Environ 12:339–346. https://doi.org/10.1890/130066

Lambin EF, Meyfroidt P (2011) Global land use change, economic globalization, and the looming land scarcity. Proc Natl Acad Sci USA 108:3465–3472. https://doi.org/10.1073/pnas.1100480108

Lang W, Chen T, Li X (2016) A new style of urbanization in China: transformation of urban rural communities. Habitat Int 55:1–9. https://doi.org/10.1016/j.habitatint.2015.10.009

Li WJ, Ali SH, Zhang Q (2007) Property rights and grassland degradation: a study of the Xilingol Pasture, Inner Mongolia, China. J Environ Manag 85:461–470. https://doi.org/10.1016/j.jenvman.2006.10.010

Liu M, Dries L, Heijman W et al (2015) Tragedy of the commons or tragedy of privatisation? The impact of land tenure reform on grassland condition in Inner Mongolia, China. In: International conference of agricultural economists, pp 9–14

Melillo JM, Richmond TT, Yohe G (2014) Climate change impacts in the United States. US Global Change Research Program Washington, DC

Millenium Ecosystem Assessment (2005) Ecosystems and human well-being: desertification synthesis. Island Press Washington, DC

Prestele R, Alexander P, Rounsevell MDA et al (2016) Hotspots of uncertainty in land-use and land-cover change projections: a global-scale model comparison. Glob Chang Biol 22:3967–3983. https://doi.org/10.1111/gcb.13337

Reid RS, Kruska RL, Muthui N et al (2000) Land-use and land-cover dynamics in response to changes in climatic, biological and socio-political forces: the case of Southwestern Ethiopia. Landscape Ecol 15:339–355. https://doi.org/10.1023/A:1008177712995

Schneider A, Mertes CM, Tatem AJ et al (2015) A new urban landscape in East-Southeast Asia, 2000–2010. Environ Res Lett 10:34002. https://doi.org/10.1088/1748-9326/10/3/034002

Sleeter BM, Sohl TL, Loveland TR et al (2013) Land-cover change in the conterminous United States from 1973 to 2000. Glob Environ Change 23:733–748. https://doi.org/10.1016/j.gloenvcha.2013.03.006

Zanaga D, Van De Kerchove R, De Keersmaecker W et al. (2021) ESA WorldCover 10 m 2020 v100. Meteosat Second Generation Evapotranspiration, vol 1–27. https://doi.org/10.5281/zenodo.5571936

Zhu Z, Woodcock CE (2014) Continuous change detection and classification of land cover using all available Landsat data. Rem Sens Environ 144: 152–171

54 Conservation I—Assessing the Spatial Relationship Between Burned Area and Precipitation

Harriet Branson and Chelsea Smith

Overview

The purpose of this chapter is to introduce the need for fire activity and rainfall trend monitoring to inform conservation management practices.

Practical conservation requires an understanding of key environmental factors such as fire and rainfall, which impact the amount of forage and habitat available for a variety of species. This chapter will guide you through how to create fire and rainfall time series and visualize this data. At the end of this chapter, you will be able to present this information in an accessible way on a graph to help inform conservation management practices such as early burning and supplementary feeding.

Learning Outcomes

- Understanding why fire and rainfall trends are useful in conservation management.
- Creating a time series of burned areas.
- Writing a function to map areal mean rainfall calculations over an `ImageCollection`.
- Generating an interactive graph displaying fire and rainfall trends over a 10-year period.

Helps if you know how to

- Import images and image collections, filter, and visualize (Part I).

H. Branson (✉) · C. Smith
Fauna & Flora International, David Attenborough Building, Pembroke St., Cambridge CB2 3QZ, UK
e-mail: Harriet.Branson@fauna-flora.org

C. Smith
e-mail: chelsea.v.smith@fauna-flora.org

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_54

- Create a graph using `ui.Chart` (Chap. 4).
- Summarize an image with `reduceRegion` (Chap. 9).
- Write a function and `map` it over an `ImageCollection` (Chap. 12).
- Work with CHIRPS rainfall data (Chap. 14).
- Mask cloud, cloud shadow, snow/ice, and other undesired pixels (Chap. 15).

54.1 Introduction to Theory

Globally, biodiversity is under threat from changing climates, habitat loss, and fragmentation. Conservation work to protect, manage, and restore ecosystems is vitally important to maintain global biodiversity, which supports climate regulation and a host of other ecosystem services we depend upon (Reddy 2021).

Remote sensing offers a valuable tool to collect, analyze, and display data over an entire ecosystem. Environmental variables such as fire and rainfall have direct impacts on habitat and forage availability (Holden et al. 2018). Monitoring of these variables can help inform conservation, such as fire management to reduce the severity of fires and amount of habitat lost (Ribeiro et al. 2021). Monitoring rainfall patterns can help conservationists understand which climate conditions species prefer in their habitats and if the conditions are at risk of changing (Pinto-Ledezma and Cavender-Bares 2021).

In this chapter, we will see how using the Earth Engine platform allows conservationists to scope the fire and rainfall conditions for any site using global datasets. The data we visualize is being directly used in real-world conservation by informing fire management, as well as indirectly through the identification of climate conditions to which species are adapted.

54.2 Practicum

54.2.1 Section 1: Assess Area of Interest

The first step is to upload and explore the area of interest. Northern Mozambique's Niassa Reserve is home to 40% of the country's entire elephant population, and is a haven for two of Africa's threatened carnivores: lion, and wild dog. Fire is a key ecological process in forested savanna ecosystems that are prevalent in the Niassa Reserve, and the knowledge of the fire regime is an important factor in forest fire management (Nhongo et al. 2020). Our area of interest (AOI) has very stark wet and dry seasons; take a look at the satellite imagery basemap that shows the landscape in the wet season.

```
// ** Upload the area of interest ** //
var AOI = ee.Geometry.Polygon([
    [
        [37.72, -11.22],
        [38.49, -11.22],
        [38.49, -12.29],
        [37.72, -12.29]
    ]
]);
Map.centerObject(AOI, 9);
Map.addLayer(AOI, {
    color: 'white'
}, 'Area of interest');
```

54.2.2 Section 2: Load the MODIS Burned Area Dataset

Next, we will use the MCD64A1 dataset, which is a global layer representing burned area at 500 m resolution from 2001 to the present. The layer is also accompanied by a band named `'BurnDate'`, which enables you to disaggregate into daily fire data.

First, we will filter the MODIS `ImageCollection` based on the timespan requirements. We are looking at fire patterns over the past 10 years. The MCD64A1 dataset comes with three main bands, `'BurnDate'`, `'Uncertainty'`, and `'QA'` (quality assurance). In this case, we select only the `'BurnDate'` band, which associates each pixel of burned area with a day-of-year value.

```
// ** MODIS Monthly Burn Area ** //

// Load in the MODIS Monthly Burned Area dataset.
var dataset = ee.ImageCollection('MODIS/006/MCD64A1')
    // Filter based on the timespan requirements.
    .filter(ee.Filter.date('2010-01-01', '2021-12-31'));

// Select the BurnDate band from the images in the
collection.
var MODIS_BurnDate = dataset.select('BurnDate');
```

Next, we can create the function that will calculate the area of pixels associated with each day. The function is structured so that it will map over each image in the `MODIS_BurnDate ImageCollection`. It begins by using the command pixelArea, which calculates the area of each pixel in square meters. To limit the calculation to specifically burned areas, we need to use updateMask with our input img as the variable. Because we want our final area calculation to be in square kilometers instead of square meters, we divide the value by 1,000,000 (1e6).

The reduceRegion command gathers the sum of burned area within our input area of interest at the correct scale of the input image (500 m for our MODIS_BurnDate collection). After this, the getNumber command recalls the area calculation per image per day of year and appends it as a new band area on the ImageCollection.

Since the MODIS ImageCollection has a 'system:time_start' embedded within each image, we can select the area band and continue without the 'BurnDate'.

```
// A function that will calculate the area of pixels in
each image by date.
var addArea = function(img) {
    var area = ee.Image.pixelArea()
        .updateMask(
            img
        ) // Limit area calculation to areas that have
burned data.
        .divide(1e6) // Divide by 1,000,000 for square
kilometers.
        .clip(AOI) // Clip to the input geometry.
        .reduceRegion({
            reducer: ee.Reducer.sum(),
            geometry: AOI,
            scale: 500,
            bestEffort: true
        }).getNumber(
            'area'
        ); // Retrieve area from the reduce region
calculation.
    // Add a new band to each image in the collection
named area.
    return img.addBands(ee.Image(area).rename('area'));
};

// Apply function on image collection.
var burnDateArea = MODIS_BurnDate.map(addArea);

// Select only the area band as we are using system time
for date.
var burnedArea = burnDateArea.select('area');
```

To show the total amount of burned area over the past 10 years, we can plot this on a time-series graph. By defining the `xProperty` with `'system:time_start'` for the date variable on this graph, we can plot the total area burned with time.

```
// Create a chart that shows the total burned area over
time.
var burnedAreaChart =
    ui.Chart.image
    .series({
        imageCollection: burnedArea, // Our image
collection.
        region: AOI,
        reducer: ee.Reducer.mean(),
        scale: 500,
        xProperty: 'system:time_start' // time
    })
    .setSeriesNames(['Area']) // Label for legend.
    .setOptions({
        title: 'Total monthly area burned in AOI',
        hAxis: {
            title: 'Date', // The x axis label.
            format: 'YYYY', // Years only for date format.
            gridlines: {
                count: 12
            },
            titleTextStyle: {
                italic: false,
                bold: true
            }
        },
        vAxis: {
            title: 'Total burned area (km²)', // The y-axis
label
            maxValue: 2250, // The bounds for y-axis
            minValue: 0,
            titleTextStyle: {
                italic: false,
                bold: true
            }
        },
        lineWidth: 1.5,
        colors: ['d74b46'], // The line color
    });
print(burnedAreaChart);
```

Code Checkpoint A39a. The book's repository contains a script that shows what your code should look like at this point.

Question 1. We have calculated the area burned in square kilometers; which line of code would you have to change, and how, to calculate the area burned in hectares?

Question 2. By hovering your mouse over the graph, which month has the greatest area burnt, and does this vary year on year?

Question 3. The most appropriate fire dataset and analysis to use for your study will vary depending on location and scale of analysis. If you wanted to understand the impact of fires on koala habitat in southern Australia, what would be the most appropriate analysis? Hint: Consider the analysis run in Chap. 46 and the work by Bonney et al. (2020).

54.2.3 Section 3: Areal Mean Rainfall Time Series

To inform habitat management and further understand fire patterns within northern Mozambique, we have to also consider patterns in rainfall. To do this, we will use the Climate Hazards Group InfraRed Precipitation with Station (CHIRPS) Precipitation dataset. This dataset measures precipitation levels every five days from 1981 to present day at 500 m resolution to make a long-term quasi-global dataset.

First, we will define our temporal range (matching our burned area data) from 2010 to 2021, and set the advancing dates from these years. After this, we create a sequence of years and months that will be used to filter the dataset chronologically.

After setting these parameters, filter the CHIRPS dataset using the start and end date, and sort chronologically in descending order using the same `'system:time_start'` property used in Sect. 54.2.1. Filter the bounds to the AOI, and select the precipitation band from the `ImageCollection`.

```
// Load in the CHIRPS rainfall pentad dataset.
var chirps = ee.ImageCollection('UCSB-CHG/CHIRPS/PENTAD');

// Define the temporal range
var startyear = 2010;
var endyear = 2021;

// Set the advancing dates from the temporal range.
var startdate = ee.Date.fromYMD(startyear, 1, 1);
var enddate = ee.Date.fromYMD(endyear, 12, 31);
```

```
// Create a list of years
var years = ee.List.sequence(startyear, endyear);
// Create a list of months
var months = ee.List.sequence(1, 12);

// Filter the dataset based on the temporal range.
var Pchirps = chirps.filterDate(startdate, enddate)
    .sort('system:time_start',
        false) // Sort chronologically in descending
order.
    .filterBounds(AOI) // Filter to AOI
    .select('precipitation'); // Select precipitation band
```

Once the dataset has been filtered, we can calculate the monthly precipitation using a function. The function maps the input, `y`, over the list of years generated above. The function then returns the total precipitation for the month, alongside a date variable and the `′system:time_start′` property. The command `millis` is used to keep the system number that refers to the date collected.

Print the `ImageCollection` for checking.

```
// Calculate the precipitation per month.
var MonthlyRainfall = ee.ImageCollection.fromImages(
    years.map(function(
        y
    ) { // Using the list of years based on temporal range.
        return months.map(function(m) {
            var w = Pchirps.filter(ee.Filter
                    .calendarRange(y, y, 'year'))
                .filter(ee.Filter.calendarRange(m, m,
                    'month'))
                .sum(); // Calculating the sum for the
                        month
            return w.set('year', y)
                .set('month', m)
                .set('system:time_start', ee.Date
                    .fromYMD(y, m, 1).millis()
                ) // Use millis to keep the system time
                      number.
                .set('date', ee.Date.fromYMD(y, m,
                    1));
        });
    }).flatten());
// Print the image collection.
print('Monthly Precipitation Image Collection',
MonthlyRainfall);
```

Once the monthly precipitation levels have been calculated, we can use a reducer to calculate the mean total rainfall across our AOI, also known as the areal mean rainfall (AMR), and plot these on a time-series graph. This process is very similar to the burned area chart in Sect. 54.2.2.

```
// ** Chart: CHIRPS Precipitation ** //

// Create a chart displaying monthly rainfall over a
temporal range.
var monthlyRainfallChart =
    ui.Chart.image
    .series({
        imageCollection: MonthlyRainfall.select(
            'precipitation'), // Select precipitation band
        region: AOI,
        reducer: ee.Reducer
            .mean(), // Use mean reducer to calculate AMR
        scale: 500,
        xProperty: 'system:time_start' // Use system time
start for x-axis
    })
    .setSeriesNames(['Precipitation']) // /The label legend
    .setOptions({
        title: 'Total monthly precipitation in AOI', // Add
title
        hAxis: {
            title: 'Date',
            format: 'YYYY', // Year only date format
            gridlines: {
                count: 12
            },
            titleTextStyle: {
                italic: false,
                bold: true
            }
        },
        vAxis: {
            title: 'Precipitation (mm)', // The y-axis
label
            maxValue: 450, // The bounds for y-axis
            minValue: 0,
            titleTextStyle: {
                italic: false,
                bold: true
            }
        },
        lineWidth: 1.5,
        colors: ['4f5ebd'],
    });
print(monthlyRainfallChart);
```

Print the monthly rainfall chart. Using the chart, we can see which months receive rainfall, and can categorize these as the wet season. With data from the chart, we will categorize any month that receives over 5 mm of rainfall as a wet season month. We can then calculate total seasonal rainfall across our AOI by aggregating wet season months together.

```
// 2010/2011 wet season total
var year = 2010; // Adjust year
var startDate = ee.Date.fromYMD(year, 11, 1); // Adjust
months/days
var endDate = ee.Date.fromYMD(year + 1, 5, 31); // Adjust
months/days
var filtered = chirps
    .filter(ee.Filter.date(startDate, endDate));
var Rains10_11Total =
filtered.reduce(ee.Reducer.sum()).clip(AOI);

// 2011/2012 wet season total
var year = 2011; // Adjust year
var startDate = ee.Date.fromYMD(year, 11, 1); // Adjust
months/days
var endDate = ee.Date.fromYMD(year + 1, 5, 31); // Adjust
months/days
var filtered = chirps
    .filter(ee.Filter.date(startDate, endDate));
var Rains11_12Total =
filtered.reduce(ee.Reducer.sum()).clip(AOI);
```

Question 4. Classify the remaining wet seasons using the areal mean rainfall chart, and calculate the total seasonal rainfall over the AOI using the code you have just learned, to calculate wet seasons in 2010–2011 and 2011–2012.

Question 5. We have created a 10-year time series. Is this a long enough time period to start to consider changes in rainfall patterns?

Code Checkpoint A39b. The book's repository contains a script that shows what your code should look like at this point.

54.2.4 Section 4: Visualizing Fire and Rainfall Time Series

Now that we have visualized patterns in both burned area and precipitation separately, it is important to assess these patterns together. To do this, we need to combine the two image collections in order to plot them on the same graph.

Once they are combined, we can begin creating a multi-variable time-series chart. By keeping the `'system:time_start'` property on both sets of image collections, we are able to easily plot each variable temporally. Following this, we set both series' names and set `interpolateNulls` to true to provide continuous precipitation data that can be plotted alongside the near daily burned area data.

To enable two y-axes, we need to set some series parameters. By defining our `targetAxisIndex` as 0 and 1, we can then set our `vAxes` to match, using two sets of parameters with different labels and different bounds for ease of plotting. This is useful since the burned area and precipitation datasets have different minimum and maximum values, so it would be difficult to analyze on the same axis.

You can print the chart to the **Console** for export if necessary, but for interactivity, we will add it to the map later.

```
// ** Combine: CHIRPS Average Rainfall & MODIS Monthly Burn
** //

// Combine the two image collections for joint analysis
var bpMerged = burnedArea.merge(MonthlyRainfall);
print('Merged image collection', bpMerged);

// ** Chart: CHIRPS Average Rainfall & MODIS Monthly Burn
** //
// Plot the two time series on a graph
var bpChart =
    ui.Chart.image.series({
        imageCollection: bpMerged, // The merged image
collection
        region: AOI,
        reducer: ee.Reducer.mean(),
        scale: 500,
        xProperty: 'system:time_start' // Use system time
start for synchronous plotting
    })
    .setSeriesNames(['Burned Area', 'Precipitation']) //
Label series
    .setChartType('LineChart') // Define chart type
    .setOptions({
        title: 'Relationship between burned area and
rainfall in Chuilexi',
        interpolateNulls: true, // Interpolate nulls to
provide continuous data
        series: { // Use two sets of series with a target
axis to create the two y-axes needed for plotting
            0: { // 0 and 1 reference the vAxes settings
below
                targetAxisIndex: 0,
                type: 'line',
                lineWidth: 1.5,
                color: 'd74b46'
            },
            1: {
                targetAxisIndex: 1,
                type: 'line',
                lineWidth: 1.5,
                color: '4f5ebd'
            },
        },
        hAxis: {
            title: 'Date',
            format: 'YYYY',
            gridlines: {
                count: 12
            },
            titleTextStyle: {
                italic: false,
                bold: true
            }
        },
        vAxes: {
            0: {
                title: 'Burned area (km²)', // Label left-
hand y-axis
                baseline: 0,
                viewWindow: {
                    min: 0
                },
                titleTextStyle: {
                    italic: false,
                    bold: true
                }
            },
            1: {
                title: 'Precipitation (mm)', // Label
right-hand y-axis
                baseline: 0,
                viewWindow: {
                    min: 0
                },
                titleTextStyle: {
                    italic: false,
                    bold: true
                }
            },
        },
        curveType: 'function' // For smoothing
    });
bpChart.style().set({
    position: 'bottom-right',
    width: '492px',
    height: '300px'
});
```

Once we have created our final chart that displays burned area and precipitation, we can build some legends on the map for the spatial data. Using two different functions, we can create a horizontal legend with a set gradient palette and custom markers that correspond to the precipitation data.

The burned area legend is simpler in that we are creating a square of red to indicate that any pixel marked red on the image was burned at that particular time point.

```
// ** Legend: Rainfall ** //
var rain_palette = ['#ffffcc', '#a1dab4', '#41b6c4',
'#2c7fb8',
    '#253494'
];

function ColorBar(rain_palette) {
    return ui.Thumbnail({
        image: ee.Image.pixelLonLat().select(0),
        params: {
            bbox: [0, 0, 1, 0.1],
            dimensions: '300x15',
            format: 'png',
            min: 0,
            max: 1,
            palette: rain_palette,
        },
        style: {
            stretch: 'horizontal',
            margin: '0px 22px'
        },
    });
}

function makeRainLegend(lowLine, midLine, highLine,
lowText, midText,
    highText, palette) {
    var labelheader = ui.Label(
        'Total precipitation in wet season (mm)', {
            margin: '5px 17px',
            textAlign: 'center',
            stretch: 'horizontal',
            fontWeight: 'bold'
        });
    var labelLines = ui.Panel(
        [
            ui.Label(lowLine, {
                margin: '-4px 21px'
            }),
            ui.Label(midLine, {
                margin: '-4px 0px',
                textAlign: 'center',
                stretch: 'horizontal'
            }),
            ui.Label(highLine, {
                margin: '-4px 21px'
            })
        ],
        ui.Panel.Layout.flow('horizontal'));
    var labelPanel = ui.Panel(
        [
            ui.Label(lowText, {
                margin: '0px 14.5px'
            }),
            ui.Label(midText, {
                margin: '0px 0px',
                textAlign: 'center',
                stretch: 'horizontal'
            }),
            ui.Label(highText, {
                margin: '0px 1px'
            })
        ],
        ui.Panel.Layout.flow('horizontal'));
    return ui.Panel({
        widgets: [labelheader, ColorBar(rain_palette),
            labelLines, labelPanel
        ],
        style: {
            position: 'bottom-left'
        }
    });
}
Map.add(makeRainLegend('|', '|', '|', '0', '250', '500',
['#ffffcc',
    '#a1dab4', '#41b6c4', '#2c7fb8', '#253494'
]));

// ** Legend: Burned area ** //
var burnLegend = ui.Panel({
    style: {
        position: 'top-left',
        padding: '8px 15px'
    }
```

```
});

var makeRow = function(color, name) {
    var colorBox = ui.Label({
        style: {
            backgroundColor: '#' + color,
            padding: '10px',
            margin: '0 10px 0 0'
        }
    });
    var description = ui.Label({
        value: name,
        style: {
            margin: 'o o 6px 6px'
        }
    });
    return ui.Panel({
        widgets: [colorBox, description],
        layout: ui.Panel.Layout.Flow('horizontal')
    });
};

var burnPalette = ['FF0000'];
var names = ['Burned area'];
for (var i = 0; i < 1; i++) {
    burnLegend.add(makeRow(burnPalette[i], names[i]));
}
Map.add(burnLegend);
```

Now that we have our legends complete, we can add our double variable time-series chart to the map, and center the map on our area of interest.

```
Map.centerObject(AOI, 9); // Centre the map on the AOI
Map.add(
    bpChart
); // Add the merged burned area & precipitation chart to
the map
```

While visualizing the data on a static chart is useful for further analysis and identifying patterns, it is also useful to see the spatial data corresponding to a particular date or fire. To do this, we can add an interactive element to the chart. If you click a particular point on the chart, it will reveal the corresponding image for burned area and precipitation on the map.

To do this, we need to create a function that uses the burned area and precipitation chart (`bpChart`) and executes `onClick`, displaying the relevant data based on the input values. By utilizing the `'system:time_start'` property, we can query the date, which will then be used to identify the first image (`first`) that appears within the list of images, within the area of interest. We can then format the legend box within the map so that it shows the date and time of the layer selected.

Following this, we can set the symbology of each layer (burned area in red, and the precipitation color gradient indicated in the legend settings), and display the relevant layer with the date text string on the map (Fig. 54.1).

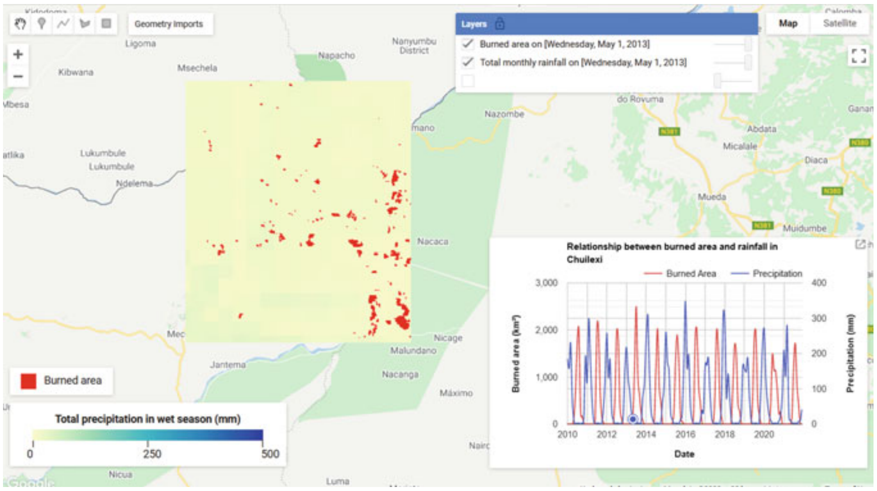

Geometry Imports
Layers
Burned area on [Wednesday, May 1, 2013]
Total monthly rainfall on [Wednesday, May 1, 2013]
Map
Satellite
Burned area
Total precipitation in wet season (mm)
0
250
500
Relationship between burned area and rainfall in Chuilexi
Burned Area
Precipitation
Burned area (km²)
Precipitation (mm)
Date
3,000
2,000
1,000
0
400
300
200
100
0
2010
2012
2014
2016
2018
2020

Fig. 54.1 Final result of the code once you have clicked on the chart to select a month to display, here showing the burned area and total precipitation at the beginning of the dry season in May 2021

```
// ** Chart: Adding an interactive query ** //

// Add a function where if you click on a point in the map
it displays the burned area and rainfall for that date
bpChart.onClick(function(xValue, yValue, seriesName) {
    if (!xValue) return;
    // Show layer for date selected on the chart
    var equalDate = ee.Filter.equals('system:time_start',
        xValue);
    // Search for the layer in the image collection that
links to the selected date
    var classificationB = ee.Image(MODIS_BurnDate.filter(
        equalDate).first()).clip(AOI).select('BurnDate');
    var classificationR = ee.Image(MonthlyRainfall.filter(
        equalDate).first()).clip(AOI).select(
        'precipitation');
    var burnImage =
ee.Image(MODIS_BurnDate.filter(equalDate)
        .first());
    var date_string = new Date(xValue).toLocaleString(
        'en-EN', {
            dateStyle: 'full'
        });
    var rainImage =
ee.Image(MonthlyRainfall.filter(equalDate)
        .first());
    var date_stringR = new Date(xValue).toLocaleString(
        'en-EN', {

            dateStyle: 'full'
        });
    // Reset the map layers each time a new date is clicked
    Map.layers().reset([classificationB]);
    Map.layers().reset([classificationR]);
    var visParamsBurnLayer = { // Visualisation for burned
area
        min: 0,
        max: 365,
        palette: ['red']
    };
    var visParamsRainLayer = { // Visualisation for rain
        min: 0,
        max: 450,
        palette: ['#ffffcc', '#a1dab4', '#41b6c4',
            '#2c7fb8', '#253494'
        ]
    };
    // Add the layers to the map
    Map.addLayer(classificationR, visParamsRainLayer,
        'Total monthly rainfall on [' + date_string + ']');
    Map.addLayer(classificationB, visParamsBurnLayer,
        'Burned area on [' + date_string + ']');
});
```

Code Checkpoint A39c. The book's repository contains a script that shows what your code should look like at this point.

Question 6. Considering the importance of environmental variables in conservation work, what other datasets and common earth observation analysis could provide data to inform conservation management?

54.3 Synthesis

In this chapter, you have learned how to create a time series of burned area, using the global MODIS burned area product. You also calculated areal mean rainfall using the CHIRPS dataset, and graphed total seasonal rainfall using your time-series chart. Additionally, you can now merge two image collections to show two variables on one interactive chart in the Earth Engine map.

With the code you have learned in this chapter, you can now create area burned and areal mean rainfall time series for your own region of interest anywhere in the world. Recreate the analysis in a different environment, and consider extending the time series over a longer period; can you detect any trends?

Monitoring vegetation is also important in conservation and can be significantly impacted by fire and rainfall trends. Try to modify the code you have learned to calculate areal mean NDVI from the NOAA CDR AVHRR daily NDVI dataset. Can you add this as another variable in the interactive chart?

54.4 Conclusion

In this chapter, we understand and map the dynamic relationship between fire and rainfall, and how this can influence conservation action and land management needs. We began by mapping burned areas using MODIS Burned Area Monthly Global 500 m to understand how much of, and where, the landscape is affected by burning. Following this, we understood how to access rainfall data and calculate areal mean rainfall, plotting this on a graph to understand changes and patterns over time. By combining the burned area and rainfall data, we can see how rainfall (or lack thereof) can exacerbate burning, and begin to spot patterns in the landscape. This analysis, which would often be undertaken in the field or by hand using satellite imagery, is made accessible via Earth Engine—not only because we can perform this on a workstation that only requires an internet connection rather than computing power, but also because Earth Engine generates quick, consistent results that can inform conservation management practices.

References

Bonney MT, He Y, Myint SW (2020) Contextualizing the 2019–20 kangaroo island bushfires: quantifying landscape-level influences on past severity and recovery with Landsat and Google Earth Engine. Remote Sens 12:1–32. https://doi.org/10.3390/rs12233942

Holden ZA, Swanson A, Luce CH et al (2018) Decreasing fire season precipitation increased recent Western US forest wildfire activity. Proc Natl Acad Sci USA 115:E8349–E8357. https://doi.org/10.1073/pnas.1802316115

Nhongo E, Fontana D, Guasselli L (2020) Spatio-temporal patterns of wildfires in the Niassa Reserve–Mozambique, using remote sensing data. bioRxiv, 1–7. https://doi.org/10.1101/2020.01.16.908780

Pinto-Ledezma JN, Cavender-Bares J (2021) Predicting species distributions and community composition using satellite remote sensing predictors. Sci Rep 11:1–12. https://doi.org/10.1038/s41598-021-96047-7

Reddy CS (2021) Remote sensing of biodiversity: what to measure and monitor from space to species? Biodivers Conserv 30:2617–2631. https://doi.org/10.1007/s10531-021-02216-5

Ribeiro NS, Armstrong AH, Fischer R et al (2021) Prediction of forest parameters and carbon accounting under different fire regimes in Miombo woodlands, Niassa Special Reserve. Northern Mozambique. for Policy Econ 133:102625. https://doi.org/10.1016/j.forpol.2021.102625

Conservation II—Assessing Agricultural Intensification Near Protected Areas

55

Pradeep Koulgi and M. D. Madhusudan

Overview

Protected Areas (PAs) in many densely populated tropical regions are often small in area, and are enormously influenced by the broader production landscapes in which they are found. Changes in the agricultural matrix surrounding a PA can have a profound impact on the PA's wildlife and on neighboring resident human communities. In this chapter, we will examine greening trend changes in the exteriors of 186 PAs in Western India from 2000 to 2021 using MODIS Terra vegetation indices, a Sen's slope linear trend estimator, and other summary techniques available in Earth Engine. We will use these techniques to investigate how these greening trends are distributed in relation to the precipitation regimes of a given PA site.

Learning Outcomes

- Computing a metric of a monotonic trend (Sen's slope) in dry-season pixel greenness for each pixel.
- Inferring the nature and intensity of change in agricultural practice based on the trend metric.
- Exploring the relationship between changes in vegetation greenness and ecosystem type as determined by average annual precipitation.

P. Koulgi (✉) · M. D. Madhusudan
National Centre for Biological Sciences, Tata Institute of Fundamental Research, Bellary Road, Bangalore, Karnataka 560065, India
e-mail: pradeep.koulgi@gmail.com

J. A. Cardille et al. (eds.), *Cloud-Based Remote Sensing with Google Earth Engine*,
https://doi.org/10.1007/978-3-031-26588-4_55

Helps if you know how to

- Import images and image collections, filter, and visualize (Part I).
- Perform basic image analysis: select bands, compute indices, create masks (Part II).
- Calculate and interpret vegetation indices (Chap. 5)
- Use reducers to implement linear regression between image bands (Chap. 8).
- Write a function and map it over an `ImageCollection` (Chap. 12).
- Map an annual reducer across multiple years (Chaps. 12, and 13).
- Write a function and map it over a collection (Chap. 12).
- Conduct basic vector analyses: vectorizing and buffering (Part V).
- Write a function and map it over a `FeatureCollection` (Chaps. 23, and 24).

55.1 Introduction to Theory

In many regions of the densely populated tropics, Protected Areas (PAs) are often small, and their interfaces with production landscapes are sharp. As a result, changes in the agriculture surrounding a PA can have a profound impact on the PA's wildlife, as well as on their interactions with local human communities. For example, across India's Gangetic Plains, increased crop irrigation via groundwater exploitation has enabled its traditional rainfed agriculture with long fallow periods and annual crops (cereals, legumes, and oilseeds) to be to be replaced by year-round agriculture (Chen et al 2019; Maina et al 2022). This intensification can have subtle yet significant changes in the landscape in terms of spatial regimes of primary productivity, and to wildlife in terms of both dietary and habitat resources. These changes, in turn, can influence wildlife interactions with people (Kumar et al. 2018).

In this chapter, we describe greening trends in vegetation in the exteriors of PAs in Western India, and ask how these greening trends are distributed in relation to the precipitation regimes of a given PA site. Based on our experience and our reading of literature pertaining to wildlife in India, we hypothesize that as crop irrigation via groundwater extraction increases, farming in the moisture-limited semi-arid tracts of India becomes more independent of seasonal differences, and their constituent habitats become structurally more complex. These changes could help create novel anthropogenic habitats that can be accessed by adaptable wildlife species living in PAs, such as leopards and elephants (Lenin 2010; Odden et al. 2014). While such changes to habitat structure and resource availability could improve functional connectivity between PAs (e.g., Rodrigues et al. 2022), intensification can also lead to a greater overlap between humans and wildlife, possibly leading to greater conflict over crops and livestock (Kumar et al. 2018). The potential trade-offs involved between conservation and conflict imply that it is crucial to assess and map land use changes around PAs over time, and eventually to relate this to changes in wildlife distribution and behavior, for which this chapter provides a framework.

55.2 Practicum

55.2.1 Section 1: Initializing Parameters

The Normalized Difference Vegetation Index (NDVI) estimates vegetation greenness; the MODIS Terra vegetation indices products, MOD13Q1.006, provides historical 16-day composites of it at a global span and 250 m per pixel resolution. This dataset is used here to estimate a monotonic trend in annual changes of dry-season vegetation greenness around a set of PAs in India. Various parameters used in the script are initialized first.

55.2.1.1 Section 1.1: Annual Dry-Season Maximum NDVI Calculation

The MODIS vegetation indices dataset MOD13Q1.006 is initialized, along with the NDVI band name and relevant scale values. The dataset starts from the year 2000 and extends to the present. For this analysis, we use data from 2000 to 2021 as the sequence of full years the data are available for. The annual dry season in much of India occurs over three to five months, starting roughly in January. Here, a period of 90 days (spanning January through March) starting on the first day of each year is taken to be the dry season. Convenient names are chosen for bands holding NDVI and time values for regression analysis.

```
var modis_veg = ee.ImageCollection('MODIS/006/MOD13Q1');
var ndviBandName = 'NDVI';
var ndviValuesScaling = 0.0001;
var modisVegScale = 250; // meters
var maxNDVIBandname = 'max_dryseason_ndvi';
var yearTimestampBandname = 'year';
var years = ee.List.sequence(2000, 2021, 1);
var drySeasonStart_doy = 1;
var drySeasonEnd_doy = 90;
```

55.2.1.2 Section 1.2: Boundaries of PAs of Interest

In the code below, a set of PAs in the western part of India is identified for study. This collection of PAs in the western belt of India spans a wide range of rainfall regimes, from very moist areas in the southern Western Ghats to very arid regions in the Thar Desert of Rajasthan in the north, with diverse rainfall regimes in between. This belt is home to a diverse array of forests and savanna ecosystems, ranging from moist ecosystems to semi-arid and arid ecosystems. The landscapes in this belt also have a complex, diverse intervening matrix of natural and intensively human-utilized lands making up wildlife corridors. The size of the buffer area around each PA is also defined.

```
var paBoundaries = ee.FeatureCollection(
    'projects/gee-book/assets/A3-10/IndiaMainlandPAs');
var boundaryBufferWidth = 5000; // meters
var bufferingMaxError = 30; // meters
// Choose PAs in only the western states
var western_states = [
    'Rajasthan', 'Gujarat', 'Madhya Pradesh',
    'Maharashtra', 'Goa', 'Karnataka', 'Kerala'
];
var western_pas = paBoundaries
    .filter(ee.Filter.inList('STATE', western_states));
```

55.2.1.3 Section 1.3: Regression Analysis

The Sen's slope linear trend estimator (Sen 1968) is a non-parametric estimator useful for monotonic trend estimation applications using remotely sensed indices. Its suitability for applications with remotely sensed indices comes from a key difference when compared to linear regression using the least squares estimator: while linear regression assumes that the regression residuals are normally distributed, Sen's slope estimator makes no such assumptions about the statistical structure of the data it is being applied on. The reducer for this Sen's slope regression is built into the Earth Engine API, and in the code below, its x and y variables are initialized.

```
var regressionReducer = ee.Reducer.sensSlope();
var regressionX = yearTimestampBandname;
var regressionY = maxNDVIBandname;
```

55.2.1.4 Section 1.4: Surface Water Layer to Mask Water Pixels from Assessment

NDVI values of pixels spanning water bodies tend to be highly noisy and can mislead trend analyses. Masking out all pixels that were ever water during the period of interest can mitigate this problem. The European Union's Joint Research Centre mapped monthly surface water from 1984 to 2021, including the maximum water extent detected during the period.

```
// Selects pixels where water has ever been detected
between 1984 and 2021
var surfaceWaterExtent =
ee.Image('JRC/GSW1_3/GlobalSurfaceWater')
    .select('max_extent');
```

55.2.1.5 Section 1.5: Average Annual Precipitation Layer

To relate the estimated trends in vegetation greenness to rainfall regime, we use WorldClim's estimated long-term average annual precipitation data.

```
var rainfall =
ee.Image('WORLDCLIM/V1/BIO').select('bio12');
```

55.2.1.6 Section 1.6: Visualization and Saving Parameters

The estimated metric of change in vegetation greenness can be visualized as a raster on the map. Additionally, the relationship between changes in vegetation greenness and precipitation around each PA can be charted using a scatter plot. The visualization parameters for these are defined.

```
var regressionResultVisParams = {
    min: -3,
    max: 3,
    palette: ['ff8202', 'ffffff', '356e02']
};
var regressionSummaryChartingOptions = {
    title: 'Yearly change in dry-season vegetation
greenness ' +
        'in PA buffers in relation to average annual
rainfall',
    hAxis: {
        title: 'Annual Precipitation'
    },
    vAxis: {
        title: 'Median % yearly change in vegetation
greenness ' +
            'in 5 km buffer'
    },
    series: {
        0: {
            visibleInLegend: false
        }
    },
};
```

Code Checkpoint A310a. The book's repository contains a script that shows what your code should look like at this point.

55.2.2 Section 2: Raster Processing for Change Analysis

The 16-day composite NDVI rasters are filtered, processed, and reduced according to parameters appropriately defined and initialized above to generate a raster of percentage annual change in greenness as a metric for vegetation change. A summary of this for each PA buffer is also calculated. Finally, these results are visualized.

55.2.2.1 Section 2.1: Annual Dry-Season Maxima of NDVI

The source MODIS vegetation indices dataset is first filtered to the dry-season days for all years and the NDVI band selected. From this `ImageCollection` containing only dry-season observations for each year, maximum dry-season NDVI is calculated. This represents the vegetation at its greenest condition within the dry season each year. The maximum NDVI is the least likely to be influenced by episodic changes, such as fires that occur during this dry period. An image band containing the year value is also created and combined with the maximum NDVI, in preparation for time versus NDVI regression analysis in the next step. This is accomplished through defining a function `annualDrySeasonMaximumNDVIAndTime` to do this for each year, and then mapping that function over all the years in our period of interest.

```
function annualDrySeasonMaximumNDVIAndTime(y) {
    // Convert year y to a date object
    var yDate = ee.Date.fromYMD(y, 1, 1);
    // Calculate max NDVI for year y
    var yMaxNdvi = drySeasonNdviColl
        // Filter to year y
        .filter(ee.Filter.date(yDate, yDate.advance(1,
'year')))
        // Compute max value
        .max()
        // Apply appropriate scale, as per the dataset's
        // technical description for NDVI band.
        .multiply(ndviValuesScaling)
        // rename the band to be more comprehensible
        .rename(maxNDVIBandname);
    // Create an image with constant value y, to be used in
regression. Name it something comprehensible.
    // Name it something comprehensible.
    var yTime = ee.Image.constant(y).int().rename(
        yearTimestampBandname);
    // Combine the two images into a single 2-band image,
and return
    return ee.Image.cat([yMaxNdvi, yTime]).set('year', y);
}

// Create a collection of annual dry season maxima
// for the years of interest.  Select the NDVI band and
// filter to the collection of dry season observations.
var drySeasonNdviColl = modis_veg.select([ndviBandName])
    .filter(ee.Filter.calendarRange(drySeasonStart_doy,
        drySeasonEnd_doy, 'day_of_year'));
// For each year of interest, calculate the NDVI maxima and
create a corresponding time band
var dryseason_coll = ee.ImageCollection.fromImages(
    years.map(annualDrySeasonMaximumNDVIAndTime)
);
```

55.2.2.2 Section 2.2: Annual Regression to Estimate Average Yearly Change in Greenness

The dry-season maximum NDVI collection with time band is reduced with a Sen's slope regression reducer to estimate the linear rate of change of dry-season maximum greenness for the years from 2000 to 2021. Be sure to use the `select` operation to select the x and y variables for regression, in that order, as expected by the `ee.Reducer.sensSlope`. The resulting image will have a slope and an offset band, which are the slope and y-intercept of the trend estimation, respectively.

The values in the slope band are the pixel-wise annual rate of change in maximum dry-season NDVI values. The magnitude of these values signifies the intensity of change over the years, and their sign indicates whether the change manifested as greening (positive sign) or browning (negative sign) of the vegetation.

```
var ss = dryseason_coll.select([regressionX,
regressionY]).reduce(
    regressionReducer);

// Mask surface water from vegetation change image
var ss = ss.updateMask(surfaceWaterExtent.eq(0));
```

55.2.2.3 Section 2.3: Summarize Estimates of Change in Buffer Regions of PAs of Interest

In order to investigate the vegetation greening and browning changes within the buffer areas of our PAs of interest, buffer regions have to be defined. This can be done by computing the geometry "difference" between the buffered version of each PA and the PA itself. A function `extractBufferRegion` is defined to perform this for each PA feature, and that function is mapped over the feature collection with boundaries of all the PAs of interest.

```
function extractBufferRegion(pa) {
    //reduce vertices in PA boundary
    pa = pa.simplify({
        maxError: 100
    });
    // Extend boundary into its buffer
    var withBuffer = pa.buffer(boundaryBufferWidth,
        bufferingMaxError);
    // Compute the buffer-only region by "subtracting"
boundary with buffer from boundary
    // Subtracting the whole set of boundaries eliminates
inclusion of forests from adjacent PAs into buffers.
    var bufferOnly =
withBuffer.difference(paBoundaries.geometry());

    return bufferOnly;
}

// Create buffer regions of PAs
var pa_buff = western_pas.map(extractBufferRegion);
```

The greenness condition of vegetation in the dry season is inherently variable in large diverse landscapes like around the PAs of choice here, and it depends strongly on the type of vegetation itself, such as deciduous or evergreen tree cover, grasslands, shrublands, and croplands. In order to take this inherent variability into account, the raw rate of change value can be converted into a percent rate of change by normalizing the change value against a baseline value. Here, the maximum year 2000 dry-season NDVI is used as the baseline in each pixel. The slope values are normalized using this baseline and converted to percent values. This can be visualized on the map.

In order to understand the overall picture at a PA buffer-region level, the median of percent rate of vegetation change is calculated for every PA buffer. And to relate this to the amount of rainfall in a buffer region, the corresponding median of average annual rainfall over the region is also calculated. These can be visualized in a chart.

```
// Normalize the metric of NDVI change to a baseline (dry-
season max NDVI in the very first year)
var baselineNdvi =
dryseason_coll.select([maxNDVIBandname]).filter(ee
    .Filter.eq('year', years.getNumber(0))).first();
var stats =
ss.select('slope').divide(baselineNdvi).multiply(100)
    .rename('vegchange');

// Combine it with average annual rainfall data
stats = stats.addBands(rainfall.rename('rainfall'));

// Calculate mean of change metric over buffer regions of
each PA of interest
var paBufferwiseMedianChange = stats.reduceRegions({
    collection: pa_buff,
    reducer: ee.Reducer.median(),
    scale: 1000,
    tileScale: 16
});
```

Code Checkpoint A310b. The book's repository contains a script that shows what your code should look like at this point.

55.2.3 Section 3: Visualizing Results

The degree of vegetation change, with a chosen palette, is visualized as a map (Fig. 55.1), and the PA buffer-wise median value is charted as a scatter plot (Fig. 55.2). In the map, green color suggests significant vegetation greening, brown color suggests significant vegetation browning, and white color suggests no detectable change, as chosen in the palette. The vegetation in large areas in the surroundings of Nauradehi Wildlife Sanctuary, as highlighted by Fig. 55.1, appears to have experienced greening or no detectable change. PA buffer-wise summaries showing moderate to high positive values indicate that, across the buffers of the PAs, there appears to have been a greater extent of greening, rather than browning, of vegetation cover.

The degree of vegetation change is also arranged by amount of rainfall in a chart, to visualize how buffer regions of PAs in moist regions fare relative to those in arid regions.

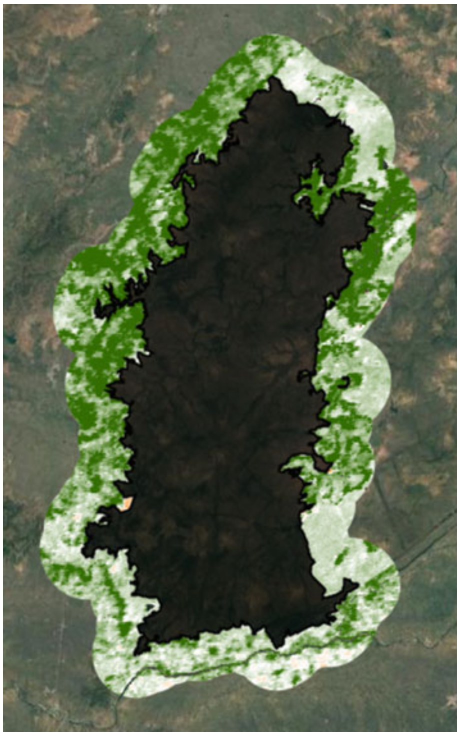


Fig. 55.1 Map of percent vegetation greenness change for 2000–2021 in a 5 km buffer area around Nauradehi Wildlife Sanctuary (polygon in black shade), India. Note the relative absence of vegetation browning

```
var medianChangeChart = ui.Chart.feature.byFeature({
    features: paBufferwiseMedianChange,
    xProperty: 'rainfall',
    yProperties: ['vegchange']
}).setOptions(regressionSummaryChartingOptions).setChartTyp
e(
    'ScatterChart');
print(medianChangeChart);

Map.centerObject(western_pas, 9);
Map.setCenter(79.2205, 23.3991, 9);
Map.setOptions('SATELLITE');
Map.addLayer(stats.select('vegchange').clipToCollection(pa_
buff),
    regressionResultVisParams, 'yearly % change');
Map.addLayer(western_pas, {}, 'Western PAs');
```

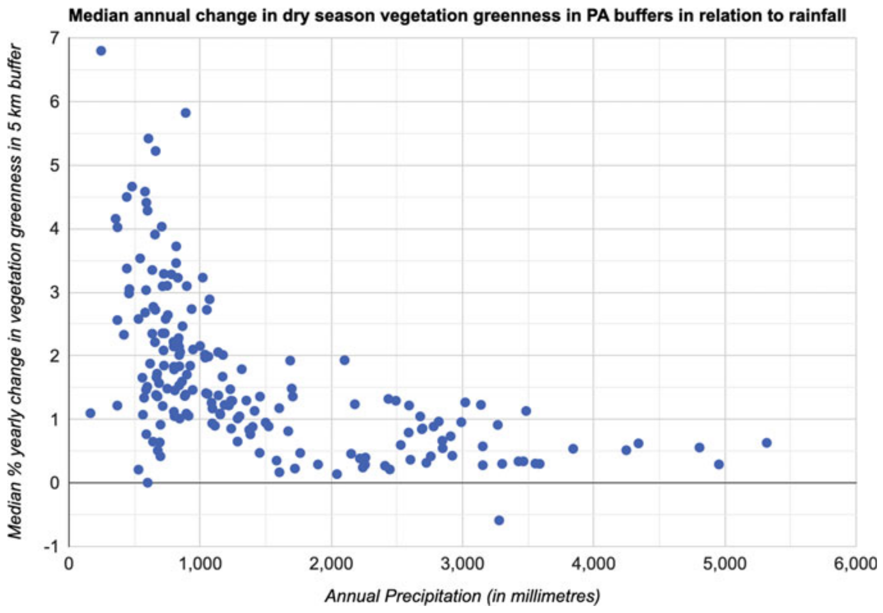


Fig. 55.2 Median yearly percent change in vegetation greenness observed in 5 km exterior buffers of PAs in India ($N = 186$), arranged along a precipitation gradient. Greening is denoted by a positive percent change in Sen's slope of dry-season maximum NDVI between 2000 and 2021. The greatest greening extent (> 2%) is observed in PAs that fall in the semi-arid zone (annual precipitation $\leq$ 1000 mm)

Code Checkpoint A310c. The book's repository contains a script that shows what your code should look like at this point.

Question 1. Create a map of percent vegetation greenness change for a different PA than the one shown in Fig. 55.1. How does it compare with the map made for Nauradehi Wildlife Sanctuary?

Question 2. We used MODIS Terra vegetation indices for our greening trend analysis in the previous example. Can you alter the analysis to calculate an NDVI time series from Landsat imagery for one PA? How do you think the results would change with Landsat's 30 m spatial resolution instead of 250 m from MODIS?

Question 3. Can you repeat this analysis, dividing the time period of 21 years into two approximate halves, and try to determine whether the slope of vegetation greening observed has been similar across the two periods, or whether greening has accelerated or decelerated between the two time periods?

55.3 Synthesis

Assignment 1. In this chapter, you learned how to estimate greening trends surrounding PAs in Western India, a densely populated tropical region. In your opinion, how generalizable are these observed patterns across space and time?

Assignment 2. Perform this same PA analysis in another region in India. Then, perform the same analysis in a tropical region in another part of the world. How do the results compare with ours from Western India?

55.4 Conclusion

In this chapter, we estimated greening trends in Western India PAs using MODIS Terra vegetation indices, Sen's slope, and reducer functions. We demonstrated that the immediate exterior buffers of most PAs (184 out of 186) show a positive—i.e., greening—trend in their vegetation cover over the two-decade time frame between 2000 and 2021. Note, however, that in this example, we do not estimate the significance level of the greening trend values modeled in our analyses. While there is considerable variation in the greening extent seen across these PA buffers, we do see clearly that PAs that fall within the semi-arid zone (annual precipitation $\leq$ 1000 mm) seem to show the greatest extent of greening (Fig. 55.2). Such a change in the land cover characteristics of the matrix surrounding PAs in the semi-arid zone suggests that they are more likely to be sites where the distribution of wildlife, especially of more adaptable species, could show expansion into the agricultural landscape beyond PA boundaries, as well as greater rates of conflict with humans over crops and livestock. Corroborating this, of course, requires ground survey data on wildlife, which were unavailable in this example.

References

Chen C, Park T, Wang X et al (2019) China and India lead in greening of the world through land-use management. Nat Sustain 2:122–129. https://doi.org/10.1038/s41893-019-0220-7

Kumar MA, Vijayakrishnan S, Singh M (2018) Whose habitat is it anyway? Role of natural and anthropogenic habitats in conservation of charismatic species. Trop Conserv Sci 11:1940082918788451. https://doi.org/10.1177/1940082918788451

Lenin J (2010) Sugarcane leopards. In: Current conservation, special: wildlife-human conflict. https://www.currentconservation.org/sugarcane-leopards/

Maina FZ, Kumar SV, Albergel C, Mahanama SP (2022) Warming, increase in precipitation, and irrigation enhance greening in High Mountain Asia. Commun Earth Environ 3:1–8. https://doi.org/10.1038/s43247-022-00374-0

Odden M, Athreya V, Rattan S, Linnell JDC (2014) Adaptable neighbours: movement patterns of GPS-collared leopards in human dominated landscapes in India. PLoS ONE 9:e112044. https://doi.org/10.1371/journal.pone.0112044

Rodrigues RG, Srivathsa A, Vasudev D (2022) Dog in the matrix: envisioning countrywide connectivity conservation for an endangered carnivore. J Appl Ecol 59:223–237. https://doi.org/10.1111/1365-2664.14048

Sen PK (1968) Estimates of the regression coefficient based on Kendall's tau. J Am Stat Assoc 63:1379–1389. https://doi.org/10.1080/01621459.1968.10480934