

A primer on understanding Google Earth Engine APIs

Rui S. Reis^{ab}, Nuno Datia^{ab}, M. P. M. Pato^{bc}

^aISEL - Instituto Superior de Engenharia de Lisboa, Instituto Politécnico de Lisboa

^bNovaLincs, FCT – Universidade Nova de Lisboa

^cInstituto de Biofísica e Engenharia Biomédica, FC-UL

ruisreis@hotmail.com {datia,mpato}@deetc.isel.pt

Abstract—This article introduces the rationale behind the usage of the *Google Earth Engine*, and the advantages it offers, as an alternative to handle large volumes of georeferenced data using the existing tools we know as *Geographic Information Systems* on premises.

Google Earth Engine is an efficient development framework that presents itself in two basic flavors: one online integrated development environment which uses the browser *JavaScript*'s engine; and two APIs that can be deployed on either a *Python* or a *NodeJS* environment.

After presenting a limited number of use cases, representative of the *Google Earth Engine* design patterns, and building a prototype class using both variants, we conclude that both platforms are merely proxy APIs to the *Google Earth Engine* and do not have any measurable performance difference. However, since they run on fundamentally diverse contexts — a *JavaScript*'s engine on an internet browser, that integrates seamlessly with *Google Maps*, and a *Python* environment — it is argued that their utility depends on the user requirements instead of being true alternatives.

Keywords: *Google Earth Engine*, *Javascript*, *Python*, *Code Editor*, *Georeferenced Data*, *Multi-spectral Data*.

I. INTRODUCTION

Google Earth Engine [1] (*GEE*) is, primarily, a distributed parallel computing platform. It is designed around a functional language pattern, even though supported on an object model, and a map - reduce [2] distributed workload paradigm.

Leveraging the sheer computing power delivered by the *Google* infrastructure and a multi petabyte georeferenced data repository, *GEE* is an efficient development framework to handle all tasks related with selecting, computing calculations and displaying georeferenced data.

A. Georeferenced data

Working with georeferenced data is a grueling task considering the large volumes of information, the complexity and diversity of storage formats.

Using, as an example, remote sensing multi-spectral data gathered by instruments on board of satellites, it is easy to understand the complexity of obtaining, interpreting and making calculations using this kind of data:

- Multi-spectral data is arranged in bands that store the reflectance measurements on a range of wavelengths. For instance, the Level-1C instrument's aboard the *Sentinel 2* [3] constellation read 13 reflectance bands and 3 additional data quality bands. For the *Sentinel 2* each

coordinate, a pair longitude and latitude values, represents a $10m^2$ area and is associated to a vector of 16 values, one for each of the reflectance and quality bands;

- This data is organized using a set of rules that are, generally, specific to each satellite. Seldom an interpretation layer must be used to transform the source format to one of the standard (or “de facto”) file formats, so that it can be used by one of the existing libraries (e.g. *GDAL* [4]);
- The calculations require a lot of resources for storing and processing. Until recently, many researchers opted almost exclusively to use calculated products, which are datasets with multi-spectral calculated data, mostly in the form of indices, like *NDVI* [5]. These were published by organizations (profitable or not) like *Copernicus*¹ or *VITO*², and the availability of these datasets is delayed in time, considering the actual date of retrieval;
- The usage of calculated products might reduce the complexity of the data, for instance a *NDVI* dataset has a single value for each coordinate, but the information volume is still very large. If we take a single day of data for a $3.245km^2$ area in Portugal of *NDVI* gathered by the *Proba-V* [6] instruments, where each coordinate represents a $300m^2$ area, we will get an approximately 1Gb [7] *GeoTIFF* [8] file.

However, besides the storage requirements, to make additional calculations on this data, an adequate tool must be used. Consider using, for example, the *GDAL* [4] library embedded in an integrated *Geographical Information System (GIS)* tool, *QGIS* [9]. Using this on premises³ setup, and the *NDVI* dataset described previously, the calculation of the arithmetic average of the *NDVI* value, on every coordinate, for two approximate areas of $300km^2$ and $170km^2$, took close to 5 minutes (using a computer with an Intel Core i5-6200U, 8Gb RAM and a 256Gb SSD) [7].

The multi-spectral data is very sensitive to the presence of clouds and atmospheric aerosols. This means that multi-spectral data is potentially sparse due to the varying weather conditions and pollution. Methods like *Maximum Value Composite* [10], that require handling several of the previously described datasets, in order to obtain significant *NDVI* values, will require even larger amounts of storage and computing resources [7].

¹<https://www.copernicus.eu/en>

²<https://vito.be/en>

³The software is installed and runs on computers on the premises of the person or organization using the software, rather than at a remote facility.

So, the main challenges to handle remote sensing multi-spectral data are:

- The quantity of storage resources needed;
- The data transformation into convenient formats; and
- The computing power to enable efficient calculations on these significant volumes of complex data.

B. Google Earth Engine

The *GEE* is a recent *Cloud* platform built to handle large volumes of georeferenced data, using Google's storage and computational resources.

It tackles the challenge of understanding the complex organization of remote sensing data, while unifying its representation around a set of common formats supported by the *API*⁴. This data is stored in a large database that contains some static datasets but, most importantly, live datasets, from sources that produce new data periodically, which are ingested by *GEE* on a regular basis, namely remote sensing multi-spectral satellite data.

In short, the repository contains, to date, a catalog of some 600 datasets from 50 different sources (devices and organizations) [11]. Remote sensing data is gathered from 30 satellites, or satellite constellations. All this, according to *Google* [12], represented a volume of more than *20PB*⁵ in 2018.

The *API* exposes an extensive set of operations that can be used to explore the public repository or other user defined datasets, that are kept in a private assets area. There is a common set of objects that structure vector based data and georeferenced bitmaps which are organized in the repository as collections. Most functions work on these collections: filtering, sorting and computing calculations over their data. Primitive data types (e.g. numbers, strings, etc...) and sets (e.g. lists, dictionaries, etc...) are also supported by the *API*.

The paradigm is fully functional, since a call to *GEE* is self contained, but it's built around a set of objects that wrap each of the data types, and expose methods that operate on them. The main development environment is the *Code Editor*, which is a browser based tool that interacts with *GEE* platform using the *JavaScript*'s engine and is able to use other *Cloud* based tools from *Google*, especially *Google Maps* and *Google Charts*. This technology mix makes it a valuable tool whenever user interaction with a rich visual interface is needed, specifically map overlays and prototyping.

GEE can expose applications using, what is called, Earth Engine Applications⁶. However, *GEE* also exposes an *API* using *Python* or *NodeJS* libraries⁷ that leverage the development of applications that do not fully depend on the *Google* infrastructure and leapfrog some of the difficulties of using the browser based *Code Editor*, which will be addressed further.

⁴Application Programming Interface

⁵1PB \equiv 2⁵⁰bytes

⁶More information can be found at <https://www.earthengine.app/>

⁷At the time of writing, *GEE* is on the verge of a major update. It is not clear if the *NodeJS* is still a priority.

All of these are particularly important given the fact that *Google* states, "Earth Engine is not subject to any Service-Level Agreement (*SLA*) or deprecation policy" [13], which might force the developer to use asynchronous resubmitting strategies to properly make use of *GEE*.

This article was wrote based on the experience of using the *GEE* framework as a development platform [7]. It will pinpoint the challenges of using *Code Editor*, as well as the advantages of making use of it to produce eye catching visual information, and the quick development of prototypes, as opposed to using the *Python* library to leverage the usage of *GEE* in a rich development environment⁸.

C. Motivation

The learning curve of *GEE* began with exploring the *Code Editor* and the *JavaScript API* in a particular use case scenario [7].

Several issues aroused while using the *Code Editor*, but two were prominent:

- The complexity of cross domain scripts that include, in the same control flow, both local and distributed processes and data structures;
- The need to extract data that might be used in other applications (e.g. *Microsoft Excel*) in such a way that the whole process could be streamlined and automated.

It soon became clear that *Code Editor* is a valuable tool to produce maps and overlays that might be used to illustrate results obtained, using georeferenced data, but clearly inadequate to integrate *GEE* data or functionality in more elaborate scenarios.

Even though the *Code Editor* is a comprehensive tool that is adequate to explore the framework and interact with the *Google Cloud* applications ecosystem, there are some obstacles using *GEE* in some use cases, specially those that require the extraction of information to be reused over time:

- The execution of long running tasks in the browser's environment does not allow a proper progress notification system, the console output is integrated in the graphical user interface and there is no way to integrate the execution path with another control flow, namely using a callback mechanism;
- The browser's *JavaScript* engine does not distinguish between a badly written piece of code and a proper long running process, which results in the user being prompted to confirm if the execution should be aborted;
- Finally, and most important, the data extraction using the browser is challenging, and it's integration in a data flow isn't feasible, though the platform supports batch processes that can persist data using *Cloud* storage (e.g. *Google Drive* or *Google Cloud*).

There is also the issue of concurrency. Sometimes it is necessary to harness the temporary unavailability of storage and processing quota due to concurrency issues between users' processes. In *Code Editor* it is not easy to control this behavior,

⁸*Google* recently has announced a third path, the *Colaboratory* hosted *Jupyter Notebooks* which inherently uses *Python*.

specially while interacting with *Google Maps*. So, whenever the platform fails to respond to user queries, or the user quota is exhausted, an exception is thrown and handled by the *Code Editor* and thus the process control flow is interrupted.

Though somewhat over simplifying, we can state that using the *Python API* obviates most of this issues, but loses the benefits of integrating with *Google Maps*.

Google made an effort to keep both *APIs* syntactically equivalent [14]. Most differences are related to the grammar of each supporting language, *JavaScript* and *Python*, but the bottom line is that they are inherently similar.

D. Testing environment

In the next sections we will browse some of the main *GEE API* concepts in order to provide insights on this framework.

To use *GEE* the user must have a *Google* account and request access to the platform⁹.

The *Code Editor* does not require any additional setup. Using the internet browser, after being authenticated using the *Google* account, the user navigates to the site using a *URL*¹⁰.

Using the *Python* environment requires some additional steps [14] that include the installation of the *Python* package *earthengine-api*.

All experiments were performed using the *Google Chrome* browser (version 81.0.4044.122) to access the *Code Editor*, and a *Python* environment (version 3.7.6, 64 bit) including the *GEE earthengine-api* package (version 0.1.219). The host was a computer running *Microsoft Windows 10 Enterprise* (version 1709, build 16299.1686), using an Intel Core i7-8550U CPU, 16Gb RAM and a 500Gb SSD.

E. Organization

The rest of the paper is organized as follows: Section 2 will elaborate on the several issues faced during the previous work that led to the usage of both the *Code Editor* and the *Python API* in different scenarios. In this section we also exemplify a few design patterns of the *GEE*, common to all *APIs*, and explain the interaction between the local development environment and the remote *GEE*. In order to demonstrate all these caveats, in section 3 we introduce two simple implementations of a same class interface using both environments. Section 4 presents the conclusions of this work.

II. BASIC CONCEPTS

GEE is a distributed parallel computing platform dedicated to store and process georeferenced data, which explores the most adequate programming paradigms to handle very large volumes of data. Each operation is mostly self contained in a functional paradigm pattern, and parallelism is implemented via *map-reduce* [2] mechanism.

Both development environments, *JavaScript* and *Python*, make an effort to hide the complexity of the object model and underlying processes.

This section is not supposed to be a *GEE* reference guide, neither a tutorial. It is, instead, an overview of the *API*, highlighting its most relevant patterns and data organization structures. Each different aspect we wish to highlight uses an object of the *GEE* model to be used as a showcase.

Though both *APIs* are almost syntactically equivalent, all the following use cases will be presented using the *Python* environment stressing the differences whenever they occur. Nonetheless, the *GEE* user guide singles out some differences[14] (see Table I).

TABLE I: Some common syntax differences between *JavaScript* and *Python* (source *GEE* user guide).

Description	<i>JavaScript</i>	<i>Python</i>
Function definition	function fun(){}	def fun():
Variable definition	var a = "value"	a = "value"
Logical operators	and() or() not()	And() Or() Not()
Multi-line method chain	fa() .fb() .fc();	fa()\n.fb()\n.fc()
Dictionary keys	{"key": "value"} or {key: "value"}	{"key": "value"}
Boolean	true false	True False
Null values	null	None
Comment	//	#

In both environments, *GEE* objects are referenced through an *ee* namespace, using a dot notation. In *CodeEditor* it is implicit and points to an existing object that supports the *API*, while in *Python* it references a package explicitly using **import ee**.

A. Initializing GEE

Any interaction with *GEE* will have to be preceded by the establishment of a secure context which will provide user authentication and session data.

A *Python* application that uses *GEE* will always have to include, beforehand, the call in Listing 1 in order to authenticate the user and establish a connection to the *GEE* platform.

Listing 1: *GEE* session initialization.

```
import ee
ee.Initialize()
```

This is a major difference compared to using *JavaScript* in the *Code Editor*. The *Code Editor* is a *Google* integrated *Cloud*

⁹<https://signup.earthengine.google.com/>

¹⁰<https://code.earthengine.google.com/>

application, the user authentication and initialization procedures are transparent.

B. Primitive types - handling numbers

This section will address the number data type, but most considerations will apply to other *GEE* primitive type wrappers.

The `ee.Number` object encapsulates a numeric value and exposes a set of functions that act upon it.

For instance, let's consider the encapsulation of the π constant. One could represent the value of π using a *GEE* object and keep it in a variable `number`. By the way, we could also represent the calculation of the cosine of this value, by setting a variable called `cosine` (see Listing 2).

Notice the symbolic nature of both variables, stressing that, at the local *Python* environment (or for this matter, *JavaScript*), they are only *JavaScript Object Notation (JSON)* representations that maybe used to compose complex *GEE* operations, with no intrinsic value. It is this representation that will be submitted to the *GEE* for execution.

Listing 2: Cosine calculation of the π constant in *GEE* using `ee.Number`

```
import math
number=ee.Number(math.pi)
cosine=number.cos()
print(cosine)
```

```
ee.Number({
  "type": "Invocation",
  "arguments": { "input": 3.141592653589793 },
  "functionName": "Number.cos"
})
```

In this section, whenever `print()` is used to output a result to the console, the code listings will be divided in two regions, split by a single line: the top region contains the *Python* code; the bottom region reflects the output to the console.

Printing the content of the `cosine` variable does not trigger any call to the *GEE* platform. From the generated output, it is self explanatory that the variable contains *GEE*'s internal representation of a call to a `cos` function using, as single parameter, the numeric constant.

Every object in *GEE* exposes a `getInfo()` method. Whenever this is called on an instance of a given object, the representation contained at the local environment is submitted to *GEE*, for evaluation, and the result is returned to the caller in the form of a *JSON* encoded object (see Listing 3).

Listing 3: Executing and obtaining the value of `cosine` in *GEE*

```
result=coseno.getInfo()
print(result)
```

```
-1.0
```

Note that `result` contains a *Python* native object that is the result of the evaluation of the representation contained in `cosine` on the *GEE* platform, that is, $\cos(\pi) = -1$.

In short, at the local environment level, representations of *GEE* can be composed to translate increasingly complex operations. The execution and evaluation of these operations occurs in the *GEE* platform explicitly when `getInfo()`, or some other function with similar behavior, are invoked, returning the operation result as a *JSON* encoded object.

C. Non primitive types - encoding dates

A calendar date is wrapped by an `ee.Date` object, which also exposes some functions that act upon it. One of these functions, `fromYMD()`, acts as a constructor and encodes a timestamp given its year, month and day numeric representations.

Listing 4, once again, underlines the symbolic proxy nature of the *GEE* object representation in the local *Python* environment.

Listing 4: Setting a date in variable `today`

```
today=ee.Date.fromYMD(2019,5,4)
print(today)
```

```
ee.Date({
  "type": "Invocation",
  "arguments": {"year": 2019, "month": 5, "day": 4
  },
  "functionName": "Date.fromYMD"
})
```

Using the `advance()` function, varying the offset parameter, it is possible to represent yesterday and tomorrow (see Listing 5).

Listing 5: Setting variables `yesterday` and `tomorrow`.

```
yesterday=today.advance(-1,"day")
tomorrow=today.advance(1,"day")
print(tomorrow)
```

```
ee.Date({
  "type": "Invocation",
  "arguments": {
    "date": {
      "type": "Invocation",
      "arguments": { "year": 2019, "month": 5, "day": 4 },
      "functionName": "Date.fromYMD"
    },
    "delta": 1,
    "unit": "day"
  },
  "functionName": "Date.advance"
})
```

Evaluating the variable `tomorrow` (see Listing 6), using `getInfo()`, the operation result value is returned.

Listing 6: Evaluating `tomorrow`

```
print(tomorrow.getInfo())
```

```
{'type': 'Date', 'value': 1557014400000}
```

The result of an `ee.Date` object does not translate to a primitive local timestamp, instead a dictionary is returned that contains the numeric value that translates to the number of milliseconds since midnight of the first of January 1970, also known as an Unix epoch.

D. Distribute using map operations - Lists

Lists are encapsulated in `ee.List` objects. Consider the representation of a list containing the previous three timestamp variables in Listing 7.

Listing 7: List dates

```
dates = ee.List([yesterday, today, tomorrow])
```

Exploring parallel processing, the workload could be distributed using the *map-reduce* paradigm. All objects that support iteration, including `ee.List`, expose a `map` function which has a single parameter, that will be a reference to another function, which has a specific set of features: (i) it will accept the item to process as parameter; (ii) returns the result of the process; (iii) must be self contained in the sense that it may not depend on anything that is not in the local scope of the function.

So, consider the function `mapper()` in Listing 8, built around these constraints. Note that it receives `element` as a single parameter. It does not have the context of the parameter type, so casting will be necessary. Finally it returns the numeric representation of the day.

This last particular feature is important to illustrate that the data type of the result of the `map` function might be different from the elements in `dates`. In this case, originally, the list has `ee.Date` members and the `mapper` function returns `ee.Number` instances.

Listing 8: A parallel processing map function.

```
def mapper(element):
    return ee.Date(element).get("day")

distribute=dates.map(mapper)
```

Note that `distribute` is just a representation at the local level. In order to process the implicit operation and obtain the result, `getInfo()` might be used (see Listing 9) to obtain the evaluated list.

Listing 9: Result of `distribute` evaluation.

```
print(distribute.getInfo())

[3, 4, 5]
```

E. Reducing and aggregating - using image collections

Georeferenced bitmaps¹¹ are kept in *GEE* using collections `ee.ImageCollection` of images `ee.Image` which, in turn, may either translate to a single image or sets of multi-spectral bands. Those that are stored in the *GEE* repository are singled out by unique string identifiers.

Images might contain more than a set of values for each represented coordinate. These values can be part of calculations using discrete or aggregation operations. The calculation may affect a single coordinate or all of them. All these operations are gathered around the concept of “band math” [15] in *GEE*.

¹¹Also know as raster images

Aggregation operation’s parallelism and distribution are supported on *reducers*.

Demonstrating this feature, consider Listing 10 which filters *Sentinel 2* [3] data, for the month of May 2019, bounded by the portuguese mainland geometry and excluding all images obscured by more than 20% of clouds.

Listing 10: Reducer operation on a given region.

```
images=ee.ImageCollection("COPERNICUS/S2")\
    .filterDate("2019-05-01", "2019-05-31")\
    .filterBounds(portugal.geometry())\
    .filter(ee.Filter.lt("CLOUDY_PIXEL_PERCENTAGE",
        20))
band1=images\
    .first()\
    .select("B4")
print(band1.reduceRegion(reducer=ee.Reducer.minMax(),
    \
        geometry=portugal.
            geometry(),\
        scale=10,\
        maxPixels=1e9).getInfo())

{'B4_max': 6186, 'B4_min': 392}
```

The variable `band1` represents the eldest image (first in the collection), the band “B4” contains the reflectance values for the red wave length. The reducer obtains the minimum and maximum values observed in the set of coordinates bounded by the geometry of the portuguese mainland, using a scale of $10m^2$, capped by a maximum of 10^9 coordinates.

Another perspective is to aggregate values across a set of bands to compute another band. The aggregate operation acts on every set of values in the scope of each coordinate (see Listing 11), resulting in a new set of values as a band.

Listing 11: Reducer operation resulting in a band.

```
band2=images\
    .select("B4")\
    .reduce(ee.Reducer.median())
print(band2.getInfo())

{'type': 'Image'
, 'bands': [{'id': 'B4_median'
, 'data_type': {'type': 'PixelType'
, 'precision': 'double'
, 'min': 0.0
, 'max': 65535.0}
, 'crs': 'EPSG:4326'
, 'crs_transform': [1.0, 0.0, 0.0, 0.0, 1.0, 0.0]}]}
```

In this case, *GEE* uses some syntactic sugar [16] that eases the burden of writing the whole expression. `reduce(ee.Reducer.median())` can be rewritten simply `median()`.

Also note that the result of Listing 11 is a single band, named after the original band concatenated with the reducer name, in this case “B4.median”.

III. BUILDING A COMPARISON PROTOTYPE

The main objective is to distinguish between:

- The *Code Editor* which is an integrated development environment, internet browser based, that uses *JavaScript* as the support language; and

- A *Python* local development environment using the *GEE API* package.

So, more than just comparing two *APIs*, this section will differentiate two development platforms.

Note that there is not a proper efficiency measure between both platforms. The only true difference, that can be discarded for this matter, is the overhead of the local processing. In the case of the *Code Editor* it would be difficult to single out the performance of a script running in the context of the internet browser without proper instrumentation of the code. Let's keep in mind that the relevant processing, and thus the performance indicators, depend totally on the *Google's* infrastructure.

Compute	Peak Mem	Count	Description
19.013	194M	399	no description available
1.688	981k	54	Encoding pixels to image
1.227	72M	58	Table query
0.742	19k	3009	(plumbing)
0.290	1.3M	216	Algorithm Collection.draw computing pixels
0.085	168M	58	Table decode
0.082	6.7k	57	Table metadata
0.029	9.9k	171	Algorithm Collection.draw
0.019	7.0k	3	Algorithm Geometry.bounds
0.016	600	324	Algorithm Image.visualize computing pixels
0.015	7.3k	168	Algorithm Image.visualize
0.012	3.5M	4	Algorithm Collection.geometry
0.000	144	1	Listing collection
-	28k	6	Algorithm Collection.iterate

Fig. 1: Sample of the *GEE Code Editor* profiler after running a script

Even though it does not make sense to benchmark any of these scenarios, there is a profiling tool in the *Code Editor* that may give us some insight on the usage of the *Google's* infrastructure for a specific script execution, regarding user processing and memory quota. See Figure 1 for an illustration of the profiler output after executing a script in *Code Editor*.

In Figure 1, the columns represent: (i) The “compute” cost of the operation; (ii) the “peak mem[ory]” usage of the operation; (iii) the “count” of running instances of the operation; and finally (iii) the “description” of the operation.

A. The prototype class

A prototype class, named *Territory*, was designed so that the difference between both support languages, using the same feature set, can be exemplified. The structure of the *Territory* class is show in Figure 2.

This class will wrap partial access to a dataset of the territory boundaries of countries and regions which exists in the *GEE* repository¹².

The following features will be provided: (i) a list of all country codes contained in the dataset by using the `countries` method; (ii) a list of all regions¹³ in the database for a given country code, implemented by the `regions` method; finally

¹²Specifically the LSIB: Large Scale International Boundary Polygons, Simplified published by the United States Department of State

¹³The concept of region is prone to be equivocal. For instance, in Portugal the dataset establishes three regions: mainland, Madeira and Azores.

- (iii) the geometry of the boundaries for a given region returning a `ee.Geometry` representation, using the `geometry` method.

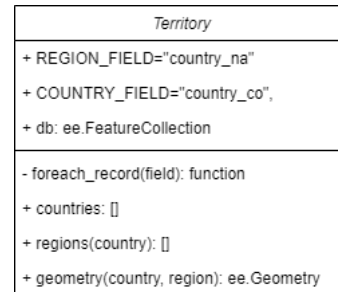


Fig. 2: *Unified Modeling Language (UML)* class diagram for the *Territory* class

Listing 12: *Territory* class implemented using *JavaScript*

```

exports.Territory = function()
{
  var me = {
    REGION_FIELD: "country_na",
    COUNTRY_FIELD: "country_co",
    db: ee.FeatureCollection("USDOS/LSIB_SIMPLE/2017"),
    countries: function()
    {
      var operation = this.db
        .iterate(this.foreach_record(this.COUNTRY_FIELD), ee.List([]));
      return operation
        .getInfo();
    },
    regions: function(country)
    {
      var operation = this.db
        .filter(ee.Filter.eq(this.COUNTRY_FIELD, country))
        .iterate(
          this.foreach_record(this.REGION_FIELD),
          ee.List([]));
      return operation
        .getInfo();
    },
    geometry: function(country, region)
    {
      var operation = this.db
        .filter(
          ee.Filter.and(
            ee.Filter.eq(this.COUNTRY_FIELD, country),
            ee.Filter.eq(this.REGION_FIELD, region)));
      return operation;
    },
    foreach_record: function(field) {
      return function(record, list)
      {
        return ee.List(list)
          .add(record.get(field))
      }
    }
  }
  return me;
}
  
```

Note that the first two methods will return native representations of a list of strings as defined in each of the *APIs* support languages but the latter will return a *JSON* proxy

representation of an `ee.FeatureCollection` object (see Listing 2 and Listing 3 to illustrate the distinction).

There is also a conceptual gap between the *UML* class diagram and the implementation using both supporting languages. Neither one supports the visibility scope, and the private members cannot be strictly enforced using the *JavaScript*'s syntax (see the implementation in Listing 12).

The visibility scope is only supported partially, but not constrained, in *Python* using the double underscore notation convention (see Listing 13).

Listing 13: Territory class implemented using *Python*

```
import ee

class Territory:
    REGION_FIELD = "country_na"
    COUNTRY_FIELD = "country_co"

    @property
    def db(self):
        return ee.FeatureCollection("USDOS/LSIB_SIMPLE/2017")

    def countries(self):
        operation = self.db\
            .iterate(self.__foreach_record(self.COUNTRY_FIELD), ee.List([]))
        return operation\
            .getInfo()

    def regions(self, country):
        operation = self.db\
            .filter(ee.Filter.eq(self.COUNTRY_FIELD, country))\
            .iterate(self.__foreach_record(self.REGION_FIELD), ee.List([]))
        return operation\
            .getInfo()

    def geometry(self, country, region):
        operation = self.db\
            .filter(\
                ee.Filter.And(\
                    ee.Filter.eq(self.COUNTRY_FIELD, country),\
                    ee.Filter.eq(self.REGION_FIELD, region)))
        return operation

    def __foreach_record(self, field):
        def __iterator(record, list):
            return ee.List(list).add(record.get(field))
        return __iterator
```

The iterator method works similarly to a map function (see Listing 8), except it accepts a list and returns another list, instead of a single element.

Observing both Listing 12 and Listing 13 it becomes clear that the relevant differences occur due to the diverse syntactical rules, but the object model is the same.

B. Referencing and reusing

In both environments a process was developed in order to output both: the full listing of country codes and the list of region names for Portugal.

To reuse the class in *JavaScript* one could include the class definition in the same script but that would lead to repeating the same code whenever needed with all the associated

drawbacks. However, the *JavaScript* browser based platform is well built and supports the development of reusable code by enabling the linkage of libraries using two major syntactical contributions to the *JavaScript* grammar: (i) a library script maybe referenced using a function (named `requires`) which has a single parameter, the location of the library file, in the user's code repository in the *Code Editor*; (ii) a predicate (named `export`) which exposes *JavaScript* variables and functions that will be referenced using the previous linked library.

For instance, considering a script named `bar` that implements a function named `baz`:

```
export.baz = function()...
```

A client script would use this library using:

```
foo = requires("bar")
foo.baz()
```

In Listing 12 the primitive `export` is used to expose the `Territory` class, while in Listing 14 the reference to a file is done using the function `require`. The return for this function will be a reference to the file kept in a `lib` variable. The reference to the `Territory` class is then possible by prefixing it with the `lib` variable.

Assuming the class in Listing 12 is placed in a file named *Territory*, in the user private assets in the *Code Editor* (named *users/geeprimer/lib*), the main function will be implemented using the script in Listing 14.

Listing 14: Main process script in *JavaScript*

```
var lib = require("users/geeprimer/lib:Territory");

function main()
{
    print("Starting...");
    var territories = new lib.Territory();

    print("Obtaining_country_codes:");
    var countries = territories.countries();
    print(countries);

    print("Obtaining_Portugal's_regions:");
    var regions = territories.regions("PO");
    print(regions);
}

main();
```

The same behavior in *Python* uses the builtin constructs. This sets a first meaningful difference between both *APIs* (see Listing 15).

Suppose the class in Listing 13 is placed in a file named *territory.py* in the same path as Listing 15, the main function will be implemented using the script in Listing 15. The same behavior might be extended to the standard packaging model used by *Python* developing new modules.

Another relevant difference, already described in section II, is the need in *Python* to use the initialization procedure that is implicit when using *JavaScript* in the *Code Editor*.

Listing 15: Main process script in *Python*

```
import ee
from territory import Territory
```

```
def main():
    print("Starting...")
    ee.Initialize()
    territories = Territory()

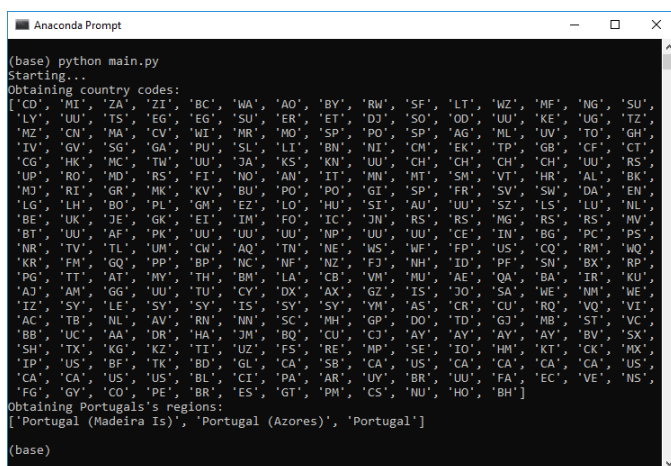
    print("Obtaining_country_codes:")
    countries = territories.countries()
    print(countries)

    print("Obtaining_Portugal's_regions:")
    regions = territories.regions("PO")
    print(regions)

main()
```

C. Console output and execution flow

Both *APIs* implement a console to support basic user interaction. The main script process uses the console to present the user with two human readable lists: one corresponding to the country codes; and another with the region names for Portugal's territorial boundaries. A function `print`, with the same syntax in both languages, is used to output text to the console.



```
(base) python main.py
Starting...
Obtaining country codes:
['CD', 'MI', 'ZA', 'ZI', 'BC', 'WA', 'AO', 'BY', 'RW', 'SF', 'LT', 'WZ', 'MF', 'NG', 'SU',
'LY', 'UU', 'TS', 'EG', 'EG', 'SU', 'ER', 'ET', 'DJ', 'SO', 'OD', 'UU', 'KE', 'UG', 'TZ',
'NZ', 'CU', 'MA', 'CV', 'WI', 'MR', 'MO', 'SP', 'PO', 'SP', 'AG', 'ML', 'UV', 'TO', 'GH',
'IV', 'GV', 'SG', 'GA', 'PU', 'SL', 'LI', 'BN', 'NI', 'CM', 'EK', 'TP', 'GB', 'CF', 'CT',
'CG', 'HK', 'MC', 'TM', 'UU', 'JA', 'KS', 'KN', 'UU', 'CH', 'CH', 'CH', 'UU', 'RS', 'RS',
'UP', 'RO', 'HD', 'RS', 'FI', 'MO', 'AN', 'ET', 'MN', 'HT', 'SH', 'VT', 'HR', 'AL', 'BK',
'NJ', 'RE', 'GR', 'HK', 'KV', 'BU', 'PO', 'PO', 'MN', 'HT', 'FR', 'SV', 'SW', 'DA', 'EN',
'LG', 'LH', 'BO', 'PL', 'GM', 'EZ', 'LO', 'HU', 'SI', 'AU', 'UU', 'SZ', 'LS', 'LU', 'NL',
'BE', 'UK', 'JE', 'GK', 'EI', 'IM', 'FO', 'IC', 'JN', 'RS', 'RS', 'MG', 'RS', 'RS', 'MV',
'BT', 'UU', 'AF', 'PK', 'UU', 'UU', 'NP', 'UU', 'UU', 'CE', 'IN', 'BG', 'PC', 'PS',
'NR', 'TV', 'TL', 'UM', 'CW', 'AO', 'TN', 'NE', 'WS', 'WF', 'FP', 'US', 'CO', 'RM', 'WQ',
'KR', 'FM', 'GO', 'PP', 'BP', 'NC', 'NF', 'NZ', 'FJ', 'NH', 'ID', 'PF', 'SN', 'BX', 'RP',
'PG', 'TT', 'AT', 'MY', 'TH', 'BM', 'LA', 'CB', 'VM', 'NU', 'AE', 'QA', 'BA', 'IR', 'KU',
'AJ', 'AM', 'GG', 'UU', 'TU', 'CY', 'DX', 'AX', 'GZ', 'IS', 'JO', 'SA', 'WE', 'NH', 'WE',
'IZ', 'SV', 'LE', 'SV', 'SY', 'IS', 'SY', 'SY', 'YH', 'AS', 'CR', 'CU', 'RO', 'VO', 'VE',
'AC', 'TB', 'NL', 'AV', 'RM', 'MM', 'SG', 'MH', 'GP', 'DO', 'TD', 'GJ', 'MB', 'ST', 'VC',
'BB', 'UC', 'AA', 'DR', 'HA', 'JM', 'BO', 'CU', 'CJ', 'AY', 'AY', 'AY', 'AV', 'BV', 'SX',
'SH', 'TK', 'KG', 'KZ', 'TI', 'UZ', 'FS', 'RE', 'MP', 'SE', 'TO', 'HM', 'KT', 'CK', 'HX',
'JP', 'US', 'BF', 'TK', 'BD', 'GL', 'CA', 'SB', 'CA', 'US', 'CA', 'CA', 'CA', 'US',
'CA', 'CA', 'US', 'US', 'BL', 'CI', 'PA', 'AR', 'UY', 'BR', 'UU', 'FA', 'EC', 'VE', 'NS',
'FG', 'GY', 'CO', 'PE', 'BR', 'ES', 'GT', 'PM', 'CS', 'NU', 'HO', 'BH']
Obtaining Portugal's regions:
['Portugal (Madeira Is)', 'Portugal (Azores)', 'Portugal']
(base)
```

Fig. 3: Console output of the main script execution in a *Python* environment.

The output console in *Python* is the standard output stream (see Figure 3). Each of the `print` calls is presented, incrementally, to the user as soon as the data is available and the process flow maybe interrupted without losing the output already produced (neither the execution context).

Also note that both *Territory* class methods used (`countries` and `regions`) return the *GEE* operation result by using `getInfo()` (see Listing 13). In *Python* this is essential to evaluate the operation result in *GEE* as shown in Listing 2 and Listing 3.

The *Code Editor* lives in a *HTML* document context and the user console is rendered as a *HTML* object (see Figure 4).

The *Code Editor* behavior is quite different from the *Python's* standard approach: (i) the console output is rendered once at the end of the script execution, it does not behave incrementally like a regular text console; (ii) if the process is interrupted all the execution context and console output are

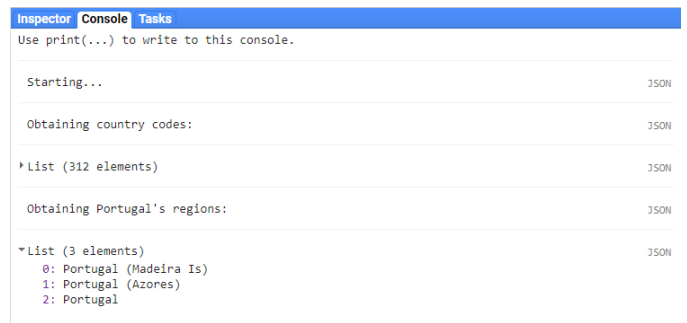


Fig. 4: Console output of the main script execution in the *Code Editor*.

lost, and during the script execution there is no way to present the user with any kind of progress information; and (iii) the output is rendered using presentation rules that are adequate to a rich graphical user interface (note the expanded panel which exposes the region names list output in Figure 4).

These behavioral differences are mostly visual and are side-effects of a deeper distinction between running the same script using a *Python* environment and the *JavaScript's* engine on the internet browser: (i) there is no difference to the internet browser's *JavaScript* engine between a long running script and a poorly written snippet of *JavaScript* code, the protection mechanism will prompt the user if the process takes too long to execute; and (ii) controlling the execution flow and recovering from a timeout error is challenging because the script is executing in a single blocking shared thread.

Finally the *Code Editor* makes some assumptions concerning the use of the `print` function. If the *Territory* class methods `countries` and `regions` are modified so that the return value (see Listing 12) is the operation instead of using `operation.getInfo()`, the content of the console in *Code Editor* would be the same as the one shown in Figure 4. What happens is that the *Code Editor* assumes that the user wants to dump the operation value instead of its *JSON* representation and implicitly uses the `getInfo()` method. This happens whenever a *GEE* object is passed to the `print` function.

However, it must be highlighted that this assumption is related to the console output in the *Code Editor*. The correct implementation of the *Territory* class is the one in Listing 12.

D. Extracting data

One of the most challenging issues related to using the *Code Editor* is the way it provides functionality to generate data that can be reused by other processes or integrated in a proper workflow.

There is an `Export` object that supports exporting data to a *Cloud* based storage (either *Google Drive* or *Google Cloud Storage*) or to the assets folder of the *GEE* user in the *Code Editor*. It is possible to persist data in the form of an image, a video (sequence of images) or a collection of georeferenced data using a set of standard formats like Comma Separated Files (*CSV*) or other dedicated formats like *GeoJSON*.

In the *Code Editor*, the procedure begins with creating an export task using the `Export` object (see Listing 16).

Listing 16: Creating an export task in *Code Editor*

```
var lib = require("users/geeprimer/lib:Territory");

function extract()
{
  var territories = new lib.Territory();
  Export.table.toDrive(
    {collection:territories.db,
     description:"ExportRegions",
     selectors: [
       territories.COUNTRY_FIELD,
       territories.REGION_FIELD]});
}

extract();
```

Once created, the new task will be presented in the “Tasks” list in the user interface (see Figure 5).

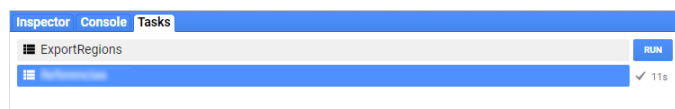


Fig. 5: Tasks list in the *Code Editor* showing the newly created task.

When the user starts the task, using the “Run” button, a dialog is presented where the original settings for the export task maybe redefined (see Figure 6). Pressing the “Run” button in the dialog the task will be started.

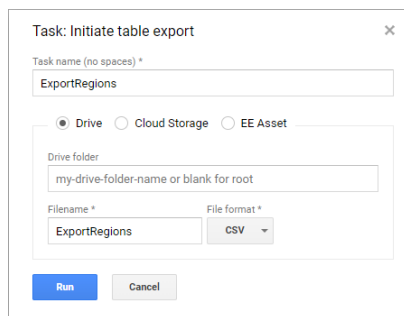


Fig. 6: Tasks list in the *Code Editor* showing the newly created task.

In this case, once generated, a *CSV* file with two columns with all the country code and corresponding regions’ names will be written to the user’s *Google Drive*.

Even though this process makes sense to most users, whom usage requirements are fulfilled by the *Code Editor*, it isn’t easy to integrate it in a process workflow neither is it practical to extract data to be used by other systems. Note that, in the example, the user interaction is needed for the process to be executed.

If the same data was to be part of an execution flow, the script in Listing 17 would extract the same data in a `data` variable, that could then be used as the input to some other function in *Python*.

Listing 17: Extracting the same data using *Python*

```
import ee
from territory import Territory

def extract():
  ee.Initialize()
  territories = Territory()

  data = territories\
    .db\
    .select(
      [territories.COUNTRY_FIELD,
       territories.REGION_FIELD],
      None,
      False)\
    .getInfo()

  return [feature["properties"] for feature in data[
    "features"]]

data = extract()
```

Note that, if the requirement is to generate a file with some kind of standard format, that could then be reused in a data flow, the resulting structure kept in the `data` variable could now be stored in a file using one of the many libraries in *Python*, for instance a “Pickle” file [17].

Listing 18: Creating an export task in *Python*

```
import ee
from territory import Territory

def extract():
  ee.Initialize()
  territories = Territory()

  return ee.batch.Export.table.toDrive(
    collection=territories.db,
    description="ExportRegions",
    selectors=[
      territories.COUNTRY_FIELD,
      territories.REGION_FIELD])

task = extract()
# Execute the task
task.start()
# The execution status of the task
task.status()

{'state': 'READY',
 'description': 'ExportRegions',
 'creation_timestamp_ms': 1587978085189,
 'update_timestamp_ms': 1587978085189,
 'start_timestamp_ms': 0,
 'task_type': 'EXPORT_FEATURES',
 'id': 'TNMJECYKYC6AXMMQ2INHXAOD',
 'name': 'projects/earthengine-legacy/operations/
TNMJECYKYC6AXMMQ2INHXAOD'}
```

Export tasks, like the example in Listing 16, might be of use in some other integration schemes: (i) some of the datasets generated by a *GEE* process can be considerably large and it would be inefficient, and sometimes impractical, to reuse using the *Python API* to serve as an intermediate input to some other process; (ii) a dataset resulting from a *GEE* application can have a single objective to be another item in the user assets in the *GEE* repository which might be reused by the user in other *GEE* processes or shared with other *GEE* users; or (iii) a prototype designed in the *Code Editor* may not have

any other objective other than generate a file to be delivered to a compatible *Cloud* storage.

While Listing 18 reproduces the same export task that was created in the *Code Editor* (see Listing 16), there are, however, two differences: (i) the export task is triggered by a call to `task.start`; and (ii) the execution status can be obtained by calling `task.status`.

The *Python API* fully supports the export tasks feature and allows a *GEE* application to implement asynchronous batch processes.

E. Imagery

The *Google Maps* platform is part of the user interface of the *Code Editor* and it is fully integrated. Points, lines and polygons maybe drawn directly in the *Google Maps* pane, and imported to a script, creating variables that describe the geometry drawn by the user.

Using *JavaScript* it is also possible to render georeferenced bitmaps or geometries (set of vectors) as layers in the *Google Maps* interface.

Listing 19: Render the territorial boundaries of Portugal's mainland as a layer in *Google Maps* using *JavaScript*

```
var lib = require("users/geeprimer/lib:Territory");

function overlay()
{
  var territories = new lib.Territory();
  var mainland = territories.geometry("PO", "Portugal");
  Map.centerObject(mainland);
  Map.addLayer(mainland, {color: "blue"}, "Portugal's mainland", true);
}

overlay();
```

Consider the script in Listing 19, it will draw a layer in the *Google Maps* pane with the territorial boundaries of the Portuguese mainland. The geometry is drawn in blue and the result would be similar to the one in Figure 7.



Fig. 7: *Google maps* pane in the *Code Editor* showing Portugal's mainland boundaries in blue.

Reproducing the same using the *Python* environment is not possible because there is no integration with *Google Maps* which is an internet browser based tool.

However, it is possible to draw just the Portugal's mainland boundaries using the script in Listing 20.

Using the *URL* representation in `url`, an internet browser may be used to visualize the image or the resource referenced by the *URL*, and maybe streamed into a local file using a standard *Python* library.

Listing 20: Draw the territorial boundaries of Portugal's mainland in a bitmap and obtain it's *URL* using *Python*

```
import ee
from territory import Territory

def overlay():
  ee.Initialize()
  territories = Territory()

  geometry = territories\
    .geometry("PO", "Portugal")
  image = geometry\
    .draw("blue")\
    .getThumbURL(
      {"dimensions": "1024x768", \
       "region": geometry\
         .geometry()\
         .bounds()\
         .getInfo(), \
       "format": "png"})

  return image

url = overlay()
```

However, if we observe the generated image, it is noticeable that it is skewed and distorted. This apparent anomaly is the result of drawing the geometry projected in a flat surface instead of the Earth's surface. The rendering of this data has to consider a reference system and geodetic datum [18], in order to make sense. To overcome this issue, we would have to use more features out of the *API* scope.

One option would be to use *Google Colaboratory* [14], which is a browser based interface that runs in the context of a remote *Python* environment. It can be used to support a small number of features similar to *Google Maps*.

In short, the distinction between both platforms is summarized in Table II.

TABLE II: Summary of the main differences between the *JavaScript* in the *Code Editor* environment and using the *API* in a *Python* environment.

<i>JavaScript</i> <i>Code Editor</i>	<i>Python</i>
Internet browser based tool, no deployment or setup needed.	There is a setup process that the user must follow in order to use the <i>API</i> .
Authentication and initialization are transparent to the user	Prior to using <i>GEE</i> , a initialization step is always needed.
Code libraries can be used using dedicated grammar contributions and the <i>Code Editor</i> code repository	Builtin constructs used to import packages and code is organized using <i>Python's</i> standard rules.

Table II (continued)

<i>JavaScript Code Editor</i>	<i>Python</i>
The console is presented to the user only at the end of the script. If interrupted, the whole context of the execution, up to then, is lost.	The console is a true character stream. The execution context, if the flow is interrupted, is kept.
Data may be extracted in the form of files, in batch processing, and requires human interaction. The supported formats are limited.	Export task feature is fully supported and can be triggered, and monitored, using code. However, data can be extracted using conventional patterns and can be streamed using other formats supported by <i>Python</i> .
Program flow and error recovery is limited due to the fact that the execution environment is an internet browser.	The interaction with <i>GEE</i> is similar to any other <i>API</i> in <i>Python</i> .
Integration with <i>Google Maps</i> is tight, and produces high quality imagery with ease.	There is no limitation obtaining georeferenced data, producing imagery requires more effort when compared to the <i>Code Editor</i> .

IV. CONCLUSION

GEE is a distributed processing platform, leveraged on the *Google* infrastructure, and a multi petabyte repository of georeferenced data, incrementally updated, presented to the user through an unified object model that hides the complexity of the sources, both supported on an *API* that is available using either *JavaScript* or *Python*. The features *GEE* offers significantly ease the burden of working with multi-spectral data compared to a typical on premises use of *GIS* tools.

Some of the main platform concepts and design patterns were illustrated with code samples that present an overall perspective of the *GEE* as well as exposing some of the syntactical and semantic differences between *JavaScript* and *Python APIs*.

Any of the *APIs* developed by *Google* are merely proxies to the *GEE*. The engine's performance is not warranted by any *SLA* from *Google*, and is affected by the number of concurrent users, and the usage level, at any time. Therefore, there is not a meaningful execution time difference between the internet browser hosted *Code Editor* and the *Python API* using a local environment. The *Code Editor* has a profiling feature that exposes an insight on the execution cost and memory usage that is of limited utility for this matter.

A prototype was implemented, of a class interface, using both *JavaScript* and *Python*. The prototype was used to distinguish between all the unique features in the different execution environments.

We conclude that, while the *Code Editor* is adequate to

prototype *GEE* applications that interact with *Google Maps* to produce imagery and map overlays, the *Python* environment *API* is more flexible and is able to integrate seamlessly in a typical software architecture.

REFERENCES

- [1] Noel Gorelick, Matt Hancher, Mike Dixon, Simon Iyushchenko, David Thau, and Rebecca Moore. *Google Earth Engine: Planetary-scale geospatial analysis for everyone. Remote Sensing of Environment*, 202:18–27, 2017.
- [2] Jeffrey Dean and Sanjay Ghemawat. *Mapreduce: Simplified data processing on large clusters*. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.
- [3] ESA. Sentinel 2 user guide. <https://earth.esa.int/web/sentinel/user-guides/sentinel-2-msi>, 2019. Accessed on 2020-04-25.
- [4] GDAL. GDAL - geospatial data abstraction library. <https://www.gdal.org/>, 2019. Accessed on 2020-04-25.
- [5] NASA. Measuring vegetation (NDVI & EVI). https://earthobservatory.nasa.gov/Features/MeasuringVegetation/measuring_vegetation_2.php, 2018. Accessed on 2020-04-25.
- [6] Vito Remote Sensing. Product types - proba-v. <http://proba-v.vgt.vito.be/en/product-types>, 2019. Accessed on 2020-04-25.
- [7] Rui S. Reis, Célia Gouveia, Nuno Datia, and M. P. M. Pato. Modelo preditivo de recuperação da vegetação afetada por incêndios florestais. In *INForum 2019 Atas do 11º Simpósio de Informática*, page 461–472. NOVA.FCT Editorial, 2019.
- [8] OGC. OGC GeoTIFF standard. <https://www.ogc.org/standards/geotiff>, 2020. Accessed on 2020-04-25.
- [9] QGIS. <https://qgis.org/en/site/>, 2019. Accessed on 2020-04-25.
- [10] Brent N Holben. Characteristics of maximum-value composite images from temporal AVHRR data. *International journal of remote sensing*, 7(11):1417–1434, 1986.
- [11] Google. Earth Engine data catalog. <https://developers.google.com/earth-engine/datasets/>, 2019. Accessed on 2020-04-25.
- [12] Google. Share your analyses using Earth Engine apps. <https://medium.com/google-earth/share-your-analyses-using-earth-engine-apps-1ac29939903f>, 2018. Accessed on 2020-04-25.
- [13] Google. Earth Engine data catalog. <https://developers.google.com/earth-engine/>, 2020. Accessed on 2020-04-26.
- [14] Google. Google Earth Engine guides, python installation. https://developers.google.com/earth-engine/python_install, 2019. Accessed on 2020-04-25.
- [15] Google. Google Earth Engine guides, band math. <https://developers.google.com/earth-engine/getstarted#band-math>, 2020. [Online; accessed 2020-01-15].
- [16] P. J. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, 01 1964.
- [17] Python Software Foundation. pickle — Python object serialization. <https://docs.python.org/3/library/pickle.html>, 2019. [Online; accessed 2020-04-25].
- [18] Wikipedia contributors. World Geodetic System — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=World_Geodetic_System&oldid=938668810, 2020. [Online; accessed 2020-02-03].