maap-docs

May 04, 2023

Contents

1 Getting Started 1

2 Science Examples 13

3 Technical Tutorials 77

4 System Reference Guide 153

5 Indices and tables 219

CHAPTER 1

Getting Started

1.1 Welcome

Welcome to the user documentation for the **Multi-Mission Algorithm and Analysis Platform (MAAP)**, a collaborative NASA/ESA project for the support of aboveground biomass research.

1.1.1 About the MAAP

The MAAP platform is designed to combine data, algorithms, and computational abilities for the processing and sharing of data related to NASA's GEDI, ESA's BIOMASS, and NASA/ISRO's NISAR missions. These missions generate vastly greater amounts of data than previous Earth observation missions. There are unique challenges to processing, storing, and sharing the relevant data due to the high data volume as well as with the data being collected from varied satellites, aircraft, and ground stations with different resolutions, coverages, and processing levels.

MAAP aims to address unique challenges by making it easier to discover and use biomass relevant data, integrating the data for comparison, analysis, evaluation, and generation. An algorithm development environment (ADE) is used to create repeatable and sharable science tools for the research community. The software is open source and adheres to ESA's and NASA's commitment to open data.

NASA and ESA are collaborating to further the interoperability of biomass relevant data and metadata. Tools have been developed to support a new approach to data stewardship and there is a data publication workflow for organizing and storing data and generating metadata to be discoverable in a cloud-based centralized location. The platform and data stewardship approaches are designed to ease barriers and promote collaboration between researchers, providers, curators, and experts across NASA and ESA.

This guide aims to help users get started with using the platform for searching, visualizing, accessing, processing, querying, and sharing biomass relevant data to the MAAP. These data, collected from satellites, aircraft, and ground stations, are organized into collections and granules. Collections are a grouping of files that share the same product specification. Granules are the individual files which are independently described, inventoried, and retrieved within a collection. Granules inherit additional attributes from their containing collection. Explanations of the various functions available in MAAP to use in the ADE will also be explored.

1.1.2 Planned Topics

The following is a list of existing and planned concepts to be covered in the completed version of this guide:

User Guides

- How To Git Clone a Repository Into the Algorithm Development Environment (ADE)
- Searching
  - How to Use the MAAP Earthdata Search Client Graphical User Interface
  - How to Search Collections in MAAP
  - How to Search Granules in MAAP
  - How to Search for and Compile a List of Granule IDs for Batch Processing
  - Available CMR Collections in MAAP
- Visualizing
  - How to Visualize Web Map Tile Service (WMTS) Layers
  - How to Visualize 3D Tiles Layers
- Accessing
  - How to Access Data from the MAAP
  - How to Access EDAV Data via Web Coverage Service
- Querying
  - How to Query Data from the MAAP via Python Client
  - Fields Within the Gedi Cal/Val Data
  - How to Query ATL08 Entwine Point Tiles
- User Data
  - How to Share Data Products to the MAAP Data Store

Registering an Algorithm

- How Do I Register My Algorithm to MAS?
- How Do I Delete My Algorithm from MAS?
- How Do I View Algorithms Already Registered in MAS?
- Running a Data Processing System (DPS) Job

How Do I Run a DPS Job?

- How Do I Check the Status of My DPS Job?
- How Do I Get the Results from My DPS Job?

Frequently Asked Questions (FAQs)

- How Do I Name My Workspace Something Useful?
- How Do I Share My Workspace with Another MAAP User?
- How Do I Copy My EarthData Search into My Jupyter Notebook?
- How Do I Import Granules over from My EarthData Search into My Jupyter Notebook?
- Which Files in My Workspace Are Persistent, and Which Ones Are Only Available in My Workspace?

Platform Technical Documentation

- ADE Workspace
  - How to Create Workspaces
  - How to Share Workspaces
- ADE Projects
  - How to Create a Project
  - How to Add a Project to Your Workspace
  - How to Update Your Project from inside Your Workspace
- ADE EarthData Search
  - How to Find Data
  - How to Add Data to a Notebook
  - How to Visualize Data in a Notebook
- ADE Common Mapping Client
  - How to Get Started
  - API Documentation
- ADE Jupyter Notebook Magics
  - Load the Inline Magics in Notebook
  - Available Magics
- ADE Workspace and Data Collaboration
  - Organizations
  - How to Share Data
- ADE SSH
  - SSH Access
- ADE User Interface Do's/Don't
  - Side Panel
  - Workspace Administration
  - How to Manage a Workspace
- DPS Algorithms
  - List Algorithms
  - Describe an Algorithm
  - Register an Algorithm
  - Delete an Algorithm
- DPS Job Submission
  - List Previous Jobs
  - Execute a Job
  - Check Job Status

  - Check Job Results
  - Delete Job
  - Dismiss Job

API Documentation

- `cmr` Operations Related to CMR
- `mas` Operations to Register an Algorithm
- `dps` Operations to Interface with HySDS Mozart
- `wmts` Retrieve Tiles
- `wms` WMS GetMap
- `members` Operations for MAAP Members
- `query` Operations for Query Service

1.2 Getting started with the MAAP

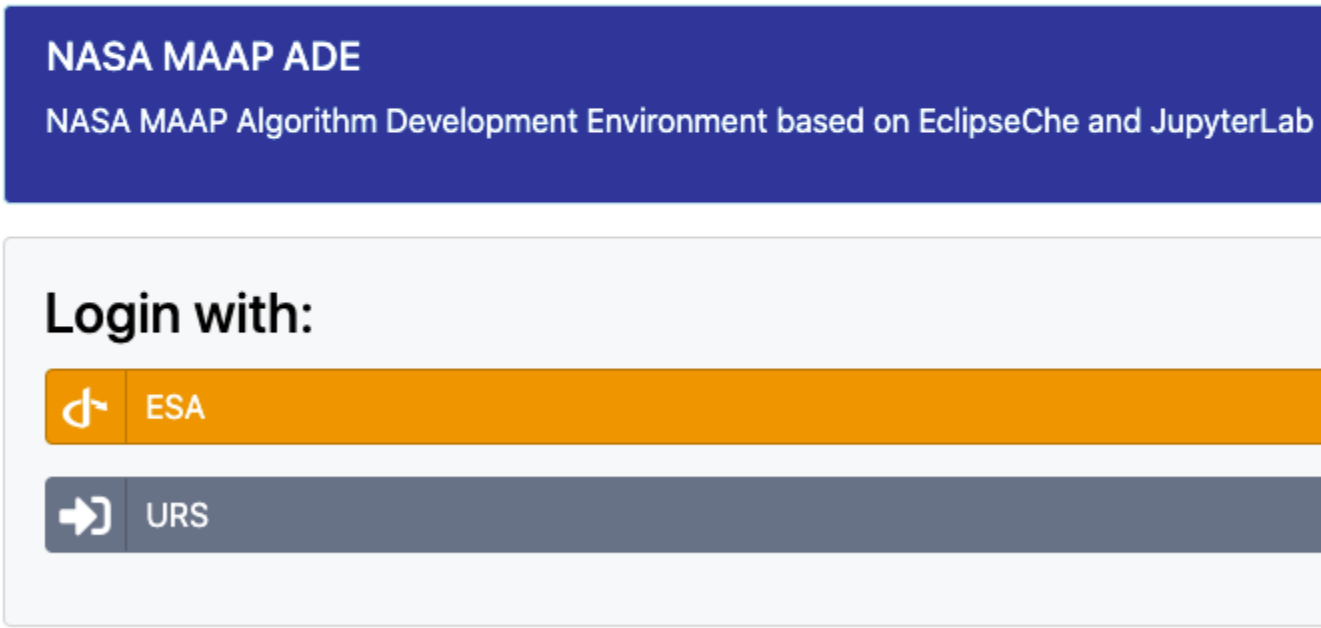
Learn how to sign up for an account and access all the services of the MAAP.

1.2.1 Signing up for an Earthdata Login account

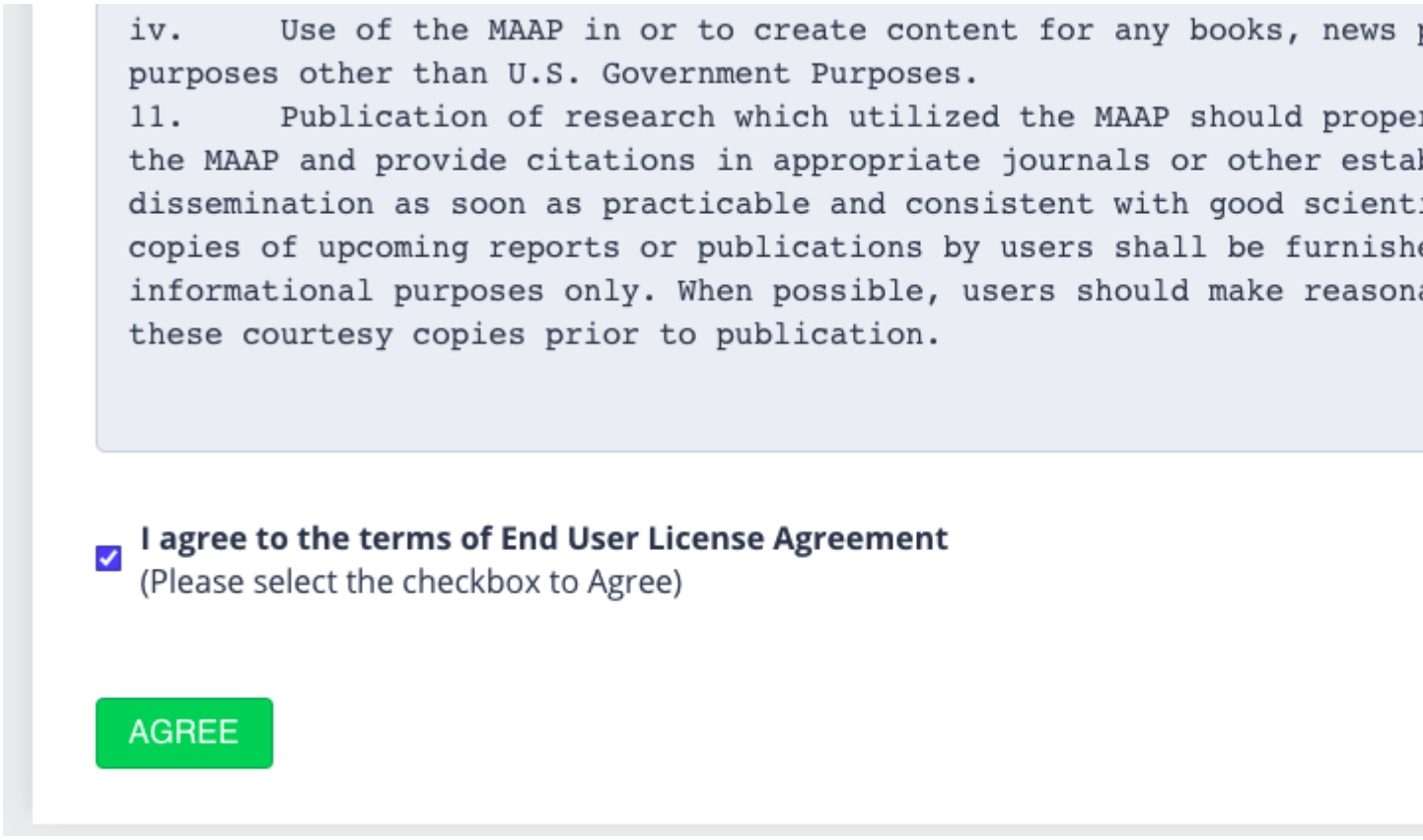
The MAAP offers accounts for NASA users through Earthdata Login. Before accessing the MAAP as a NASA user, you will need to create an Earthdata Login account. Anyone can register for an Earthdata Login profile here: https://urs.earthdata.nasa.gov/users/new.

1.2.2 Signing up for a new MAAP account

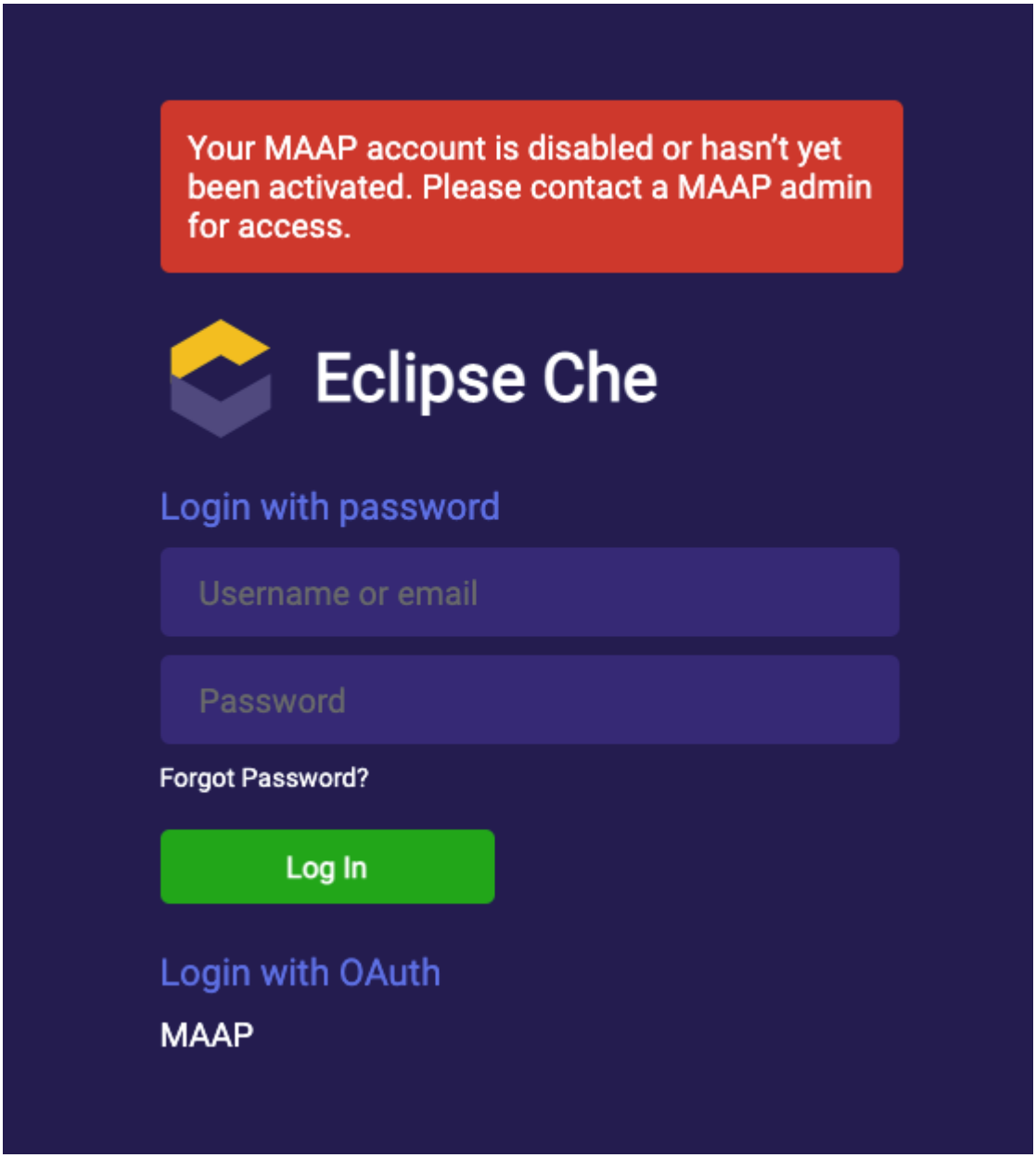
Once registered, you can register for a MAAP account by navigating to the MAAP ADE at http://ade.ops.maap-project.org. On your first visit, select the "URS" login button shown here:



If this is your first visit to the MAAP, you will be asked to agree to the MAAP Terms of Use:



Once registered, you should be redirected back to the MAAP ADE showing a disabled account message similar to this:



At this point, a MAAP administrator will approve your account, which will grant you access to the MAAP ADE. This process typically takes 5-10 minutes. To check on the status of your pending account, contact the MAAP team at support@maap-project.org.

Once your MAAP account is approved, you will receive an email notification using the address of your Earthdata Login account to let you know that your access is enabled.

1.2.3 Code repository access

After creating your MAAP account, you can create a code repository by navigating to the MAAP GitLab account at https://repo.ops.maap-project.org. This GitLab account is connected to your ADE workspaces automatically when signing into the ADE.

1.2.4 Uploading your public SSH key

In order to access your ADE workspaces using SSH, you'll need to upload your public SSH key to your MAAP profile using the MAAP portal at https://ops.maap-project.org.

Click on your profile name (or the "Login" button) in the top right corner shown here:

On your profile page, click the "Choose File" button to upload your key.

After uploading your key, the SSH connection will be enabled **after your next** login into the ADE.

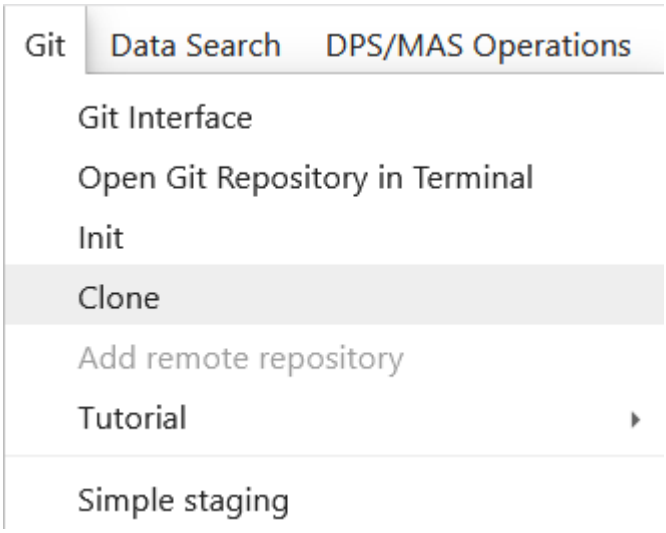
1.2.5 Note on future URL changes

The "*.ops.maap-project.org" URLs referenced in this guide will be converted to "*.maap-project.org" once the version one of MAAP is released in the near future. At that time, these URLs will be updated accordingly.

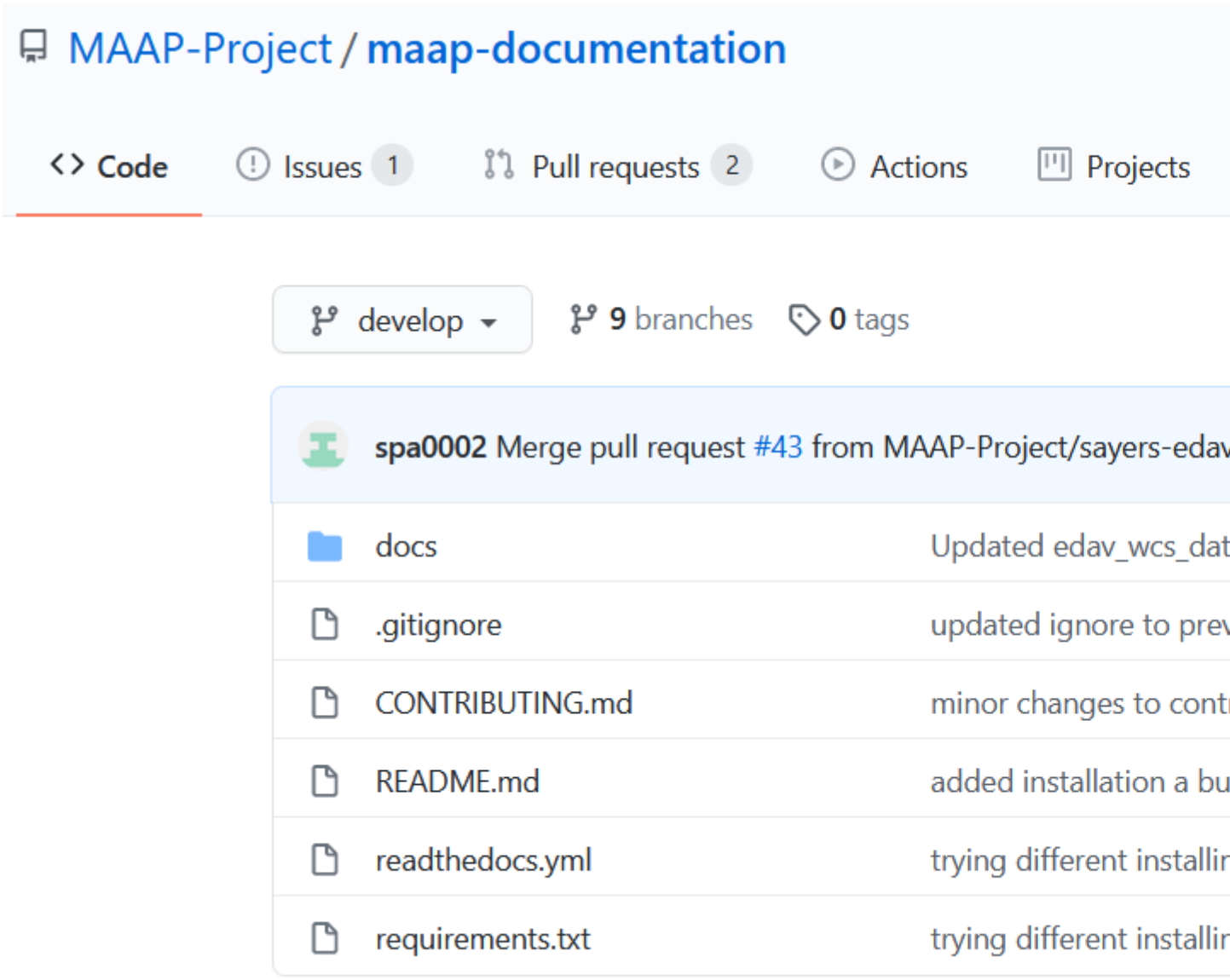
1.3 How To Git Clone a Repository Into the Algorithm Development Environment (ADE)

This example walks through cloning a repository into the ADE. Cloning a repository allows you to open, edit, and run files contained within the cloned repository. In this example, we look at cloning the "MAAP-Project/maap-documentation" Git repository, so that you are able to experiment with the code examples contained within this user documentation.

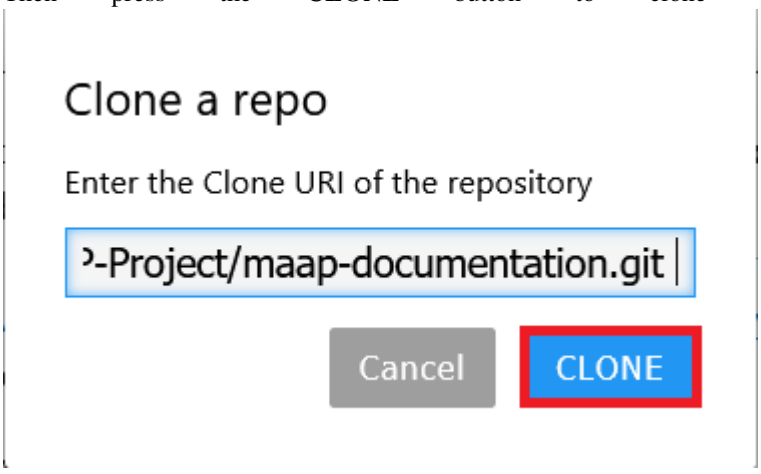
When inside of a workspace, navigate to **Git** tab at the top of the Jupyter window. Click it to see the option to **Clone**.



We can also access the "Clone a repo" dialogue box by selecting the **File Browser** tab on the JupyterLab sidebar and using the **Git Clone** button located at the top of the sidebar. The "Clone a repo" dialogue box prompts you to enter the URI of the repository you wish to clone. For this example we enter " https://github.com/MAAP-Project/maap-documentation.git ". This can be found by visiting the GitHub site for the "MAAP-Project/maap-documentation" Git repository and clicking the **Code** button.

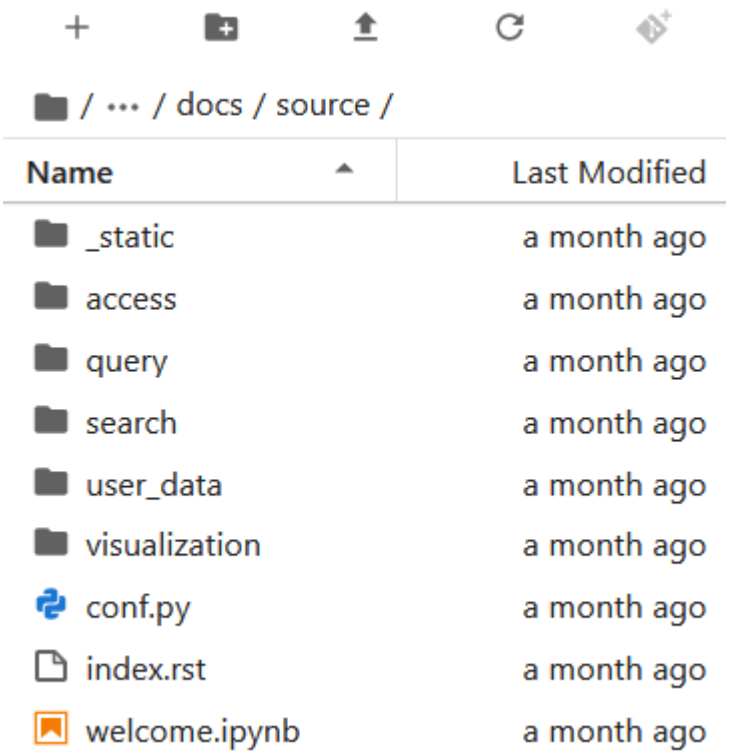


Then press the **CLONE** button to clone the repository into the ADE.



With the **File Browser** tab on the JupyterLab sidebar selected, a folder named "maap-documentation" should now

appear under the “projects” directory. Folders for the various sections of the guide can be found in the “docs/source/” directory.



To open the IPython Notebook for an example, go to a section directory and double-click on appropriate “.ipynb” file. For more information about the using Git in Jupyterlab, see https://github.com/jupyterlab/jupyterlab-git .

CHAPTER 2

Science Examples

2.1 Harmonized Landsat Sentinel (HLS) Search and Composite

In this tutorial, we will search the LPDAAC for HLS 30m optical imagery that intersects an AOI. We will filter the catalog based on a cloud cover % and build a maximum-NDVI composite image, including a suite of popular indices. We will begin by installing any packages we need and importing the packages that we will use.

If needed the following packages should be installed:

[1]:
```
# cleanup data: removes data files that should be replaced
!rm -rf ./local-s3.json
!rm -rf ./sample.json

if False:
    !conda install -c conda-forge pystac-client -y
    !conda install -c conda-forge rio-tiler -y
```

Prerequisites

- geopandas
- folium
- pystac-client
- rio_tiler

[2]:
```
# Uncomment the following lines to install these packages if you haven't already.
# !pip install geopandas
# !pip install folium
# !pip install pystac-client
# !pip install rio_tiler
```

We will now import a suite of packages that we will need:

[3]:
```
from maap.maap import MAAP
maap = MAAP(maap_host='api.ops.maap-project.org')
import geopandas as gpd
import folium
import h5py
import pandas
import matplotlib
import matplotlib.pyplot as plt
from shapely.geometry import Polygon
from pystac_client import Client
import datetime
import os
import rasterio as rio
import boto3
import json
import botocore
from rasterio.session import AWSSession
from rio_tiler.io import COGReader
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
```

```
/opt/conda/lib/python3.7/site-packages/geopandas/_compat.py:115: UserWarning: The␣
↪Shapely GEOS version (3.11.1-CAPI-1.17.1) is incompatible with the GEOS version␣
↪PyGEOS was compiled with (3.8.1-CAPI-1.13.3). Conversions between both will be slow.
  shapely_geos_version, geos_capi_version_string
```

2.1.1 Creating an AOI

To begin we will create a polygon in a forested area in Virginia, USA. We will do this by providing a number of lat/lon coordinates and creating an AOI.

[4]:
```
lon_coords = [-80, -80, -79., -79, -80]
lat_coords = [39, 38.2, 38.2, 39, 39]

polygon_geom = Polygon(zip(lon_coords, lat_coords))
crs = 'epsg:4326'
AOI = gpd.GeoDataFrame(index=[0], crs=crs, geometry=[polygon_geom])
AOI_bbox = AOI.bounds.iloc[0].to_list()
```

We can visualize this polygon using a folium interactive map.

[5]:
```
m = folium.Map([38.5,-79.3], zoom_start=9, tiles='OpenStreetMap')
folium.GeoJson(AOI).add_to(m)
folium.LatLngPopup().add_to(m)
m
```

[5]:
```
<folium.folium.Map at 0x7ff7aba868d0>
```

2.1.2 Accessing the HLS Data

To be able to access the HLS imagery we need to activate a 'rasterio' AWS session. This will give us the required access keys that we need to search and read data from the LPDAAC S3 bucket.

[6]:
```
def get_aws_session_DAAC():
    """Create a Rasterio AWS Session with Credentials"""
    creds = maap.aws.earthdata_s3_credentials('https://data.lpdaac.earthdatacloud.
↪nasa.gov/s3credentials')
    boto3_session = boto3.Session(
        aws_access_key_id=creds['accessKeyId'],
        aws_secret_access_key=creds['secretAccessKey'],
        aws_session_token=creds['sessionToken'],
        region_name='us-west-2'
    )
    return AWSSession(boto3_session)
```

[7]:
```
print('Getting AWS credentials...')
aws_session = get_aws_session_DAAC()
print('Finished')
```

```
Getting AWS credentials...
Finished
```

Now that we have our session credentials set up, we can search the HLS catalog using the following function, filtering by spatial extent (our AOI) and a time window:

[8]:
```
def query_stac(year, bbox, max_cloud, api, start_month_day, end_month_day):
    print('opening client')
    catalog = Client.open(api)

    date_min = str(year) + '-' + start_month_day
    print('start_month_day:\t\t', start_month_day)
    print('end_month_day:\t\t', end_month_day)
    date_max = str(year) + '-' + end_month_day
    start_date = datetime.datetime.strptime(date_min, "%Y-%m-%d")
    end_date = datetime.datetime.strptime(date_max, "%Y-%m-%d")
    start = start_date.strftime("%Y-%m-%dT00:00:00Z")
    end = end_date.strftime("%Y-%m-%dT23:59:59Z")

    print('start date, end date:\t\t', start, end)

    print('\nConducting HLS search now...')

    search = catalog.search(
        collections=["HLSL30.v2.0"],
        datetime=[start,end],
        bbox=bbox,
        max_items=500, # for testing, and keep it from hanging
        # query={"eo:cloud_cover":{"lt":20}} #doesn't work
    )
    print(f"Search query parameters:\n{search}\n")
    results = search.get_all_items_as_dict()

    return results
```

Here we run our STAC search and write the results out to a JSON file so we can access it later.

[9]:
```
search = query_stac(2020, AOI_bbox, 20, 'https://cmr.earthdata.nasa.gov/stac/LPCLOUD',
↪ '06-01', '09-30')

with open("./sample.json", "w") as outfile:
    json.dump(search, outfile)
```

```
opening client
start_month_day:                06-01
end_month_day:                  09-30
start date, end date:             2020-06-01T00:00:00Z 2020-09-30T23:59:59Z

Conducting HLS search now...
Search query parameters:
<pystac_client.item_search.ItemSearch object at 0x7ff7aba2b6d0>
```

So far, we have not filtered by cloud cover, which is a common filtering parameter for optical imagery. We can use the metadata files included in our STAC search to find the amount of cloud cover in each file and decide if it meets our threshold or not. We will set a cloud cover threshold of 50%. While we are doing this, we will also change the URLs to access the optical imagery from "https://" to AWS S3 URLs ("S3://").

[10]:
```
def write_local_data_and_catalog_s3(catalog, bands, save_path, cloud_cover, s3_path=
↪"s3://"):
    '''Given path to a response json from a sat-api query, make a copy changing urls␣
↪to local paths'''
    creds = maap.aws.earthdata_s3_credentials('https://data.lpdaac.earthdatacloud.
↪nasa.gov/s3credentials')
    aws_session = boto3.session.Session(
        aws_access_key_id=creds['accessKeyId'],
        aws_secret_access_key=creds['secretAccessKey'],
        aws_session_token=creds['sessionToken'],
        region_name='us-west-2')
    s3 = aws_session.client('s3')

    with open(catalog) as f:
        clean_features = []
        asset_catalog = json.load(f)

        # Remove duplicate scenes
        features = asset_catalog['features']

        for feature in features:
            umm_json = feature['links'][6]['href']
            umm_data = !curl -i {umm_json}
            for line in umm_data:
                if "CLOUD_COVERAGE" in line:
                    cc_perc = (int(line.split("CLOUD_COVERAGE")[-1].split('"')[4]))
                    if cc_perc > cloud_cover:
                        pass
                    else:
                        try:
                            for band in bands:
                                output_file = feature['assets'][band]['href'].replace(
↪'https://data.lpdaac.earthdatacloud.nasa.gov/', s3_path)
                                bucket_name = output_file.split("/")[2]
                                s3_key = "/".join(output_file.split("/")[3:])
                                head = s3.head_object(Bucket = bucket_name, Key = s3_
↪key, RequestPayer='requester')
                                if head['ResponseMetadata']['HTTPStatusCode'] == 200:
                                    feature['assets'][band]['href'] = output_file
                            clean_features.append(feature)
                        except botocore.exceptions.ClientError as e:
                            if e.response['Error']['Code'] == "404":
```

(continues on next page)

(continued from previous page)

```
                                print(f"The object does not exist. {output_file}")
                            else:
                                raise
        # save and updated catalog with local paths
        asset_catalog['features'] = clean_features
        local_catalog = catalog.replace('sample', 'local-s3')
        with open(local_catalog,'w') as jsonfile:
            json.dump(asset_catalog, jsonfile)

        return local_catalog
```

When run, this will create a new JSON file that will only include files that meet our cloud cover threshold and have S3 URLs.

[11]:
```
local_cat = write_local_data_and_catalog_s3('./sample.json', ['B02','B03','B04','B05',
↪'B06','B07','Fmask'], './', 60, s3_path="s3://")
```

Now that we have images that meet our requirements, we will composite them into a multiband image for our AOI. We will composite the image on a band-by-band basis, so we first have to get a list of all the file paths for each band.

[12]:
```
def GetBandLists(inJSON, bandnum, tiles=['']):
    bands = dict({2:'B02', 3:'B03', 4:'B04', 5:'B05', 6:'B06', 7:'B07',8:'Fmask'})
    BandList = []
    with open(inJSON) as f:
        response = json.load(f)
    for i in range(len(response['features'])):
        try:
            getBand = response['features'][i]['assets'][bands[bandnum]]['href']
            # check 's3' is at position [:2]
            if getBand.startswith('s3', 0, 2):
                BandList.append(getBand)
        except Exception as e:
            print(e)

    BandList.sort()
    return BandList
```

We will build a band list of file paths for each image band. We will also access the 'fmask' band to mask out clouds.

[13]:
```
blue_bands = GetBandLists('./local-s3.json', 2)
green_bands = GetBandLists('./local-s3.json', 3)
red_bands = GetBandLists('./local-s3.json', 4)
nir_bands = GetBandLists('./local-s3.json', 5)
swir_bands = GetBandLists('./local-s3.json', 6)
swir2_bands = GetBandLists('./local-s3.json', 7)
fmask_bands = GetBandLists('./local-s3.json', 8)

print('number of each images in each band = ', len(blue_bands))
```

```
number of each images in each band =  21
```

2.1.3 Reading in HLS data and creating composite

We will not read all of the HLS data, as we only need what's included in our AOI. To do this "windowed read" we need to know the dimensions, in pixels, of the window. To do this we need to convert our AOI to a projected coordinate system (UTM) and calculate the number of columns and rows we will need, using a pixel resolution of 30m.

[14]:
```
def get_shape(bbox, res=30):
    left, bottom, right, top = bbox
    width = int((right-left)/res)
    height = int((top-bottom)/res)

    return height,width

# convert to m
AOI_utm = AOI.to_crs('epsg:32617')
height, width = get_shape(AOI_utm.bounds.iloc[0].to_list())
```

When building a maximum-NDVI composite, the first data we need is the red and NIR bands to make an NDVI band for each image. We will use 'riotiler' to perform a windowed read of our data. We will also read the 'fmask' layer as we can use this to mask out unwanted pixels.

[15]:
```
def ReadData(file, in_bbox, height, width, epsg="epsg:4326", dst_crs="epsg:4326"):
    '''Read a window of data from the raster matching the tile bbox'''
    with COGReader(file) as cog:
        img = cog.part(in_bbox, bounds_crs=epsg, max_size=None, dst_crs=dst_crs,␣
↪height=height, width=width)

    return (np.squeeze(img.as_masked().astype(np.float32)) * 0.0001)
```

Our AWS session has an expiry time. Now would be a good time to renew our access key to ensure we do not encounter any timeout issues.

[16]:
```
print('Getting AWS credentials...')
aws_session = get_aws_session_DAAC()
print('Finished')
```

```
Getting AWS credentials...
Finished
```

Using our renewed session, for each band we will read in the relevant band in each of our images, creating an array of image bands.

[17]:
```
with rio.Env(aws_session):
    print('reading red bands...')
    red_stack = np.array([ReadData(red_bands[i], AOI_bbox, height, width, epsg="epsg:
↪4326", dst_crs="epsg:4326") for i in range(len(red_bands))])
    print('reading nir bands...')
    nir_stack = np.array([ReadData(nir_bands[i], AOI_bbox, height, width, epsg="epsg:
↪4326", dst_crs="epsg:4326") for i in range(len(nir_bands))])
    print('reading fmask bands...')
    fmask_stack = np.array([ReadData(fmask_bands[i], AOI_bbox, height, width, epsg=
↪"epsg:4326", dst_crs="epsg:4326") for i in range(len(fmask_bands))])
    print('finished')

print("number of red_bands = ", np.shape(red_stack)[0])
```

```
reading red bands...
reading nir bands...
reading fmask bands...
finished
number of red_bands =  21
```

Now that we have our red band array and NIR band array we can make an NDVI image. While we do this, we will also apply our 'fmask' to remove any pixels that contain clouds.

[18]:
```
ndvi_stack =  np.ma.array(np.where(((fmask_stack==1)), -9999, (nir_stack-red_stack)/
↪(nir_stack+red_stack)))
ndvi_stack = np.where((~np.isfinite(ndvi_stack)) | (ndvi_stack>1), -9999, ndvi_stack)
```

```
/opt/conda/lib/python3.7/site-packages/ipykernel_launcher.py:1: RuntimeWarning:␣
↪divide by zero encountered in true_divide
  """Entry point for launching an IPython kernel.
```

At this point, we can plot our images and see which ones contain cloud coverage. These images have gaps where there is no data or cloud coverage.

[19]:
```
fig, axes = plt.subplots(3,7, figsize=(33,30))

for i, ax in enumerate(axes.flat):
    ndvi_stack[i] = np.where((ndvi_stack[i]>1) | (ndvi_stack[i]<-1), 0, ndvi_stack[i])
    ax.axes.xaxis.set_visible(False)
    ax.axes.yaxis.set_visible(False)
    ax.imshow(ndvi_stack[i], cmap='viridis', clim=(0.1, 0.5))
```

Now that we have a stack of NDVI bands, we can create an index band that maps the pixel position from each band where the NDVI value is greatest. We can use this to locate the pixels we want to use in our composite.

[20]:
```
# Create Bool mask where there is no value in any of the NDVI layers
print("Make NDVI valid mask")
print("shape:\t\t", np.ma.array(ndvi_stack).shape)
MaxNDVI = np.ma.max(np.ma.array(ndvi_stack),axis=0)
BoolMask = np.ma.getmask(MaxNDVI)
del MaxNDVI

## Get the argmax index positions from the stack of NDVI images
print("Get stack nan mask")
ndvi_stack = np.ma.array(ndvi_stack)
print("Calculate Stack max NDVI image")
NDVImax = np.nanargmax(ndvi_stack,axis=0)
## create a tmp array (binary mask) of the same input shape
NDVItmp = np.ma.zeros(ndvi_stack.shape, dtype=bool)

## for each dimension assign the index position (flattens the array to a LUT)
print("Create LUT of max NDVI positions")
for i in range(np.shape(ndvi_stack)[0]):
    NDVItmp[i,:,:]=NDVImax==i
```

```
Make NDVI valid mask
shape:          (21, 3006, 2951)
Get stack nan mask
Calculate Stack max NDVI image
Create LUT of max NDVI positions
```

Now that we have our NDVI lookup table and a list of all the images for each band, we can make composites based on the maximum NDVI value. For this, we will use the following two functions.

[21]:
```
def CollapseBands(inArr, NDVItmp, BoolMask):
    inArr = np.ma.masked_equal(inArr, 0)
    inArr[np.logical_not(NDVItmp)]=0
    compImg = np.ma.masked_array(inArr.sum(0), BoolMask)
    #print(compImg)
    return compImg

def CreateComposite(file_list, NDVItmp, BoolMask, in_bbox, height, width, epsg, dst_
↪crs):
    MaskedFile = [ReadData(file_list[i], in_bbox, height, width, epsg, dst_crs) for i␣
↪in range(len(file_list))]
    Composite=CollapseBands(MaskedFile, NDVItmp, BoolMask)
    return Composite
```

For each band, we will read all the images (for the required band) and create a composite based on the maximum NDVI value.

[22]:
```
aws_session = get_aws_session_DAAC()
with rio.Env(aws_session):
    print('Creating Blue Composite')
    blue_comp = CreateComposite(blue_bands, NDVItmp, BoolMask, AOI_bbox, height,␣
↪width, "epsg:4326", "epsg:4326")
    print('Creating Green Composite')
    green_comp = CreateComposite(green_bands, NDVItmp, BoolMask, AOI_bbox, height,␣
↪width, "epsg:4326", "epsg:4326")
    print('Creating Red Composite')
    red_comp = CreateComposite(red_bands, NDVItmp, BoolMask, AOI_bbox, height, width,
↪"epsg:4326", "epsg:4326")
```

(continues on next page)

(continued from previous page)

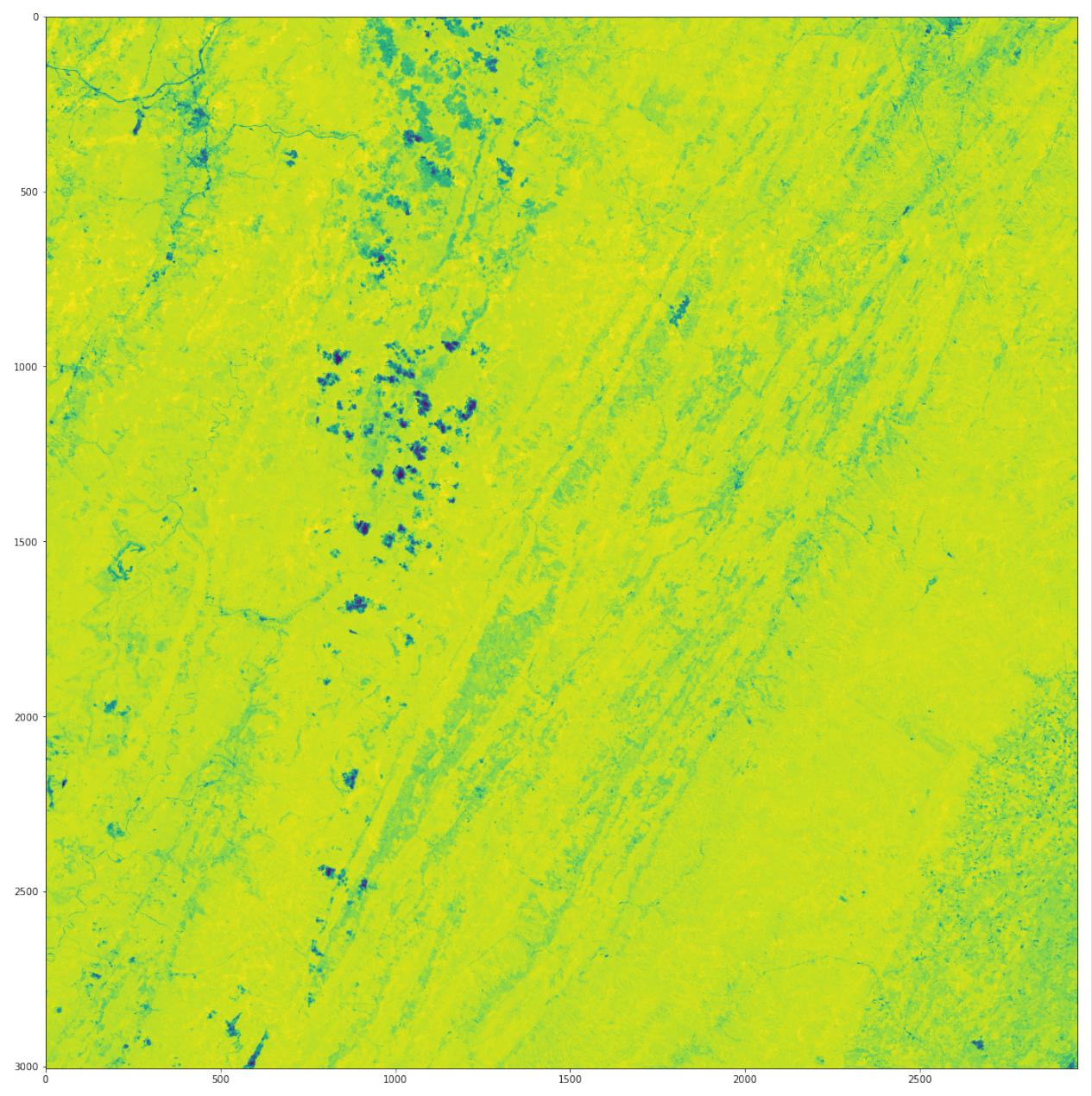
```
    print('Creating NIR Composite')
    nir_comp = CreateComposite(nir_bands, NDVItmp, BoolMask, AOI_bbox, height, width,
↪"epsg:4326", "epsg:4326")
    print('Creating SWIR Composite')
    swir_comp = CreateComposite(swir_bands, NDVItmp, BoolMask, AOI_bbox, height,␣
↪width, "epsg:4326", "epsg:4326")
    print('Creating SWIR2 Composite')
    swir2_comp = CreateComposite(swir2_bands, NDVItmp, BoolMask, AOI_bbox, height,␣
↪width, "epsg:4326", "epsg:4326")
    print('Creating NDVI Composite')
    ndvi_comp = CollapseBands(ndvi_stack, NDVItmp, BoolMask)
    print('Creating fmask Composite')
    fmask_comp = CollapseBands(fmask_stack, NDVItmp, BoolMask)
```

```
Creating Blue Composite
Creating Green Composite
Creating Red Composite
Creating NIR Composite
Creating SWIR Composite
Creating SWIR2 Composite
Creating NDVI Composite
Creating fmask Composite
```

We can look at our NDVI composite image and see we now have a complete image for our AOI.

[23]:
```
fig, axes = plt.subplots(1,1, figsize=(20,20))
ax.axes.xaxis.set_visible(False)
ax.axes.yaxis.set_visible(False)
plt.imshow(np.where(fmask_comp==1, -9999, ndvi_comp))
```

[23]:
```
<matplotlib.image.AxesImage at 0x7ff8d3fdd5d0>
```



Now that we have a 7-band composite image, we can use these bands to calculate a suite of common vegetation indices using the following functions.

[24]:
```
# SAVI
def calcSAVI(red, nir):
    savi = ((nir - red)/(nir + red + 0.5))*(1.5)
    print('\tSAVI Created')
    return savi

# NDMI
def calcNDMI(nir, swir):
    ndmi = (nir - swir)/(nir + swir)
    print('\tNDMI Created')
```

(continues on next page)

(continued from previous page)

```
    return ndmi

# EVI
def calcEVI(blue, red, nir):
    evi = 2.5 * ((nir - red) / (nir + 6 * red - 7.5 * blue + 1))
    print('\tEVI Created')
    return evi

# NBR
def calcNBR(nir, swir2):
    nbr = (nir - swir2)/(nir + swir2)
    print('\tNBR Created')
    return nbr

# MSAVI
def calcMSAVI(red, nir):
    msavi = (2 * nir + 1 - np.sqrt((2 * nir + 1)**2 - 8 * (nir - red))) / 2
    print('\tMSAVI Created')
    return msavi
```

We can call these functions and make our additional indices.

[25]:
```
# calculate covars
print("Generating covariates")
SAVI = calcSAVI(red_comp, nir_comp)
#print("NDMI")
NDMI = calcNDMI(nir_comp, swir_comp)
#print("EVI")
EVI = calcEVI(blue_comp, red_comp, nir_comp)
#print("NBR")
NBR = calcNBR(nir_comp, swir2_comp)
MSAVI = calcMSAVI(red_comp, nir_comp)
```

```
Generating covariates
        SAVI Created
        NDMI Created
        EVI Created
        NBR Created
        MSAVI Created
```

We have a suite of 12 bands now, and we can merge them together into a single 12-band image stack.

[26]:
```
print("\nCreating raster stack...\n")
stack = np.transpose([blue_comp, green_comp, red_comp, nir_comp, swir_comp, swir2_
↪comp, ndvi_comp, SAVI, MSAVI, NDMI, EVI, NBR], [0, 1, 2])
stack = np.where(fmask_comp==1, -9999, stack)
print(np.shape(stack))
```

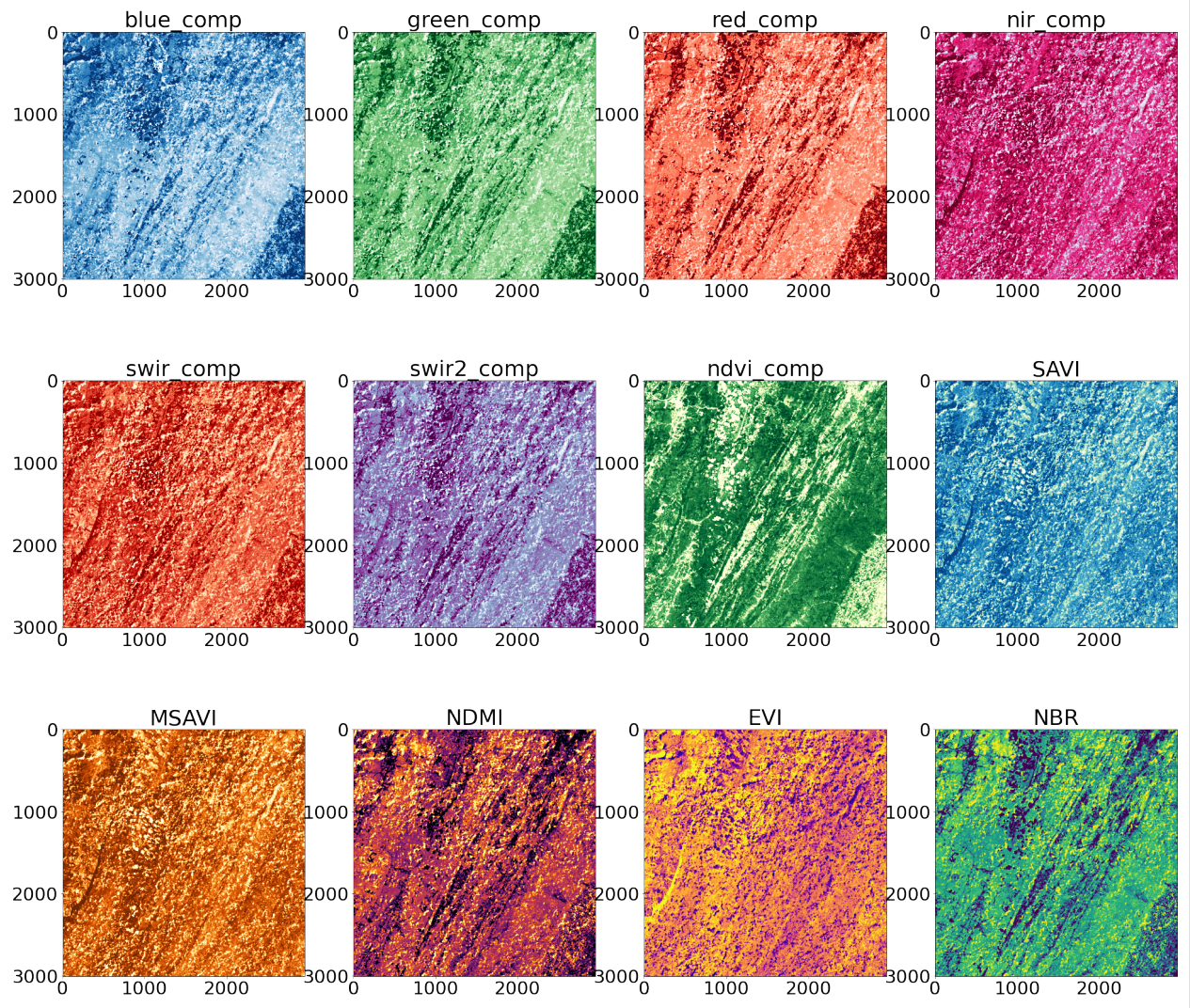
```

Creating raster stack...

(12, 3006, 2951)
```

2.1.4 Display Results

We can look at each of these bands by using 'matplotlib' to plot each one individually.

[27]:
```
n: int = len(stack)
#topo_cmaps = ["bone","Spectral", "magma", "RdBu", "coolwarm"]
topo_cmaps = ['Blues','Greens','Reds','PuRd','OrRd','BuPu','YlGn','GnBu','YlOrBr',
↪'inferno','plasma', 'viridis']
print(stack.shape)
bandnames = ['blue_comp', 'green_comp', 'red_comp', 'nir_comp', 'swir_comp', 'swir2_
↪comp', 'ndvi_comp', 'SAVI', 'MSAVI', 'NDMI', 'EVI', 'NBR']

font = {'size'   : 30}
matplotlib.rc('font', **font)

#axs = [[0,0],[0,1],[0,2],[0,3],[0,4],[0,5],[1,0],[1,1],[1,2],[1,3],[1,4],[1,5],[2,0],
↪[2,1],[2,2],[2,3],[2,4],[2,5]]
fig, axes = plt.subplots(3,4, figsize=(33,30))
print(axes.flat)
for i, ax in enumerate(axes.flat):
    ax.imshow(stack[i], cmap=topo_cmaps[i], clim=(np.percentile(stack[i], 10), np.
↪percentile(stack[i], 90)))
    ax.set_title(bandnames[i])
```

```
(12, 3006, 2951)
<numpy.flatiter object at 0x5609d86b7ce0>
```



We can also visualize our composite NDVI band on our interactive 'folium' map. You can see that even though we found multiple images, by using a windowed read we were able to just read and process the data we needed.

[28]:
```
m = folium.Map(location=[38.6, -78.5], zoom_start=9, tiles='CartoDB positron')
AOI_bx = AOI.bounds
#folium.GeoJson(AOI, style_function=lambda x: {'fillColor': 'orange','opacity':0}).
↪add_to(m)
geo_r = folium.raster_layers.ImageOverlay(np.ma.getdata(ndvi_comp), opacity=1,␣
↪bounds=[[AOI_bx['miny'][0],AOI_bx['minx'][0]],[AOI_bx['maxy'][0],AOI_bx['maxx
↪'][0]]])
geo_r.add_to(m)

m
```

[28]:
```
<folium.folium.Map at 0x7ff7aba43810>
```

2.2 Global Ecosystem Dynamics Investigation (GEDI) Level 2A Product Tutorial

This tutorial aims to provide information and code to help users get started working with the GEDI Level 2A (GEDI02_A) product using the MAAP. Information about the GEDI02_A product may be found at the Data Set Landing Page. We will start by importing the packages which will allow us to search for, access, explore, and visualize GEDI02_A product data.

Note: This Jupyter notebook utilizes the `folium` and `pystac_client` packages. If you do not have these packages installed, uncomment the lines and run the following code block.

[1]:
```
# !pip install folium
# !pip install pystac_client
```

For this tutorial, we will import `boto3`, `folium`, `h5py`, `pandas`, `exists` from `os.path` and `Client` from `pystac_client` as shown in the following codeblock.

[2]:
```
# Install packages
import boto3
import folium
import h5py
import os
import pandas as pd
from maap.maap import MAAP
from os.path import exists
from pystac_client import Client
```

2.2.1 Searching for and accessing GEDI02_A data

As of the time of the writing of this tutorial (2/10/23), two recommended ways for searching and accessing GEDI02_A data for use on the MAAP ADE are through the `maap-stac` as well as through NASA's Common Metadata Repository (CMR). The methods for using these two ways are different and documented in the following two sub-sections.

Via `maap-stac`

To search for data from the GEDI02_A product, we will use the `Client` package to open the `maap-stac` URL <https://stac.maap-project.org/>`__ and assign this to a variable (`client` in this case).

[3]:
```
# Open the maap-stac URL with the Client package
URL = 'https://stac.maap-project.org/'
client = Client.open(URL)
```

Now we can use the client specified above to search for data within the GEDI02_A product. Let's search for the first item that is found in the GEDI02_A collection and assign this to a variable (`search` in this case).

[4]:
```
collection = 'GEDI02_A' # assign collection name
# Search for 1st item found in the collection
search = client.search(
    max_items = 1,
    collections = collection,
)
```

Let's inspect this item using the `get_all_items()` function.

[5]:
```
# Inspect first item
item = search.get_all_items()[0]
item
```

[5]:
```
<Item id=GEDI02_A_2021272190541_O15849_04_T03030_02_003_02_V002>
```

After running the code block above, you should receive an output such as the output shown above. Click the arrow next to the item to see information such as the ID, bounding box coordinates, datetime, and more. In order to access the data, we will use the `item` variable from the above section in order to extract the necessary information to set and display `bucket`, `key`, and `filename` variables. These will be useful to use as arguments when downloading the file.

[6]:
```
# Use the item variable to extract information about the bucket, key, and file name
href = item.assets['data'].href
path_parts = href.split('/')
bucket = path_parts[2]
key = href.split(bucket)[1][1:]
filename = path_parts[-1]
# Display arguments
bucket, key, filename
```

[6]:
```
('nasa-maap-data-store',
 'file-staging/nasa-map/GEDI02_A___002/2021.09.29/GEDI02_A_2021272190541_O15849_04_
↪T03030_02_003_02_V002.h5',
 'GEDI02_A_2021272190541_O15849_04_T03030_02_003_02_V002.h5')
```

Now let's set an `s3` variable using the `boto3.client` function. We can use the function `download_file` along with the arguments we set in the previous block to download the GEDI02_A data we need.

[7]:
```
# Set s3 variable
s3 = boto3.client('s3')
# If file already exists, do not download file
if exists(filename):
    print("File already downloaded")
# Otherwise, download the file
else:
    s3.download_file(bucket, key, filename)
    print("Finished downloading")
```

```
Finished downloading
```

After the previous block has finished running, we should see the message `Finished downloading` and the file should appear in the same directory that the Jupyter notebook is in.

Via NASA's CMR

To search for data from the GEDI02_A product using NASA's CMR, we invoke the `MAAP` constructor, setting the `maap_host` argument to `'api.ops.maap-project.org'`.

[8]:
```
# Invoke the MAAP using the MAAP host argument
maap = MAAP(maap_host='api.ops.maap-project.org')
```

Now we can use the `searchGranule` function to find granule data within the collection, using the collection short name "GEDI_02A". Note that we can use `searchGranule`'s `cmr_host` argument to specify a CMR instance external to MAAP and the `readable_granule_name` argument to find granules matching either granule UR or producer granule id (please see the API documentation for more information). In order to download data from NASA's CMR, we will set a variable to the first result from the `results` we obtained.

[9]:
```
# Search for granule data using CMR host name and collection short name, and readable_
↪granule_name arguments
results = maap.searchGranule(
    cmr_host='cmr.earthdata.nasa.gov',
    short_name='GEDI02_A',
    readable_granule_name = "GEDI02_A_2021272190541_O15849_04_T03030_02_003_02_V002.h5
↪")
# Download first result
filename = results[0].getData()
```

If desired, the `print` function can be utilized to see the file name and directory.

[10]:
```
# Print file directory
print(filename)
```

```
./GEDI02_A_2021272190541_O15849_04_T03030_02_003_02_V002.h5
```

2.2.2 Explore

Now that we have downloaded the data, let's look into what it contains.

[11]:
```
# Create variable containing info from the file we downloaded
gediL2A = h5py.File(filename, 'r')
```

GEDI02_A data has data for 8 different beams. Let's create a list of beam names to help explore the data.

[12]:
```
# Create list of beam names
beamNames = [g for g in gediL2A.keys() if g.startswith('BEAM')]
beamNames
```

[12]:
```
['BEAM0000',
 'BEAM0001',
 'BEAM0010',
 'BEAM0011',
 'BEAM0101',
 'BEAM0110',
 'BEAM1000',
 'BEAM1011']
```

Now let's explore the information available for one of the beams (in this case 'BEAM0000').

[13]:
```
# Get list of objects in the data pertaining to 'BEAM0000'
beam = beamNames[0]
gediL2A_objs = []
gediL2A.visit(gediL2A_objs.append)
gediSDS = [o for o in gediL2A_objs if isinstance(gediL2A[o], h5py.Dataset)]
[i for i in gediSDS if beam in i][0:20]
```

[13]:
```
['BEAM0000/ancillary/l2a_alg_count',
 'BEAM0000/beam',
 'BEAM0000/channel',
 'BEAM0000/degrade_flag',
 'BEAM0000/delta_time',
 'BEAM0000/digital_elevation_model',
 'BEAM0000/digital_elevation_model_srtm',
 'BEAM0000/elev_highestreturn',
 'BEAM0000/elev_lowestmode',
```

(continues on next page)

(continued from previous page)

```
 'BEAM0000/elevation_bias_flag',
 'BEAM0000/elevation_bin0_error',
 'BEAM0000/energy_total',
 'BEAM0000/geolocation/elev_highestreturn_a1',
 'BEAM0000/geolocation/elev_highestreturn_a2',
 'BEAM0000/geolocation/elev_highestreturn_a3',
 'BEAM0000/geolocation/elev_highestreturn_a4',
 'BEAM0000/geolocation/elev_highestreturn_a5',
 'BEAM0000/geolocation/elev_highestreturn_a6',
 'BEAM0000/geolocation/elev_lowestmode_a1',
 'BEAM0000/geolocation/elev_lowestmode_a2']
```

2.2.3 Visualize

Now that we've seen the various labels within the /BEAM0000 group, let's use this information to visualize the GEDI orbit path for our scenes. To start, we shall get samples for various shots, the beam number, longitude, latitude, and quality flags. We can use these samples to create and display a `pandas` dataframe.

[14]:
```
# Set variables for shot, beam number, longitude, latitude, and quality flag samples
lonSample, latSample, shotSample, qualitySample, beamSample = [], [], [], [], []
lats = gediL2A[f'{beamNames[0]}/lat_lowestmode'][()]
lons = gediL2A[f'{beamNames[0]}/lon_lowestmode'][()]
shots = gediL2A[f'{beamNames[0]}/shot_number'][()]
quality = gediL2A[f'{beamNames[0]}/quality_flag'][()]
for i in range(len(shots)):
    if i % 100 == 0:
        shotSample.append(str(shots[i]))
        lonSample.append(lons[i])
        latSample.append(lats[i])
        qualitySample.append(quality[i])
        beamSample.append(beamNames[0])
# Create a pandas dataframe containing the sample information
latslons = pd.DataFrame({'Beam': beamSample, 'Shot Number': shotSample, 'Longitude':␣
↪lonSample,
                         'Latitude': latSample, 'Quality Flag': qualitySample})
# Display the dataframe
latslons
```

[14]:

	Beam	Shot Number	Longitude	Latitude	Quality Flag
0	BEAM0000	158490000400502383	52.337698	0.599011	0
1	BEAM0000	158490000400502483	52.367554	0.556129	0
2	BEAM0000	158490000400502583	52.396932	0.514800	0
3	BEAM0000	158490000400502683	52.426548	0.472746	0
4	BEAM0000	158490000400502783	52.456248	0.430695	0
...	...	...	...	...	...
1685	BEAM0000	158490000400670883	135.837810	-51.605016	0
1686	BEAM0000	158490000400670983	135.921291	-51.605653	0
1687	BEAM0000	158490000400671083	136.003206	-51.606299	0
1688	BEAM0000	158490000400671183	136.086629	-51.606741	0
1689	BEAM0000	158490000400671283	136.168523	-51.607159	0

[1690 rows x 5 columns]

We can now create a map of the orbit path using the dataframe that we have created and utilizing the `folium.Map` and `folium.Circle` functions. Include `map` in the code block to inspect the map which is created within the Jupyter notebook.

[15]:
```
# Create a map
map = folium.Map(
    location=[
        latslons.Latitude.mean(),
        latslons.Longitude.mean()
    ], zoom_start=3, control_scale=True
)
# Add variables to the map
for index, location_info in latslons.iterrows():
    folium.Circle(
        [location_info["Latitude"], location_info["Longitude"]],
        popup=f"Shot Number: {location_info['Shot Number']}"
    ).add_to(map)
# Display map
map
```

[15]:
```
<folium.folium.Map at 0x7f04c05a8690>
```

References:

- https://lpdaac.usgs.gov/resources/e-learning/getting-started-gedi-l2a-version-2-data-python/
- https://cmr.earthdata.nasa.gov/search/site/docs/search/api.html

2.3 Introduction to the GEDI02_B Product

Source: https://lpdaac.usgs.gov/products/gedi02_bv002/

The Global Ecosystem Dynamics Investigation (GEDI) mission aims to characterize ecosystem structure and dynamics to enable radically improved quantification and understanding of the Earth's carbon cycle and biodiversity. The GEDI instrument produces high resolution laser ranging observations of the 3-dimensional structure of the Earth. GEDI is attached to the International Space Station (ISS) and collects data globally between 51.6° N and 51.6° S latitudes at the highest resolution and densest sampling of any light detection and ranging (lidar) instrument in orbit to date. Each GEDI Version 2 granule encompasses one-fourth of an ISS orbit and includes georeferenced metadata to allow for spatial querying and subsetting.

The purpose of the GEDI Level 2B Canopy Cover and Vertical Profile Metrics product (GEDI02_B) is to extract biophysical metrics from each GEDI waveform. These metrics are based on the directional gap probability profile derived from the L1B waveform. Metrics provided include canopy cover, Plant Area Index (PAI), Plant Area Volume Density (PAVD), and Foliage Height Diversity (FHD). The GEDI02_B product is provided in HDF5 format and has a spatial resolution (average footprint) of 25 meters.

The GEDI02_B data product contains 96 layers for each of the eight-beam ground transects (or laser footprints located on the land surface). Datasets provided include precise latitude, longitude, elevation, height, canopy cover, and vertical profile metrics. Additional information for the layers can be found in the GEDI Level 2B Data Dictionary.

2.3.1 Opening GEDI02_B files and exploring its datasets

This tutorial opens and exampines a GEDI02_B file and demonstrates how to plot a subset of points in order to understand the spatial extent of that file.

- Author: Aimee Barciauskas
- Date: December 16, 2022

Resources used: * Getting Started with GEDI L2B Version 2 Data in Python (LPDAAC Tutorial)

[1]:
```
import boto3
import folium
import h5py
from os.path import exists
from pystac_client import Client
import pandas as pd
```

[2]:
```
# STAC API root URL
URL = 'https://stac.maap-project.org/'
client = Client.open(URL)
```

[3]:
```
collection = 'GEDI02_B'
search = client.search(
    max_items = 1,
    collections = collection,
)
```

[4]:
```
item = search.get_all_items()[0]
item
```

[4]:
```
<Item id=GEDI02_B_2021299195557_O16268_02_T06679_02_003_01_V002>
```

2.3.2 Access and open the GEDI02_B HDF5 file

Below, we break down the S3 URL into its bucket and key components to support downloading the file with S3. Note, this will only work on platforms configured with access to the GEDI directory of the NASA MAAP Data Store (`s3://nasa-maap-data-store/GEDI*`). Further, it is strongly preferred that any transfer of data is done on AWS services in AWS region us-west-2 in order for data transfer costs to be minimized.

[5]:
```
href = item.assets['data'].href
path_parts = href.split('/')
bucket = path_parts[2]
key = href.split(bucket)[1][1:]
filename = path_parts[-1]
bucket, key, filename
```

[5]:
```
('nasa-maap-data-store',
 'file-staging/nasa-map/GEDI02_B___002/2021.10.26/GEDI02_B_2021299195557_O16268_02_
↪T06679_02_003_01_V002.h5',
 'GEDI02_B_2021299195557_O16268_02_T06679_02_003_01_V002.h5')
```

[14]:
```
# don't want to run this more than once!
s3 = boto3.client('s3')

if exists(filename):
    print("file already downloaded")
else:
    s3.download_file(bucket, key, filename)
    print("finished downloading")
```

```
file already downloaded
```

[15]:
```
gediL2B = h5py.File(filename, 'r')
```

[16]:
```
beamNames = [g for g in gediL2B.keys() if g.startswith('BEAM')]
beamNames
```

[16]:
```
['BEAM0000',
 'BEAM0001',
 'BEAM0010',
 'BEAM0011',
 'BEAM0101',
 'BEAM0110',
 'BEAM1000',
 'BEAM1011']
```

2.3.3 List all the science data sets

to understand what type of data is stored in this dataset better.

[ ]:
```
beam = beamNames[0]
gediL2B_objs = []
gediL2B.visit(gediL2B_objs.append)                                          #␣
↪Retrieve list of datasets
gediSDS = [o for o in gediL2B_objs if isinstance(gediL2B[o], h5py.Dataset)]  # Search␣
↪for relevant SDS inside data file
[i for i in gediSDS if beam in i][0:20]
```

2.3.4 Plot a subset of the points

First, we create a subset of points to facilitate plotting.

[19]:
```
lonSample, latSample, shotSample, qualitySample, beamSample = [], [], [], [], []  #␣
↪Set up lists to store data

# Open the SDS
lats = gediL2B[f'{beamNames[0]}/geolocation/lat_lowestmode'][()]
lons = gediL2B[f'{beamNames[0]}/geolocation/lon_lowestmode'][()]
shots = gediL2B[f'{beamNames[0]}/geolocation/shot_number'][()]
quality = gediL2B[f'{beamNames[0]}/l2b_quality_flag'][()]

# Take every 1000th shot and append to list
for i in range(len(shots)):
    if i % 1000 == 0:
        shotSample.append(str(shots[i]))
        lonSample.append(lons[i])
        latSample.append(lats[i])
        qualitySample.append(quality[i])
        beamSample.append(beamNames[0])

# Write all of the sample shots to a dataframe
latslons = pd.DataFrame({'Beam': beamSample, 'Shot Number': shotSample, 'Longitude':␣
↪lonSample, 'Latitude': latSample,
                         'Quality Flag': qualitySample})
latslons
```

[19]:

	Beam	Shot Number	Longitude	Latitude	Quality Flag
0	BEAM0000	162680000200086772	71.176248	-0.652634	0
1	BEAM0000	162680000200087772	71.475272	-0.228396	0

(continues on next page)

(continued from previous page)

```
2   BEAM0000  162680000200088772   71.772920   0.194259            0
3   BEAM0000  162680000200089772   72.070465   0.616924            0
4   BEAM0000  162680000200090772   72.369240   1.040828            0
..       ...                 ...         ...        ...          ...
89  BEAM0000  162680000200175772  102.130037  35.159365            0
90  BEAM0000  162680000200176772  102.593719  35.509484            0
91  BEAM0000  162680000200177772  103.061891  35.857398            0
92  BEAM0000  162680000200178772  103.522337  36.194505            0
93  BEAM0000  162680000200179772  103.755363  36.363258            0

[94 rows x 5 columns]
```

[22]:
```
map = folium.Map(
    location=[
        latslons.Latitude.mean(),
        latslons.Longitude.mean()
    ], zoom_start=3, control_scale=True
)

for index, location_info in latslons.iterrows():
    folium.Circle(
        [location_info["Latitude"], location_info["Longitude"]],
        popup=f"Shot Number: {location_info['Shot Number']}"
    ).add_to(map)

map
```

[22]:
```
<folium.folium.Map at 0x124ce5520>
```

2.4 Search GEDI L2B and plot canopy heights

In this tutorial, we will use the integrated Earthdata search function in MAAP ADE (https://ade.ops.maap-project.org/) to search for some GEDI L2B data for an area of interest. We will then download some of this data and read its attributes in preparation for some analysis. We will perform a spatial subset on the data to reduce data volumes and we will finally make some basic plots of our data. We will start by importing some of the modules we will need.

Prerequisites

- geopandas
- folium

[1]:
```
# Uncomment the following lines to install these packages if you haven't already.
# !pip install geopandas
# !pip install folium
```

[2]:
```
from maap.maap import MAAP
maap = MAAP(maap_host='api.ops.maap-project.org')
import geopandas as gpd
import folium
import h5py
import pandas
import matplotlib
import matplotlib.pyplot as plt
import shapely
```

```
/opt/conda/lib/python3.7/site-packages/geopandas/_compat.py:115: UserWarning: The␣
↪Shapely GEOS version (3.11.1-CAPI-1.17.1) is incompatible with the GEOS version␣
↪PyGEOS was compiled with (3.8.1-CAPI-1.13.3). Conversions between both will be slow.
  shapely_geos_version, geos_capi_version_string
```

We will search and download GEDI L2B data using the bounding box of a vector AOI. Firstly, an AOI over the Shenandoah National Park will be created and then we will plot its location on a map.

[3]:
```
# Using bounding coordinates to create a polygon around Shenadoah National Park
lon_coords = [-78.32129105072025, -78.04618813890727, -78.72985973163064, -79.
↪0158578082679, -78.32129105072025]
lat_coords = [38.88703610703791, 38.74909216350823, 37.88789051477522, 38.
↪03177640342157, 38.88703610703791]

polygon_geom = shapely.geometry.polygon.Polygon(zip(lon_coords, lat_coords))
crs = 'epsg:4326'
AOI = gpd.GeoDataFrame(index=[0], crs=crs, geometry=[polygon_geom])
```

We can get the bounding box of the AOI so we can use it as a spatial limit on our data search. GeoPandas has a function for returning the spatial coordinates of a bounding box:

[4]:
```
# Get the bounding box of the shp
bbox = AOI.bounds
# print the bounding box coords
print('minx = ', bbox['minx'][0])
print('miny = ', bbox['miny'][0])
print('maxx = ', bbox['maxx'][0])
print('maxy = ', bbox['maxy'][0])
```

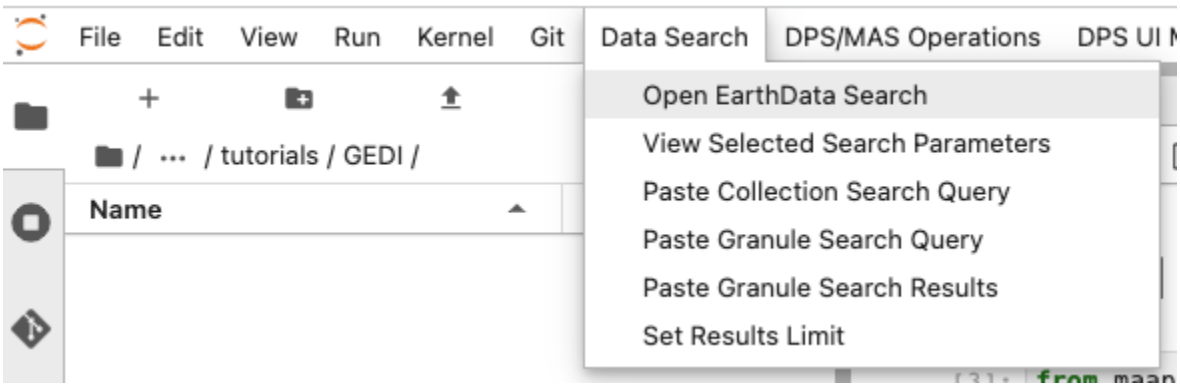
```
minx =  -79.0158578082679
miny =  37.88789051477522
maxx =  -78.04618813890727
maxy =  38.88703610703791
```

Let's look at our AOI on an interactive map using folium.

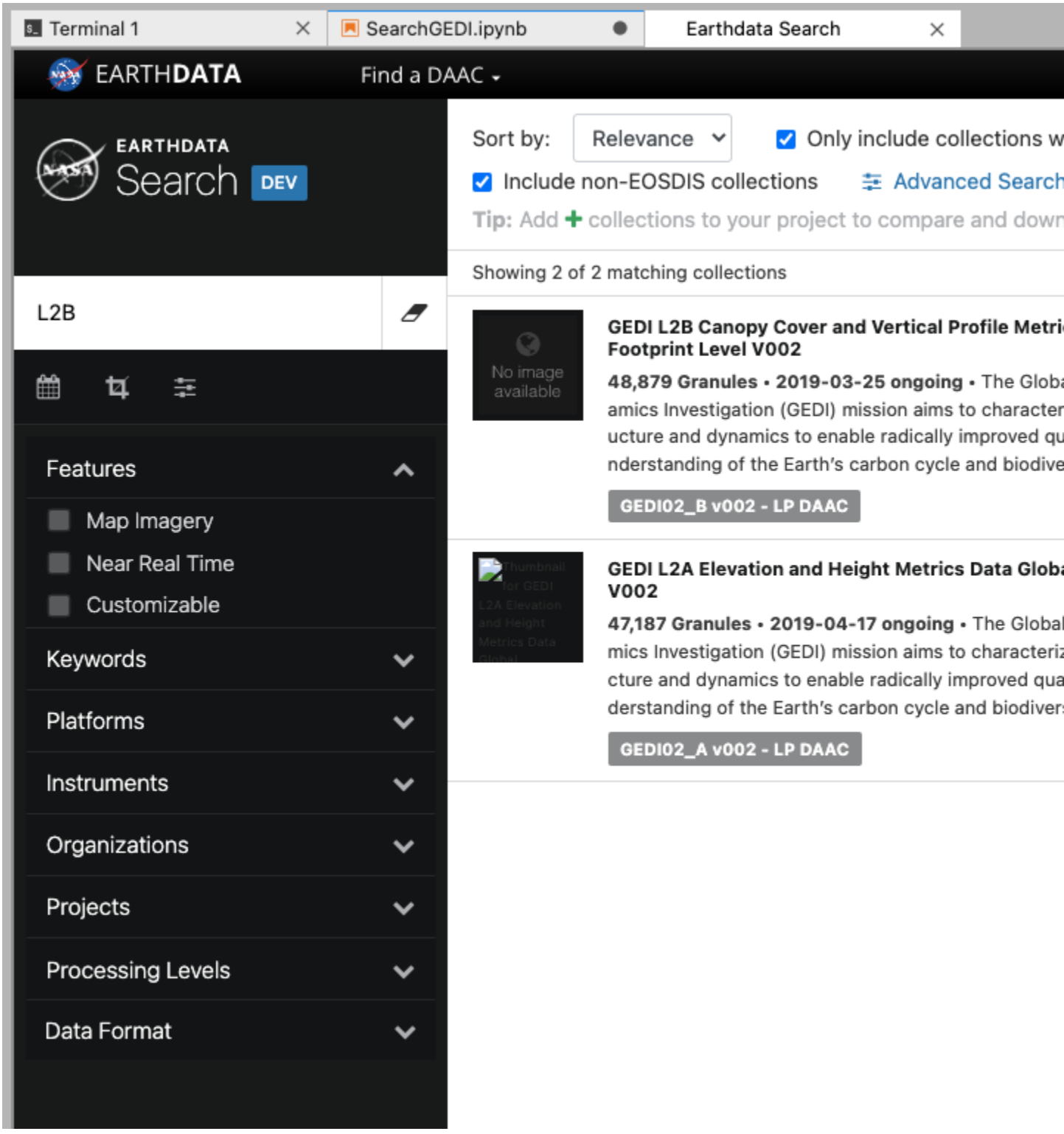
[5]:
```
m = folium.Map(location=[38.5, -78], zoom_start=9, tiles='CartoDB positron')
geo_j = folium.GeoJson(data=AOI, style_function=lambda x: {'fillColor': 'orange'})
geo_j.add_to(m)
m
```

[5]:
```
<folium.folium.Map at 0x7f1bece5c090>
```

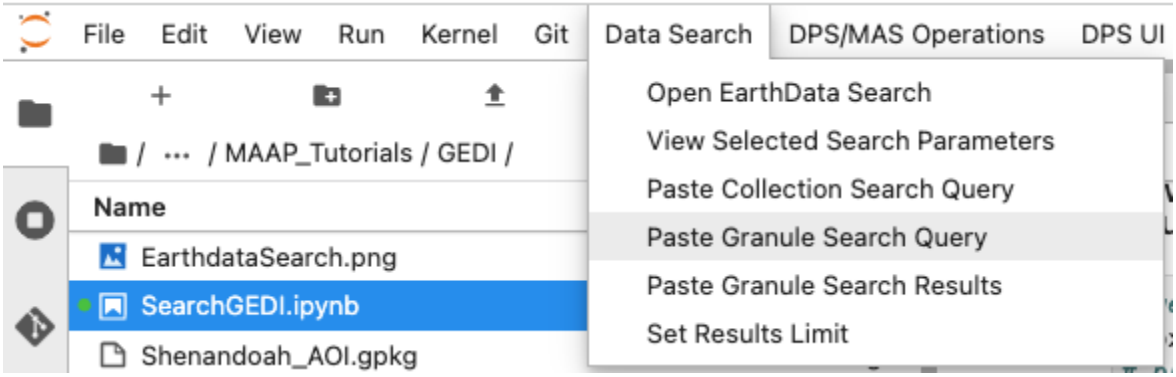
Search Data: To search GEDI data we can use the EarthData search integration in the MAAP ADE. Open the Earthdata search toolbar by clicking on the following:



This will open up the EarthData search interface in a new tab. We can use the search bar to search GEDI L2B data. By entering "L2B" in the search bar, we can see the relevant GEDI data is filtered. Click on the dataset to get more details.



While we have been searching for data, the MAAP ADE has been keeping a track of our search parameters. This means that we can easily insert our search for GEDI data straight into our notebook.



This gives us our search parameters in our notebook using the GEDI dataset id. The default limit for the number of returned results is 1000. Running this will produce 1000 results, but we can look at the first one by indexing the list of returned results. This is what the data entry looks like. You can see a large amount of metadata for the file along with the URL for where this specific file is stored.

[6]:
```
# generated from this EDSC search: https://ade.ops.maap-project.org:30052/search/
↪granules?p=C1201460047-NASA_MAAP&q=L2B&tl=1647961605!4!!
data = maap.searchGranule(concept_id="C1201460047-NASA_MAAP", limit=1000)[0]
data
```

[6]:
```
{'concept-id': 'G1201636833-NASA_MAAP',
 'collection-concept-id': 'C1201460047-NASA_MAAP',
 'revision-id': '11',
 'format': 'application/echo10+xml',
 'Granule': {'GranuleUR': 'SC:GEDI02_B.002:2433465280',
  'InsertTime': '2021-02-22T19:16:30.675Z',
  'LastUpdate': '2021-09-16T13:36:07.965Z',
  'Collection': {'ShortName': 'GEDI02_B', 'VersionId': '002'},
  'DataGranule': {'SizeMBDataGranule': '16.5413',
   'ProducerGranuleId': 'GEDI02_B_2019108002012_O01959_01_T03909_02_003_01_V002.h5',
   'DayNightFlag': 'UNSPECIFIED',
   'ProductionDateTime': '2021-02-21T14:45:08.000Z'},
  'Temporal': {'RangeDateTime': {'BeginningDateTime': '2019-04-18T00:20:12.000Z',
    'EndingDateTime': '2019-04-18T01:52:53.000Z'}},
  'Spatial': {'HorizontalSpatialDomain': {'Geometry': {'GPolygon': {'Boundary': {
↪'Point': [{'PointLongitude': '80.2890335089',
         'PointLatitude': '-4.6168623465'},
        {'PointLongitude': '82.4542052313', 'PointLatitude': '-1.5568021223'},
        {'PointLongitude': '83.6402279084', 'PointLatitude': '0.1277767863'},
        {'PointLongitude': '83.6814118126', 'PointLatitude': '0.0987901632'},
        {'PointLongitude': '82.4953794073', 'PointLatitude': '-1.5858244175'},
        {'PointLongitude': '80.3302671214',
         'PointLatitude': '-4.6459691487'}]}}}},
  'AdditionalAttributes': {'AdditionalAttribute': [{'Name': 'Dataset Status',
     'Values': {'Value': 'MAAP Standard Data Product'}},
    {'Name': 'Geolocated', 'Values': {'Value': 'true'}},
    {'Name': 'Spatial Resolution', 'Values': {'Value': '60'}},
    {'Name': 'Data Format', 'Values': {'Value': 'HDF5'}},
    {'Name': 'Band Center Frequency', 'Values': {'Value': '281759.8'}},
    {'Name': 'Swath Width', 'Values': {'Value': '4.2'}},
    {'Name': 'Field of View', 'Values': {'Value': '12'}},
    {'Name': 'Laser Footprint Diameter', 'Values': {'Value': '25'}},
    {'Name': 'Acquisition Type', 'Values': {'Value': 'Satellite Lidar'}},
    {'Name': 'Band Center Wavelength', 'Values': {'Value': '1064'}},
    {'Name': 'Reference_Ground_Track', 'Values': {'Value': '0'}},
```

(continues on next page)

(continued from previous page)

```
    {'Name': 'SEGMENT_NUMBER', 'Values': {'Value': '01'}}]},
  'OnlineAccessURLs': {'OnlineAccessURL': {'URL': 's3://nasa-maap-data-store/file-
↪staging/nasa-map/GEDI02_B___002/2019.04.18/GEDI02_B_2019108002012_O01959_01_T03909_
↪02_003_01_V002.h5',
    'URLDescription': 'File to download'}},
  'Orderable': 'true',
  'DataFormat': 'HDF5',
  'Visible': 'true'}}
```

So far, this search function requests all of the GEDI data but we can add a spatial subset filter using our AOI from above to limit the results. Adding a spatial filter returns 176 GEDI L2B files that intersect with our AOI.

[7]:
```
data_aoi = maap.searchGranule(concept_id="C1201460047-NASA_MAAP", bounding_box="-79.
↪0158578082679,37.88789051477522,-78.04618813890727,38.887036107037915", limit=1000)
print(len(data_aoi))
```

```
176
```

This is more data than we need, so let's look at the contents of a single GEDI file. Firstly we need to bring it from the server side (S3) to our local side. This can be done using the MAAP function getData. Here we will pull the 7th file in our search results.

[8]:
```
gedi_data = data_aoi[6].getData('./')
print(gedi_data)
```

```
.//GEDI02_B_2019145051352_O02537_03_T04809_02_003_01_V002.h5
```

GEDI data has 8 beams. So, we will check that all beams are in our file and print a list of the available beams.

[9]:
```
gedi_h5_file = h5py.File(gedi_data, 'r')
gedi_keys = list(gedi_h5_file.keys())
gedi_beams = ['BEAM0000', 'BEAM0001', 'BEAM0010', 'BEAM0011', 'BEAM0101', 'BEAM0110',
↪'BEAM1000', 'BEAM1011']
gedi_beams_lst = []
for gedi_beam_name in gedi_keys:
    if gedi_beam_name in gedi_beams:
        gedi_beams_lst.append(gedi_beam_name)
print(gedi_beams_lst)
```

```
['BEAM0000', 'BEAM0001', 'BEAM0010', 'BEAM0011', 'BEAM0101', 'BEAM0110', 'BEAM1000',
↪'BEAM1011']
```

For each beam, we need to get all of the data and metrics associated with it. For this, we have a function that will gather all of the metrics we want and put them into a geopandas dataframe:

[10]:
```
def get_gedi_df(gedi_h5_file, gedi_beam_name):
    gedi_beam = gedi_h5_file[gedi_beam_name]

    # Get location info.
    gedi_beam_geoloc = gedi_beam['geolocation']
    # Get land cover data.
    gedi_beam_landcover = gedi_beam['land_cover_data']

    gedi_beam_df = pandas.DataFrame(
            {'elevation_bin0'          : gedi_beam_geoloc['elevation_bin0'],
             'elevation_lastbin'       : gedi_beam_geoloc['elevation_lastbin'],
             'height_bin0'             : gedi_beam_geoloc['height_bin0'],
```

(continues on next page)

(continued from previous page)

```
            'height_lastbin'          : gedi_beam_geoloc['height_lastbin'],
            'shot_number'             : gedi_beam_geoloc['shot_number'],
            'solar_azimuth'           : gedi_beam_geoloc['solar_azimuth'],
            'solar_elevation'         : gedi_beam_geoloc['solar_elevation'],
            'latitude_bin0'           : gedi_beam_geoloc['latitude_bin0'],
            'latitude_lastbin'        : gedi_beam_geoloc['latitude_lastbin'],
            'longitude_bin0'          : gedi_beam_geoloc['longitude_bin0'],
            'longitude_lastbin'       : gedi_beam_geoloc['longitude_lastbin'],
            'degrade_flag'            : gedi_beam_geoloc['degrade_flag'],
            'digital_elevation_model': gedi_beam_geoloc['digital_elevation_model'],
            'landsat_treecover'       : gedi_beam_landcover['landsat_treecover'],
            'modis_nonvegetated'      : gedi_beam_landcover['modis_nonvegetated'],
            'modis_nonvegetated_sd'   : gedi_beam_landcover['modis_nonvegetated_sd'],
            'modis_treecover'         : gedi_beam_landcover['modis_treecover'],
            'modis_treecover_sd'      : gedi_beam_landcover['modis_treecover_sd'],
            'beam'                    : gedi_beam['beam'],
            'cover'                   : gedi_beam['cover'],
            'master_frac'             : gedi_beam['master_frac'],
            'master_int'              : gedi_beam['master_int'],
            'num_detectedmodes'       : gedi_beam['num_detectedmodes'],
            'omega'                   : gedi_beam['omega'],
            'pai'                     : gedi_beam['pai'],
            'pgap_theta'              : gedi_beam['pgap_theta'],
            'pgap_theta_error'        : gedi_beam['pgap_theta_error'],
            'rg'                      : gedi_beam['rg'],
            'rh100'                   : gedi_beam['rh100'],
            'rhog'                    : gedi_beam['rhog'],
            'rhog_error'              : gedi_beam['rhog_error'],
            'rhov'                    : gedi_beam['rhov'],
            'rhov_error'              : gedi_beam['rhov_error'],
            'rossg'                   : gedi_beam['rossg'],
            'rv'                      : gedi_beam['rv'],
            'sensitivity'             : gedi_beam['sensitivity'],
            'stale_return_flag'       : gedi_beam['stale_return_flag'],
            'surface_flag'            : gedi_beam['surface_flag'],
            'l2a_quality_flag'        : gedi_beam['l2a_quality_flag'],
            'l2b_quality_flag'        : gedi_beam['l2b_quality_flag']})

    gedi_beam_gdf = gpd.GeoDataFrame(gedi_beam_df, crs='EPSG:4326',
                                     geometry=gpd.points_from_xy(gedi_beam_df.
↪longitude_lastbin,
                                                                 gedi_
↪beam_df.latitude_lastbin))
    return gedi_beam_gdf
```

To access the data with this function, we can call the function for each beam id that we have:

[11]:
```
BEAM0000 = get_gedi_df(gedi_h5_file, 'BEAM0000')
BEAM0001 = get_gedi_df(gedi_h5_file, 'BEAM0001')
BEAM0010 = get_gedi_df(gedi_h5_file, 'BEAM0010')
BEAM0011 = get_gedi_df(gedi_h5_file, 'BEAM0011')
BEAM0101 = get_gedi_df(gedi_h5_file, 'BEAM0101')
BEAM0110 = get_gedi_df(gedi_h5_file, 'BEAM0110')
BEAM1000 = get_gedi_df(gedi_h5_file, 'BEAM1000')
BEAM1011 = get_gedi_df(gedi_h5_file, 'BEAM1011')
```

Now we can look at the data in one of the dataframes (beams). We can see that there are 108583 GEDI shots (108583

entries in each column).

[12]:
```
print(BEAM0000.head())
print('number of rows = ', len(BEAM0000))
```

```
   elevation_bin0  elevation_lastbin  height_bin0  height_lastbin  \
0      533.320205         532.870899 -9999.000000    -9999.000000
1      533.610735         533.161429 -9999.000000    -9999.000000
2      835.586133         790.954687    42.284134       -2.373195
3      822.361914         780.875651    38.280968       -3.229356
4      788.459662         760.602475    25.957445       -1.915897

         shot_number  solar_azimuth  solar_elevation  latitude_bin0  \
0  25370000300165164     -34.511044       -10.500773      51.821288
1  25370000300165165     -34.510292       -10.501078      51.821282
2  25370000300165166     -34.509689       -10.501341      51.821258
3  25370000300165167     -34.508926       -10.501648      51.821253
4  25370000300165168     -34.508163       -10.501955      51.821249

   latitude_lastbin  longitude_bin0  ...  rhov  rhov_error  rossg  \
0         51.821288     -127.096372  ...   0.6     -9999.0    0.5
1         51.821282     -127.095550  ...   0.6     -9999.0    0.5
2         51.821260     -127.094876  ...   0.6     -9999.0    0.5
3         51.821255     -127.094048  ...   0.6     -9999.0    0.5
4         51.821250     -127.093210  ...   0.6     -9999.0    0.5

            rv  sensitivity  stale_return_flag  surface_flag  \
0 -9999.000000   -14.502214                  1             1
1 -9999.000000    -8.689309                  1             1
2  3615.732178     0.965246                  0             1
3  4290.229980     0.969620                  0             1
4  2557.771240     0.943713                  0             1

   l2a_quality_flag  l2b_quality_flag                    geometry
0                 0                 0  POINT (-127.09637 51.82129)
1                 0                 0  POINT (-127.09555 51.82128)
2                 1                 1  POINT (-127.09485 51.82126)
3                 1                 1  POINT (-127.09403 51.82126)
4                 1                 1  POINT (-127.09320 51.82125)

[5 rows x 41 columns]
number of rows =  108583
```

Before displaying this data we can do a spatial subset to remove the data outside of our AOI by doing a spatial subset. We use the Geopandas clip function to clip out the GEDI data based on the extent of our AOI. This reduces the size of the dataframe from 108583 rows to 508.

[13]:
```
BEAM0000_sub = gpd.clip(BEAM0000, AOI)
len(BEAM0000_sub)
```

[13]:
```
508
```

We can display this subset of data over our AOI extent.

[14]:
```
m = folium.Map(location=[38.5, -78], zoom_start=9, tiles='CartoDB positron')

geo_j = folium.GeoJson(data=AOI, style_function=lambda x: {'fillColor': 'orange'})
geo_j.add_to(m)
```

(continues on next page)

(continued from previous page)

```
geo_g = folium.GeoJson(data=BEAM0000_sub,marker = folium.CircleMarker(radius = 1, #␣
↪Radius in metres
                                                    weight = 0, #outline weight
                                                    fill_color = '#FF0000',
                                                    fill_opacity = 1))
geo_g.add_to(m)
m
```

[14]:
```
<folium.folium.Map at 0x7f1beb7e5550>
```

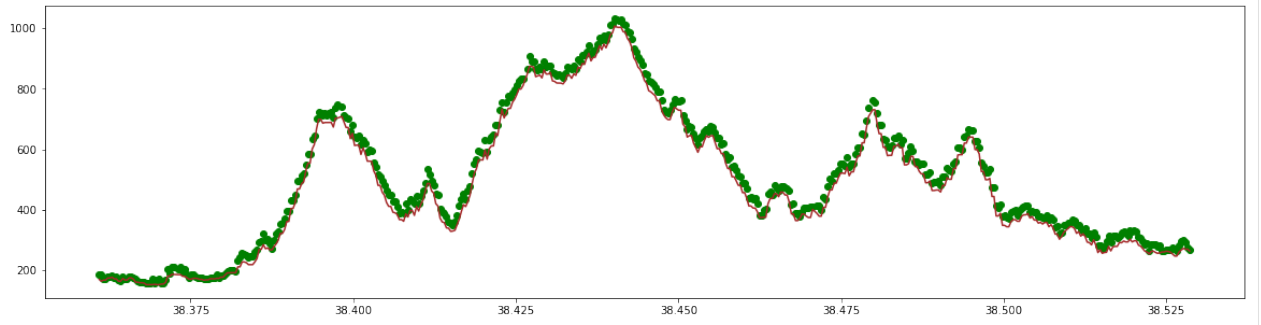
Now we have this subset of data, we can look at some of the GEDI metrics over our area of interest. The 'rh100' metric should give us the top of the canopy heights. What does this look like over Shenandoah National Park? We will look at the ground height and the 'rh100' above ground. The DEM metric is in meters (m) and the 'rh100' metric is in centimeters (cm) above the ground height. Therefore we must add the 'rh100' value to the ground height to get the total height of the tree for display purposes. The 'rh100' metric is also converted to meters (m) to normalize the units

[15]:
```
latitude = BEAM0000_sub['latitude_lastbin']
rh100 = BEAM0000_sub['rh100']
ground = BEAM0000_sub['digital_elevation_model']

TreeHeight = ground + rh100/100
```

Finally, we can make a plot of the ground surface and the canopy heights above the ground surface. We can see the forest canopies in green above the topographically complex ground in brown.

[16]:
```
plt.figure(figsize=(20, 5))
plt.scatter(latitude, TreeHeight, c='green')
plt.plot(latitude, ground, c='brown')
```

[16]:
```
[<matplotlib.lines.Line2D at 0x7f1becec0590>]
```



Now you have this basic structure you can investigate some of the other metrics and GEDI beams to understand more about the data.

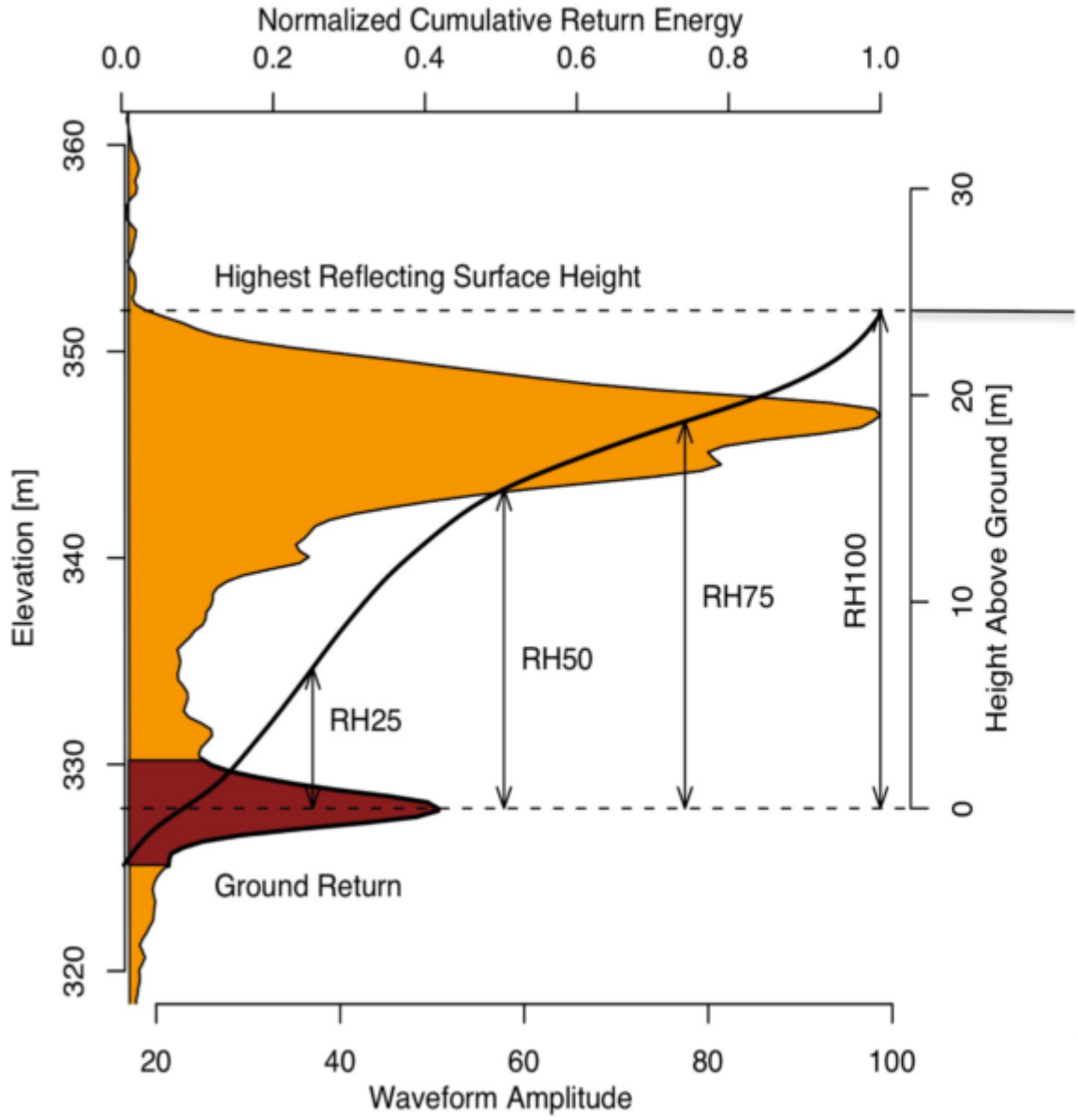
2.5 GEDI L3 Tutorial

2.5.1 GEDI Instrument

GEDI produces the first high resolution layer ranging observations of 3D structure of earth with precise measurements of forest canopy height, canopy vertical structure and surface elevation. GEDI is attached to the International Space Station (ISS) and collects data globally between 51.6° N and 51.6° S latitudes at the highest resolution and densest sampling of any light detection and ranging (lidar) instrument in orbit to date.

2.5.2 About GEDI L3 Data

GEDI L3 data products are gridded by spatially interpolating Level 2 footprint estimates of canopy cover, canopy height, Leaf Area Index (LAI), vertical foliage profile and their uncertainties. Level 2 data contains terrain elevation, canopy height, RH metrics and Leaf Area Index (LAI). The raw waveforms are collected by GEDI system and processed to provide canopy height and profile metrics. These metrics provide easy to use and interpret information about the vertical distribution of canopy material.



Source: https://gedi.umd.edu/data/products/

Figure: Sample of GEDI lidar waveform (Left). The light brown area under the curve represents the return energy from the canopy, while the dark brown area signifies the return from the underlying typography. The black line is cumulative return energey, starting from the bottom of the ground return (normalized to 0) to the top of canopy (normalized to 1). The diagram on the right shows the distribution of trees that produced the waveform.

2.5.3 [Optional] Install Python Packages

This tutorial assumes you've all packages installed. If you haven't already, uncomment the following lines to install these packages.

[1]:
```
# !pip install geopandas
# !pip install contextily
# !pip install backoff
# !pip install folium
# !pip install geojsoncontour
```

[5]:
```
from maap.maap import MAAP
import pandas as pd
import folium
from rasterio.plot import show
import rasterio
import boto3
import os
```

After importing necessary packages, the next step is to initialize MAAP constructor using `api.ops.maap-project.org` as `maap_host` argument.

[6]:
```
maap = MAAP(maap_host="api.ops.maap-project.org")
```

2.5.4 Dataset

GEDI L3 Gridded Land Surface Metrics, Version 2

This dataset provides Global Ecosystem Dynamics Investigation (GEDI) Level 3 (L3) gridded mean canopy height, standard deviation of canopy height, mean ground elevation, standard deviation of ground elevation, and counts of laser footprints per 1-km x 1-km grid cells globally within -52 and 52 degrees latitude.

Source: https://daac.ornl.gov/cgi-bin/dsviewer.pl?ds_id=1952

User Guide: https://daac.ornl.gov/GEDI/guides/GEDI_L3_LandSurface_Metrics_V2.html

Now, the next step is to seach granules from MAAP CMR. To generate following query, you can use EarthData search feature from MAAP ADE. Refer to this tutorial for more info.

[7]:
```
# generated from this EDSC search: https://ade.ops.maap-project.org:30052/search?p=!
↪C1201702030-NASA_MAAP&pg[1][v]=t&pg[1][qt]=2015-01-01T00%3A00%3A00.000Z%2C2020-12-
↪31T23%3A59%3A59.999Z&q=L3%20GEDI
results=maap.searchGranule(cmr_host='cmr.maap-project.org',concept_id="C1201702030-
↪NASA_MAAP", limit=1000)
```

The above query gives 1000 results by default. The number of necessary results can be changed using `limit` argument. We can view the `GranuleUR` from results using:

[8]:
```
[result['Granule']['GranuleUR'] for result in results]
```

[8]:
```
['GEDI_L3_LandSurface_Metrics_V2.GEDI03_elev_lowestmode_stddev_2019108_2020287_002_02.
↪tif',
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_counts_2019108_2021104_002_02.tif',
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_elev_lowestmode_stddev_2019108_2021104_002_02.
↪tif',
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_elev_lowestmode_mean_2019108_2020287_002_02.
↪tif',
```

(continues on next page)

(continued from previous page)

```
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_elev_lowestmode_mean_2019108_2021104_002_02.
↪tif',
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_rh100_stddev_2019108_2021104_002_02.tif',
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_counts_2019108_2020287_002_02.tif',
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_rh100_mean_2019108_2021104_002_02.tif',
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_rh100_mean_2019108_2020287_002_02.tif',
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_rh100_stddev_2019108_2020287_002_02.tif',
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_counts_2019108_2021216_002_02.tif',
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_rh100_mean_2019108_2021216_002_02.tif',
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_elev_lowestmode_stddev_2019108_2021216_002_02.
↪tif',
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_elev_lowestmode_mean_2019108_2021216_002_02.
↪tif',
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_rh100_stddev_2019108_2021216_002_02.tif']
```

Before downloading a particular tif, let's catch the collection and file name for first item in results:

[9]:
```
granule_ur=results[0]['Granule']['GranuleUR'].split(".")
collection_name=granule_ur[0]
file_name=granule_ur[1]
```

[10]:
```
print(f"collection name: {collection_name} | file_name: {file_name}")
```

```
collection name: GEDI_L3_LandSurface_Metrics_V2 | file_name: GEDI03_elev_lowestmode_
↪stddev_2019108_2020287_002_02
```

2.5.5 Download file from ORNL DAAC S3

To download the file from the source, temporary s3 credentials are required for `maap` package. You can explicitly request `s3_cred_endpoint` for the credentials. The code below wraps that request to get credentials and download the file to your workspace.

[14]:
```
def get_s3_creds(url):
    return maap.aws.earthdata_s3_credentials(url)

def get_s3_client(s3_cred_endpoint):
    creds=get_s3_creds(s3_cred_endpoint)
    boto3_session = boto3.Session(
            aws_access_key_id=creds['accessKeyId'],
            aws_secret_access_key=creds['secretAccessKey'],
            aws_session_token=creds['sessionToken']
    )
    return boto3_session.client("s3")

def download_s3_file(s3, bucket, collection_name, file_name):
    os.makedirs("/projects/gedi_l3", exist_ok=True) # create directories, as necessary
    download_path=f"/projects/gedi_l3/{file_name}.tif"
    s3.download_file(bucket, f"gedi/{collection_name}/data/{file_name}.tif", download_
↪path)
    return download_path
```

[15]:
```
s3_cred_endpoint= 'https://data.ornldaac.earthdata.nasa.gov/s3credentials'
s3=get_s3_client(s3_cred_endpoint)
```

[16]:
```
bucket="ornl-cumulus-prod-protected"
download_path=download_s3_file(s3, bucket, collection_name, file_name)
download_path
```

[16]:
```
'/projects/gedi_l3/GEDI03_elev_lowestmode_stddev_2019108_2020287_002_02.tif'
```

Now, we have the file in our local workspace. It's time to visualize it using `rasterio` package

2.5.6 [Optional] Visualization using Rasterio

The downloaded file is too big to read and visualize directly so we might need to scale it down and view it as a small thumbnail.

[2]:
```
def show_thumbnail(path):
    src=rasterio.open(path)
    oview = src.overviews(1)[0]
    thumbnail = src.read(1, out_shape=(1, int(src.height // oview), int(src.width //␣
↪oview)))
    show(thumbnail)
```

[ ]:
```
show_thumbnail(download_path)
```

2.5.7 [Optional] Overlay raster layer on top of folium map

To properly visualize the canopy heights, we need to display the TIF image on the map. The TIF image file may be too memory and compute-intensive for the kernel causing the process to exit.

[31]:
```
# tif=rasterio.open(download_path)
# arr=tif.read()
# bounds=tif.bounds
```

[32]:
```
# x1,y1,x2,y2=bounds
# bbox=[(bounds.bottom, bounds.left), (bounds.top, bounds.right)]
# m=folium.Map(location=[14.59, 120.98], zoom_start=10)
# img = folium.raster_layers.ImageOverlay(image=np.moveaxis(arr, 0, -1), bounds=bbox,␣
↪opacity=0.9, interactive=True, cross_origin=False, zindex=1)
# m
```

2.6 GEDI L4A Subsetting Example

2.6.1 [Optional] Install Python Packages

This notebook contains some cells marked as optional, meaning that you can use this notebook without necessarily running such cells.

However, if you do wish to run the optional cells, you must install the following Python packages, which might not already be installed in your environment:

- `geopandas`: for reading your AOI (GeoJson file), as well as for reading the job output (GeoPackage file containing the subset)
- `contextily`: for visually verifying your AOI

- `backoff`: for repeatedly polling the job status (after submission) until the job has been completed (either successfully or not)
- `folium`: for visualizing your data on a Leaflet map
- `geojsoncontour`: for converting your matplotlib contour plots to geojson

[1]:
```
# Uncomment the following lines to install these packages if you haven't already.
# !pip install geopandas
# !pip install contextily
# !pip install backoff
# !pip install folium
# !pip install geojsoncontour
```

A job can be submitted without these packages, but installing them in order to run the optional cells may make it more convenient for you to visually verify both your AOI and the subset output produced by your job.

2.6.2 Obtain Username

[2]:
```
from maap.maap import MAAP

maap = MAAP(maap_host="api.ops.maap-project.org")
username = maap.profile.account_info()["username"]
username
```

[2]:
```
'sayers'
```

2.6.3 Define the Area of Interest

You may use either a publicly available GeoJSON file for your AOI, such as those available at geoBoundaries, or you may create a custom GeoJSON file for your AOI. The following 2 subsections cover both cases.

Using a geoBoundary GeoJSON File

If your AOI is a publicly available geoBoundary, you can obtain the URL for the GeoJSON file using the function below. You simply need to supply an ISO3 value and a level. To find the appropriate ISO3 and level values, see the table on the geoBoundaries site.

[3]:
```
import requests


def get_geo_boundary_url(iso3: str, level: int) -> str:
    response = requests.get(
        f"https://www.geoboundaries.org/api/current/gbOpen/{iso3}/ADM{level}"
    )
    response.raise_for_status()
    return response.json()["gjDownloadURL"]


# If using a geoBoundary, uncomment the following assignment, supply
# appropriate values for `<iso3>` and `<level>`, then run this cell.

# Example (Gabon level 0): get_geo_boundary("GAB", 0)

# aoi = get_geo_boundary_url("<iso3>", <level>)
```

Using a Custom GeoJSON File

Alternatively, you can make your own GeoJSON file for your AOI and place it within your `my-public-bucket` folder within the ADE.

Based upon where you place your GeoJSON file under `my-public-bucket`, you can construct the URL for a job's `aoi` input value.

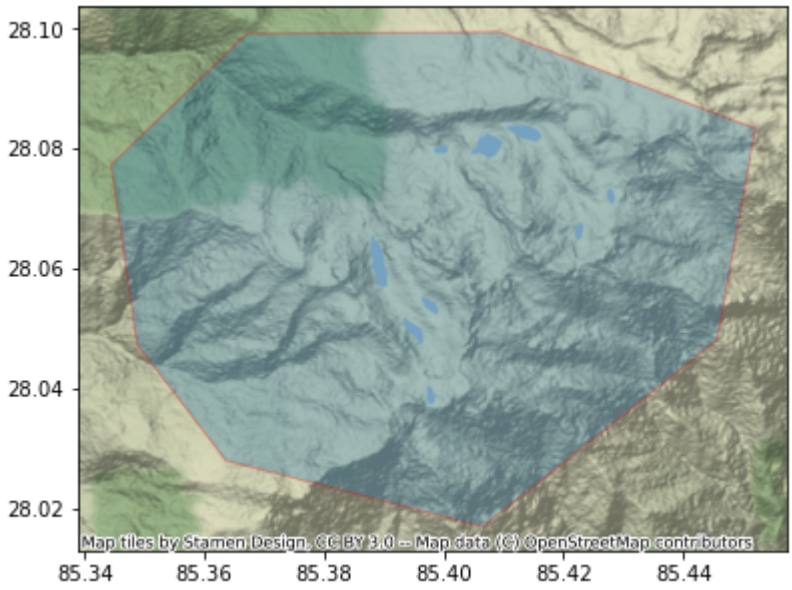
For example, if the relative path of your AOI GeoJSON file under `my-public-bucket` is `path/to/my-aoi.geojson` (avoid using whitespace in the path and filename), the URL you would supply as the value of a job's `aoi` input would be the following (where `{username}` is replaced with your username as output from the previous section):

```
f"https://maap-ops-workspace.s3.amazonaws.com/shared/{username}/path/to/my-aoi.geojson
↪"`
```

If this is the case, use the cell below.

[4]:
```
#aoi = f"https://maap-ops-workspace.s3.amazonaws.com/shared/{username}/langtang_np.
↪geojson"

#for your convenience you can use this geoJSON file but if you have your own geojson,␣
↪use the commented link as example format
aoi = f"https://maap-ops-workspace.s3.amazonaws.com/shared/anisbhsl/langtang_np.
↪geojson"
```

This example uses the AOI of Gosaikunda Lake region inside Langtang National Park. You can also create your own GeoJSON file for your AOI using sites like geojson.io

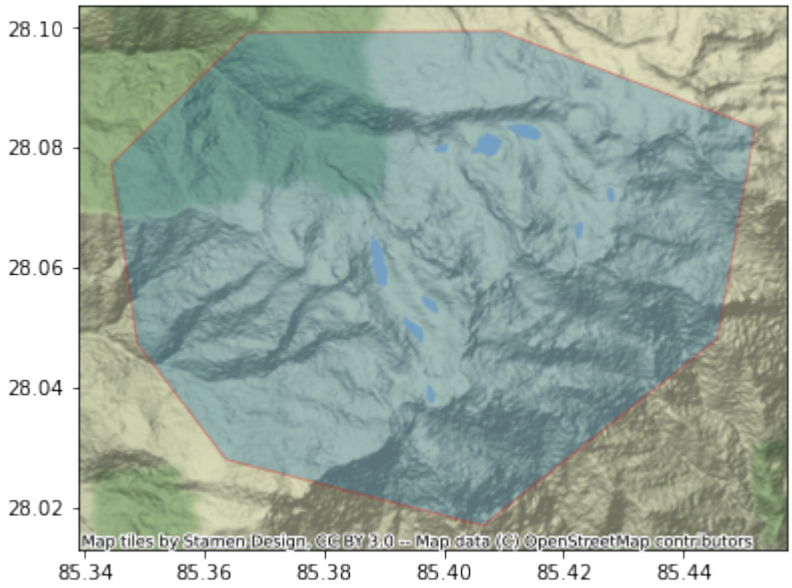


2.6.4 [Optional] Visually Verify your AOI

If you want to visually verify your AOI before proceeding, you may run the following cell, if you have the `geopandas` and `contextily` Python packages installed.

[5]:
```
try:
    import geopandas as gpd
    import contextily as ctx
except:
    print(
        "If you wish to visually verify your AOI, "
        "you must install the `geopandas` and `contextily` packages."
    )
else:
    aoi_gdf = gpd.read_file(aoi)
    aoi_epsg4326 = aoi_gdf.to_crs(epsg=4326)
    ax = aoi_epsg4326.plot(figsize=(10, 5), alpha=0.3, edgecolor="red")
    ctx.add_basemap(ax, crs=4326)
```

```
/opt/conda/lib/python3.7/site-packages/geopandas/_compat.py:115: UserWarning: The␣
↪Shapely GEOS version (3.11.1-CAPI-1.17.1) is incompatible with the GEOS version␣
↪PyGEOS was compiled with (3.8.1-CAPI-1.13.3). Conversions between both will be slow.
  shapely_geos_version, geos_capi_version_string
```



2.6.5 Submit a Job

When supplying input values for a GEDI subsetting job, to use the default value for a field (where indicated), use a dash (`"-"`) as the input value.

- `aoi` (required): URL to a GeoJSON file representing your area of interest, as explained above.
- `doi`: Digital Object Identifier (DOI) of the GEDI collection to subset, or a logical name representing such a DOI. Valid logical names: L1B, L2A, L2B, L4A
- `columns`: Comma-separated list of column names to include in the output file.
- `query`: Query expression for subsetting the rows in the output file.
- `limit`: Maximum number of GEDI granule data files to download (among those that intersect the specified AOI). (Default: 10000)

It is recommended to use `maap-dps-worker-32gb` queues when submitting a job with a large aoi.

[6]:
```
inputs = dict(
   aoi=aoi,
   doi="L4A",
   lat="lat_lowestmode",
   lon="lon_lowestmode",
   beams="all",
   columns="agbd, agbd_se,elev_lowestmode,sensitivity, geolocation/sensitivity_a2",
   query="l2_quality_flag == 1 and l4_quality_flag == 1 and sensitivity > 0.95 and␣
↪`geolocation/sensitivity_a2` > 0.95",
   limit = 10_000
)

result = maap.submitJob(
    identifier="gedi-subset",
    algo_id="gedi-subset_ubuntu",
    version="develop",
    queue="maap-dps-worker-32gb",
    username=username,
    **inputs,
)

job_id = result["job_id"]
job_id or result
```

[6]:
```
'ead0cec5-8ca5-4d0c-8517-963e0a618894'
```

2.6.6 Get the Job's Output File

Now that the job has been submitted, we can use the `job_id` to check the job status until the job has been completed.

[7]:
```
import xml.etree.ElementTree as ET
from urllib.parse import urlparse


def job_status_for(job_id: str) -> str:
    response = maap.getJobStatus(job_id)
    response.raise_for_status()

    root = ET.fromstring(response.text)
    status_element = root.find(".//{http://www.opengis.net/wps/2.0}Status")

    return status_element.text


def job_result_for(job_id: str) -> str:
    response = maap.getJobResult(job_id)
    response.raise_for_status()

    root = ET.fromstring(response.text)

    return root.find(".//{http://www.opengis.net/wps/2.0}Data").text


def to_job_output_dir(job_result_url: str) -> str:
    url_path = urlparse(job_result_url).path
    # The S3 Key is the URL path excluding the `/{username}` prefix
```

(continues on next page)

(continued from previous page)

```
    s3_key = "/".join(url_path.split("/")[2:])

    return f"/projects/my-private-bucket/{s3_key}"
```

If you have installed the `backoff` Python package, running the following cell will automatically repeatedly check your job's status until the job has been completed. Otherwise, you will have to manually repeatedly rerun the following cell until the output is either `'Succeeded'` or `'Failed'`.

[8]:
```
try:
    import backoff
except:
    job_status = job_status_for(job_id)
else:
    # Check job status every 2 minutes
    @backoff.on_predicate(
        backoff.constant,
        lambda status: status not in ["Deleted", "Succeeded", "Failed"],
        interval=120,
    )
    def wait_for_job(job_id: str) -> str:
        return job_status_for(job_id)

    job_status = wait_for_job(job_id)

job_status
```

```
INFO:backoff:Backing off wait_for_job(...) for 66.9s (Accepted)
INFO:backoff:Backing off wait_for_job(...) for 75.4s (Accepted)
INFO:backoff:Backing off wait_for_job(...) for 59.1s (Accepted)
INFO:backoff:Backing off wait_for_job(...) for 101.0s (Accepted)
INFO:backoff:Backing off wait_for_job(...) for 119.1s (Running)
INFO:backoff:Backing off wait_for_job(...) for 17.2s (Running)
INFO:backoff:Backing off wait_for_job(...) for 80.4s (Running)
```

[8]:
```
'Succeeded'
```

[9]:
```
assert job_status == "Succeeded", (
    job_result_for(job_id)
    if job_status == "Failed"
    else f"Job {job_id} has not yet completed ({job_status}). Rerun the prior cell."
)

output_url = job_result_for(job_id)
output_dir = to_job_output_dir(output_url)
output_file = f"{output_dir}/gedi_subset.gpkg"
print(f"Your subset results are in the file {output_file}")
```

```
Your subset results are in the file /projects/my-private-bucket/dps_output/gedi-
↪subset_ubuntu/develop/2022/12/16/20/37/29/132757/gedi_subset.gpkg
```

2.6.7 [Optional] Visually Verify the Results

If you installed the `geopandas` Python package, you can visually verify the output file by running the following cell.

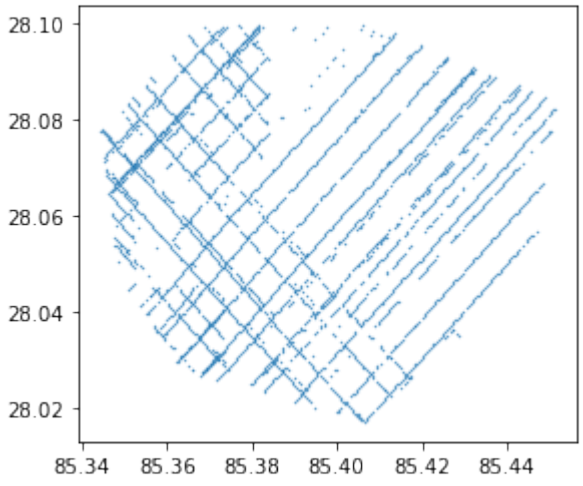
[10]:
```
try:
    import geopandas as gpd
```

(continues on next page)

(continued from previous page)

```
    import matplotlib.pyplot as plt
except:
    print(
        "If you wish to visually verify your output file, "
        "you must install the `geopandas` package."
    )
else:
    gedi_gdf = gpd.read_file(output_file)
    print(gedi_gdf.head())
    sensitivity_colors = plt.cm.get_cmap("viridis_r")
    gedi_gdf.plot(markersize = 0.1)
```

```
                                            filename  BEAM        agbd  \
0  GEDI04_A_2020347023307_O11328_02_T07169_02_002...  0000  116.018761
1  GEDI04_A_2020347023307_O11328_02_T07169_02_002...  0000  323.874420
2  GEDI04_A_2020347023307_O11328_02_T07169_02_002...  0000  295.168396
3  GEDI04_A_2020347023307_O11328_02_T07169_02_002...  0000  127.484352
4  GEDI04_A_2020347023307_O11328_02_T07169_02_002...  0000  345.088531

    agbd_se  elev_lowestmode  sensitivity  geolocation/sensitivity_a2  \
0  7.206294      2612.725342     0.963938                    0.963938
1  7.232505      2588.142090     0.965968                    0.965968
2  7.224230      2537.219727     0.969290                    0.969290
3  7.206897      2534.912842     0.980470                    0.980470
4  7.240329      2613.810059     0.979566                    0.979566

                    geometry
0  POINT (85.35394 28.04128)
1  POINT (85.35434 28.04165)
2  POINT (85.35513 28.04240)
3  POINT (85.35590 28.04316)
4  POINT (85.35628 28.04357)
```



2.6.8 Generate contour lines

Create a `lat`, `lon` mesh grid with elevation as a depth parameter. As shown in the plot above, the lines don't seem smooth. So we can apply `linear` or 'cubic' interpolation to smoothen those missing points.

[11]:
```
geometry = gedi_gdf["geometry"]
elevation=gedi_gdf["elev_lowestmode"]
```

[12]:
```
lon = geometry.x
lat = geometry.y
```

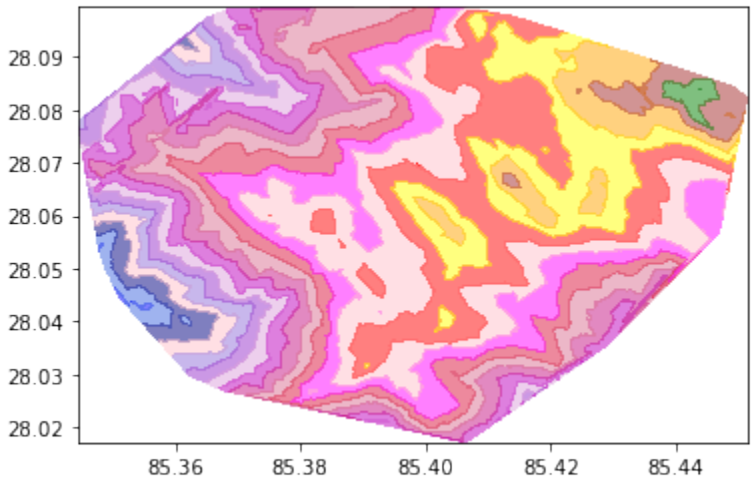
[13]:
```
import numpy as np

x=np.linspace(min(lon), max(lon), 1000)
y=np.linspace(min(lat), max(lat), 1000)
```

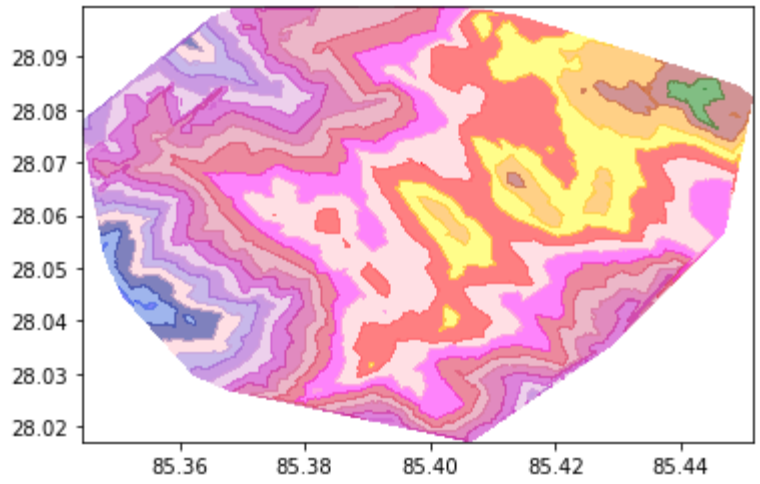
[14]:
```
from scipy.interpolate import griddata

x_mesh, y_mesh = np.meshgrid(x,y)
```

You may experiment with `nearest`, `linear`, and `cubic` interpolation methods to see which gives more smooth results.

[15]:
```
#grid the elevation
z_mesh = griddata((lon, lat), elevation, (x_mesh, y_mesh), method='linear')
```

[16]:
```
colors=['blue','royalblue', 'navy','pink',  'mediumpurple',  'darkorchid',  'plum',
↪'m', 'mediumvioletred', 'palevioletred', 'crimson',
        'magenta','pink','red','yellow','orange', 'brown','green', 'darkgreen']
levels=len(colors)
contourf = plt.contourf(x_mesh, y_mesh, z_mesh, levels, alpha=0.5,  colors=colors,␣
↪linestyles='None', vmin=elevation.min(), vmax=elevation.max())
```





Now we need to plot this contour into an interactive map for better visualization.

2.6.9 Plot the contour lines in folium

You may need to install `geojsoncontour`, `mapclassify`, and `folium`, if you don't already have them installed. We need to convert this `contourf` into geoJSON format.

[17]:
```
import folium
from folium import plugins
import branca
import geojsoncontour
```

[18]:
```
geojson = geojsoncontour.contourf_to_geojson(
    contourf=contourf,
    min_angle_deg=3.0,
    ndigits=5,
    stroke_width=1,
    unit='ft',
    fill_opacity=0.1,
)
```

[19]:
```
#create map view
m = folium.Map([lat.mean(), lon.mean()], zoom_start=12, tiles="OpenStreetMap")

folium.GeoJson(
    geojson,
    style_function=lambda x:{
        'color': x['properties']['stroke'],
        'weight':    x['properties']['stroke-width'],
        'fillColor': x['properties']['fill'],
        'opacity':   0.5,
    }
).add_to(m)

cm = branca.colormap.LinearColormap(colors, vmin=elevation.min(), vmax=elevation.
↪max()).to_step(levels)
cm.caption='Elevation (in m)'
m.add_child(cm)
```

(continues on next page)

(continued from previous page)

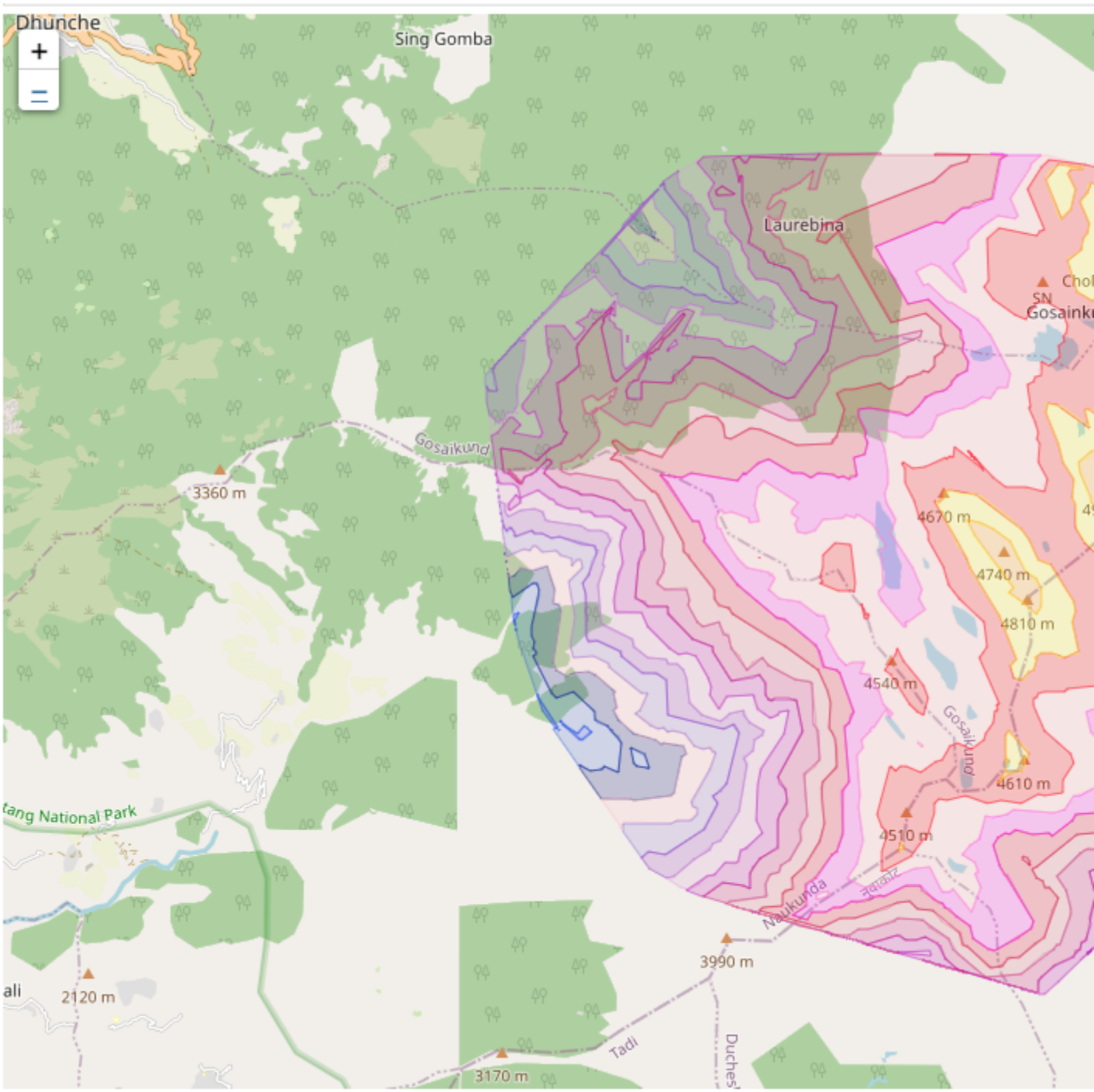
```
#legend
plugins.Fullscreen(position='topright', force_separate_button=True).add_to(m)
```

[19]:
```
<folium.plugins.fullscreen.Fullscreen at 0x7f245c76c710>
```

[20]:
```
m
```

[20]:
```
<folium.folium.Map at 0x7f245c3b24d0>
```

Now you have an interactive visualization of a contour plot.



2.7 Visualizing GEDI L4B Data

In this example, we demonstrate how to access GEDI L4B granule data on the MAAP ADE.

Within your Jupyter Notebook, start by importing the **maap package**. Then invoke the **MAAP** constructor, setting the maap_host argument to **'api.ops.maap-project.org'**.

[1]:
```
from maap.maap import MAAP
from matplotlib import pyplot
import os
import pprint
import rasterio

maap = MAAP(maap_host="api.ops.maap-project.org")
```

2.7.1 Search all collections to find the GEDI collection

Then search granules by the GEDI collection's collection ID.

[2]:
```
results = maap.searchCollection()
for collection in results:
    if collection['Collection']['ShortName'] == 'GEDI_L4B_Gridded_Biomass_2017':
        pprint.pprint(collection)
        break
```

```
{'Collection': {'AdditionalAttributes': {'AdditionalAttribute': [{'DataType': 'STRING
↪',
                                                                 'Description':
↪'MAAP '

↪'data '

↪'product '

↪'status ',
                                                                 'Name': 'Dataset '
                                                                         'Status',
                                                                 'Value': 'MAAP '
                                                                          'Standard '
                                                                          'Data '
                                                                          'Product'},
                                                                {'DataType': 'BOOLEAN
↪',
                                                                 'Description':
↪'Denotes '

↪'whether '
                                                                                 'the
↪'

↪'data '
                                                                                 'is '

↪'geolocated',
                                                                 'Name': 'Geolocated
↪',
                                                                 'Value': 'true'},
                                                                {'DataType': 'FLOAT',
                                                                 'Description':
↪'Ground '

↪'area '

↪'imaged '
```

(continues on next page)

(continued from previous page)

```
                                                                    'by '
                                                                    'a '

↪'sensoru2019s '

↪'instantaneous '

↪'field '
                                                                    'of '
                                                                    'view
↪',
                                                     'Name': 'Spatial '
                                                             'Resolution
↪',
                                                     'Value': '1'},
                                                    {'DataType': 'STRING
↪',
                                                     'Description':
↪'File '

↪'format '
                                                                    'of '
                                                                    'the
↪'

↪'associated '

↪'granules ',
                                                     'Name': 'Data '
                                                             'Format',
                                                     'Value': 'GeoTIFF'},
                                                    {'DataType': 'FLOAT',
                                                     'Description':
↪'Central '

↪'transmission '

↪'frequency '
                                                                    'of '
                                                                    'the
↪'

↪'instrument',
                                                     'Name': 'Band '
                                                             'Center '
                                                             'Frequency',
                                                     'Value': '281759.8'}
↪,
                                                    {'DataType': 'FLOAT',
                                                     'Description':
↪'Earth '

↪'surface '

↪'distance '

↪'observed '
```

(continues on next page)

(continued from previous page)

```
                                                                    'by '
                                                                    'the
↪'

↪'instrument',
                                                  'Name': 'Swath '
                                                          'Width',
                                                  'Value': '4.2'},
                                                 {'DataType': 'FLOAT',
                                                  'Description':
↪'Angular '

↪'extent '
                                                                    'a '

↪'sensor '

↪'views '

↪'across '
                                                                    'the
↪'

↪'Earthu2019s '

↪'surface',
                                                  'Name': 'Field '
                                                          'of '
                                                          'View',
                                                  'Value': '12'},
                                                 {'DataType': 'FLOAT',
                                                  'Description': 'The
↪'

↪'diameter '
                                                                    'of '
                                                                    'the
↪'

↪'laser '

↪'beam '

↪'that '

↪'reaches '
                                                                    'the
↪'

↪'ground',
                                                  'Name': 'Laser '
                                                          'Footprint '
                                                          'Diameter',
                                                  'Value': '25'},
                                                 {'DataType': 'STRING
↪',
                                                  'Description':
↪'Concise '
```

(continues on next page)

(continued from previous page)

```
↪'Platform-Instrument '
↪'data '
↪'collection '
↪'method '
                                                                    'tag
↪',
                                                          'Name':
↪'Acquisition '
                                                                  'Type',
                                                          'Value': 'Satellite
↪'
                                                                   'Lidar'},
                                                        {'DataType': 'FLOAT',
                                                         'Description':
↪'Central '
↪'transmission '
↪'wavelength '
                                                                       'of '
                                                                       'the
↪'
↪'instrument',
                                                          'Name': 'Band '
                                                                  'Center '
                                                                  'Wavelength
↪',
                                                          'Value': '1064'}]},
               'ArchiveCenter': 'ORNL_DAAC',
               'AssociatedBrowseImageUrls': {'ProviderBrowseUrl': {'Description':
↪'Gridded '
↪'mean '
↪'aboveground '
↪'biomass '
↪'density '
↪'(top) '
↪'and '
↪'standard '
↪'error '
↪'of '
↪'the '
↪'mean '
```

(continues on next page)

(continued from previous page)

```
↪'(bottom).',
                                                                    'URL': 'https://
↪daac.ornl.gov/GEDI/guides/GEDI_L4B_Gridded_Biomass_Fig1.png'}},
               'Campaigns': {'Campaign': {'LongName': 'Global Ecosystem '
                                                      'Dynamics Investigation',
                                          'ShortName': 'GEDI'}},
               'CollectionState': 'COMPLETE',
               'Contacts': {'Contact': {'OrganizationAddresses': {'Address': {'City':
↪ 'Oak '
                                                                                  ⌞
↪ 'Ridge',
                                                                                  ⌞
↪'Country': 'United '
                                                                                  ⌞
↪    'States',
                                                                                  ⌞
↪'PostalCode': '37831-6407',
                                                                                  ⌞
↪'StateProvince': 'Tennessee',
                                                                                  ⌞
↪'StreetAddress': 'ORNL '
                                                                                  ⌞
↪          'DAAC '
                                                                                  ⌞
↪          'User '
                                                                                  ⌞
↪          'Services '
                                                                                  ⌞
↪          'Office, '
                                                                                  ⌞
↪          'P.O. '
                                                                                  ⌞
↪          'Box '
                                                                                  ⌞
↪          '2008, '
                                                                                  ⌞
↪          'MS '
                                                                                  ⌞
↪          '6407, '
                                                                                  ⌞
↪          'Oak '
                                                                                  ⌞
↪          'Ridge '
                                                                                  ⌞
↪          'National '
                                                                                  ⌞
↪          'Laboratory'}},
                                         'OrganizationEmails': {'Email': 'uso@daac.
↪ornl.gov'},
                                         'OrganizationName': 'ORNL_DAAC',
                                         'OrganizationPhones': {'Phone': {'Number':
↪'(865) '
                                                                                  ⌞
↪'241-3952',
                                                                    'Type':
↪'Direct '
```

(continues on next page)

(continued from previous page)

```
↪'Line'}},
                                      'Role': 'ARCHIVER'}},
               'DOI': {'Authority': 'https://doi.org',
                       'DOI': '10.3334/ORNLDAAC/2017'},
               'DataFormat': 'GeoTIFF',
               'DataSetId': 'GEDI L4B Gridded Aboveground Biomass Density, '
                            'Version 2',
               'Description': 'This Global Ecosystem Dynamics Investigation '
                              '(GEDI) L4B product provides 1 km x 1 km (1 '
                              'km,  hereafter) estimates of mean aboveground '
                              'biomass density (AGBD) based on observations '
                              'from mission week 19 starting on 2019-04-18 to '
                              'mission week 138 ending on 2021-08-04. The '
                              'GEDI L4A Footprint Biomass product converts '
                              'each high-quality waveform to an AGBD '
                              'prediction, and the L4B product uses the '
                              'sample present within the borders of each 1 km '
                              'cell to statistically infer mean AGBD. The '
                              'gridding procedure is described in the GEDI '
                              'L4B Algorithm Theoretical Basis Document '
                              '(ATBD). Patterson et al. (2019) describes the '
                              'hybrid model-based mode of inference used in '
                              'the L4B product. Corresponding 1 km estimates '
                              'of the standard error of the mean are also '
                              'provided in the L4B product. Uncertainty is '
                              "due to both GEDI's sampling of the 1 km area "
                              '(as opposed to making wall-to-wall '
                              'observations) and the fact that L4A biomass '
                              'values are modeled in a process subject to '
                              'error instead of measured in a process that '
                              'may be assumed to be error-free.',
               'InsertTime': '2022-03-29T00:00:00.000Z',
               'LastUpdate': '2022-03-30T10:43:50.000Z',
               'LongName': 'Not provided',
               'OnlineAccessURLs': {'OnlineAccessURL': {'URL': 'https://daac.ornl.
↪gov/daacdata/gedi/GEDI_L4B_Gridded_Biomass/',
                                                       'URLDescription': 'This '
                                                                         'link '
                                                                         'allows '
                                                                         'direct '
                                                                         'data '
                                                                         'access '
                                                                         'via '
                                                                         'Earthdata
↪'
                                                                         'login'}},
               'OnlineResources': {'OnlineResource': [{'Description': 'ORNL '
                                                                      'DAAC '
                                                                      'Data '
                                                                      'Set '
                                                                      'Documentation
↪',
                                                       'Type': 'VIEW RELATED '
                                                               'INFORMATION : '
                                                               'GENERAL '
                                                               'DOCUMENTATION',
```

(continues on next page)

(continued from previous page)

```
                                                 'URL': 'https://daac.ornl.gov/
↪GEDI/guides/GEDI_L4B_Gridded_Biomass.html'},
                                                {'Description': 'Data '
                                                                'set '
                                                                'Landing '
                                                                'Page '
                                                                'DOI '
                                                                'URL',
                                                 'Type': 'CollectionURL '
                                                         ': DATA SET '
                                                         'LANDING PAGE',
                                                 'URL': 'https://doi.org/10.
↪3334/ORNLDAAC/2017'},
                                                {'Description': 'GEDI '
                                                                'L4B '
                                                                'Gridded '
                                                                'Aboveground '
                                                                'Biomass '
                                                                'Density, '
                                                                'Version '
                                                                '2: '
                                                                'GEDI_L4B_ATBD_
↪v1.0.pdf',
                                                 'Type': 'VIEW RELATED '
                                                         'INFORMATION : '
                                                         'GENERAL '
                                                         'DOCUMENTATION',
                                                 'URL': 'https://daac.ornl.gov/
↪daacdata/gedi/GEDI_L4B_Gridded_Biomass/comp/GEDI_L4B_ATBD_v1.0.pdf'},
                                                {'Description': 'GEDI '
                                                                'L4B '
                                                                'Gridded '
                                                                'Aboveground '
                                                                'Biomass '
                                                                'Density, '
                                                                'Version '
                                                                '2: '
                                                                'GEDI_L4B_
↪Gridded_Biomass.pdf',
                                                 'Type': 'VIEW RELATED '
                                                         'INFORMATION : '
                                                         'GENERAL '
                                                         'DOCUMENTATION',
                                                 'URL': 'https://daac.ornl.gov/
↪daacdata/gedi/GEDI_L4B_Gridded_Biomass/comp/GEDI_L4B_Gridded_Biomass.pdf'},
                                                {'Description': 'GEDI '
                                                                'Project '
                                                                'Site',
                                                 'Type': 'CollectionURL '
                                                         ': PROJECT '
                                                         'HOME PAGE',
                                                 'URL': 'https://gedi.umd.edu'}
↪]},
               'Platforms': {'Platform': {'Instruments': {'Instrument': {'LongName':
↪'Global '

↪'Ecosystem '
```

(continues on next page)

(continued from previous page)

```
↪'Dynamics '

↪'Investigation',
                                                                    'ShortName':
↪ 'GEDI'}},
                                        'LongName': 'International Space '
                                                    'Station',
                                        'ShortName': 'ISS',
                                        'Type': 'Space Stations/Crewed '
                                                'Spacecraft'}},
               'Price': '     0.00',
               'ProcessingLevelDescription': 'model products',
               'ProcessingLevelId': '4',
               'RevisionDate': '2022-04-20T17:07:00.000Z',
               'ScienceKeywords': {'ScienceKeyword': [{'CategoryKeyword': 'EARTH '
                                                                          'SCIENCE',
                                                      'TermKeyword': 'ECOSYSTEMS',
                                                      'TopicKeyword': 'BIOSPHERE',
                                                      'VariableLevel1Keyword': {
↪'Value': 'TERRESTRIAL '
                                                                               ⌊
↪     'ECOSYSTEMS'}},
                                                     {'CategoryKeyword': 'EARTH '
                                                                         'SCIENCE',
                                                      'TermKeyword': 'VEGETATION',
                                                      'TopicKeyword': 'BIOSPHERE',
                                                      'VariableLevel1Keyword': {
↪'Value': 'BIOMASS'}},
                                                     {'CategoryKeyword': 'EARTH '
                                                                         'SCIENCE',
                                                      'TermKeyword': 'LIDAR',
                                                      'TopicKeyword': 'SPECTRAL/
↪ENGINEERING',
                                                      'VariableLevel1Keyword': {
↪'Value': 'LIDAR '
                                                                               ⌊
↪     'WAVEFORM'}}]},
               'ShortName': 'GEDI_L4B_Gridded_Biomass_2017',
               'Spatial': {'GranuleSpatialRepresentation': 'CARTESIAN',
                           'HorizontalSpatialDomain': {'Geometry': {
↪'BoundingRectangle': {'EastBoundingCoordinate': '180.0',
                                                                               ⌊ ⌊
↪     'NorthBoundingCoordinate': '52.0',
                                                                               ⌊ ⌊ ⌊
↪     'SouthBoundingCoordinate': '-52.0',
                                                                               ⌊ ⌊
↪     'WestBoundingCoordinate': '-180.0'},
                                                                  'CoordinateSystem
↪': 'CARTESIAN'}},
                           'SpatialCoverageType': 'HORIZONTAL'},
               'SpatialInfo': {'SpatialCoverageType': 'HORIZONTAL'},
               'SpatialKeywords': {'Keyword': 'GLOBAL LAND'},
               'Temporal': {'EndsAtPresentFlag': 'false',
                            'RangeDateTime': {'BeginningDateTime': '2019-04-18T00:00:
↪00.000Z',
                                              'EndingDateTime': '2021-08-04T23:59:59.
↪000Z'}},
```

(continues on next page)

(continued from previous page)

```
                'UseConstraints': {'LicenseURL': {'Description': 'License URL '
                                                                 'for data use '
                                                                 'policy',
                                                  'MimeType': 'text/html',
                                                  'Type': 'Data Use Policy',
                                                  'URL': 'https://science.nasa.gov/
↪earth-science/earth-science-data/data-information-policy'}},
                'VersionId': '2'},
 'concept-id': 'C1202077968-NASA_MAAP',
 'format': 'application/echo10+xml',
 'revision-id': '2'}
```

Search for granules using searchGranule function with the GEDI collection ID.

[3]:
```
COLLECTIONID = collection['concept-id']
results = maap.searchGranule(concept_id=COLLECTIONID,limit=50)  # COLLECTIONID
↪'C1202077968-NASA_MAAP'
print(f'Got {len(results)} results')
results[0]['Granule']
```

```
Got 10 results
```

[3]:
```
{'GranuleUR': 'GEDI_L4B_Gridded_Biomass.GEDI04_B_MW019MW138_02_002_05_R01000M_PS.tif',
 'InsertTime': '2022-03-29T00:00:00.000Z',
 'LastUpdate': '2022-03-30T10:43:51.000Z',
 'Collection': {'ShortName': 'GEDI_L4B_Gridded_Biomass_2017',
  'VersionId': '2'},
 'DataGranule': {'SizeMBDataGranule': '20.103343',
  'Checksum': {'Value':
↪'025a141348906d5e612262218c496a2d468446ca30875439be6651d851bfbe23',
   'Algorithm': 'SHA-256'},
  'DayNightFlag': 'BOTH',
  'ProductionDateTime': '2022-03-29T00:00:00.000Z'},
 'Temporal': {'RangeDateTime': {'BeginningDateTime': '2019-04-18T00:00:00.000Z',
   'EndingDateTime': '2021-08-04T23:59:59.000Z'}},
 'Spatial': {'HorizontalSpatialDomain': {'Geometry': {'BoundingRectangle': {
↪'WestBoundingCoordinate': '-180',
     'NorthBoundingCoordinate': '52',
     'EastBoundingCoordinate': '180',
     'SouthBoundingCoordinate': '-52'}}}},
 'MeasuredParameters': {'MeasuredParameter': [{'ParameterName': 'LIDAR WAVEFORM'},
   {'ParameterName': 'BIOMASS'},
   {'ParameterName': 'TERRESTRIAL ECOSYSTEMS'}]},
 'Platforms': {'Platform': {'ShortName': 'ISS',
   'Instruments': {'Instrument': {'ShortName': 'GEDI'}}}},
 'Campaigns': {'Campaign': {'ShortName': 'GEDI'}},
 'AdditionalAttributes': {'AdditionalAttribute': [{'Name': 'Dataset Status',
    'Values': {'Value': 'MAAP Standard Data Product'}},
   {'Name': 'Geolocated', 'Values': {'Value': 'true'}},
   {'Name': 'Spatial Resolution', 'Values': {'Value': '1'}},
   {'Name': 'Data Format', 'Values': {'Value': 'GeoTIFF'}},
   {'Name': 'Band Center Frequency', 'Values': {'Value': '281759.8'}},
   {'Name': 'Swath Width', 'Values': {'Value': '4.2'}},
   {'Name': 'Field of View', 'Values': {'Value': '12'}},
   {'Name': 'Laser Footprint Diameter', 'Values': {'Value': '25'}},
   {'Name': 'Acquisition Type', 'Values': {'Value': 'Satellite Lidar'}},
```

(continues on next page)

(continued from previous page)

```
   {'Name': 'Band Center Wavelength', 'Values': {'Value': '1094'}}]},
 'OnlineAccessURLs': {'OnlineAccessURL': [{'URL': 's3://ornl-cumulus-prod-protected/
↪gedi/GEDI_L4B_Gridded_Biomass/data/GEDI04_B_MW019MW138_02_002_05_R01000M_PS.tif',
    'URLDescription': 'File to download'},
   {'URL': 'https://data.ornldaac.earthdata.nasa.gov/protected/gedi/GEDI_L4B_Gridded_
↪Biomass/data/GEDI04_B_MW019MW138_02_002_05_R01000M_PS.tif',
    'URLDescription': 'Download GEDI04_B_MW019MW138_02_002_05_R01000M_PS.tif'},
   {'URL': 'https://data.ornldaac.earthdata.nasa.gov/public/gedi/GEDI_L4B_Gridded_
↪Biomass/data/GEDI04_B_MW019MW138_02_002_05_R01000M_PS.tif.sha256',
    'URLDescription': 'Download GEDI04_B_MW019MW138_02_002_05_R01000M_PS.tif.sha256'}
↪]},
 'OnlineResources': {'OnlineResource': [{'URL': 'https://daac.ornl.gov/GEDI/guides/
↪GEDI_L4B_Gridded_Biomass.html',
    'Description': 'ORNL DAAC Data Set Documentation',
    'Type': 'VIEW RELATED INFORMATION'},
   {'URL': 'https://doi.org/10.3334/ORNLDAAC/2017',
    'Description': 'Data set Landing Page DOI URL',
    'Type': 'VIEW RELATED INFORMATION'},
   {'URL': 'https:/daac.ornl.gov/daacdata/gedi/GEDI_L4B_Gridded_Biomass/comp/GEDI_L4B_
↪ATBD_v1.0.pdf',
    'Description': 'Data Set Documentation',
    'Type': 'VIEW RELATED INFORMATION'},
   {'URL': 'https:/daac.ornl.gov/daacdata/gedi/GEDI_L4B_Gridded_Biomass/comp/GEDI_L4B_
↪Gridded_Biomass.pdf',
    'Description': 'Data Set Documentation',
    'Type': 'VIEW RELATED INFORMATION'},
   {'URL': 'https://data.ornldaac.earthdata.nasa.gov/s3credentials',
    'Description': 'api endpoint to retrieve temporary credentials valid for same-
↪region direct s3 access',
    'Type': 'VIEW RELATED INFORMATION'}]},
 'Orderable': 'true',
 'DataFormat': 'GeoTIFF',
 'Visible': 'true'}
```

2.7.2 Utilize 'getData' function to download one granule's data file

[4]:
```
download_path = './download/gedi_l4b'
os.makedirs(download_path, exist_ok=True)
download = results[0].getData(download_path)
# Print the downloaded file
download
```

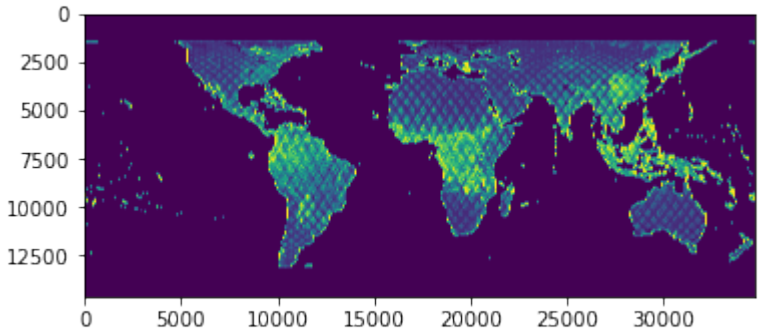
[4]:
```
'./download/gedi_l4b/GEDI04_B_MW019MW138_02_002_05_R01000M_PS.tif'
```

[5]:
```
src = rasterio.open(download)
data = src.read(1)

# Print the shape of data to verify if rasterio was able to read the file properly
print(data.shape)
```

```
(14616, 34704)
```

2.7.3 Plot the .tif file using 'matplotlib'

[6]:
```
pyplot.imshow(data)
pyplot.show()
```



2.8 ICESat-02 Advanced Topographic Laser Altimeter System-03 (ATL03) data access and exploration instructions.

About ATL03 Data: (ATL03) contains height above the WGS 84 ellipsoid (ITRF2014 reference frame), latitude, longitude, and time for all photons downlinked by the Advanced Topographic Laser Altimeter System (ATLAS) instrument on board the Ice, Cloud and land Elevation Satellite-2 (ICESat-2) observatory. The ATL03 product was designed to be a single source for all photon data and ancillary information needed by higher-level ATLAS/ICESat-2 products. As such, it also includes spacecraft and instrument parameters and ancillary data not explicitly required for ATL03. (source: https://nsidc.org/data/atl03/versions/5)

Required packages:

You will need to install the following required packages if not already present in your working environment: maap-py, pandas, geopandas, folium, shapely, h5glance, h5py

2.8.1 Import relevant python modules

[1]:
```
# Import the MAAP package
from maap.maap import MAAP

# Invoke the MAAP constructor using the maap_host argument
maap = MAAP(maap_host='api.ops.maap-project.org')

# Import pandas dataframe
import pandas as pd

# Import libraries needed for visualizing data spatial extent
import geopandas as gpd
import folium
from shapely.geometry import Polygon,Point
```

(continues on next page)

(continued from previous page)

```
# Import H5glance to interactively explore H5 file in notebook
from h5glance import H5Glance

# Import H5py to read h5 file
import h5py
```

```
/opt/conda/lib/python3.7/site-packages/geopandas/_compat.py:115: UserWarning: The␣
↪Shapely GEOS version (3.11.1-CAPI-1.17.1) is incompatible with the GEOS version␣
↪PyGEOS was compiled with (3.8.1-CAPI-1.13.3). Conversions between both will be slow.
  shapely_geos_version, geos_capi_version_string
```

2.8.2 Decide on a subset of ATL03 data to load using spatial extent and date range and visualize extent using folium

[2]:
```
# Create a variable for short name of ATL03 data
short_name = 'ATL03'

# Create spatial extent for which we will search
# ATL03 granules on NASA CMR. We have chosen a very narrow
# area over Yosemite national park. This is okay,
# as we need just a few granules for demo purposes.

# Create Latitude, Longitude list.
lat_coords = [37.700057,37.700057,37.758166,37.758166,37.700057]
lon_coords = [-119.680359,-119.680359,-119.538910,-119.538910,-119.680359]

# Create bounding box
spatial_extent = [lon_coords[0],lat_coords[0],lon_coords[2],lat_coords[2]]

# Reformat bounding box to work with NASA CMR API
spatial_extent = ','.join(str(coords) for coords in spatial_extent)

#Provide date range. It is just 1 day.
date_range = ['2021-02-02','2022-02-03']

# For folium purpose, provide the map center
map_center = [37.729139,-119.609670]

# Convert to AOI for visualizaton with folium
polygon_geom = Polygon(zip(lon_coords, lat_coords))

# Provide relevant Coordinate Reference System
crs = 'epsg:4326'

# Convert to Geodataframe and back to list in that specific reference system
AOI = gpd.GeoDataFrame(index=[0], crs=crs, geometry=[polygon_geom])
AOI_bbox = AOI.bounds.iloc[0].to_list()
```

[3]:
```
# Visualize the spatial extent using folium.
m = folium.Map(map_center, zoom_start=12, tiles='OpenStreetMap')
folium.GeoJson(AOI).add_to(m)
folium.LatLngPopup().add_to(m)
m
```

[3]:
```
<folium.folium.Map at 0x7ff1580c11d0>
```

2.8.3 Search for available granules in the NASA CMR for the given spatial extent and date range and print the number of granules available.

[4]:
```
# Provide the name of cmr host to use with maap py
nasa_cmr_host = "cmr.earthdata.nasa.gov"
# Search granule using maap-py using subset criteria identified earlier.
data = maap.searchGranule(cmr_host=nasa_cmr_host,short_name = short_name, bounding_
↪box = spatial_extent,temporal= date_range,limit=1000)
# Check to see if the granule search was successfull in finding data within spatial␣
↪and temporal extent
print(len(data))
```

```
68
```

[5]:
```
#Use maap-py's getDatafunction to extract one of the HDF files from NSIDC servers.
# Print the name of file extracted. The file will be downloaded and stored in your␣
↪current directory

ice_data = data[0].getData()
print(ice_data)
```

```
./ATL03_20210202191800_06231006_005_01.h5
```

2.8.4 Read the H5 file to understand the data structure.

There are two ways to do this. We can just list keys and then go forward exploring each key one by one.

[6]:
```
# Open the H5 file and list the keys
ice_file = h5py.File(ice_data,'r')
list(ice_file.keys())
```

[6]:
```
['METADATA',
 'ancillary_data',
 'atlas_impulse_response',
 'ds_surf_type',
 'ds_xyz',
 'gt1l',
 'gt1r',
 'gt2l',
 'gt2r',
 'gt3l',
 'gt3r',
 'orbit_info',
 'quality_assessment']
```

Or use the h5glance package to interactively list various keys, sub-keys and variables.

H5Glance lists all the keys and sub-keys and allows for the copying of path variables on the fly, all from within the Jupyter Notebook. Note: In the web version of this notebook which is shown here, not all sub-fields will be listed. But they can be accessed when using a Jupyter Notebook running on ADE or local machines.

[7]:
```
H5Glance(ice_file)
```

[7]:
```
./ATL03_20210202191800_06231006_005_01.h5/ (47 attributes)
├METADATA    (9 children) (3 attributes)
├ancillary_data     (34 children) (2 attributes)
├atlas_impulse_response    (2 children) (1 attributes)
├ds_surf_type      [int32: 5] (12 attributes)
├ds_xyz     [int32: 3] (12 attributes)
├gt1l        (5 children) (7 attributes)
├gt1r        (5 children) (7 attributes)
├gt2l        (5 children) (7 attributes)
├gt2r        (5 children) (7 attributes)
├gt3l        (5 children) (7 attributes)
├gt3r        (5 children) (7 attributes)
├orbit_info         (7 children) (2 attributes)
└quality_assessment         (9 children) (1 attributes)
```

2.8.5 Subset the data by required columns. In this case we need Latitude, Longitude, Photon Height and Along Track Distance.

We are using the copied path from the data tree generated by H5Glance above. Use h5py to read file and variables from the copied path.

[8]:
```
with h5py.File(ice_data,'r') as f:
    gt1l_lat = f['/gt1l/heights/lat_ph'][:]
    gt1l_lon = f['/gt1l/heights/lon_ph'][:]
    gt1l_height = f['/gt1l/heights/h_ph'][:]
    gt1l_dist_ph = f['/gt1l/heights/dist_ph_along'][:]
```

2.8.6 Show the subset data in a dataframe.

1. By using Pandas module.

[9]:
```
# Write latitude, longitude, photon height and along track distance for gt1l to a␣
↪dataframe
gt1l_df = pd.DataFrame({'Latitude': gt1l_lat, 'Longitude': gt1l_lon, 'Photon_Height':␣
↪gt1l_height,'Along_track_distance':gt1l_dist_ph})
gt1l_df
```

[9]:

	Latitude	Longitude	Photon_Height	Along_track_distance
0	59.482065	-115.906874	611.887878	16.894447
1	59.482065	-115.906877	580.084106	16.980614
2	59.482064	-115.906882	527.962097	17.122099
3	59.482063	-115.906885	491.036133	17.222527
4	59.482059	-115.906875	616.919922	17.593958
...	...	...	...	...
36561868	33.011390	-119.752303	69.065865	4.154013
36561869	33.011382	-119.752316	-16.501354	5.156162
36561870	33.011380	-119.752329	-108.562981	5.470425
36561871	33.011376	-119.752317	-17.947205	5.871534
36561872	33.011375	-119.752319	-37.489002	5.938596

[36561873 rows x 4 columns]

2. By using Geopandas module.

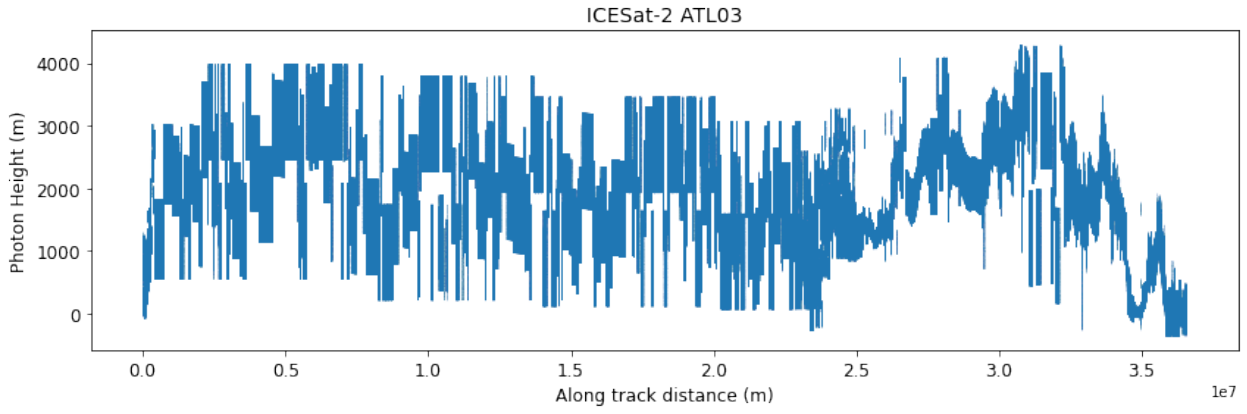
Create geopandas dataframe with a column for point locations in the geometry column, and other relevant variables.

[10]:
```
geometry = gpd.points_from_xy(gt1l_lon, gt1l_lat)
data = {'Latitude': gt1l_lat, 'Longitude': gt1l_lon, 'Photon_Height': gt1l_height,
↪'Along_track_distance':gt1l_dist_ph}
gdf = gpd.GeoDataFrame(data,geometry=geometry, crs='EPSG:4326')

# View the resulting geopandas dataframe
print(gdf.head())
```

```
    Latitude   Longitude  Photon_Height  Along_track_distance  \
0  59.482065 -115.906874     611.887878             16.894447
1  59.482065 -115.906877     580.084106             16.980614
2  59.482064 -115.906882     527.962097             17.122099
3  59.482063 -115.906885     491.036133             17.222527
4  59.482059 -115.906875     616.919922             17.593958

                      geometry
0  POINT (-115.90687 59.48207)
1  POINT (-115.90688 59.48206)
2  POINT (-115.90688 59.48206)
3  POINT (-115.90689 59.48206)
4  POINT (-115.90687 59.48206)
```

2.8.7 Visualize photon heights with respect to along track distance for this H5 file (using inputs from geodataframe)

[22]:
```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(figsize=(14, 4))
gdf.Photon_Height.plot(ax=ax, ls='', marker='.', ms=0.01)
ax.set_xlabel('Along track distance (m)', fontsize=12);
ax.set_ylabel('Photon Height (m)', fontsize=12)
ax.set_title('ICESat-2 ATL03', fontsize=14)
ax.tick_params(axis='both', which='major', labelsize=12)
```



[ ]:

2.9 Tutorial for simulated NISAR data access, reading, and visualizing amplitude and phase.

In the following tutorial, we are going to look at how to access and read simulated-NISAR data which is available from NISAR mission. This mission is supposed to be launched in 2024. At this time we are using modified UAVSAR data to simulate the data that we will get from NISAR satellite.

Before we start any work, we will need to prepare our Jupyter workspace with the necessary python packages. In this tutorial we rely on - h5py, numpy, glob, os, h5glance, osgeo, and matplotlib.

You might have some or all packages already installed. Follow the instructions below to check and install the packages required for this tutorial. The block below will check if your packages are available and install them if they are not.

[2]:
```
# ! pip install h5py numpy glob h5glance osgeo matplotlib
```

We also require that you have the gdal package installed out of the Jupyter environment to use: gdal_translate and gdalinfo. If on Mac OS, you might need to install the xcode package before being able to install gdal.

To install gdal outside the Jupyter environment, please follow the instructions here as it will vary by the OS you are using: https://gdal.org/download.html

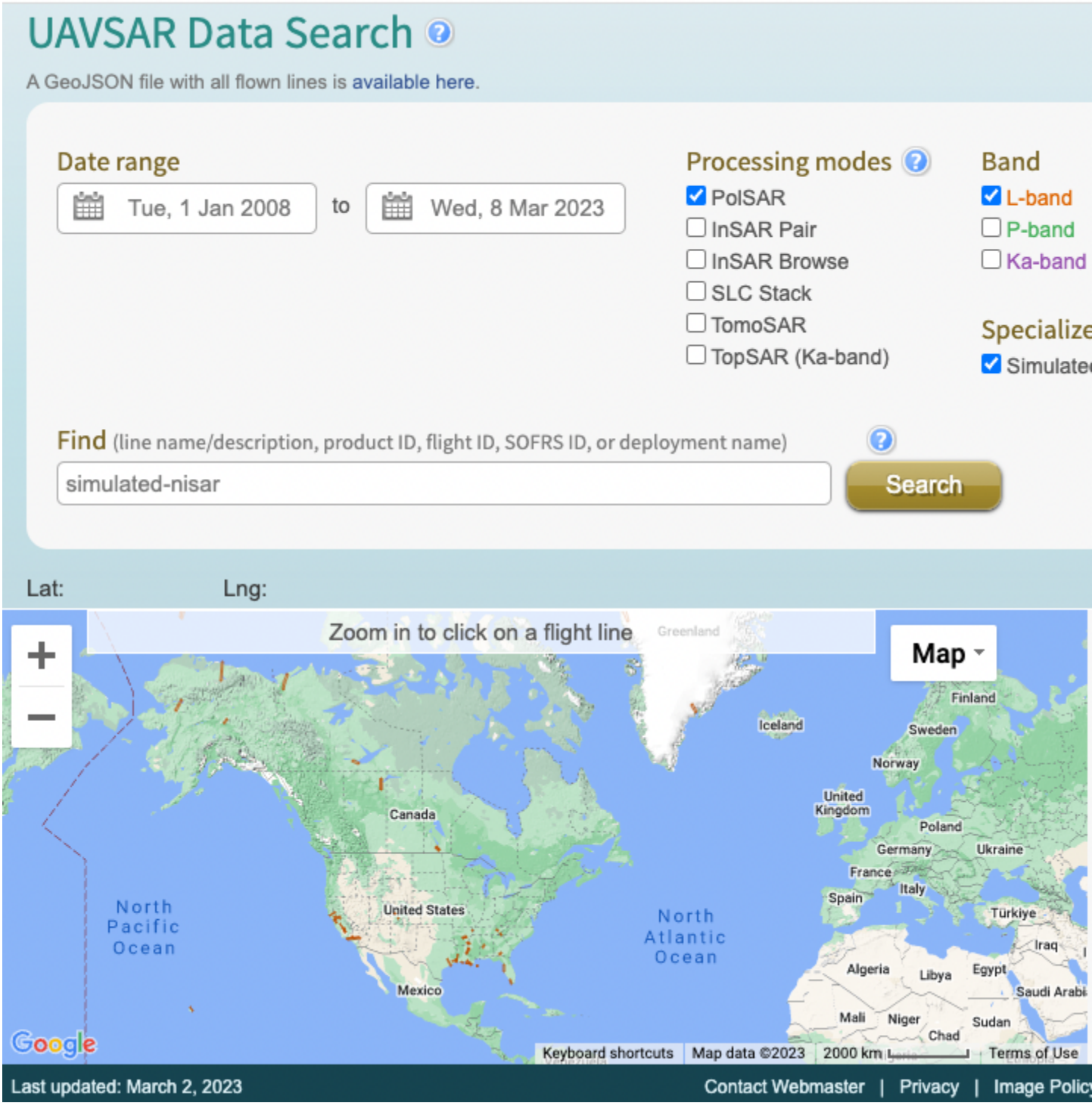
Import installed packages to read the NISAR h5 files and look at the data structure.

[35]:
```
import h5py
import numpy as np
import glob
import os
from h5glance import H5Glance
```

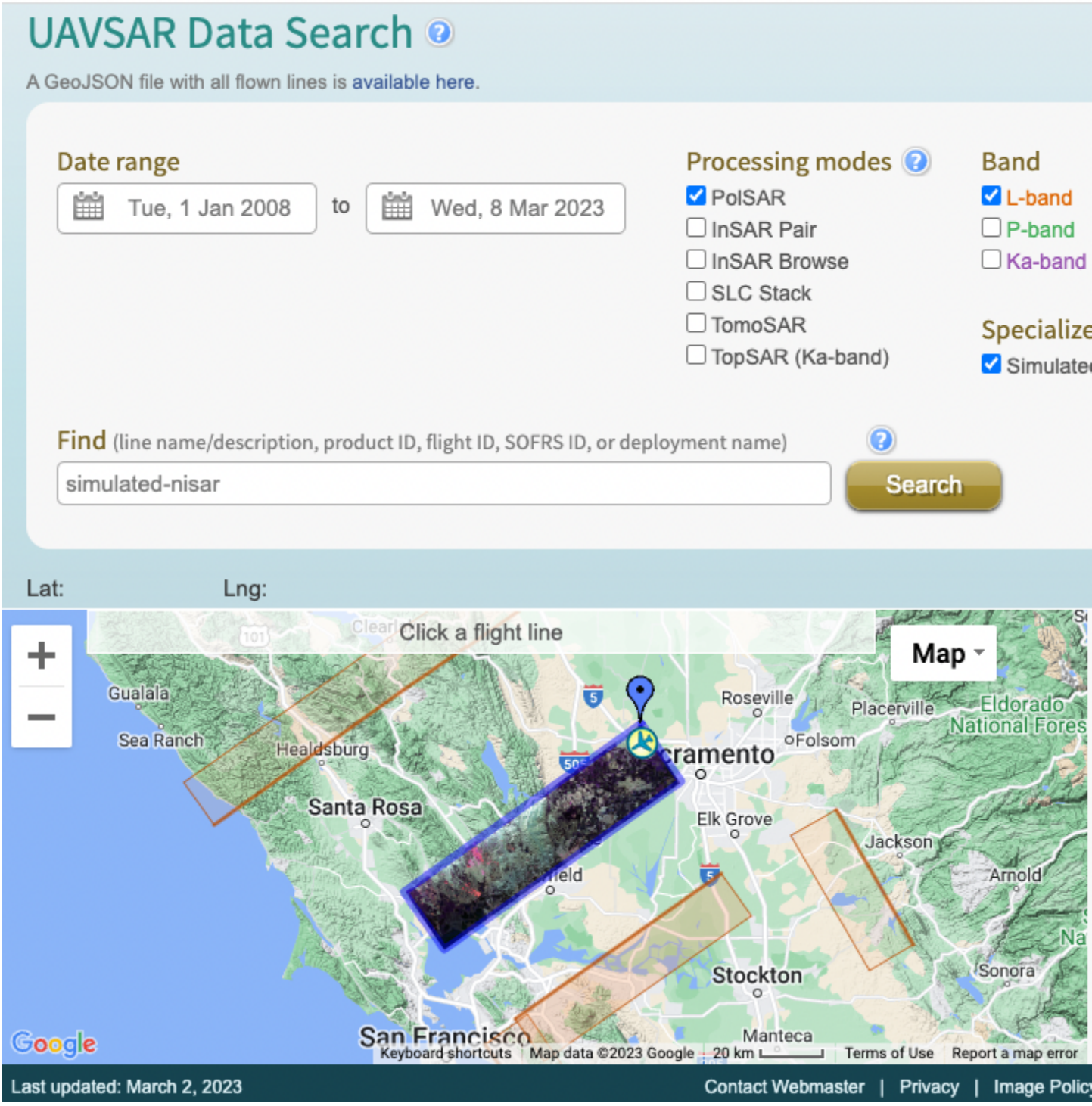
2.9.1 NISAR data can be downloaded from https://uavsar.jpl.nasa.gov/cgi-bin/data.pl

At the time of this tutorial, there are 21033 NISAR products available through the above link. There are a few ways to download data that we can use. We can directly download an SLC file or we can download an H5 file and translate it to SLC to work with this tutorial. In this tutorial, we demonstrate how to get an H5 file, translate it to SLC and use it. Users can alternatively directly download SLC files.

To select a data file from the UAVSAR website, check the Simulated NISAR under specialized products. This will show the flight lines available on the map and list them to the right of the map.



Click on one of the flight lines of interest or select them from the list.



Then click on "View", next to the selected line from the list. This will open a data product page with optional components and instructions to download data in various formats. We will use the "wget" method as specified at the bottom of this page. Be mindful that these H5 files are huge and it will take some time to download on slower internet connections.

[36]:
```
!wget https://downloaduav.jpl.nasa.gov/Release2d/SanAnd_05518_18072_003_181002_L090_
↪CX_03/SanAnd_05518_18072_003_181002_L090_CX_129_03.h5
```

```
--2023-03-08 17:36:30--  https://downloaduav.jpl.nasa.gov/Release2d/SanAnd_05518_
↪18072_003_181002_L090_CX_03/SanAnd_05518_18072_003_181002_L090_CX_129_03.h5
Resolving downloaduav.jpl.nasa.gov (downloaduav.jpl.nasa.gov)... 137.78.249.121
Connecting to downloaduav.jpl.nasa.gov (downloaduav.jpl.nasa.gov)|137.78.249.121|:
↪443... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://uavsar.jpl.nasa.gov/Release2d/SanAnd_05518_18072_003_181002_L090_CX_
↪03/SanAnd_05518_18072_003_181002_L090_CX_129_03.h5 [following]
--2023-03-08 17:36:31--  https://uavsar.jpl.nasa.gov/Release2d/SanAnd_05518_18072_003_
↪181002_L090_CX_03/SanAnd_05518_18072_003_181002_L090_CX_129_03.h5
Resolving uavsar.jpl.nasa.gov (uavsar.jpl.nasa.gov)... 137.78.249.121
Connecting to uavsar.jpl.nasa.gov (uavsar.jpl.nasa.gov)|137.78.249.121|:443...␣
↪connected.
HTTP request sent, awaiting response... 200 OK
Length: 3016099668 (2.8G)
Saving to: 'SanAnd_05518_18072_003_181002_L090_CX_129_03.h5'

SanAnd_05518_18072_ 100%[===================>]   2.81G  56.1MB/s    in 54s

2023-03-08 17:37:44 (52.8 MB/s) - 'SanAnd_05518_18072_003_181002_L090_CX_129_03.h5'␣
↪saved [3016099668/3016099668]
```

Once downloaded, the files will be saved in your current working directory.

[37]:
```
os.getcwd()
```

[37]:
```
'/projects/my-private-bucket/Nisar'
```

If you downloaded multiple H5 files, initialize an empty variable to store the names of all H5 files in the directory.

[38]:
```
nisar_list = []
```

[39]:
```
for file in glob.glob("*.h5"):
    nisar_list.append(file)
    print(file)
```

```
SanAnd_05518_18072_003_181002_L090_CX_129_03.h5
```

2.9.2 Explore the structure of NISAR *.H5 file using H5glance

[40]:
```
H5Glance(nisar_list[0])
```

[40]:
```
SanAnd_05518_18072_003_181002_L090_CX_129_03.h5/ (6 attributes)
└science    (1 children)
```

2.9.3 Use gdal to convert HH polarization for frequencyA to ENVI SLC format and visualize it.

(More example notebooks are provided here: https://github.com/isce-framework/sds-ondemand)

Before we go ahead and convert the HH polarization, check out the metadata information associated with it.

[41]:
```
!gdalinfo HDF5:"/projects/my-private-bucket/Nisar/SanAnd_05518_18072_003_181002_L090_
↪CX_129_03.h5"://science/LSAR/SLC/swaths/frequencyA/HH
```

```
Driver: HDF5Image/HDF5 Dataset
Files: /projects/my-private-bucket/Nisar/SanAnd_05518_18072_003_181002_L090_CX_129_03.
↪h5
Size is 2640, 26678
Metadata:
  contact=nisarops@jpl.nasa.gov
  Conventions=CF-1.7
  institution=NASA JPL
  mission_name=NISAR
  reference_document=TBD
  title=NISAR L1 SLC Product
Corner Coordinates:
Upper Left  (    0.0,    0.0)
Lower Left  (    0.0,26678.0)
Upper Right ( 2640.0,    0.0)
Lower Right ( 2640.0,26678.0)
Center      ( 1320.0,13339.0)
Band 1 Block=128x128 Type=CFloat32, ColorInterp=Undefined
  Metadata:
    science_LSAR_SLC_swaths_frequencyA_HH_description=Focused SLC image (HH)
    science_LSAR_SLC_swaths_frequencyA_HH_units=DN
```

Translate the HH polarization into SLC files. SLC files are Single Look Complex files. More information can be found in the product description here.

[43]:
```
!gdal_translate -of ENVI HDF5:"/projects/my-private-bucket/Nisar/SanAnd_05518_18072_
↪003_181002_L090_CX_129_03.h5"://science/LSAR/SLC/swaths/frequencyA/HH HH.slc
```

```
Input file size is 2640, 26678
0...10...20...30...40...50...60...70...80...90...100 - done.
```

Check the metadata again.

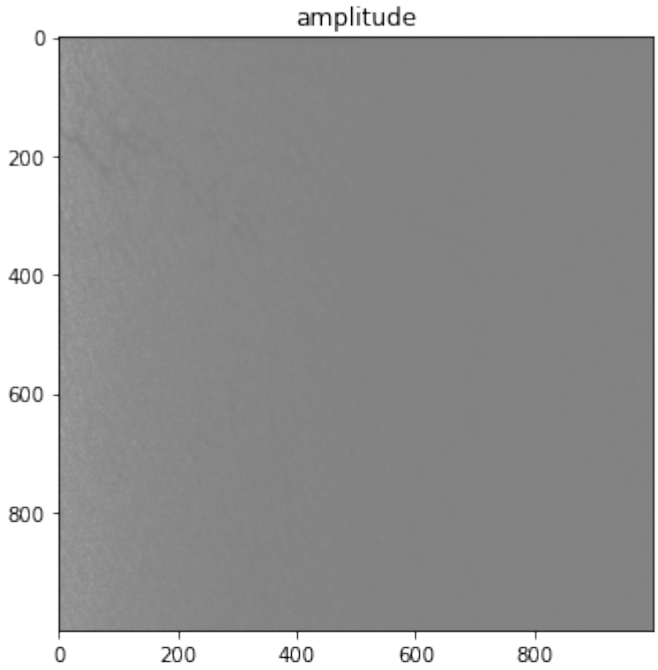
[44]:
```
!gdalinfo HH.slc
```

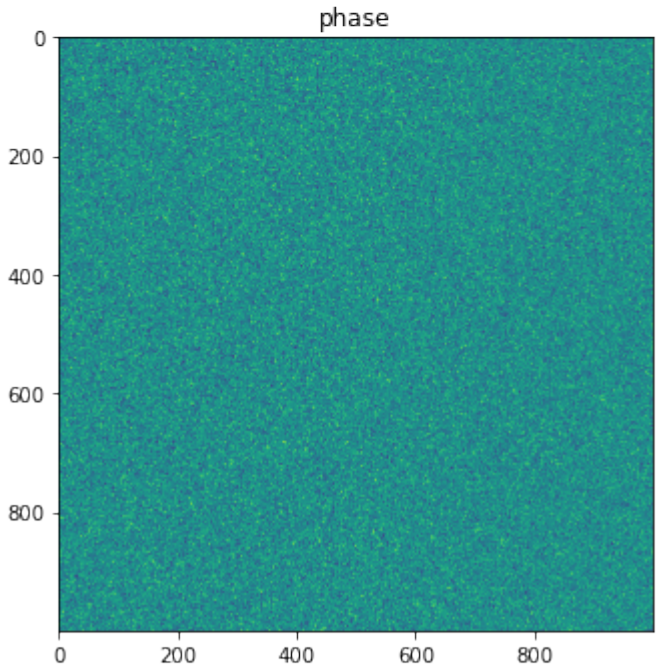
```
Driver: ENVI/ENVI .hdr Labelled
Files: HH.slc
       HH.slc.aux.xml
       HH.hdr
Size is 2640, 26678
Metadata:
  contact=nisarops@jpl.nasa.gov
  Conventions=CF-1.7
  institution=NASA JPL
  mission_name=NISAR
  reference_document=TBD
  title=NISAR L1 SLC Product
Image Structure Metadata:
  INTERLEAVE=BAND
Corner Coordinates:
Upper Left  (    0.0,    0.0)
Lower Left  (    0.0,26678.0)
Upper Right ( 2640.0,    0.0)
Lower Right ( 2640.0,26678.0)
Center      ( 1320.0,13339.0)
Band 1 Block=2640x1 Type=CFloat32, ColorInterp=Undefined
  Metadata:
    science_LSAR_SLC_swaths_frequencyA_HH_description=Focused SLC image (HH)
    science_LSAR_SLC_swaths_frequencyA_HH_units=DN
```

2.9.4 Plotting Amplitude and Phase

[45]:
```
import numpy as np
from osgeo import gdal
import matplotlib.pyplot as plt

ds = gdal.Open("HH.slc", gdal.GA_ReadOnly)

# Extract a subset of the SLC to display
x0 = 0
y0 = 10
x_offset = 1000
y_offset = 1000

slc = ds.GetRasterBand(1).ReadAsArray(x0, y0, x_offset, y_offset)
print(slc)


fig = plt.figure(figsize=(14, 12))

# Display amplitude of the slc
ax = fig.add_subplot(2,1,1)
ax.imshow(np.abs(slc), vmin = -2, vmax=2, cmap='gray')
ax.set_title("amplitude")

# Display phase of the slc
ax = fig.add_subplot(2,1,2)
ax.imshow(np.angle(slc))
ax.set_title("phase")

plt.show()
```

```
[[-0.05650587-0.00651771j  0.00275015-0.02713971j -0.01253977+0.01070294j
  ... -0.04617569-0.04110472j -0.00265791-0.05650726j
  -0.02567512+0.03241472j]
 [-0.04666522+0.00447277j -0.00942332-0.00897887j  0.04313291+0.05079061j
  ... -0.00320398+0.01296128j -0.04679057+0.07169638j
  -0.03488599+0.01855517j]
 [-0.03761201-0.02987598j  0.02749964+0.01824897j  0.02109917-0.0422815j
  ...  0.05811677+0.05686287j  0.02144936+0.04998003j
  -0.03564805+0.06198853j]
 ...
 [ 0.07378234+0.05721019j  0.35254684+0.08179998j  0.34999567-0.00636293j
  ... -0.03517671-0.06153427j -0.04970409-0.05051059j
  -0.06259989-0.03799874j]
 [-0.04348793-0.03808671j -0.04053086-0.5250713j   0.34176347-0.4646253j
  ... -0.0237219 -0.07149725j -0.0344082 -0.07497716j
   0.00491631+0.00245685j]
 [ 0.02678669-0.07999389j  0.19715716+0.11465938j  0.41922772+0.13021956j
  ... -0.00058665-0.08080925j -0.03730002-0.06477241j
   0.02169864+0.00241395j]]
```


amplitude
0
200
400
600
800
0
200
400
600
800


phase
0
200
400
600
800
0
200
400
600
800

[ ]:

CHAPTER 3

Technical Tutorials

3.1 Search

MAAP users are advised to use two catalogs:

1. Use NASA's Operational CMR to discover NASA-produced and curated data: https://cmr.earthdata.nasa.gov/search/site/docs/search/api.html.
2. Use MAAP STAC for data not found in NASA CMR, and data produced by MAAP users: https://stac.maap-project.org/docs.

Warning: The https://cmr.maap-project.org catalog will be deprecated by **May 1, 2023**. Users should request collections they need from this catalog to be made discoverable in the MAAP STAC or NASA's Operational CMR if they're not already there.

More information on each catalog and migrating from MAAP's CMR here: MAAP's Dual Catalog.

3.1.1 MAAP's Dual Catalog

MAAP users are advised to use two catalogs:

1. Use NASA's Operational CMR to discover NASA-produced and curated data: https://cmr.earthdata.nasa.gov/search/site/docs/search/api.html.
2. Use MAAP STAC for data not found in NASA CMR, and data produced by MAAP users: https://stac.maap-project.org/docs.

Warning: The https://cmr.maap-project.org catalog will be deprecated by **May 1, 2023**. Users should request collections they need from this catalog to be made discoverable in the MAAP STAC or NASA's Operational CMR if they're not already there.

More information on each catalog and migrating from MAAP's CMR is detailed in the bottom of this page.

MAAP STAC

MAAP STAC (https://stac.maap-project.org) is dedicated to datasets not accessible via NASA's CMR, such as GEDI Cal/Val datasets, ESA datasets, and user-shared data products.

STAC discovery

Users can discover data in MAAP STAC using pystac-client or https://stac-browser.maap-project.org.

API documentation is available here: https://stac.maap-project.org/docs (will return MAAP STAC results).

The general STAC API spec is here: https://api.stacspec.org/v1.0.0-rc.1/core/.

An example of using pystac-client is included above and in Searching STAC Documentation.

Data Access via STAC

Data assets (files) published to STAC have not moved from the S3 bucket `s3://nasa-maap-data-store`. ESA data is accessible via public HTTP access. NASA data in S3 is accessible publicly or via role-based bucket policy access.

Users are encouraged to use common AWS S3 libraries for NASA data access, such as Python's boto3.

Each item should have a "data" asset which includes a URL to the data.

For example, https://stac.maap-project.org/collections/BIOSAR1/items/biosar1_roi_lidar58 includes:

```
"assets": {
  "data": {
    "href": "https://bmap-catalogue-data.oss.eu-west-0.prod-cloud-ocb.orange-business.
↪com/Campaign_data/biosar1/biosar1_roi_lidar58.shx",
    "type": "application/octet-stream",
    "roles": [
      "data"
    ]
  },
}
```

NASA's Operational CMR

CMR Discovery

Users can discover data NASA's Operational CMR via its publicly accessible API: https://cmr.earthdata.nasa.gov and user interface: https://search.earthdata.nasa.gov.

CMR Search documentation can be found in Searching Collections and Searching Granules and https://cmr.earthdata.nasa.gov/search/site/docs/search/api.html.

CMR Access

For all NASA MAAP users, access to NASA'S Operational data is provided via a federated access token. Anything that is in NASA's Operational CMR should be accessed via maap-py so that the federated access token can be used. Users can also access data from LPDAAC (and possibly other DAACs in the future) without maap-py since the workspace should have access via a role-based bucket policy on the LPDAAC cloud bucket.

Anyone can access data through Earthdata Login as well.

Find more documentation about how to access data in CMR in the Access section of this documentation.

Migrating from MAAP's CMR

If you're migrating code from using https://cmr.maap-project.org, we're here to help. The documentation below should support migrating to https://cmr.earthdata.nasa.gov and https://stac.maap-project.org. If not, please contact the data team for assistance.

Migration Steps:

1. Identify where your code is using https://cmr.maap-project.org and which datasets are being discovered and accessed.
2. Once you've identified the datasets, use https://search.earthdata.nasa.gov or https://stac-browser.maap-project.org to find out if the dataset is available through NASA's Operational CMR or MAAP's STAC catalog. If you don't see your datasets in one of those places, reach out to the data team so they can prioritize that dataset for publication to MAAP STAC.
3. If the dataset is in NASA's Operational CMR and you're using MAAP's Python library `maap-py` to discover and access data, add the parameter `cmr_host="cmr.earthdata.nasa.gov"` to your `maap.searchCollection` and `maap.searchGranule` function calls. Update the `concept_id` to match the one from NASA's Operational CMR if you're using it to identify a specific collection or granule.
4. If the dataset is in MAAP STAC, use `pystac_client` (https://pystac-client.readthedocs.io/en/stable/) or an HTTP library to call the STAC HTTP API endpoints directly.

Examples:

Example of switching a granule search to NASA's Operational CMR:

The code below discovers granules from the `ABoVE LVIS L2 Geolocated Surface Elevation Product`:

```
COLLECTION_ID = 'C1200125288-NASA_MAAP'
results = maap.searchGranule(concept_id=COLLECTION_ID)
pprint(f'Got {len(results)} results')
```

This dataset exists in NASA's Operational CMR. Using https://search.earthdata.nasa.gov, I discovered the collection's `concept_id` by searching for "ABoVE LVIS L2 Geolocated Surface Elevation Product" and copying the `concept_id` from the URL of the result to modify the code below:

```
COLLECTION_ID = 'C1513105984-NSIDC_ECS'
results = maap.searchGranule(
  cmr_host='cmr.earthdata.nasa.gov',
```

(continues on next page)

(continued from previous page)

```
  concept_id=COLLECTION_ID
)
pprint(f'Got {len(results)} results')
```

Example of switching a granule search to MAAP STAC:

This code discovers granules from the `Landsat 8 Operational Land Imager (OLI) Surface Reflectance Analysis Ready Data (ARD) V1, Peru and Equatorial Western Africa, April 2013-January 2020`.

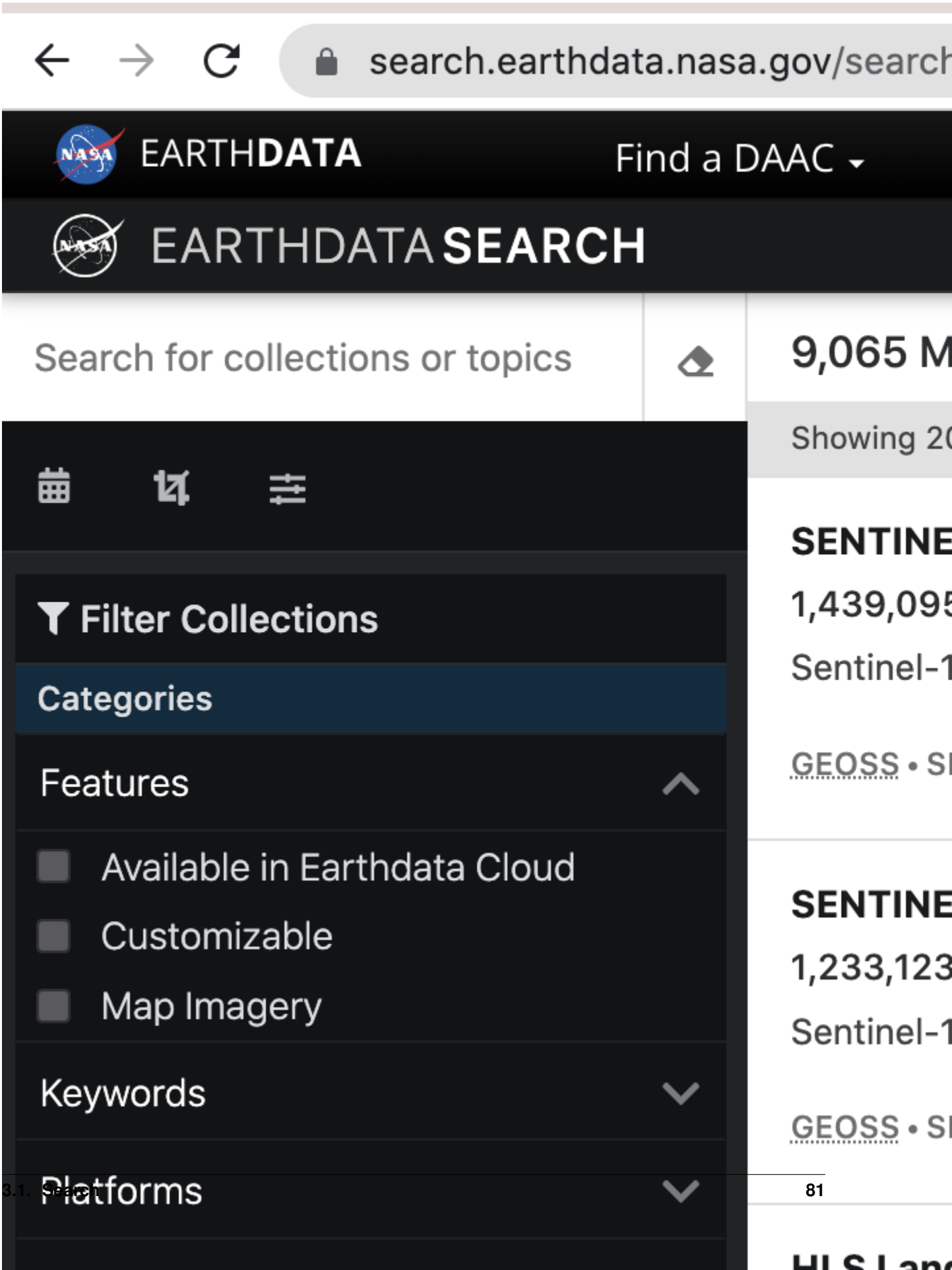
```
COLLECTION_ID = 'C1200110769-NASA_MAAP'

results = maap.searchGranule(concept_id=COLLECTION_ID)
pprint(f'Got {len(results)} results')
```

You can use https://stac-browser.maap-project.org to find the STAC collection ID for that dataset, which is `Landsat8_SurfaceReflectance`.

```
from pystac_client import Client
URL = 'https://stac.maap-project.org/'
cat = Client.open(URL)
for collection in cat.get_all_collections():
    print(collection)

collection = cat.get_collection('Landsat8_SurfaceReflectance')
items = collection.get_items()
```

3.1.2 Using the NASA Earthdata Search Client Graphical User Interface

The Earthdata Search Client (EDSC) allows users to search, preview, download, and access EOSDIS Earth observation data. NASA's EDSC is located at search.earthdata.nasa.gov. ESDC provides a graphical user interface (GUI) for NASA's Common Metadata Repository (CMR) to ease the process of discovering data.


search.earthdata.nasa.gov/search
EARTHDATA
Find a DAAC
EARTHDATA SEARCH
Search for collections or topics
9,065 M
Showing 20
SENTINE
1,439,095
Sentinel-1
GEOSS • SI
SENTINE
1,233,123
Sentinel-1
GEOSS • SI
Filter Collections
Categories
Features
Available in Earthdata Cloud
Customizable
Map Imagery
Keywords
Platforms

(*Image of the NASA EDSC GUI*)

Using the Earthdata Search Client

We can use the searchbar to search for a term (example: *GEDI*) to narrow the resulting collections. The search can be further refined by picking a temporal range from the calendar using the **Temporal** button and setting spatial boundaries using the **Spatial** button. Additionally, using the **Advanced search** lets you search by the HUC ID or region (more information about hydrological units may be found here). You can use the **Clear Filters** button to undo any filters that have been set. The sidebar to the left allows you to further refine your search by selecting one of the available facets. These include *Features*, *Keywords*, *Platforms*, *Instruments*, *Organizations*, *Projects*, and *Processing levels*.

We can also use the tools with the background map to refine our search. We can search spatially using **Search by spatial polygon**, **Search by spatial rectangle**, and **Search by spatial coordinate**. Layers may be edited using the **Edit layers** button and deleted using the **Delete layers** buttons. There are also options for **North Polar Stereographic**, **Geographic (Equirectangular)**, and **South Polar Stereographic** projections. There are options to **Zoom in** +, **Zoom home**, and **Zoom out** −. Finally, we can change the basemap by selecting the **Map layers** button.

The results of the search are displayed in the *Matching Collections* section. Collection names and summaries for each result are shown here. The **View collection details** button may be used to view related URLs and additional information about the selected collection. Also, collections may be added to a project using the **Add collection to the current project** button. Clicking anywhere else on a result allows you to see the granules within the collection available for download.

Using the Earthdata Search Client in the Algorithm Development Environment (ADE)

The EDSC can also be accessed within the ADE when using the MAAP JupyterLab IDE. When inside of a workspace, we can use the **Data Search** tab at the top of the Jupyter window to see the option to **Open EarthData Search**. We can also access the EDSC by selecting the **Commands** tab on the JupyterLab sidebar and scrolling down to the *Search* section where the **Open EarthData Search** option is located. Alternatively, we can use the search bar at the top of the JupyterLab sidebar to find this option, or press Ctrl+Shift+C to be taken directly to the search bar.

The JupyterLab Search extension is integrated with the embedded Earthdata Search Client allowing users to paste queries and query results into a notebook. This is accomplished by using the EDSC and navigating to the desired JupyterLab notebook. Next, use the **Data Search** tab at the top of the Jupyter window to see the **Paste Search Query** and **Paste Search Results** options can be selected (these can also be accessed by the **Commands** tab on the JupyterLab sidebar).

3.1.3 Searching for Collections in NASA's Operational CMR using maap-py

These examples walk through the MAAP API functionality of searching for collections based on specific parameters. Collections are groupings of files that share the same product specification. Searching for collections can be useful for finding individual files, known as granules, which are used for processing.

We begin by importing the `maap` package and creating a new MAAP class.

[1]:
```
# import the MAAP package to handle queries
from maap.maap import MAAP

# import printing package to help display outputs
from pprint import pprint
```

(continues on next page)

(continued from previous page)

```
# invoke the MAAP search client
maap = MAAP()
```

We can use the `maap.searchCollection` function to return a list of desired collections. Before using this function, let's use the `help` function to view the specific arguments and keywords for `maap.searchCollection`.

[2]:
```
# view help for the searchCollection function
help(maap.searchCollection)
```

```
Help on method searchCollection in module maap.maap:

searchCollection(limit=100, **kwargs) method of maap.maap.MAAP instance
    Search the CMR collections
    :param limit: limit of the number of results
    :param kwargs: search parameters
    :return: list of results (<Instance of Result>)
```

The help text is showing that `maap.searchCollection` accepts a limit and search parameters. The limit parameter limits the number of resulting collections returned by `maap.searchCollection`. Note that `limit=100` means that the *default limit* for results from the MAAP API is 100. `maap.searchCollection` accepts any additional search parameters that are included in the CMR. For a list of accepted parameters, please refer to the CMR Search Collections API reference.

In this example we will explore search options including:

1. Finding all Collections
2. Searching by temporal filter
3. Searching by spatial filter
4. Using the results from one search as inputs into another
5. Searching by additional attributes

Finding all Collections

Here we will demonstrate how to create a list containing all of the collections contained within the CMR. To do this, we will use the `maap.searchCollection` function without any additional search parameters.

[3]:
```
# search all collections
results = maap.searchCollection(cmr_host="cmr.earthdata.nasa.gov")

# print the number of collections
pprint(f'Got {len(results)} results')
```

```
'Got 100 results'
```

We get 100 results because of the default page limit. The result from the MAAP API is a list of collections where each element in the list is the metadata for that particular collection. To change the limit, type `limit=` and then a value within the parentheses after `maap.searchCollection()`.

Let's look at the metadata for the first collection in our list of results (`results[0]`) using `pprint`. For formatting purposes, we can use the `depth` parameter to control the number of levels of metadata detail to display. By default, there is no constraint on the depth. By setting a `depth` parameter (in this case `depth=2`), we can ensure that the next contained level is replaced by an ellipsis.

[9]:
```
# print the metadata for the first collection
# we use the depth parameter to set the layer of metadata detail in the results, with␣
↪(1) having the least detail
# (1) displays the concept ID, format, and revision ID
# adjust the depth to a larger value (6) if you would like to view all of the metadata
pprint(results[0], depth=2)
```

```
{'Collection': {'Campaigns': {...},
                'CollectionState': 'COMPLETE',
                'Contacts': {...},
                'DOI': {...},
                'DataSetId': '10 Days Synthesis of SPOT VEGETATION Images '
                             '(VGT-S10)',
                'Description': 'The VGT-S10 are near-global or continental, '
                               '10-daily composite images which are '
                               "synthesised from the 'best available' "
                               'observations registered in the course of every '
                               "'dekad' by the orbiting earth observation "
                               'system SPOT-VEGETATION. The products provide '
                               'data from all spectral bands (SWIR, NIR, RED, '
                               'BLUE), the NDVI and auxiliary data on image '
                               'acquisition parameters. The VEGETATION system '
                               'allows operational and near real-time '
                               'applications, at global, continental and '
                               'regional scales, in very broad environmentally '
                               'and socio-economically critical fields. The '
                               'VEGETATION instrument is operational since '
                               'April 1998, first with VGT1, from March 2003 '
                               'onwards, with VGT2. More information is '
                               'available on: '
                               'https://docs.terrascope.be/#/DataProducts/SPOT-VGT/
↪Level3/Level3',
                'InsertTime': '2023-04-07T08:27:18.514Z',
                'LastUpdate': '2023-04-07T08:27:18.514Z',
                'LongName': 'Not provided',
                'OnlineAccessURLs': {...},
                'OnlineResources': {...},
                'Platforms': {...},
                'ProcessingLevelId': 'NA',
                'ScienceKeywords': {...},
                'ShortName': 'urn:ogc:def:EOP:VITO:VGT_S10',
                'Spatial': {...},
                'Temporal': {...},
                'UseConstraints': {...},
                'VersionId': '1'},
 'concept-id': 'C2207472890-FEDEO',
 'format': 'application/echo10+xml',
 'revision-id': '5'}
```

The `Collection` key has all of the collection information including attributes, the archive center, spatial, and temporal information. The `concept-id` is a unique identifier for this collection. It can be used to further refine search results from the CMR, such as when searching for granule information.

Searching by Temporal Filter

Here we use a temporal filter to narrow down our results using the `temporal` keyword in our search. The temporal keyword takes datetime information in a specific format. The date format used is `YYYY-MM-DDThh:mm:ssZ` and

temporal search criteria may be either a single date or a date range. If one date is provided then it can be inferred as the start or end date. To define a start date and return all collections after the date, put a comma after the date (`YYYY-MM-DDThh:mm:ssZ,`). To define a end date and return all granules prior to the data, put a comma before the date (`,YYYY-MM-DDThh:mm:ssZ`). Lastly, to get a date range, provide the start date and end date separated by a comma (`YYYY-MM-DDThh:mm:ssZ,YYYY-MM-DDThh:mm:ssZ`).

In this example we will search for one month of data.

[5]:
```
datetimeRange = '2000-01-01T00:00:00Z,2000-01-31T23:59:59Z' # specify datetime range␣
↪to search for data in January 2000

results = maap.searchCollection(
    cmr_host = "cmr.earthdata.nasa.gov",
    temporal = datetimeRange
)
pprint(f'Got {len(results)} results')
```

```
'Got 100 results'
```

[6]:
```
collectionName = results[0]['Collection']['ShortName'] # get the collection short name
collectionDate = results[0]['Collection']['Temporal']['RangeDateTime'][
↪'BeginningDateTime'] # get the collection start time

pprint(
    f'Collection {collectionName} was acquired starting at {collectionDate}',␣
↪width=100)
```

```
'Collection GLDAS_NOAH025_3H was acquired starting at 2000-01-01T00:00:00.000Z'
```

It appears the first result correctly matches with the beginning and ending temporal search parameters. Keep in mind that the results are limited to 100 so the final collection returned may not match the end date that was searched for.

Searching by Spatial Filter

Here we will illustrate how to search for collections by a spatial filter. There are a couple of spatial filters available to search by in the CMR including point, line, polygon, and bounding box. In this example, we will explore filtering with a bounding box which is a sequence of four latitude and longitude values in the order of `[W,S,E,N]`.

[7]:
```
collectionDomain = '-42,10,42,20' # specify bounding box to search by

results = maap.searchCollection(
    cmr_host = "cmr.earthdata.nasa.gov",
    bounding_box = collectionDomain
)
pprint(f'Got {len(results)} results')
```

```
'Got 100 results'
```

[8]:
```
collectionName = results[3]['Collection']['ShortName'] # get a collection short name
collectionGeometry = results[3]['Collection']['Spatial']['HorizontalSpatialDomain'][
↪'Geometry'] # grab the spatial information from collection

pprint(f'Collection {collectionName} was acquired within the following geometry: ',␣
↪width=100)
pprint(collectionGeometry)
```

```
'Collection gov.noaa.nodc:0000029 was acquired within the following geometry: '
{'BoundingRectangle': {'EastBoundingCoordinate': '-16.25',
                       'NorthBoundingCoordinate': '46.263167',
                       'SouthBoundingCoordinate': '0.766667',
                       'WestBoundingCoordinate': '-124.041667'},
 'CoordinateSystem': 'CARTESIAN'}
```

We can see from the first collection that the spatial coordinates of the collection intersect our search box.

3.1.4 Searching for Granules in NASA's Operational CMR using maap-py

These examples will walk through the MAAP API functionality of searching granules within a collection based on specific parameters. Granules are individual files from a sensor where a group of granules make a collection within CMR. The granules are the raw data that will be used for processing.

We begin by importing the `maap` and `pprint` packages. Then invoke the `MAAP` constructor, setting the `maap_host` argument to `'api.ops.maap-project.org'`.

[1]:
```
# import the MAAP package
from maap.maap import MAAP

# import printing package to help display outputs
from pprint import pprint

# invoke the MAAP constructor using the maap_host argument
maap = MAAP(maap_host='api.ops.maap-project.org')
```

Here we view the specific arguments and keywords for the `maap.searchGranule` function.

[2]:
```
help(maap.searchGranule)
```

```
Help on method searchGranule in module maap.maap:

searchGranule(limit=20, **kwargs) method of maap.maap.MAAP instance
    Search the CMR granules

    :param limit: limit of the number of results
    :param kwargs: search parameters
    :return: list of results (<Instance of Result>)
```

As we can see from the result, `maap.searchGranule` accepts a limit keyword which limits the number of results from CMR. `maap.searchGranule()` also accepts any additional search parameters that are included in CMR. For a list of accepted parameters, please refer to the CMR Search Granules API reference.

It is important to note that *the default limit on results from the MAAP API is 20.* To increase the number of results we will specify a variable and use it in later queries.

[3]:
```
# get at max 500 results from CMR
MAX_RESULTS = 500
```

In this example we will explore search options including:

1. Searching by collection concept ID
2. Searching by temporal filter
3. Searching by spatial filter

4. Using the results from one search as inputs into another
5. Searching by additional attributes

For the next couple of examples, we will focus on the ICESat-2/ATLAS Land and Vegetation Height dataset.

Searching by Collection Short Name, Version

Here we will search by a short name and version which should uniquely identify a collection CMR. HOWEVER, some datasets exist both in the cloud and on-prem, so in the following example we actually get **2** results.

[4]:
```
atl08_collections = maap.searchCollection(
    short_name='ATL08',
    version='005',
    cmr_host='cmr.earthdata.nasa.gov'
)
len(atl08_collections)
```

[4]:
```
2
```

If you inspect the results, you will see the second result has distribution information which points to an S3 bucket location. You can see this information with the follow code: `atl08_collections[1]['Collection']['DirectDistributionInformation']`.

A simpler solution to finding just the cloud-hosted dataset is to add the `cloud_hosted="true"` parameter to our search.

[5]:
```
atl08_collections = maap.searchCollection(
    short_name='ATL08',
    version='005',
    cmr_host='cmr.earthdata.nasa.gov',
    cloud_hosted="true"
)
len(atl08_collections)
```

[5]:
```
1
```

Now we can look up the collection concept id to find only granules in the cloud-hosted ATL08 v005 dataset.

[6]:
```
COLLECTION_ID = atl08_collections[0]['concept-id']

results = maap.searchGranule(
    concept_id=COLLECTION_ID,
    cmr_host='cmr.earthdata.nasa.gov',
    limit=MAX_RESULTS)
pprint(f'Got {len(results)} results')
```

```
'Got 500 results'
```

We were able to get 500 results! There are most likely more than 500 granules in search results, but remember we limited the results to 500 granules. The result from the MAAP API is a list of granules where each element in the list is the metadata for that particular granule.

Now let's look at the metadata for the first result.

[7]:
```
# print the first granule's metadata
# we use the depth parameter to set the layer of metadata detail in the results, with␣
↪(1) having the least detail
# (1) displays the collection concept ID, concept ID, format, and revision ID
```

(continues on next page)

(continued from previous page)

```
# adjust the depth to a larger value (6) if you would like to view all of the metadata
pprint(results[0], depth=2)
```

```
{'Granule': {'AssociatedBrowseImageUrls': {...},
             'Collection': {...},
             'DataGranule': {...},
             'GranuleUR': 'ATL08_20181014001049_02350102_005_01.h5',
             'InsertTime': '2021-11-14T23:43:07.741Z',
             'LastUpdate': '2021-11-14T23:43:07.741Z',
             'OnlineAccessURLs': {...},
             'OnlineResources': {...},
             'OrbitCalculatedSpatialDomains': {...},
             'Spatial': {...},
             'Temporal': {...}},
 'collection-concept-id': 'C2153574670-NSIDC_CPRD',
 'concept-id': 'G2166182816-NSIDC_CPRD',
 'format': 'application/echo10+xml',
 'revision-id': '1'}
```

There is a lot of information in the metadata so let's break it down...

The `Granule` key has all of the granule information including attributes, browse imagery URLs, spatial, and temporal information. The `collection-concept-id` should match what you searched by and be the same for each granule. Lastly the granule specific `concept-id` is a unique identifier for this granule. This information can be used to further refine search results from CMR, specifically the granule information.

Searching by Temporal Filter

Here we will combine a search from earlier using the Collection Concept ID with a temporal filter to fine tune our results using the `temporal` keyword in our search.

The temporal keyword takes datetime information in a specific format. The date format used is `YYYY-MM-DDThh:mm:ssZ` and temporal search criteria may be either a single date or a date range. If one date is provided then it can be inferred as start or end date. To define a start date and return all granules after the date, put a comma after the date (`YYYY-MM-DDThh:mm:ssZ,`). To define an end date and return all granules prior to the data, put a comma before the date (`,YYYY-MM-DDThh:mm:ssZ`). Lastly, to get a date range, provide the start date and end date separated by a comma (`YYYY-MM-DDThh:mm:ssZ,YYYY-MM-DDThh:mm:ssZ`).

In this example we will search for one month of data.

[8]:
```
date_range = '2018-12-01T00:00:00Z,2018-12-31T23:59:59Z' # specify a date range to␣
↪search for data for Dec. 2018

results = maap.searchGranule(
    temporal=date_range,
    concept_id=COLLECTION_ID,
    limit=MAX_RESULTS,
    cmr_host="cmr.earthdata.nasa.gov"
)
pprint(f'Got {len(results)} results')
```

```
'Got 500 results'
```

[9]:
```
granuleFilename = results[0]['Granule']['DataGranule']['ProducerGranuleId'] # get the␣
↪granule file name
granuleDate = results[0]['Granule']['Temporal']['RangeDateTime']['BeginningDateTime']
↪# get the granule start time
```

(continues on next page)

(continued from previous page)

```
pprint(f'Granule {granuleFilename} was acquired starting at {granuleDate}',width=100)
```

```
'Granule ATL08_20181201001339_09680103_005_01.h5 was acquired starting at 2018-12-
↪01T00:13:48.477Z'
```

It looks like the first result correctly matches with the beginning temporal search parameter. Keep in mind that the results are limited to 500 so the final granule returned may not match the end date that was searched for.

Searching by Spatial Filter

Here we will illustrate how to search for granules by a spatial filter. There are a couple of spatial filters available to search by in CMR including point, line, polygon, and bounding box. The most simple to use is the bounding box which is a sequence of four latitude and longitude values in the order of `[W,S,E,N]`. In this example, we are going to search for data over Gabon using the `bounding_box` keyword.

[10]:
```
granule_bbox = '8.79799563969,-3.97882659263,14.4254557634,2.32675751384' # specify␣
↪bounding box to search by

COLLECTION_ID = 'C1000000240-LPDAAC_ECS' # Collection title: "NASA Shuttle Radar␣
↪Topography Mission Global 1 arc second V003"

results = maap.searchGranule(
    concept_id=COLLECTION_ID,
    bounding_box=granule_bbox,
    cmr_host="cmr.earthdata.nasa.gov"
)
pprint(f'Got {len(results)} results')
```

```
'Got 20 results'
```

[11]:
```
granule_filename = results[0]['Granule']['DataGranule']['ProducerGranuleId'] # get␣
↪the granule file name
geometry = results[0]['Granule']['Spatial']['HorizontalSpatialDomain']['Geometry'] #␣
↪grab the spatial information from granule

pprint(f'Granule {granule_filename} was acquired within the following geometry: ',␣
↪width=100)
pprint(geometry)
```

```
'Granule S03E012.SRTMGL1.hgt.zip was acquired within the following geometry: '
{'BoundingRectangle': {'EastBoundingCoordinate': '13.00027778',
                       'NorthBoundingCoordinate': '-1.99972222',
                       'SouthBoundingCoordinate': '-3.00027778',
                       'WestBoundingCoordinate': '11.99972222'}}
```

We can see from the first granule that the spatial coordinates of the granule intersect our search box.

The MAAP API provides rich functionality to interact with the CMR instance within the MAAP platform. Users can search datasets programmatically by many parameters and even combine parameters such as spatial and temporal filters to refine results.

3.1.5 Searching for and Compiling a List of Granule IDs for Batch Processing

While using the MAAP ADE, you may wish to create a list of granule IDs to be used for batch processing, granules being the individual files from a sensor that are used for processing. For this example, we will imagine a scenario that

we wish to produce a biomass estimate for a single HLS tile, and then expand that estimate over a larger area. In order to produce this expanded estimate, we will create a list of the HLS files which fall within a certain area.

We start by importing the `maap` and `pprint` packages and creating a new MAAP class.

[1]:
```
# import the MAAP package
from maap.maap import MAAP

# import printing package to help display outputs
from pprint import pprint

# create MAAP class
maap = MAAP()
```

We can use the `searchGranule` function to search for "HLS Landsat Operational Land Imager Surface Reflectance and TOA Brightness Daily Global 30m v2.0" granules. Click here for more information about searching for granules with the `searchGranule` function. Since the default limit on results from the MAAP API is 20, we specify a variable to use in our search query.

[2]:
```
# get at max 1000 results from CMR
MAX_RESULTS = 1000
```

To filter our search to HLS data, we create a variable with the collection ID of the HLS collection. Using the collection concept ID is the preferred method to filter by a collection, as it is a unique identifier which avoids ambiguity.

[3]:
```
COLLECTION_ID = 'C2021957657-LPCLOUD' # specifying the collection ID for the HLS␣
↪dataset
```

Next, we search for granules using the collection ID and a spatial filter. We can use a bounding box as our spatial filter. The bounding box is a sequence of four latitude and longitude values in the order of [W,S,E,N]. For this example, let's search for granules from the HLS data using a bounding box for the country Peru.

[4]:
```
collection_bbox = '-81.4109425524,-18.3479753557,-68.6650797187,-0.0572054988649' #␣
↪specify bounding box to search by

# getting results from granule search using the bounding box, collection ID, and␣
↪results limit
results = maap.searchGranule(
    cmr_host='cmr.earthdata.nasa.gov',
    bounding_box=collection_bbox,
    concept_id=COLLECTION_ID,
    limit=MAX_RESULTS
)
pprint(f'Got {len(results)} results') # print the number of results
```

```
'Got 1000 results'
```

We were able to get 805 results. Each element in the list `results` contains the metadata for one of the granules returned by the search. Within this metadata is the key `concept-id`, which is the unique identifier for each granule. To create a list of granule IDs, we create a new list and use a for loop to add the `concept-id` from each element of `results` into the new list.

[5]:
```
granuleID_list = [] # create list for granule IDs

for result in results: # loop through each element (granule) in the list `results`
    granuleID_list.append(result['concept-id']) # add the concept id for each granule␣
↪to `granuleID_list`
```

(continues on next page)

(continued from previous page)

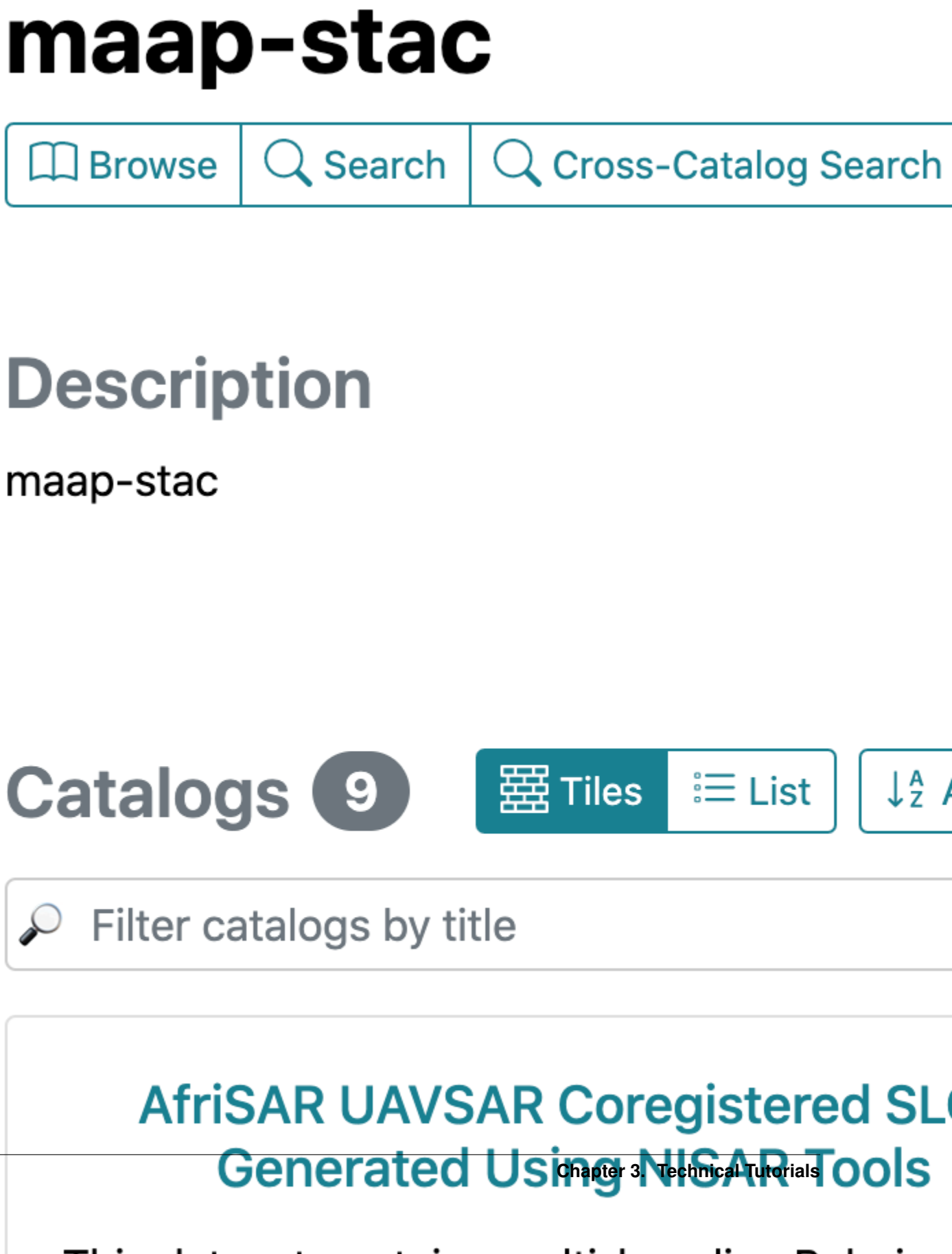
```
# You can uncomment the line below to see the result
# print(granuleID_list)
```

Now we have a list of all the granule IDs for granules in the Landsat 8 collection that fall within the bounding box for the country Peru within `granuleID_list`.

3.1.6 Searching the STAC Catalog

This tutorial provides a basic introduction to searching the MAAP STAC catalog (https://stac.maap-project.org/) using `pystac-client`.

Another method of searching the STAC catalog is via the STAC browser.


maap-stac
Browse
Search
Cross-Catalog Search
Description
maap-stac
Catalogs 9
Tiles
List
Filter catalogs by title
AfriSAR UAVSAR Coregistered SL
Generated Using NISAR Tools

About the STAC Catalog

At this time, the STAC catalog provides discovery of a subset of MAAP datasets. These datasets were selected because MAAP CMR analytics indicated selected datasets were being searched for the most. The data files have not been moved at all in the process of publishing datasets to STAC.

Data will continue to be added to the STAC catalog with priority given to datasets which are known to be in-use by MAAP UWG members through CMR metrics, S3 metrics, direct collaboration with data team members and by request.

Prerequisites

- pystac-client
- rioxarray (for opening a raster as an xarray dataset)

Authorship

- Author: Aimee Barciauskas
- Date: December 13, 2022
- Resources used: https://pystac-client.readthedocs.io/en/stable/tutorials/pystac-client-introduction.html

[1]:
```
%%capture
!pip install -U pystac-client
```

[2]:
```
from pystac_client import Client
```

STAC Client

We first connect to an API by retrieving the root catalog, or landing page, of the API with the Client.open function.

[3]:
```
# STAC API root URL
URL = 'https://stac.maap-project.org/'
cat = Client.open(URL)
cat
```

[3]:
```
<Client id=stac-fastapi>
```

CollectionClient

As with a static catalog the get_collections function will iterate through the Collections in the Catalog. Notice that because this is an API it can get all the Collections through a single call, rather than having to fetch each one individually.

[4]:
```
for collection in cat.get_all_collections():
    print(collection)
```

```
<CollectionClient id=GlobCover_09>
<CollectionClient id=AfriSAR_UAVSAR_Coreg_SLC>
<CollectionClient id=BIOSAR1>
<CollectionClient id=AfriSAR_AGB_Maps_1681>
<CollectionClient id=Landsat8_SurfaceReflectance>
<CollectionClient id=ABLVIS1B>
<CollectionClient id=AfriSAR_UAVSAR_Ungeocoded_Covariance>
<CollectionClient id=GlobCover_05_06>
<CollectionClient id=Global_PALSAR2_PALSAR_FNF>
<CollectionClient id=GEDI02_A>
```

(continues on next page)

(continued from previous page)

```
<CollectionClient id=AFLVIS2>
<CollectionClient id=AFRISAR_DLR>
<CollectionClient id=icesat2-boreal>
<CollectionClient id=AfriSAR_UAVSAR_Normalization_Area>
<CollectionClient id=GEDI_CalVal_Lidar_Data_Compressed>
<CollectionClient id=GEDI_CalVal_Field_Data>
<CollectionClient id=ESACCI_Biomass_L4_AGB_V3_100m_2010>
<CollectionClient id=ESACCI_Biomass_L4_AGB_V3_100m_2018>
<CollectionClient id=ESACCI_Biomass_L4_AGB_V3_100m_2017>
<CollectionClient id=AfriSAR_UAVSAR_Geocoded_SLC>
<CollectionClient id=Global_PALSAR2_PALSAR_Mosiac>
<CollectionClient id=AFRISAR_DLR2>
<CollectionClient id=AfriSAR_UAVSAR_KZ>
<CollectionClient id=GEDI_CalVal_Lidar_Data>
<CollectionClient id=GEDI02_B>
<CollectionClient id=ABoVE_UAVSAR_PALSAR>
<CollectionClient id=Global_Forest_Change_2000-2017>
```

[5]:
```
collection = cat.get_collection('ESACCI_Biomass_L4_AGB_V3_100m_2010')
collection
```

[5]:
```
<CollectionClient id=ESACCI_Biomass_L4_AGB_V3_100m_2010>
```

STAC Items

The main functions for getting items return iterators, where pystac-client will handle retrieval of additional pages when needed. Note that one request is made for the first ten items, then a second request for the next ten.

[6]:
```
items = collection.get_items()

# flush stdout so we can see the exact order that things happen
def get_ten_items(items):
    for i, item in enumerate(items):
        print(f"{i}: {item}")
        if i == 9:
            return

print('First page', flush=True)
get_ten_items(items)
```

```
First page
0: <Item id=S50W080_ESACCI-BIOMASS-L4-AGB_SD-MERGED-100m-2010-fv3.0>
1: <Item id=S50W080_ESACCI-BIOMASS-L4-AGB-MERGED-100m-2010-fv3.0>
2: <Item id=S50W070_ESACCI-BIOMASS-L4-AGB_SD-MERGED-100m-2010-fv3.0>
3: <Item id=S50W070_ESACCI-BIOMASS-L4-AGB-MERGED-100m-2010-fv3.0>
4: <Item id=S50W060_ESACCI-BIOMASS-L4-AGB_SD-MERGED-100m-2010-fv3.0>
5: <Item id=S50W060_ESACCI-BIOMASS-L4-AGB-MERGED-100m-2010-fv3.0>
6: <Item id=S40W080_ESACCI-BIOMASS-L4-AGB_SD-MERGED-100m-2010-fv3.0>
7: <Item id=S40W080_ESACCI-BIOMASS-L4-AGB-MERGED-100m-2010-fv3.0>
8: <Item id=S40W070_ESACCI-BIOMASS-L4-AGB_SD-MERGED-100m-2010-fv3.0>
9: <Item id=S40W070_ESACCI-BIOMASS-L4-AGB-MERGED-100m-2010-fv3.0>
```

Discover the URL of one item using xarray

[7]:
```
item = collection.get_item('S50W080_ESACCI-BIOMASS-L4-AGB_SD-MERGED-100m-2010-fv3.0')
item.assets['data'].href
```

[7]:
```
'https://bmap-catalogue-data.oss.eu-west-0.prod-cloud-ocb.orange-business.com/esacci/
↪ESACCI_Biomass_L4_AGB_V3_100m/2010/S50W080_ESACCI-BIOMASS-L4-AGB_SD-MERGED-100m-
↪2010-fv3.0.tif'
```

3.2 Visualize

3.2.1 Visualizing Web Map Tile Service (WMTS) Layers

At this time, there are two collections (AFLVIS2 and AfriSAR_UAVSAR_Coreg_SLC) for which we can visualize WMTS layers. WMTS layers can be visualized using the Common Mapping Client (CMC). The CMC is a starter-kit for web-based mapping applications which utilizes several common mapping funtionalities. This example demonstrates how to visualize WMTS layers using `ipyCMC`, a Jupyter Lab widget for the CMC.

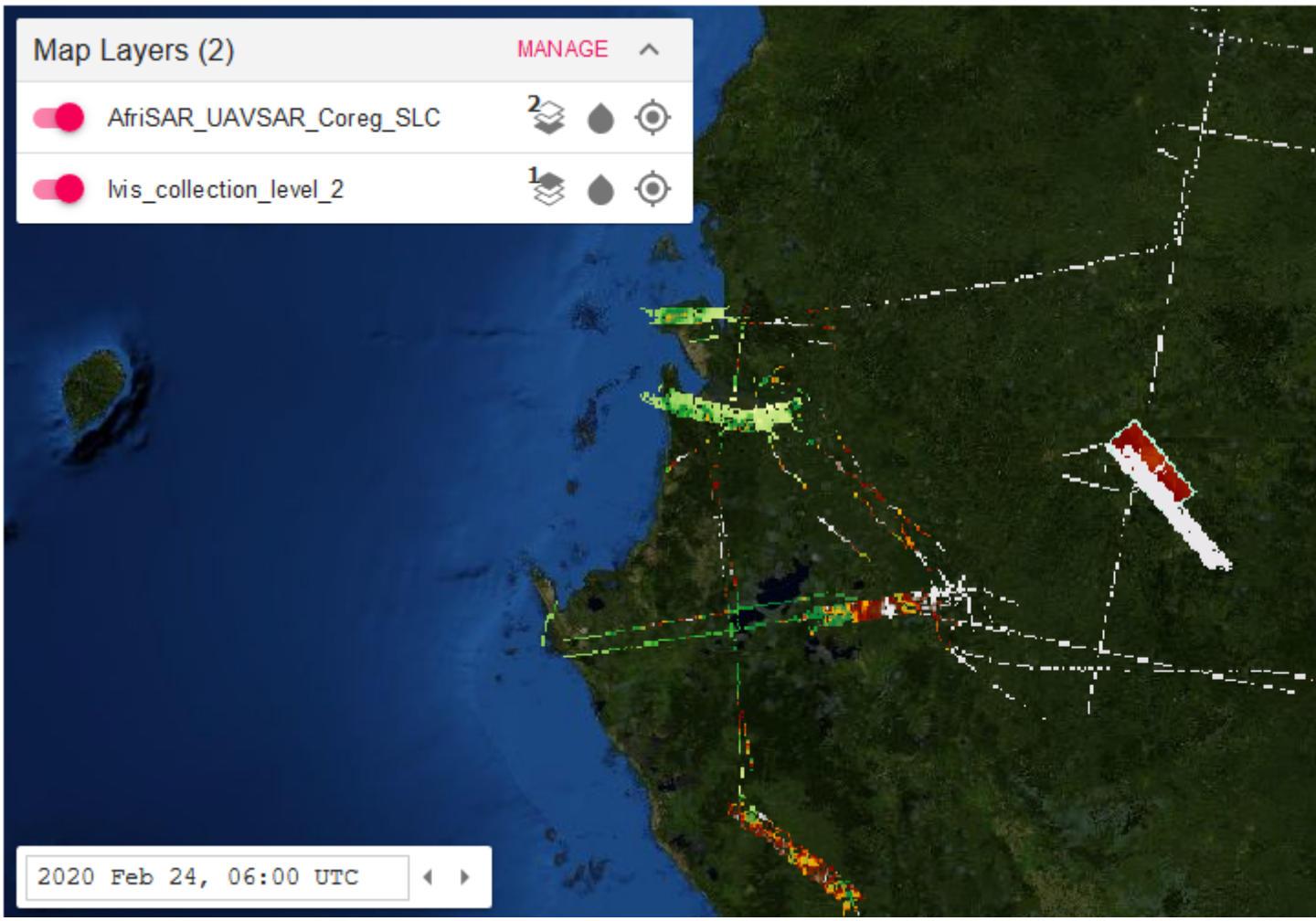
First, we import the `ipycmc` package.

[1]:
```
# Import the ipycmc module
import ipycmc
```

We utilize the CMC widget to visualize data in the MAAP Algorithm Development Environment (ADE).

[2]:
```
# utilize the CMC widget
w = ipycmc.MapCMC()
w
```

```
MapCMC()
```



The CMC user interface provides a variety of tools for analyzing data. In the top left corner, the Map Layers drop-down menu allows us to turn map layers on and off and provides tools for managing layer positioning, setting layer opacity, and zooming to a layer. At the lower right end of the display are tools for displaying the data, adjusting the zoom level, and selecting data. The 'Switch to 3D map' button allows us to switch between 2D and 3D maps. The 'Home' button zooms to the global extent and the 'Zoom In' and 'Zoom Out' buttons zoom the data to appear nearer or farther away. The 'Select Area' button allows us to select an area using a variety of geometrical methods, the 'Select Basemap' button allows us to select from a list of available basemaps, and the 'Fullscreen' button toggles full screen mode. At the bottom right, the cursor location on the map is displayed in Latitude/Longitude decimal degrees. At the bottom left of the user interface, there is a text box allowing us to change the date and time and arrows to change the day.

Visualizing Collections

To visualize the available collections, we utilize the `load_layer_config()` function, handling the url as a WMTS xml file.

[3]:
```
# visualize the available set of layers
w.load_layer_config("https://api.ops.maap-project.org/api/wmts/GetCapabilities",
↪"wmts/xml")
```

Once the layers load, the number displayed in the Map Layers drop-down menu should increase. Pressing the downward arrow on the drop-down menu displays the loaded layers. Turn the layers on and off by pressing the toggle to the

left of the layer names and zoom to a layer by pressing the crosshair button.

Visualizing Single Granules

We can also visualize a single granule which is within one of the available collections by using the granule UR with the `load_layer_config()` function and handling the url as a WMTS file.

[4]:
```
# visualize a single granule using the granule UR
w.load_layer_config("https://api.ops.maap-project.org/api/wmts/GetCapabilities?short_
↪name=AFLVIS2&granule_ur=SC:AFLVIS2.001:138348905", "wmts/xml")
```

This example here shows how we can add a layer by providing a granule UR. These granule URs can be extracted from the metadata by searching using CMR and the MAAP API. Please see the search granule example for specifics on how to search for granules and extract the UR - https://maap-project.readthedocs.io/en/latest/search/granules.html.

3.2.2 Visualizing SRTM dataset from MAAP CMR STAC using MosaicJSON

In this notebook, we discover SRTM COGs fromMAAP's CMR STAC API Proxy**and generate a mosaic.**

MosaicJSON

A common challenge in visualizing spatial data is data is stored across many files representing small spatial extents.

MosaicJSON is a specification created by DevelopmentSeed which aims to be an open standard for representing metadata about a spatial mosaic of many COG files.

> MosaicJSON can be seen as a cloud friendly virtual raster (see GDAL's VRT) enabling spatial and temporal indexing for a list of Cloud-Optimized GeoTIFF.

Ref: https://github.com/developmentseed/mosaicjson-spec

Data

The NASA Shuttle Radar Topographic Mission (SRTM) has provided digital elevation data (DEMs) for over 80% of the globe. This data is currently distributed free of charge by USGS and is available for download from the National Map Seamless Data Distribution System, or the USGS FTP site.

At MAAP, we've converted this elevation data into Cloud-Optimized GeoTIFFs (COGs) so they can be efficiently queried and visualized. These COGs are available in the MAAP CMR.

Titiler Endpoint

By default, TiTiler has a complete `mosaicjson` endpoint. For this demo we are going to use an instance of TiTiler deployed by MAAP at https://titiler.maap-project.org

Requirements

To be able to run this notebook you'll need the following requirements:

- rasterio
- folium

- requests
- tqdm
- rio-tiler (2.0b8) (Optional)
- cogeo-mosaic (Optional)

```
pip install rasterio folium requests tqdm
pip install rio-tiler cogeo-mosaic --pre
```

Get the Data

[1]:
```
import requests
from pprint import pprint

from rio_tiler.io import COGReader
from rasterio.features import bounds as featureBounds

from folium import Map, TileLayer, GeoJson
```

Fetch SRTM COG STAC items

[2]:
```
stac_endpoint = "https://cmr-stac.dit.maap-project.org/stac/NASA_MAAP/search"

stac_json = {
    "collections": ["SRTMGL1_COD.v001"],
    "bbox": "4,42,16,48",
    "limit": 120
}

headers = {
            "Content-Type": "application/json",
            "Accept": "application/geo+json, application/json",
        }

# Read Items

r_stac = requests.post(stac_endpoint, headers=headers, json=stac_json)
items = r_stac.json()
```

Map the data bounds

[3]:
```
geojson = {'type': 'FeatureCollection', 'features': items['features']}

bounds = featureBounds(geojson)
zoom_start = 5

m = Map(
    tiles="OpenStreetMap",
    location=((bounds[1] + bounds[3]) / 2,(bounds[0] + bounds[2]) / 2),
    zoom_start=zoom_start
)
```

(continues on next page)

(continued from previous page)

```
geo_json = GeoJson(
    data=geojson,
    style_function=lambda x: {
        'opacity': 1, 'dashArray': '1', 'fillOpacity': 0, 'weight': 1
    },
)
geo_json.add_to(m)
m
```

[3]:
```
<folium.folium.Map at 0x7fb86b15aed0>
```

2. Create Mosaic

We're using the TiTiler deployed by MAAP

[4]:
```
titiler_endpoint = "https://titiler.maap-project.org"  # MAAP titiler endpoint
```

[5]:
```
%time
from rio_tiler.io import COGReader

first_cog = geojson['features'][0]['assets']['browse']['href']
with COGReader(first_cog) as cog:
    info = cog.info()
```

```
CPU times: user 4 µs, sys: 0 ns, total: 4 µs
Wall time: 7.63 µs
```

[6]:
```
from cogeo_mosaic.mosaic import MosaicJSON

# We are creating the mosaicJSON using the geojson
# SRTMGL1 CODs have a "browse" asset type, so we can create a mosaic of these type=
↪"image/tiff" assets
# accesor function provide access to those
mosaicdata = MosaicJSON.from_features(geojson.get('features'), minzoom=6,␣
↪maxzoom=info.maxzoom, accessor=lambda feature : feature['assets']['browse']['href'])
```

[7]:
```
# Uploading the mosaicjson to the TiTiler
r = requests.post(
    url=f"{titiler_endpoint}/mosaics",
    headers={
        "Content-Type": "application/vnd.titiler.mosaicjson+json",
    },
    json=mosaicdata.dict(exclude_none=True)).json()

pprint(r)
```

```
{'id': '7e9554c7-a41d-407d-a835-d6e6add92c40',
 'links': [{'href': 'https://titiler.maap-project.org/mosaics/7e9554c7-a41d-407d-a835-
↪d6e6add92c40',
            'rel': 'self',
            'title': 'Self',
            'type': 'application/json'},
           {'href': 'https://titiler.maap-project.org/mosaics/7e9554c7-a41d-407d-a835-
↪d6e6add92c40/mosaicjson',
```

(continues on next page)

(continued from previous page)

```
            'rel': 'mosaicjson',
            'title': 'MosaicJSON',
            'type': 'application/json'},
           {'href': 'https://titiler.maap-project.org/mosaics/7e9554c7-a41d-407d-a835-
↪d6e6add92c40/tilejson.json',
            'rel': 'tilejson',
            'title': 'TileJSON',
            'type': 'application/json'},
           {'href': 'https://titiler.maap-project.org/mosaics/7e9554c7-a41d-407d-a835-
↪d6e6add92c40/tiles/{z}/{x}/{y}',
            'rel': 'tiles',
            'title': 'Tiles',
            'type': 'application/json'},
           {'href': 'https://titiler.maap-project.org/mosaics/7e9554c7-a41d-407d-a835-
↪d6e6add92c40/WMTSCapabilities.xml',
            'rel': 'wmts',
            'title': 'WMTS',
            'type': 'application/json'}]}
```

The response from the post request gives endpoints for different services (eg. mosaicjson, tilejson, tiles, wmts, etc). We're fetching the `tilejson` endpoint.

[8]:
```
tilejson_endpoint = list(filter(lambda x: x.get("rel") == "tilejson", dict(r)["links
↪"]))
```

NOTE: You have to zoom to "minzoom" to load the data.

SECOND NOTE: The important bit is the "tiles" endpoint returned from `f"{titiler_endpoint}/mosaics`. This endpoint (e.g. `https://titiler.maap-project.org/mosaics/4199126b-9313-435a-b4b5-17802716b7b1/tiles/{z}/{x}/{y}`) could be used in any map client which can tile `x,y,z` layers.

[9]:
```
r_te = requests.get(
    tilejson_endpoint[0].get('href')
).json()

pprint(r_te)

tiles = TileLayer(
    tiles=f"{r_te['tiles'][0]}?rescale=0,4000",
    min_zoom=r_te["minzoom"],
    max_zoom=r_te["maxzoom"],
    opacity=1,
    attr="USGS"
)

tiles.add_to(m)
m
```

```
{'bounds': [2.9997222, 40.9997222, 17.0002778, 49.0002778],
 'center': [10.0, 45.0, 6],
 'maxzoom': 12,
 'minzoom': 6,
 'name': '7e9554c7-a41d-407d-a835-d6e6add92c40',
 'scheme': 'xyz',
 'tilejson': '2.2.0',
 'tiles': ['https://titiler.maap-project.org/mosaics/7e9554c7-a41d-407d-a835-
↪d6e6add92c40/tiles/{z}/{x}/{y}@1x'],
```

(continues on next page)

(continued from previous page)

```
 'version': '1.0.0'}
```

[9]:
```
<folium.folium.Map at 0x7fb86b15aed0>
```

[1]:
```
#!pip install folium branca
```

[2]:
```
import branca
from folium import Map, TileLayer
import json
from matplotlib import cm
import requests
```

3.2.3 Discover data from STAC

[3]:
```
stac_endpoint = "https://stac.maap-project.org"
titiler_endpoint = "https://titiler.ops.maap-project.org"
collection = "AfriSAR_AGB_Maps_1681"
items_response = requests.get(f"{stac_endpoint}/collections/{collection}/items").
↪json()
url = items_response['features'][0]['assets']['data']['href']
url
```

[3]:
```
's3://nasa-maap-data-store/file-staging/nasa-map/AfriSAR_AGB_Maps_1681___1/Rabi_AGB_
↪50m.tif'
```

3.2.4 Get data values from the `/statistics` endpoint

[4]:
```
# You can use gdalinfo /vsis3/nasa-maap-data-store/file-staging/nasa-map/SRTMGL1_COD__
↪_001/N21W089.SRTMGL1.tif -stats
# or you can get metadata from titiler.
stats_response = requests.get(
    f"{titiler_endpoint}/cog/statistics",
    params = {
        "url": url
    }
).json()
```

[5]:
```
bins = stats_response['b1']['histogram'][1]
bin_ranges = [[bins[i], bins[i+1]] for i in range(len(bins)-1)]
bin_ranges
```

[5]:
```
[[0.08733899891376495, 51.720603942871094],
 [51.720603942871094, 103.35386657714844],
 [103.35386657714844, 154.9871368408203],
 [154.9871368408203, 206.62039184570312],
 [206.62039184570312, 258.253662109375],
 [258.253662109375, 309.8869323730469],
 [309.8869323730469, 361.52020263671875],
 [361.52020263671875, 413.1534423828125],
 [413.1534423828125, 464.7867126464844],
 [464.7867126464844, 516.4199829101562]]
```

3.2.5 Pick a color map and create a linear mapping

[6]:
```
# There are many pre-defined colormaps supported by matplotlib.
# Some are listed below but a complete list be found here: https://matplotlib.org/3.1.
↪0/tutorials/colors/colormaps.html
# You may define custom color maps, but using the predefined ones makes life easier.
sequential_cmaps = [
    'Greys', 'Purples', 'Blues', 'Greens', 'Oranges', 'Reds',
    'YlOrBr', 'YlOrRd', 'OrRd', 'PuRd', 'RdPu', 'BuPu',
    'GnBu', 'PuBu', 'YlGnBu', 'PuBuGn', 'BuGn', 'YlGn'
]

sequential_cmaps2 = [
    'binary', 'gist_yarg', 'gist_gray', 'gray', 'bone', 'pink',
    'spring', 'summer', 'autumn', 'winter', 'cool', 'Wistia',
    'hot', 'afmhot', 'gist_heat', 'copper']
```

[7]:
```
# Here we create a list of pairs, each pair containing a data value interval range␣
↪(aka "bin")
# with a color value in RGBA (see https://www.w3schools.com/css/css3_colors.asp)
# First we create a list of colors
rgbas = [[int(value) for value in rgb] for rgb in cm.ScalarMappable(cmap='gist_earth
↪').to_rgba(x=bins[:-1], bytes=True)]
# Then we use python's zip function to pair rgba values with data values (https://www.
↪w3schools.com/python/ref_func_zip.asp)
color_map = list(zip(bin_ranges, rgbas))
# some tweaking may be necessary
color_map
```

[7]:
```
[([0.08733899891376495, 51.720603942871094], [0, 0, 0, 255]),
 ([51.720603942871094, 103.35386657714844], [18, 48, 119, 255]),
 ([103.35386657714844, 154.9871368408203], [37, 102, 124, 255]),
 ([154.9871368408203, 206.62039184570312], [54, 135, 111, 255]),
 ([206.62039184570312, 258.253662109375], [67, 151, 77, 255]),
 ([258.253662109375, 309.8869323730469], [123, 167, 82, 255]),
 ([309.8869323730469, 361.52020263671875], [169, 179, 91, 255]),
 ([361.52020263671875, 413.1534423828125], [191, 164, 100, 255]),
 ([413.1534423828125, 464.7867126464844], [221, 186, 167, 255]),
 ([464.7867126464844, 516.4199829101562], [253, 250, 250, 255])]
```

[8]:
```
# Let's also create a legend using the RGBA values and bins so our map visualization␣
↪can be interpreted!
legend = branca.colormap.StepColormap(rgbas, index=bins, vmin=round(bins[0], 2),␣
↪vmax=round(bins[-1], 2))
```

3.2.6 Preview the data

[9]:
```
# Create a json string of the colormap, so it can be passed as a parameter to titiler
↪'s API.
cmap = json.dumps(color_map)

# We fetch tilejson from titiler endpoint, to build a better map with appropriate␣
↪bounds and zoom level
tilejson_response = requests.get(
    f"{titiler_endpoint}/cog/tilejson.json",
```

(continues on next page)

(continued from previous page)

```
    params = {
        "url": url,
        "colormap": cmap
    }
).json()

bounds = tilejson_response["bounds"]
m = Map(
    location=((bounds[1] + bounds[3]) / 2,(bounds[0] + bounds[2]) / 2),
    zoom_start=tilejson_response["minzoom"] + 1
)

# We add a TileLayer using the tilejson_response "tiles" value which is the XYZ␣
↪endpoint of titiler.
aod_layer = TileLayer(
    tiles=tilejson_response["tiles"][0],
    opacity=1,
    attr="Swisstopo"
)
aod_layer.add_to(m)

# Finally, we add the legend.
legend.caption = 'Estimated aboveground biomass in Mg/ha (megagram per hectare)'
legend.add_to(m)

m
```

[9]:
```
<folium.folium.Map at 0x7ff922a031d0>
```

3.2.7 Algorithm Development Environment (ADE) Visualization Example

MosaicJSON

A common challenge in visualizing spatial data is data is stored across many files representing small spatial extents.

MosaicJSON is a specification created by DevelopmentSeed which aims to be an open standard for representing metadata about a spatial mosaic of many COG files.

> MosaicJSON can be seen as a cloud friendly virtual raster (see GDAL's VRT) enabling spatial and temporal indexing for a list of Cloud-Optimized GeoTIFF.

Ref: https://github.com/developmentseed/mosaicjson-spec

Visualizing COGs generated in the MAAP Algorithm Development Environment (ADE)

This notebook visualizes COGs generated in the ADE using the python Common Mapping Client**, an open source project of NASA and JPL.**

Any Cloud Optimized GeoTIFF (COG), or group of COGs, in an ADE workspace can be visualized in a dynamic map by using a tiling service hosted in MAAP.

Steps: 1. Make a list of TIFFs in your workspace to use as a single layer 2. Generate a MosaicJSON file from this list of files (or a GeoJSON index) 3. Combine the MosaicJSON with other tiler visualization parameters to register a layer with your visualization tool.

[1]:
```
import glob
import ipycmc
import os
from pprint import pprint
import requests
import urllib

#!pip install cogeo_mosaic
from cogeo_mosaic.mosaic import MosaicJSON
from cogeo_mosaic.backends import MosaicBackend
```

Build a list of files

You can either make a list of file paths, or create a GeoJSON layer with a column containing the file paths. The paths need to be **S3** paths currently.

[2]:
```
# Local Path to your COGs
dps_output = "/projects/shared-buckets/alexdevseed/landsat8/viz/"

# titiler endpoint
titiler_endpoint = f"https://titiler.maap-project.org"

# Search for files to include, use recursive if nested folders (common in DPS output)
files = glob.glob(os.path.join(dps_output, "Landsat*_dps.tif"), recursive=False)

def local_to_s3(url):
    ''' A Function to convert local paths to s3 urls'''
    return url.replace("/projects/shared-buckets", "s3://maap-ops-workspace/shared")

tiles = [local_to_s3(file) for file in files]
print(tiles)
```

```
['s3://maap-ops-workspace/shared/alexdevseed/landsat8/viz/Landsat8_30542_comp_cog_
↪2015-2020_dps.tif', 's3://maap-ops-workspace/shared/alexdevseed/landsat8/viz/
↪Landsat8_30543_comp_cog_2015-2020_dps.tif', 's3://maap-ops-workspace/shared/
↪alexdevseed/landsat8/viz/Landsat8_30822_comp_cog_2015-2020_dps.tif', 's3://maap-ops-
↪workspace/shared/alexdevseed/landsat8/viz/Landsat8_30823_comp_cog_2015-2020_dps.tif
↪']
```

How to find the S3 path

You might be wondering how to find the S3 path to begin with. Right now the easiest way is to right click on a file in the file explorer on the left panel, and **Get Presigned S3 Url**.

It will look something like `https://maap-ops-workspace.s3.amazonaws.com/shared/alexdevseed/landsat8/viz/Copernicus_30438_covars_cog_topo_stack.tif?AWSAccessKeyId...`

The first part of the url is the bucket name: `maap-ops-workspace`. After the next `/`, it then matches to the local path.

Future versions of MAAP will include functions to do this part for you...

Make a mosaic

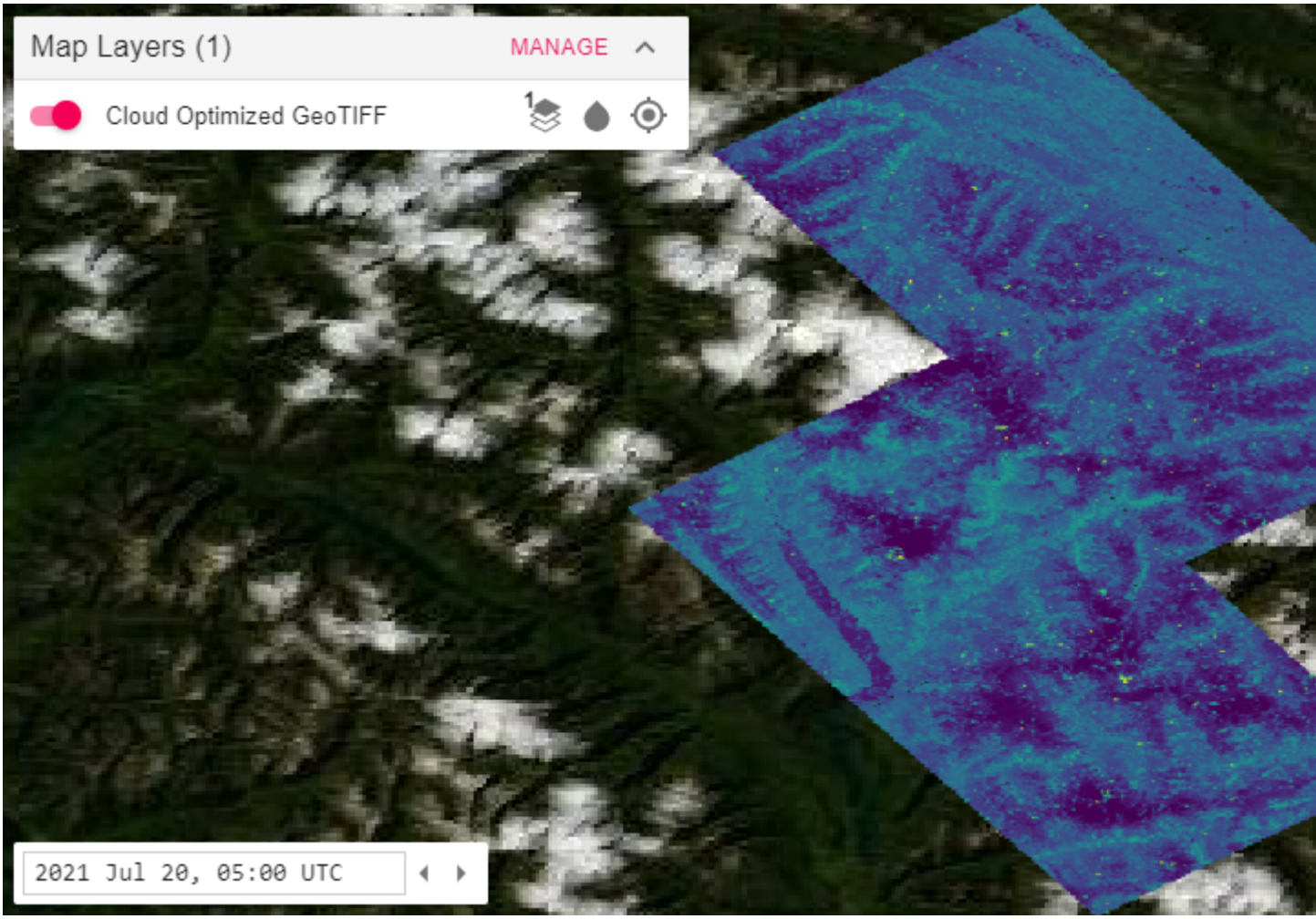
[3]:
```
# Now take the list of S3 paths and generate a mosaic
# TODO: if you have a lot of files (more than 100), creating a GeoJSON index and␣
↪using from_features will be more efficient.

mosaicdata = MosaicJSON.from_urls(tiles, minzoom=9, maxzoom=16)
```

[4]:
```
r = requests.post(
    url=f"{titiler_endpoint}/mosaics",
    headers={
        "Content-Type": "application/vnd.titiler.mosaicjson+json",
    },
    json=mosaicdata.dict(exclude_none=True)).json()

mosaicid = r['id']
```

Make a Map

[5]:
```
w = ipycmc.MapCMC()
w
```

```
MapCMC()
```



[6]:
```
# Build a WMTS call
"""
All of this is subject to change in a future version
The important parameters for users:
  url : the S3 path to the MosaicJSON file,
  bidx (band number),
  rescale (required if using non Byte data type),
  colormap_name or colormap

Other parameters are possible, see https://titiler.maap-project.org/docs#/MosaicJSON/
↪wmts_mosaicjson_WMTSCapabilities_xml_get
"""

wmts_url = f"https://titiler.ops.maap-project.org/mosaics/{mosaicid}/WMTSCapabilities.
↪xml"
params = {
    "tile_format":"png",
    "tile_scale":"1",
    "pixel_selection":"first",
    "TileMatrixSetId":"WebMercatorQuad",
    "bidx":"6", # Select which band to use
    "resampling_method":"nearest",
    "rescale":"0,1", # Values in data are from 0 to 1
```

(continues on next page)

(continued from previous page)

```
    "return_mask":"true",
    "colormap_name":"viridis" # Any colormap from matplotlib will work
}

wmts_call = "?".join([wmts_url, urllib.parse.urlencode(params)])

# Note Jupyter bug add amp; incorrectly when printing the url
wmts_call
```

[6]:
```
'https://titiler.ops.maap-project.org/mosaics/e613b570-7318-4d88-92c4-1c5373c6b513/
↪WMTSCapabilities.xml?tile_format=png&tile_scale=1&pixel_selection=first&
↪TileMatrixSetId=WebMercatorQuad&bidx=6&resampling_method=nearest&rescale=0%2C1&
↪return_mask=true&colormap_name=viridis'
```

[7]:
```
# This adds a new layer to the map above, call Cloud Optimized GeoTIFF
w.load_layer_config(wmts_call, "wmts/xml")
```

3.3 Access

3.3.1 Accessing Data from the MAAP

In this example, we demonstrate how to access data from the MAAP using the `getLocalPath()` function. At this time, this procedure is the same for user-contributed data added to the store.

We import the `os` module, import the `MAAP` package, and create a new MAAP class.

[1]:
```
# import os module
import os

# import the MAAP package
from maap.maap import MAAP

# create MAAP class
maap = MAAP()
```

For this example, the `site_name` additional attribute is used to search for granules that have been tagged as part of the *Mondah Forest Gabon* research site. For more information about searching for granules in MAAP, please see https://maap-project.readthedocs.io/en/latest/search/granules.html.

[2]:
```
# assign Mondah Forest Gabon site name
SHORTNAME = "AFLVIS2"
SITENAME = 'Mondah Forest Gabon'

# search for granules with site name
results = maap.searchGranule(short_name=SHORTNAME,site_name=SITENAME)
```

We assign a variable (in this case, `data_file`) to the first result of our search from the cell above.

[3]:
```
# grab first result
data_file = results[0]
```

A data directory is then set, and if the directory does not already exist, it is created. The file from our search is then downloaded into the file system in this directory. Here, the function `getLocalPath()` is accessing the data stored

on the MAAP's Simple Storage Service (S3) bucket and downloading it directly to the path provided.

[4]:
```
# set data directory
dataDir = './data'

# check if directory exists -> if directory doesn't exist, directory is created
if not os.path.exists(dataDir):
    os.mkdir(dataDir)

# download file from search into data directory
data = data_file.getLocalPath(dataDir)
data
```

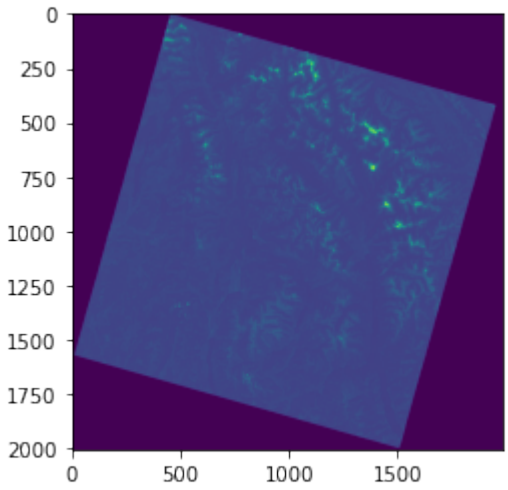
[4]:
```
'./data/LVIS2_Gabon2016_0308_R1808_038002.TXT'
```

Now we can see that the data directory has been created and the 'LVIS2_Gabon2016_0308_R1808_038002.TXT' file is downloaded into the directory. The downloaded file remains in the data directory until the user deletes it.

3.3.2 Accessing Data from AWS Requester Pays Buckets

Some data is cloud available but in requester pays buckets. In this example, we use Rasterio, Boto3, and MAAP's `aws.requester_pays_credentials()` function to retrieve data within the `usgs-landsat` requester pays bucket.

[5]:
```
import boto3
import rasterio as rio

from maap.maap import MAAP
from rasterio.plot import show
from rasterio.session import AWSSession


maap = MAAP(maap_host='api.ops.maap-project.org')
credentials = maap.aws.requester_pays_credentials()

boto3_session = boto3.Session(
    aws_access_key_id=credentials['aws_access_key_id'],
    aws_secret_access_key=credentials['aws_secret_access_key'],
    aws_session_token=credentials['aws_session_token']
)

aws_session = AWSSession(boto3_session, requester_pays=True)
file_s3 = 's3://usgs-landsat/collection02/level-2/standard/oli-tirs/2015/044/025/LC08_
↪L2SP_044025_20150812_20200908_02_T1/LC08_L2SP_044025_20150812_20200908_02_T1_SR_B2.
↪TIF'
with rio.Env(aws_session):
    with rio.open(file_s3, 'r') as src:
        # list of overviews
        oviews = src.overviews(1)
        # get second item from list to retrieve a thumbnail
        oview = oviews[1]
        # read first band of file and set shape of new output array
        thumbnail = src.read(1, out_shape=(1, int(src.height // oview), int(src.width␣
↪// oview)))
# now display the thumbnail
show(thumbnail)
```



[5]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff88c062650>
```

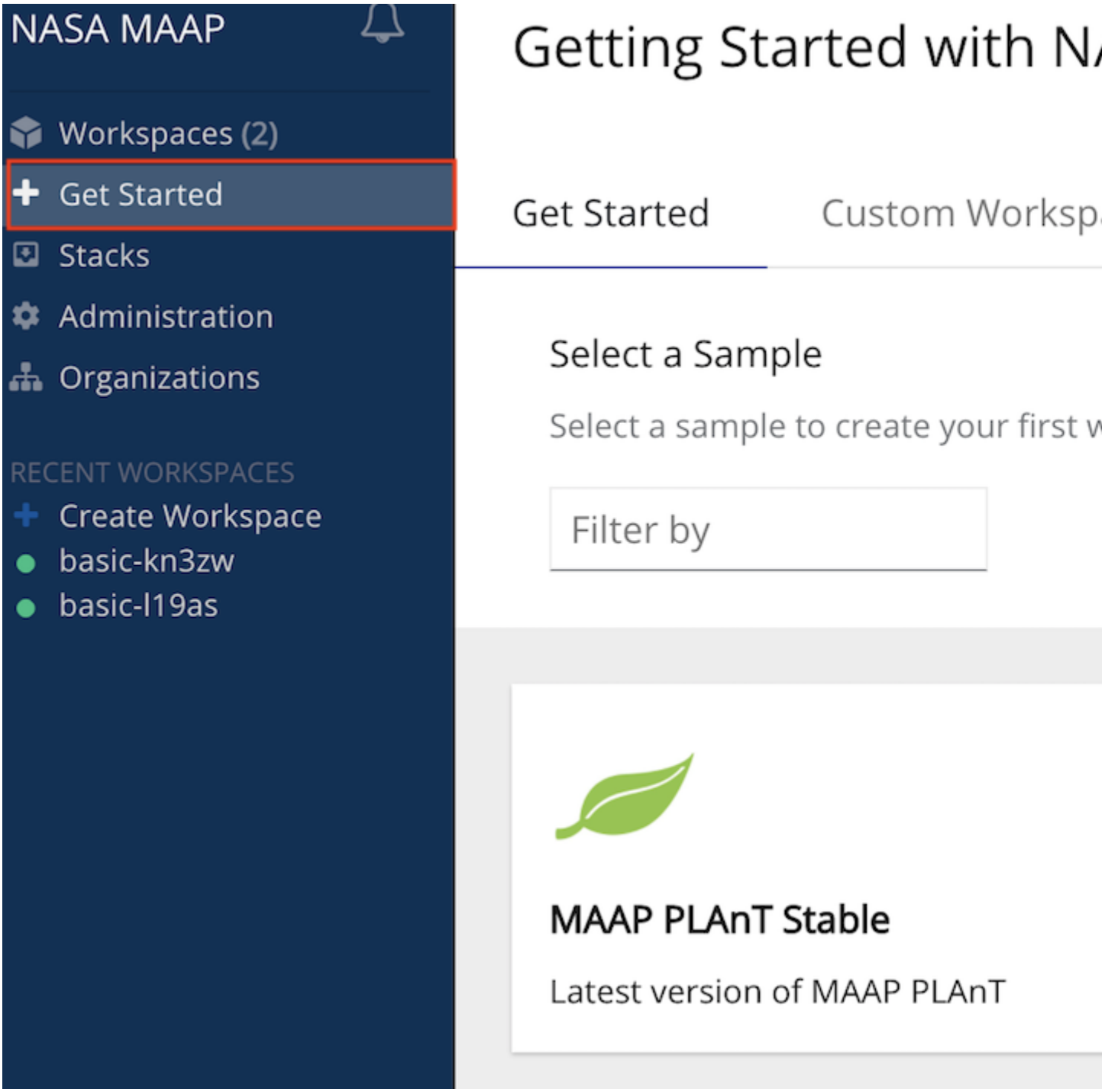
You may adjust the expiration time of the AWS credentials generated by `maap.aws.requester_pays_credentials()`:

[ ]:
```
# Credential expiration time in seconds (defaults to 12 hours)
maap.aws.requester_pays_credentials(expiration=3600)
```

3.3.3 Accessing data provided by NASA's Distributed Active Archive Centers (DAACs)

It is possible to download data provided by DAACs, including data which is not cataloged by the MAAP's CMR, using the NASA MAAP ADE. This data is hosted externally from the MAAP but can be accessed using the NASA MAAP ADE's authentication systems.

In order to do this, we start by creating a Jupyter workspace within the NASA MAAP ADE. Using the left-hand navigation, select "+ Get Started" and then select the "Jupyter - MAAP Basic Stable" workspace.



Alternatively, you can create a workspace using the "Workspaces" interface. See Create Workspace for more information.

Accessing data from Jupyter Notebooks in your workspace

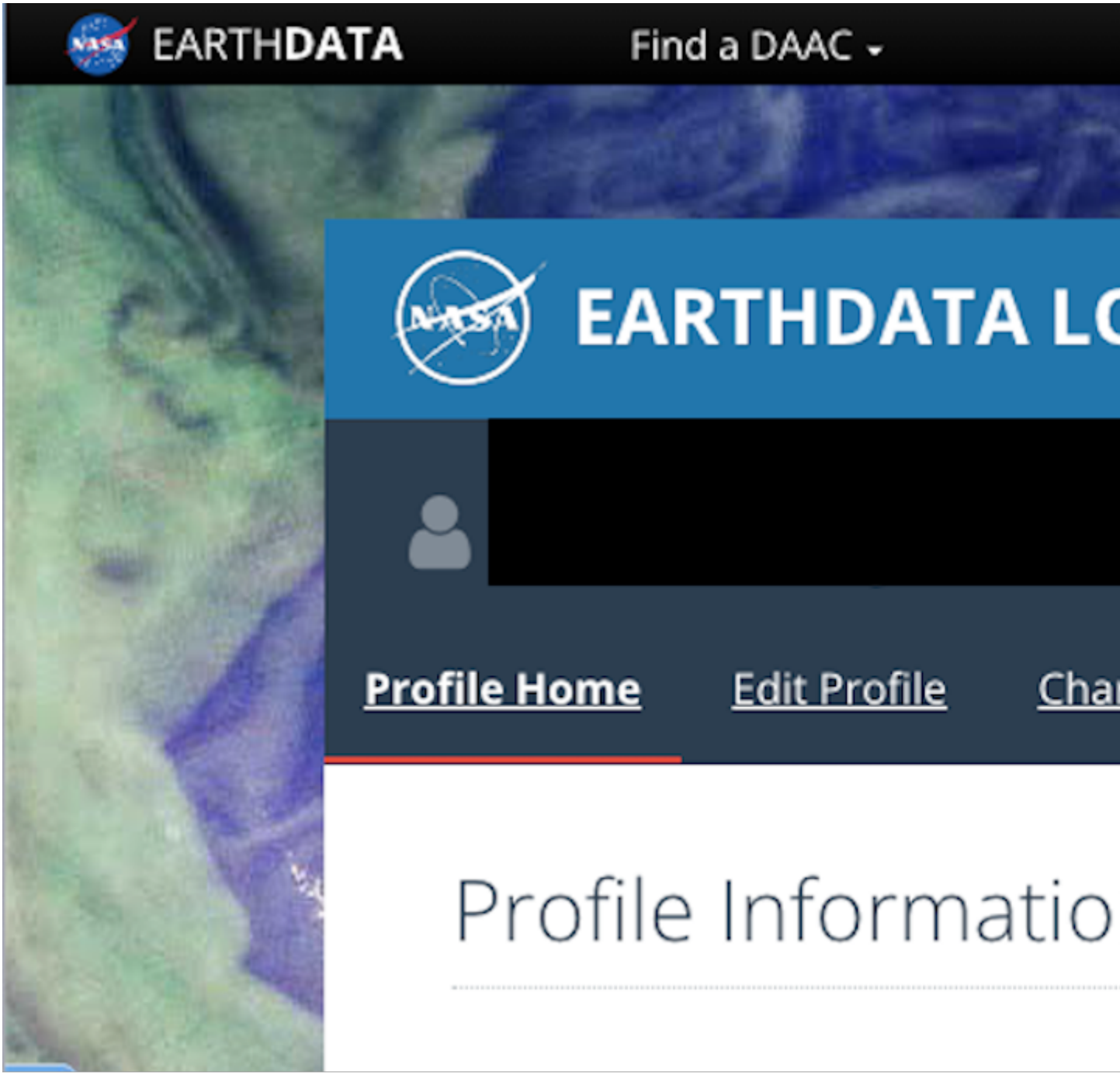
Within your Jupyter Notebook, start by importing the `maap` package. Then invoke the `MAAP` constructor, setting the `maap_host` argument to `'api.ops.maap-project.org'`.

[1]:
```
# import the maap package
from maap.maap import MAAP
# invoke the MAAP constructor using the maap_host argument
maap = MAAP(maap_host='api.ops.maap-project.org')
```

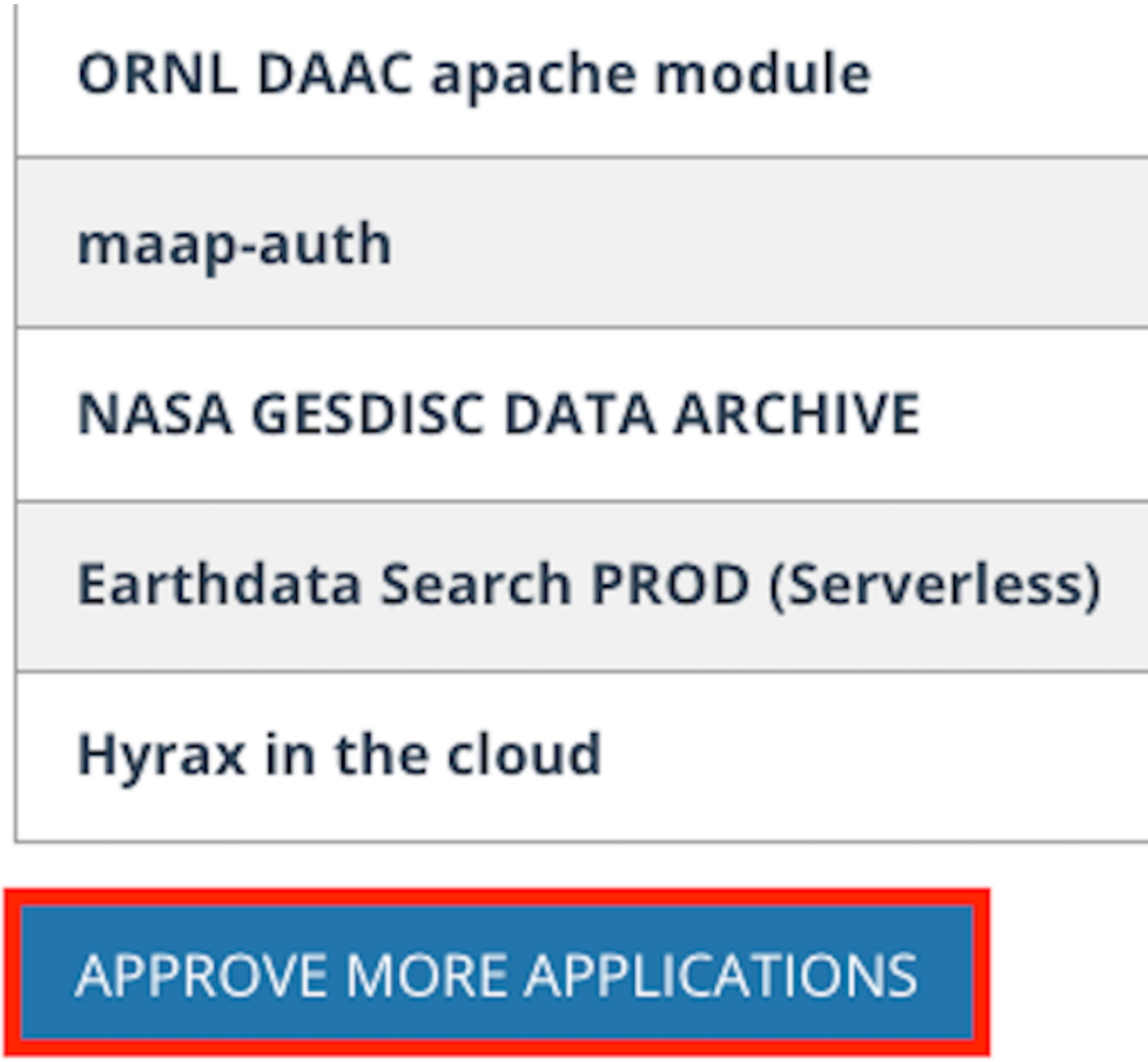
Granting Earthdata Login access to your target DAAC application

In order to access external DAAC data from the NASA MAAP ADE, MAAP uses your Earthdata Login profile to send a data request to the desired DAAC application.

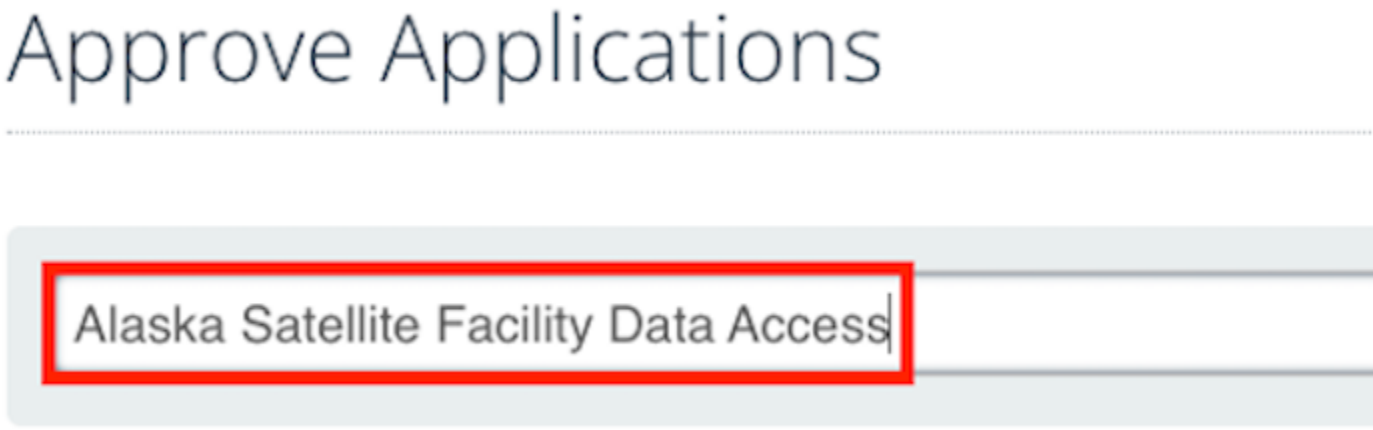
Some DAAC applications (such as 'Alaska Satellite Facility Data Access') must be authorized before you can use them. Login or register at https://urs.earthdata.nasa.gov/ in order to see the applications that you have authorized. From the profile page, click on the 'Applications' tab and select 'Authorized Apps' from the drop-down menu.



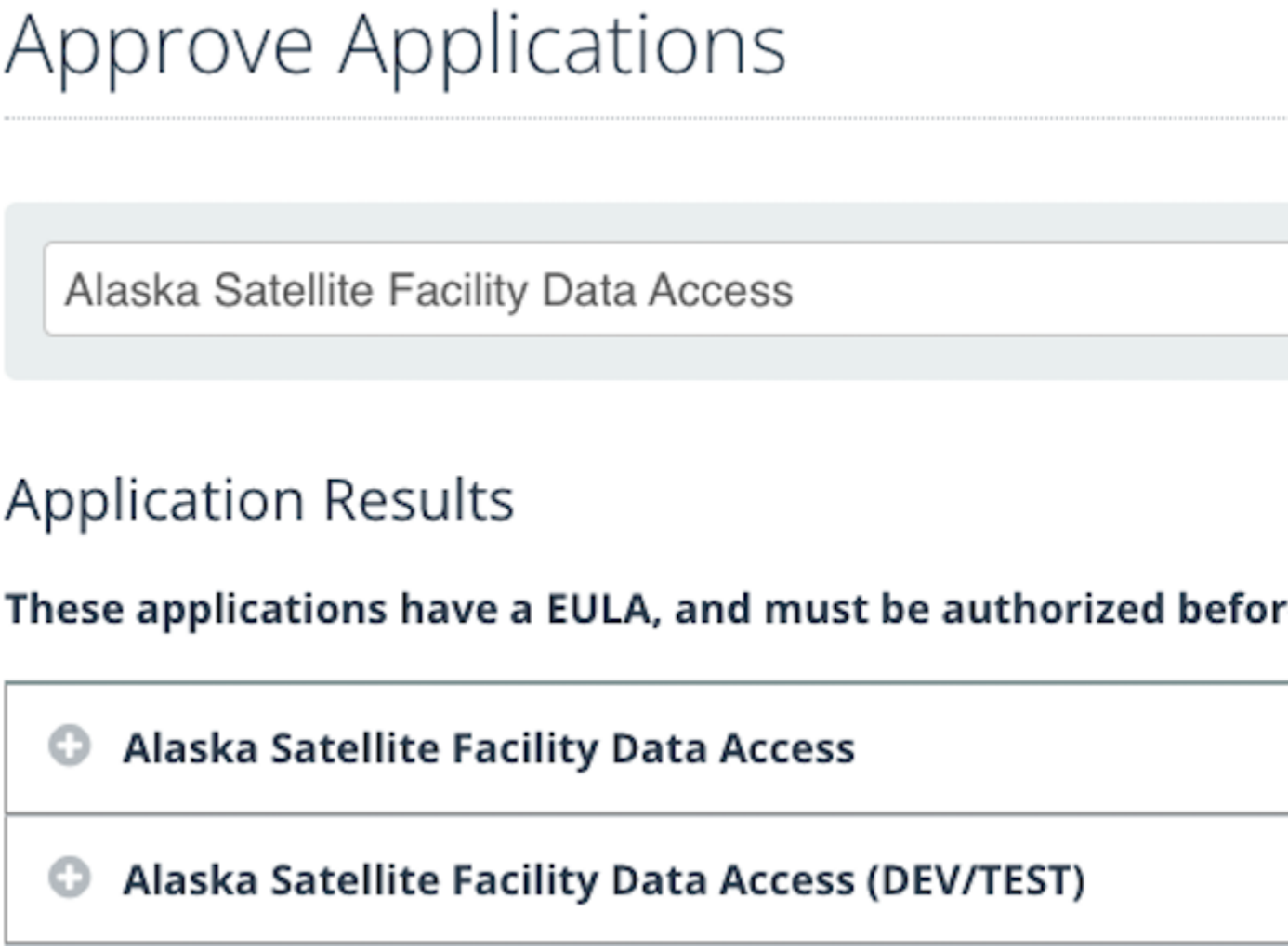
This takes you to the Approved Applications page which lists the applications you have authorized. To add more applications, scroll down to the bottom of the page and click the 'APPROVE MORE APPLICATIONS' button which takes you to the Application search page.



Enter the desired application name within the search box and click the 'SEARCH' button. After this, a list of search results appears.



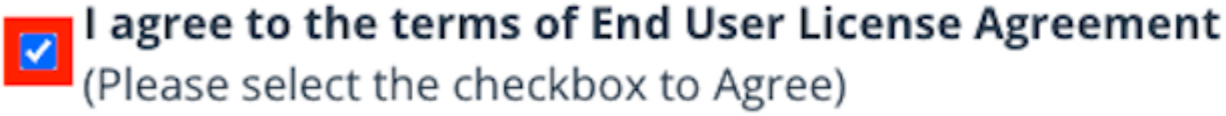
Once you find the desired application, click the 'AUTHORIZE' button next to the name.



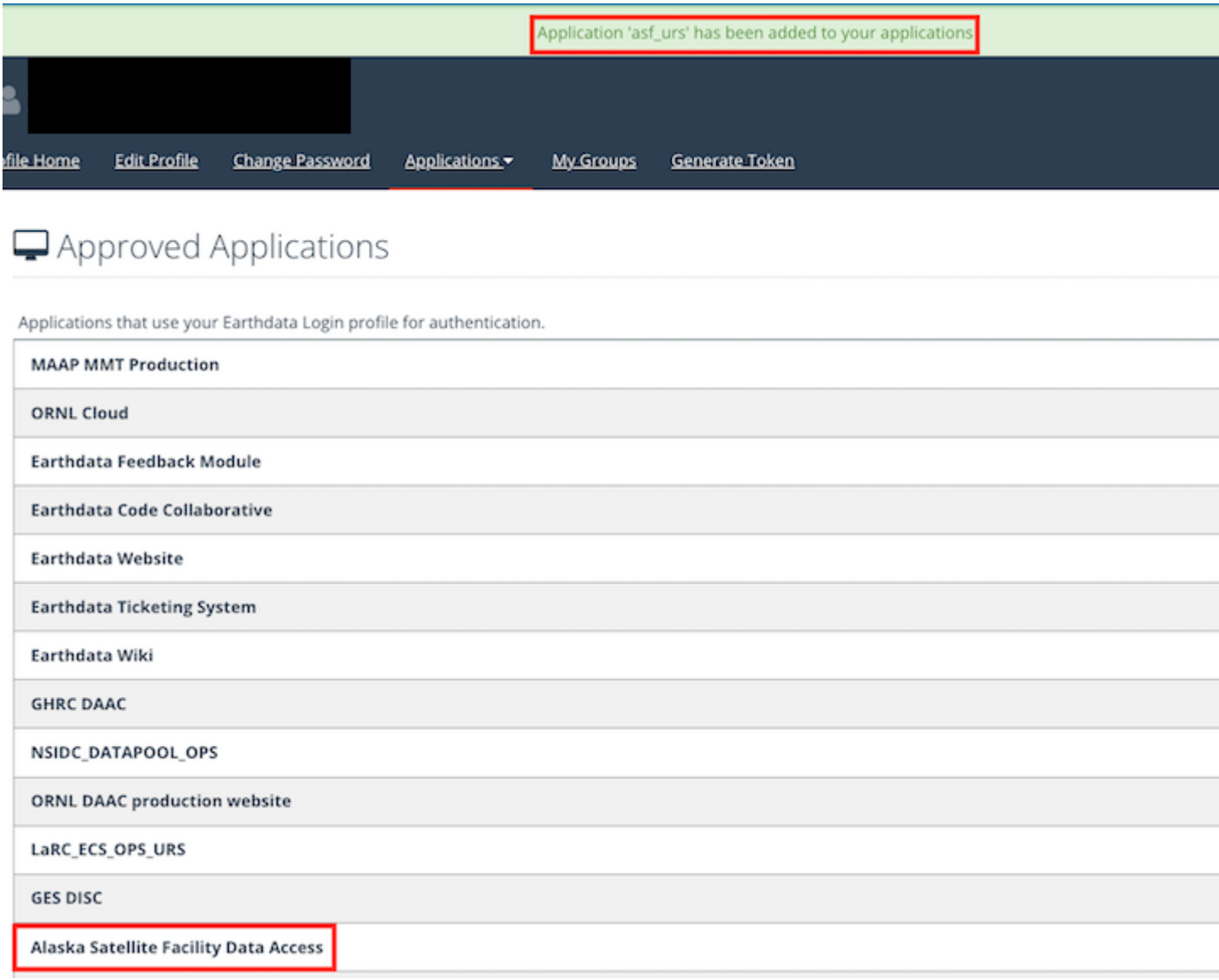
You are then presented with its End User License Agreement. In order to have authorization, you need to select the 'I

agree to the terms of End User License Agreement' checkbox and then click the 'AGREE' button.

```
I agree to clearly mark all ERS-1 and ERS-2 data, i
which it is reproduced, in such a way that the data
are clear to see.

    Data: Dataset: ERS-2[1], ESA [year of data acqu
          ASF DAAC [day/month/year of data access]

    Images: © ESA [year of data acquisition]
```



After this is done, you are then shown the Approved Applications page again and the desired application should now be listed.



Note that if Earthdata Login access is not granted to your target DAAC application, the following example will result in a 401-permission error.

Accessing Sentinel-1 Granule Data from the Alaska Satellite Facility (ASF)

Search for a granule using the `searchGranule` function (for more information on searching for granules, see Searching for Granules in MAAP). Then utilize the `getData` function, which downloads granule data if it doesn't already exist locally. We can use `getData` to download the first result from our granule search into the file system and assign it to a variable (in this case `download`). Note that you will need to authorize the 'Alaska Satellite Facility Data Access' application before downloading any results from our search (see the above section for more information concerning authorizing applications).

[2]:
```
# search for granule data using the short_name argument
results = maap.searchGranule(short_name='SENTINEL-1A_DP_GRD_HIGH')
# download first result
download = results[0].getData()
```

Note that we can then use the `print` function to see the file name and directory.

[3]:
```
# print file directory
print(download)
```

```
./S1A_S3_GRDH_1SDH_20140615T034444_20140615T034512_001055_00107C_8977.zip
```

Accessing Harmonized Landsat Sentinel-2 (HLS) Level 3 Granule Data from the Land Processes Distributed Active Archive Center (LP DAAC)

We use a similar approach in order to access HLS Level 3 granule data. Note that this data is not cataloged by the MAAP's CMR but we can use `searchGranule`'s `cmr_host` argument to specify a CMR instance external to MAAP.

[4]:
```
# search for granule data using CMR host name and short name arguments
results = maap.searchGranule(
    cmr_host='cmr.earthdata.nasa.gov',
    short_name='HLSL30')
# download first result
download = results[0].getData()
```

As in the previous example, we can use the `print` function to see the file name and directory.

[5]:
```
# print file directory
print(download)
```

```
./HLS.L30.T59WPT.2013101T001445.v2.0.B09.tif
```

3.3.4 Accessing Global Ecosystem Dynamics Investigation (GEDI) Level 3 Granule Data

In this example, we demonstrate how to access GEDI Level 3 granule data on the MAAP ADE.

Within your Jupyter Notebook, start by importing the `maap` package. Then invoke the `MAAP` constructor, setting the `maap_host` argument to `'api.ops.maap-project.org'`.

[1]:
```
# import os module
import os
# import the maap package to handle queries
from maap.maap import MAAP
# invoke the MAAP constructor using the maap_host argument
maap = MAAP(maap_host='api.ops.maap-project.org')
```

Search for a granule using the `searchGranule` function (for more information on searching for granules, see Searching for Granules in MAAP). Note that we can use `searchGranule`'s `cmr_host` argument to specify `cmr.maap-project.org` as the CMR instance.

[2]:
```
# search for granule data using CMR host name and collection concept ID arguments
results = maap.searchGranule(
    cmr_host='cmr.maap-project.org',
    collection_concept_id='C1201702030-NASA_MAAP'
)
```

Let's view the list of `GranuleURs` within our `results`:

[3]:
```
# show all of the items
[item['Granule']['GranuleUR'] for item in results]
```

[3]:
```
['GEDI_L3_LandSurface_Metrics_V2.GEDI03_elev_lowestmode_stddev_2019108_2020287_002_02.
↪tif',
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_counts_2019108_2021104_002_02.tif',
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_elev_lowestmode_stddev_2019108_2021104_002_02.
↪tif',
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_elev_lowestmode_mean_2019108_2020287_002_02.
↪tif',
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_elev_lowestmode_mean_2019108_2021104_002_02.
↪tif',
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_rh100_stddev_2019108_2021104_002_02.tif',
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_counts_2019108_2020287_002_02.tif',
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_rh100_mean_2019108_2021104_002_02.tif',
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_rh100_mean_2019108_2020287_002_02.tif',
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_rh100_stddev_2019108_2020287_002_02.tif',
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_counts_2019108_2021216_002_02.tif',
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_rh100_mean_2019108_2021216_002_02.tif',
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_elev_lowestmode_stddev_2019108_2021216_002_02.
↪tif',
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_elev_lowestmode_mean_2019108_2021216_002_02.
↪tif',
 'GEDI_L3_LandSurface_Metrics_V2.GEDI03_rh100_stddev_2019108_2021216_002_02.tif']
```

For this example, we are interested in downloading `GEDI03_elev_lowestmode_stddev_2019108_2021104_002_02.tif`.

[4]:
```
# select item
results[2]['Granule']['GranuleUR']
```

[4]:
```
'GEDI_L3_LandSurface_Metrics_V2.GEDI03_elev_lowestmode_stddev_2019108_2021104_002_02.
↪tif'
```

Now utilize the `getData` function, which downloads granule data if it doesn't already exist locally. We can use `getData` to download the third result from our granule search into the file system and assign its local path to a variable (in this case `download`).

[5]:
```
# download granule item
local_dir = '/projects/local_data'  # download directory (absolute path or relative␣
↪to current directory)
os.makedirs(local_dir, exist_ok=True) # create directories, as necessary
download = results[2].getData(local_dir)  # default download directory is current␣
↪directory, if no directory is given
```

We can then use the `print` function to see the file name and directory.

[6]:
```
# print path to downloaded file
print(download)
```

```
/projects/local_data/GEDI03_elev_lowestmode_stddev_2019108_2021104_002_02.tif
```

3.3.5 Accessing Global Ecosystem Dynamics Investigation (GEDI) Level 4A Granule Data

In this example, we demonstrate how to access GEDI Level 4A granule data on the MAAP ADE.

Within your Jupyter Notebook, start by importing the `maap` package. Then invoke the `MAAP` constructor, setting the `maap_host` argument to `'api.ops.maap-project.org'`.

[1]:
```
# import os module
import os
# import the maap package to handle queries
from maap.maap import MAAP
# invoke the MAAP constructor using the maap_host argument
maap = MAAP(maap_host='api.ops.maap-project.org')
```

Search for a granule using the `searchGranule` function (for more information on searching for granules, see Searching for Granules in MAAP). Note that we can use `searchGranule`'s `cmr_host` argument to specify `cmr.maap-project.org` as the CMR instance. Then utilize the `getData` function, which downloads granule data if it doesn't already exist locally. We can use `getData` to download the first result from our granule search into the file system and assign its local path to a variable (in this case `download`).

[2]:
```
# search for granule data using CMR host name, collection concept ID, and Granule UR␣
↪arguments
results = maap.searchGranule(
    cmr_host='cmr.maap-project.org',
    collection_concept_id='C1202028193-NASA_MAAP',
    granule_ur='GEDI_L4A_AGB_Density_V2_1.GEDI04_A_2019107224731_O01958_01_T02638_02_
↪002_02_V002.h5')
# download first result
local_dir = '/projects/local_data'  # Download directory (absolute path or relative␣
↪to current directory)
os.makedirs(local_dir, exist_ok=True) # Create directories, as necessary
download = results[0].getData(local_dir)  # Default download directory is current␣
↪directory, if no directory is given
```

We can then use the `print` function to see the file name and directory.

[3]:
```
# print path to downloaded file
print(download)
```

```
/projects/local_data/GEDI04_A_2019107224731_O01958_01_T02638_02_002_02_V002.h5
```

3.3.6 Accessing Cloud Optimized Data

The following is an example that uses Shuttle Radar Topography Mission (SRTM) Cloud Optimized GeoTIFF data from the MAAP data store, via MAAP CMR API search. In this example, we read in elevation data using a bounding box tile.

First we install any necessary packages. Please note that the following block of code only needs to be run if needed within a new workspace and that you may need to restart the kernel to use updated packages.

[1]:
```
# only run this block if needed in a new workspace
!pip install -U folium geopandas rasterio>=1.2.3 rio-cogeo
```

```
WARNING: You are using pip version 21.0; however, version 21.1.3 is available.
You should consider upgrading via the '/opt/conda/bin/python3.7 -m pip install --
↪upgrade pip' command.
```

Let's import the `maap` and `pprint` packages and invoke the MAAP.

[2]:
```
# import the maap package to handle queries
from maap.maap import MAAP
```

(continues on next page)

(continued from previous page)

```
# import printing package to help display outputs
from pprint import pprint
# invoke the MAAP
maap = MAAP()
```

We can use the `maap.searchCollection` function to search for the desired collection, in this case the SRTMGL1_COD collection and set the results of the function to a variable. More information about searching for collections may be found here.

[3]:
```
# search for SRTMGL1_COD collection
collection_info = maap.searchCollection(short_name="SRTMGL1_COD", limit=1000)
```

Perhaps we are only interested in granules (files) from the collection which are within a certain area. We create a string with bounding box values and use the `maap.searchGranule` function with the bounding box values. More information about searching for granules may be found here.

[4]:
```
# set bounding box
bbox = '-101.5,45.5,-100.5,46.5'
# retreive granules from the collection that are within the bounding box
granule_results = maap.searchGranule(short_name="SRTMGL1_COD", bounding_box=bbox ,␣
↪limit=20)
```

Let's check how many granules we are working with.

[5]:
```
# show number of granules in results
len(granule_results)
```

[5]:
```
4
```

Inspecting the Results

Now we can work on inspecting our results. In order to do this, we import the `geopandas`,`shapely`, and `folium` packages.

[6]:
```
# import geopandas to work with geospatial data
import geopandas as gpd
# import shapely for manipulation and analysis of geometric objects
import shapely as shp
# import folium to visualize data in an interactive leaflet map
import folium
```

```
/opt/conda/lib/python3.7/site-packages/geopandas/_compat.py:110: UserWarning: The␣
↪Shapely GEOS version (3.8.0-CAPI-1.13.1 ) is incompatible with the GEOS version␣
↪PyGEOS was compiled with (3.8.1-CAPI-1.13.3). Conversions between both will be slow.
  shapely_geos_version, geos_capi_version_string
```

With the above packages imported, we create a function to make polygons from granule items passed to it.

[7]:
```
def make_polygons(item):
    """
    Returns shapely.geometry.polygon.Polygon
    Parameters:
    -----------
    item : dictionary
        A result from granule search results returned by maap.searchGranules()
```

(continues on next page)

(continued from previous page)

```
    """
    # get boundary information from granule result
    bounds = item['Granule']['Spatial']['HorizontalSpatialDomain']['Geometry'][
↪'BoundingRectangle']
    # get boundary values from `bounds` and convert to floating point number
    item_bbox = [float(value) for value in bounds.values()]
    # use boundary values to create a shapely polygon for the function to return
    bbox_polygon = shp.geometry.box(
        item_bbox[0],
        item_bbox[1],
        item_bbox[2],
        item_bbox[3]
    )
    return bbox_polygon
```

With our `make_polygons` function, we can use the `gpd.GeoSeries` function to create a GeoSeries of all the polygons created from our granule results. According to the GeoPandas documentation, a GeoSeries is a "Series [a type of one-dimensional array] object designed to store shapely geometry objects". We use 'EPSG:4326' (WGS 84) for the coordinate reference system. Then we can check the GeoSeries.

[8]:
```
# create GeoSeries of all polygons from granule results with WGS 84 coordinate␣
↪reference system
geometries = gpd.GeoSeries([make_polygons(item) for item in granule_results], crs=
↪'EPSG:4326')
# check GeoSeries
geometries
```

[8]:
```
0    POLYGON ((-100.99972 46.00028, -100.99972 44.9...
1    POLYGON ((-100.99972 47.00028, -100.99972 45.9...
2    POLYGON ((-99.99972 46.00028, -99.99972 44.999...
3    POLYGON ((-99.99972 47.00028, -99.99972 45.999...
dtype: geometry
```

Now we create a list from our bounding box values. Then we use the `centroid` function to get the centroid of our bounding box and set to a point. Next we use `folium.Map` to create a map. For the map's parameters, let's set the centroid point coordinates as the location, "cartodbpositron" for the map tileset, and 7 as the starting zoom level. With our map created, we can create a dictionary containing style information for our bounding box. Then we can use `folium.GeoJson` to create GeoJsons from our `geometries` GeoSeries and add them to the map. We also use the `folium.GeoJson` function to create a GeoJson of a polygon created from our bounding box list, name it, and add our style information. Finally, we check the map by displaying it.

[9]:
```
# create list of bounding box values
bbox_list =  [float(value) for value in bbox.split(',')]
# get centroid point from bounding box values
center = shp.geometry.box(*bbox_list).centroid
# create map with folium with arguments for lat/lon of map, map tileset, and starting␣
↪zoom level
m = folium.Map(
    location=[center.y,center.x],
    tiles="cartodbpositron",
    zoom_start=7,
)
# create style information for bounding box
bbox_style = {'fillColor': '#ff0000', 'color': '#ff0000'}
# create GeoJson of `geometries` and add to map
folium.GeoJson(geometries, name="tiles").add_to(m)
```

(continues on next page)

(continued from previous page)

```
# create GeoJson of `bbox_list` polygon and add to map with specified style
folium.GeoJson(shp.geometry.box(*bbox_list),
               name="bbox",
               style_function=lambda x:bbox_style).add_to(m)
# display map
m
```

[9]:
```
<folium.folium.Map at 0x7f3e4abf5d50>
```

Creating a Mosaic

Let's check the raster information contained in our granule results. In order to do this, we import some more packages. To read and write files in raster format, import `rasterio`. From `rasterio` we import `merge` to copy valid pixels from an input into an output file, `AWSSession` to set up an Amazon Web Services (AWS) session, and `show` to display images and label axes. Also import `boto3` in order to work with AWS. From `matplotlib`, we want to import `imshow` which allows us to display images from data. Import `numpy` to work with multi-dimensional arrays and `numpy.ma` to work with masked arrays. From `pyproj`, import `Proj` for converting between geographic and projected coordinate reference systems and `Transformer` to make transformations.

[10]:
```
# import rasterio for reading and writing in raster format
import rasterio as rio
# copy valid pixels from input files to an output file.
from rasterio.merge import merge
# set up AWS session
from rasterio.session import AWSSession
# display images, label axes
from rasterio.plot import show
# import boto3 to work with Amazon Web Services
import boto3
# display images from data
from matplotlib.pyplot import imshow
# import numpy to work with multi-dimensional arrays
import numpy as np
# import numpy.ma to work with masked arrays
import numpy.ma as ma
# convert between geographic and projected coordinates and make transformations
from pyproj import Proj, Transformer
```

Finally, we import `os` and run some code in order to speed up Geospatial Data Abstraction Library (GDAL) reads from Amazon Simple Storage Service (S3) buckets by skipping sidecar (connected) files.

[11]:
```
# speed up GDAL reads from S3 buckets by skipping sidecar files
import os
os.environ['GDAL_DISABLE_READDIR_ON_OPEN'] = 'EMPTY_DIR'
```

Now that we have the necessary packages, let's get a list of S3 urls to the granules. To start, set up an AWS session. The S3 urls are contained within the `granule_results` list, so we loop through the granules in our results to get the S3 urls and add them to a new list. We can then use the `sort` function to sort the S3 urls in an acending order. Then we can check our S3 url list.

[12]:
```
# set up AWS session
aws_session = AWSSession(boto3.Session())
# get the S3 urls to the granules
file_S3 = [item['Granule']['OnlineAccessURLs']['OnlineAccessURL']['URL'] for item in␣
↪granule_results]
```

(continues on next page)

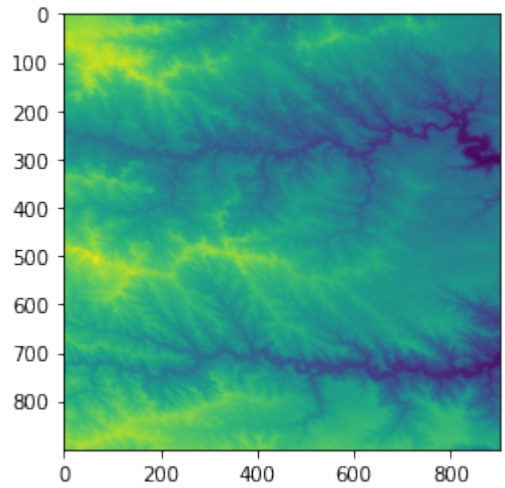
(continued from previous page)

```
# sort list in ascending order
file_S3.sort()
# check list
file_S3
```

[12]:
```
['s3://nasa-maap-data-store/file-staging/nasa-map/SRTMGL1_COD___001/N45W101.SRTMGL1.
↪tif',
 's3://nasa-maap-data-store/file-staging/nasa-map/SRTMGL1_COD___001/N45W102.SRTMGL1.
↪tif',
 's3://nasa-maap-data-store/file-staging/nasa-map/SRTMGL1_COD___001/N46W101.SRTMGL1.
↪tif',
 's3://nasa-maap-data-store/file-staging/nasa-map/SRTMGL1_COD___001/N46W102.SRTMGL1.
↪tif']
```

Looks good. We can now check to see that we can read the AWS files and display a thumbnail. Pass the boto3 session to `rio.Env`, which is the GDAL/AWS environment for `rasterio`. Use the `rio.open` function to read in one of the Cloud Optimized GeoTIFFs. Now let's use the `overviews` command to get a list of overviews. Overviews are versions of the data with lower resolution, and can thus increase performance in applications. Let's get the second overview from our list for retrieving a thumbnail. Retrieve a thumbnail by reading the first band of our file and setting the shape of the new output array. The shape can be set with a tuple of integers containing the number of datasets as well as the `height` and `width` of the file divided by our integer from the overview list. Now use the `show` function to display the thumbnail.

[13]:
```
# prove that we can read the AWS files
# for more information - https://automating-gis-processes.github.io/CSC/notebooks/L5/
↪read-cogs.html
with rio.Env(aws_session):
    with rio.open(file_S3[1], 'r') as src:
        # list of overviews
        oviews = src.overviews(1)
        # get second item from list to retrieve a thumbnail
        oview = oviews[1]
        # read first band of file and set shape of new output array
        thumbnail = src.read(1, out_shape=(1, int(src.height // oview), int(src.width␣
↪// oview)))
# now display the thumbnail
show(thumbnail)
```



[13]:
```
<AxesSubplot:>
```

Since we verified that we can read the AWS files and display a thumbnail, we can create a mosaic from all of the rasters in our `file_S3` list. To do this, again pass the boto3 session to `rio.Env`. Then create a list which contains all of the read in Cloud Optimized GeoTIFFs (this may take a while).

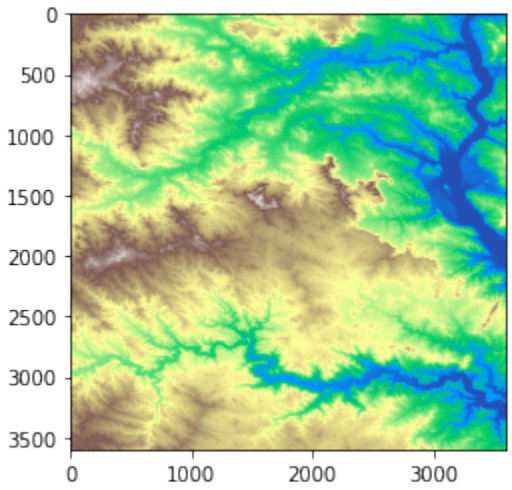
[14]:
```
# create a mosaic from all the images
with rio.Env(aws_session):
    sources = [rio.open(raster) for raster in file_S3]
```

Now we can use the `merge` function to merge these source files together using our list of bounding box values as the bounds. `merge` copies the valid pixels from input rasters and outputs them to a new raster.

[15]:
```
# merge the source files
mosaic, out_trans = merge(sources, bounds = bbox_list)
```

Lastly, we use `ma.masked_values` for masking all of the NoData values, allowing the mosaic to be plotted correctly. `ma.masked_values` returns a MaskedArray in which a data array is masked by an approximate value. For the parameters, let's use our mosaic as the data array and the integer of the "nodata" value as the value to mask by. Now we can use `show` to display our masked raster using `matplotlib` with a "terrain" colormap.

[16]:
```
# mask the NoData values so it can be plotted correctly
masked_mosaic = ma.masked_values(mosaic, int(sources[0].nodatavals[0]))
# display the masked mosaic
show(masked_mosaic, cmap = 'terrain')
```



[16]:
```
<AxesSubplot:>
```

3.3.7 Accessing EDAV Data via Web Coverage Service

This example demonstrates how to retrieve raster data from EDAV using a Web Coverage Service (WCS). A WCS lets you access coverage data with multiple dimensions online. The downloaded data is subsetted from data hosted on the MAAP and is available with the full resolution and values.

Setting up the Environment

We start by installing the libraries that are used to query the WCS connection point, and then to load, explore, and plot the raster data. We use `rasterio` for reading and writing raster formats, `rio-cogeo` for creating and validating Cloud Optimized GEOTIFF (COG) data, and `owslib` for interacting with Open Geospatial Consortium (OGC) services.

[1]:
```
# install libraries
# %pip is a magic command that installs into the current kernel
# -q means quiet (to give less output)
%pip install -q rasterio
%pip install -q rio-cogeo
%pip install -q owslib
```

After installing the libraries, note that you may see multiple messages that you need to restart the kernel. Then import `rasterio`, `show` from `rasterio.plot` to display images with labeled axes, and `WebCoverageService` from `owslib.wcs` to program with an OGC web service.

[2]:
```
# import rasterio
import rasterio as rio
# import show
from rasterio.plot import show
# import WebCoverageService
from owslib.wcs import WebCoverageService
```

Querying the WCS

Now we can configure the WCS source, use the `getCoverage` function to request a file in GeoTIFF format, and save what is returned to our workspace.

[3]:
```
# configure the WCS source
EDAV_WCS_Base = "https://edav-wcs.adamplatform.eu/wcs"
wcs = WebCoverageService(f'{EDAV_WCS_Base}?service=WCS', version='2.0.0')
```

[4]:
```
# request imagery to download
response = wcs.getCoverage(
    identifier=['test_afrisar_onera_ClopeTB10_biomass_COG'], # coverage ID
    format='image/tiff', # format what the coverage response will be returned as
    filter='false', # define constraints on query
    scale=1, # resampling factor (1 full resolution, 0.1 resolution degraded of a␣
↪factor of 10)
    subsets=[('Long',11.54,11.8),('Lat',-0.3,0.0)] # subset the image by lat / lon
)

# save the results to file as a tiff
results = "EDAV_example.tif"
with open(results, 'wb') as file:
    file.write(response.read())
```

We can use `gdalinfo` to provide information about our raster dataset to make sure the data is valid and contains spatial metadata.

[5]:
```
# gives information about the dataset
!gdalinfo {results}
```

```
Driver: GTiff/GeoTIFF
Files: EDAV_example.tif
Size is 2836, 3403
Coordinate System is:
GEOGCRS["WGS 84",
    DATUM["World Geodetic System 1984",
        ELLIPSOID["WGS 84",6378137,298.257223563,
```

(continues on next page)

(continued from previous page)

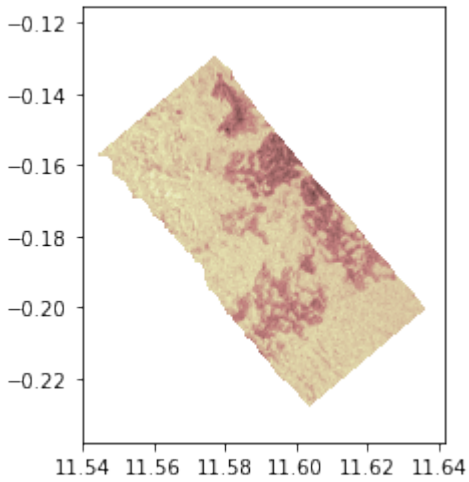
```
            LENGTHUNIT["metre",1]]],
    PRIMEM["Greenwich",0,
        ANGLEUNIT["degree",0.0174532925199433]],
    CS[ellipsoidal,2],
        AXIS["geodetic latitude (Lat)",north,
            ORDER[1],
            ANGLEUNIT["degree",0.0174532925199433]],
        AXIS["geodetic longitude (Lon)",east,
            ORDER[2],
            ANGLEUNIT["degree",0.0174532925199433]],
    ID["EPSG",4326]]
Data axis to CRS axis mapping: 2,1
Origin = (11.539968877500000,-0.115643000000000)
Pixel Size = (0.000035932500000,-0.000035932500000)
Metadata:
  AREA_OR_POINT=Area
Image Structure Metadata:
  COMPRESSION=LZW
  INTERLEAVE=BAND
Corner Coordinates:
Upper Left  (  11.5399689,  -0.1156430) ( 11d32'23.89"E,  0d 6'56.31"S)
Lower Left  (  11.5399689,  -0.2379213) ( 11d32'23.89"E,  0d14'16.52"S)
Upper Right (  11.6418734,  -0.1156430) ( 11d38'30.74"E,  0d 6'56.31"S)
Lower Right (  11.6418734,  -0.2379213) ( 11d38'30.74"E,  0d14'16.52"S)
Center      (  11.5909212,  -0.1767821) ( 11d35'27.32"E,  0d10'36.42"S)
Band 1 Block=2836x1 Type=Float32, ColorInterp=Gray
  NoData Value=0
```

Reading the Data

We can now use `rio.open` with our `results` path string and return an opened dataset object. We can set a variable (`rast`) to what is read from this dataset object. Then, we utilize the function `show` to display the raster using Matplotlib.

[6]:
```
# take path and return opened dataset object, set variable to read dataset object
with rio.open(results, 'r') as src:
        rast = src.read()
```

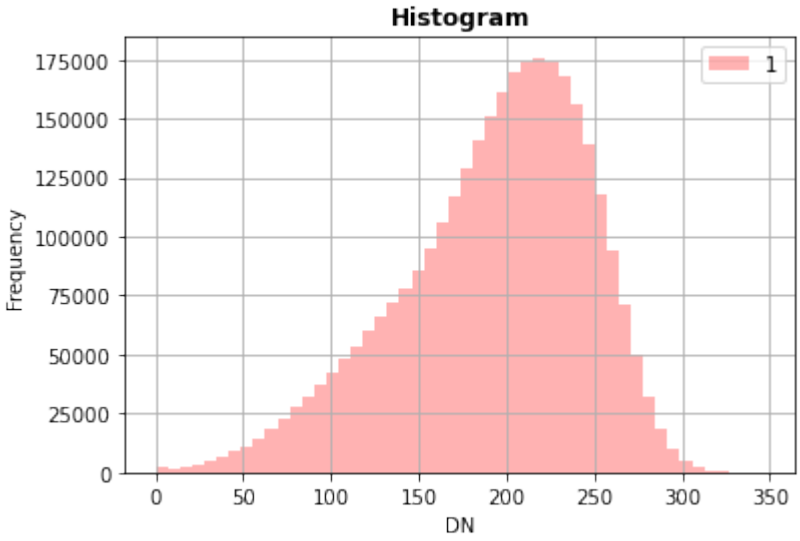
[7]:
```
# make a plot
show(rast, transform=src.transform, cmap='pink')
```



[7]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f27bc568c90>
```

We now have a visual of our raster. Let's import and employ the `show_hist` function from `rasterio.plot` to generate a histogram of the raster.

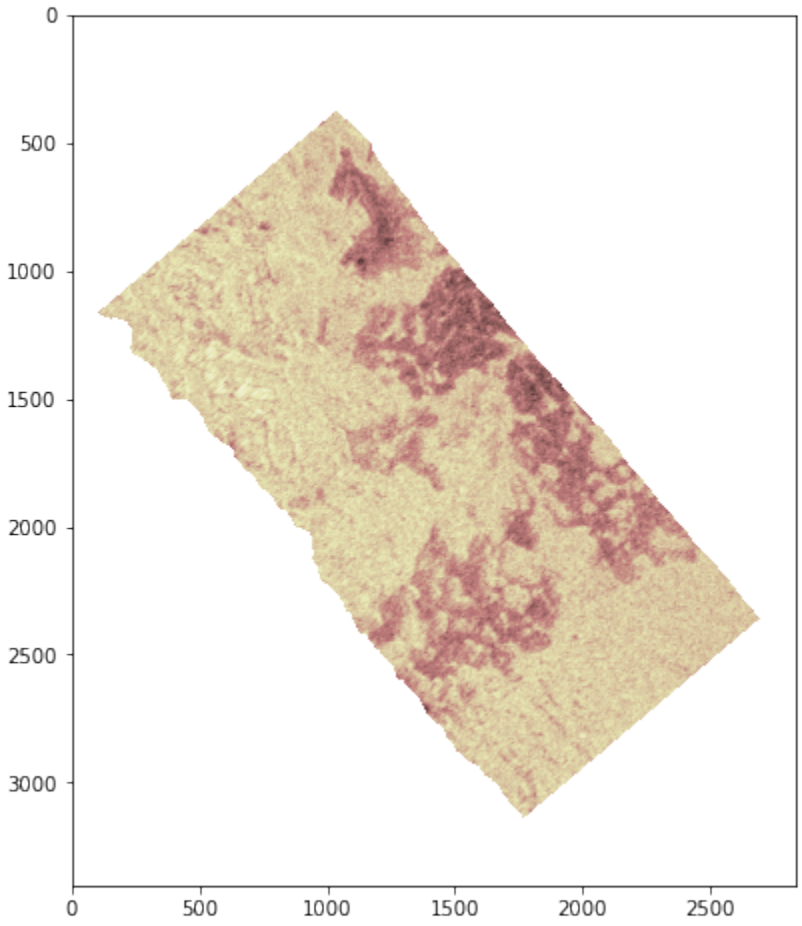
[8]:
```
# import show_hist
from rasterio.plot import show_hist
# create histogram
show_hist(rast,
          bins=50,# number of bins to compute histogram across
          alpha=.3,# transparancy
          title="Histogram"# figure title
         )
```



We can also generate a plot using Matplotlib. Let's import `matplotlib.pyplot` and `numpy` and make a new plot. To do this, use the `plt.subplots` function to return a figure and a single "Axes" instance. Then remove single-dimensional entries from the shape of our array using `np.squeeze` and display the data as an image using `imshow`. Now, we can set the norm limits for image scaling using the `set_clim` function.

[9]:
```
# import matplotlib.pyplot
import matplotlib.pyplot as plt
# import numpy
import numpy as np
```

[10]:
```
# set figure and single "Axes" instance
fig, ax = plt.subplots(1, figsize=(8,8))
# remove single-dimensional entries from the shape of the variable rast
#     and display the image
edavplot = ax.imshow(np.squeeze(rast), cmap='pink')
```



Newer Method `rioxarray`

Another way to work with raster data is with the `rasterio` "xarray" extension. Let's install and import `rioxarray` and create a plot using the `open_rasterio` and `plot` functions.

[11]:
```
# install rasterio xarray extension
%pip install -q rioxarray
```

```
WARNING: Running pip as the 'root' user can result in broken permissions and␣
↪conflicting behaviour with the system package manager. It is recommended to use a␣
↪virtual environment instead: https://pip.pypa.io/warnings/venv
WARNING: You are using pip version 22.0.3; however, version 23.0.1 is available.
You should consider upgrading via the '/opt/conda/bin/python -m pip install --upgrade␣
↪pip' command.
Note: you may need to restart the kernel to use updated packages.
```

(continues on next page)

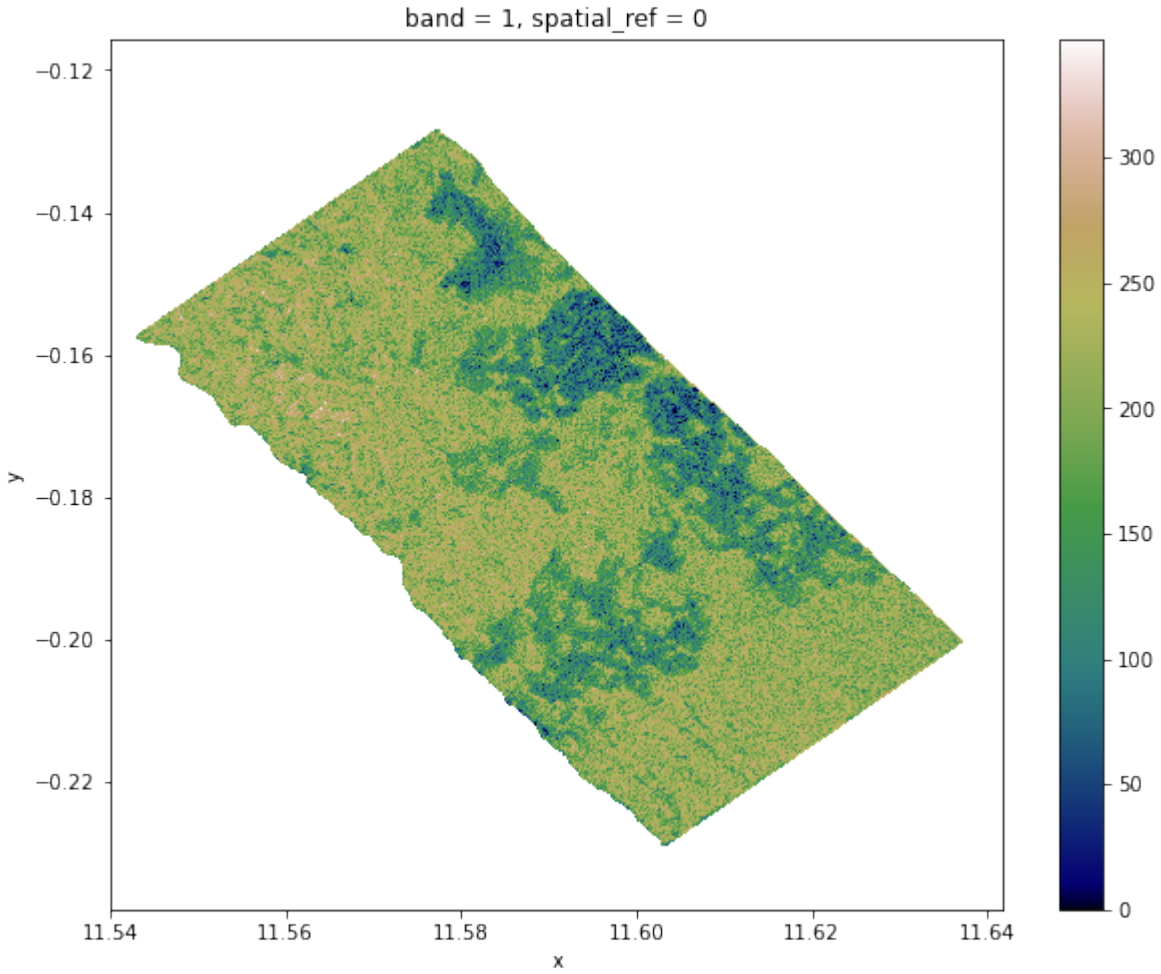
(continued from previous page)

[12]:
```
# import rasterio xarray extension
import rioxarray
```

[13]:
```
# opens results with rasterio to set dataarray
edav_x = rioxarray.open_rasterio(results)
```

[14]:
```
# plot dataarray
edav_x.plot(cmap="gist_earth", figsize=(10,8))
```

[14]:
```
<matplotlib.collections.QuadMesh at 0x7f275e7e5310>
```



References

- WCS Adapted from: Jan Verbesselt, Jorge Mendes de Jesus, Aldo Bergsma, Dainius Masiliūnas, David Swinkels, Corné Vreugdenhil. Handling Raster data with Python - 2020-01-20 https://geoscripting-wur.github.io/PythonRaster/
- OWSLib https://github.com/geopython/OWSLib
- rioxarray https://corteva.github.io/rioxarray

3.4 Query

3.4.1 How to Query Data from the MAAP via Python Client

Supported collections can be subsetted through the MAAP Query Service. At the time of writing (03/22/2021), the GEDI Calibration/Validation Field Survey Dataset is the only valid dataset for this service. However, more data will be made available for querying as the MAAP team continues to develop expanded services for the platform. Users can interact with the service through the MAAP Python client.

First, we import the `json` module, import the `MAAP` package, and create a new MAAP class.

[18]:
```
# import the json module
import json
# import the MAAP package
from maap.maap import MAAP
# create MAAP class
maap = MAAP()
```

How to Use `maap.executequery()`

We use the `executeQuery()` function to return a response object, containing the server's response to our HTTP request. This object can be used to view the response headers, access the raw data of the response, or parse the response as a JavaScript Object Notation (JSON). JSON is a data-interchange format, designed to be easy for humans to read and write.

`executeQuery` Parameters

- `src` - a dictionary-like object specifying the dataset to query. Currently, the only option is as follows:

```
{
  "Collection": {
    "ShortName": "GEDI Cal/Val Field Data_1",
    "VersionId": "2"
  }
}
```

- `query` - a dictionary-like object specifying the parameters for query. A dictionary can include `bbox`, `where`, `fields`, and `table`:
  - `bbox` - a list of floating point numbers identifying a bounding box of geographic coordinates.
  - `where` - a dictionary-like object which maps fields to required values within a query.
  - `fields` - a list of fields to return, a subset of all fields available for the corresponding dataset.
  - `table` - the name of the table to query. At this time, the only valid options are "tree" or "plot" corresponding to tables in the GEDI Cal/Val database.
- `poll_results` - a parameter which must be `True` to use the `timeout` parameter.
- `timeout` - the waiting period for a response. This indicates the maximum number of seconds to wait for a response. Note that timeout has a default value of '180' and requires that the `poll_results` parameter be `True`. Depending on the request, it may be necessary to modify the timeout to make sure the server has enough time to process the request.

- `wait_interval` - number of seconds to wait between each poll for results. `wait_interval` is only used if `poll_results=True` (default 0.5). -`max_redirects` - the maximum number of redirects to follow when scheduling an execution (default 5).

Query Searching for a Project Name

In this example, we create a dictionary containing a `Collection` key, which contains entries for `ShortName` and `VersionId`. This is used later in the `executeQuery()` function. For this example, we use the GEDI Calibration/Validation Field Survey Dataset collection. More information about dictionaries in Python can be found here.

[19]:
```
# create dictionary with a "Collection" key containing short name and version ID␣
↪entries:
collection  = {
  "Collection": {
      "ShortName": "GEDI Cal/Val Field Data_1",
      "VersionId": "2"
  }
}
```

We also create a function (in this example named `fetch_results`) which utilizes the `executeQuery` function to return results of queries. Within this function, we use the `executeQuery` function to get a response object. Within the `executeQuery` function, our `collection` dictionary is assigned to `src`, a dictionary-like object specifying the dataset to query. We also have the query used in the argument assigned to `query`, a dictionary-like object which specifies the parameters for the query. We set `timeout` to the timeout used in the argument (default is 180 seconds) and set `poll_results` to `True` in order to set the maximum waiting period for a response. We can check the 'Content-Type' header of our response to see the content type of the response. In the following code, the 'Content-Type' header is checked to determine if it is JSON or not, in order to set an appropriate variable to return.

[20]:
```
def fetch_results(query={}, timeout=180):
    """
    Function which utilizes the `executeQuery` function to return the results of␣
↪queries.
    """
    # use the executeQuery() function to get a response object
    response = maap.executeQuery(
      # dictionary-like object specifying the dataset to query
      src = collection,
      # dictionary-like object specifying the parameters for query
      query = query,
      # must be True to use the timeout parameter
      poll_results = True,
      # max waiting period for a response in seconds
      timeout = timeout
    )
    # if the 'Content-Type' is json, creates variable with json version of the␣
↪response
    if (response.headers.get("Content-Type") == "application/json"):
        data = response.json()
    # if the 'Content-Type' is not json, creates variable with unicode content of the␣
↪response
    else:
        data = response.text
    # returns `data` as json string
    return json.loads(data)
```

Now that we have our collection dictionary and our function to return the results of queries, let's use a print statement to display the first project name from a query which utilizes the `bbox` and `fields` parameters within `query`. The `bbox`

parameter is a GeoJSON-compliant bounding box ([minX, minY, maxX, maxY]) which is used to filter data spatially. GeoJSON is a format for encoding geographic data structures. More information about the bounding box can be found in the standard specification of the GeoJSON format, located here - https://tools.ietf.org/html/rfc7946#section-5. The `fields` parameter is a list of fields to return in the query response. In this case, we assign 'project' to `fields`.

[21]:
```
# prints the first project name in the results of the query as a json string
print(json.dumps(fetch_results({"bbox": [9.31, 0.53, 9.32, 0.54], "fields": ["project
↪"]})[0], indent=2))
```

```
{
  "project": "gabon_mondah"
}
```

Inspecting a Single Observation

In the previous example, we displayed the project name for a result from our query, but let's say we wished to see all of the fields and their associated values for a result. In this example, we make another query, this time only specifying the bounding box. The print statement displays the variables for a single observation. A list of the variables and their units and descriptions can be found here.

[22]:
```
# prints the fields with values for the first result in the results of the query as a␣
↪json string
print(json.dumps(fetch_results({"bbox": [9.315, 0.535, 9.32, 0.54]})[0], indent=2))
```

```
{
  "project": "gabon_mondah",
  "plot": "NASA11",
  "subplot": "1",
  "survey": "AfriSAR_ESA_2016",
  "private": 0,
  "date": "2016-02-01",
  "region": "Af",
  "vegetation": "TropRF",
  "map": 3083.93471636915,
  "mat": 25.6671529098763,
  "pft.modis": "Evergreen Broadleaf trees",
  "pft.name": null,
  "latitude": 0.538705025207016,
  "longitude": 9.31982893597376,
  "p.sample": 0,
  "p.stemmap": 0,
  "p.origin": "C",
  "p.orientation": -2.18195751718555,
  "p.shape": "R",
  "p.majoraxis": 100,
  "p.minoraxis": 100,
  "p.geom": "POLYGON ((535537.75 59601.25, 535627.75 59590.5, 535642.25 59489.25,␣
↪535543 59498.5, 535537.75 59601.25, 535537.75 59601.25))",
  "p.epsg": 32632,
  "p.area": 10000,
  "p.mindiam": 0.01,
  "sp.geom": "POLYGON((535537.510093 59494.109258, 535537.526710 59519.109253, 535562.
↪526704 59519.092636, 535562.510088 59494.092642, 535537.510093 59494.109258))",
  "sp.ix": 1,
  "sp.iy": 4,
  "sp.shape": "R",
  "sp.area": 625,
```

(continues on next page)

(continued from previous page)


```
  "sp.mindiam": 0.01,
  "pai": null,
  "lai": null,
  "cover": null,
  "dft": "EA",
  "agb": null,
  "agb.valid": null,
  "agb.lower": null,
  "agb.upper": null,
  "agbd.ha": null,
  "agbd.ha.lower": null,
  "agbd.ha.upper": null,
  "sn": null,
  "snd.ha": null,
  "sba": null,
  "sba.ha": null,
  "swsg.ba": null,
  "h.t.max": null,
  "sp.agb": null,
  "sp.agb.valid": null,
  "sp.agbd.ha": null,
  "sp.agbd.ha.lower": null,
  "sp.agbd.ha.upper": null,
  "sp.sba.ha": null,
  "sp.swsg.ba": null,
  "sp.h.t.max": null,
  "l.project": "jpl_mondah",
  "l.instr": null,
  "l.epsg": 32632,
  "l.date": null,
  "tree.date": "2016-02-10",
  "family": "Myristicaceae",
  "species": "Coelocaryon sp.",
  "pft": null,
  "wsg": 0.49353216374269,
  "wsg.sd": 0.0941339098036177,
  "tree": "5501",
  "stem": "1",
  "x": 535539.544644182,
  "y": 59496.0746845487,
  "z": null,
  "status": 1,
  "allom.key": 2,
  "a.stem": 0.0437435361085843,
  "h.t": 9.46667,
  "h.t.mod": 17.0934257579035,
  "d.stem": 0.236,
  "d.stem.valid": 1,
  "d.ht": 1.3,
  "c.w": null,
  "m.agb": 145.005237378992,
  "id": 2077,
  "geom_obj":
↪"0103000020E6100000010000000500000032A8C5A385A32240A1FCE9F75E39E13F32892DA985A32240A012DE4C393BE13F
↪",
  "wwf.ecoregion": "Central African mangroves"
}
```

Query Using Multiple Parameters With `where`

In the output of the previous example, we can see the field `"species"`. Let's say we are interested in finding observations for the `"gabon_mondah"` project within the same bounding box as the previous example which have the same species. We can do this using `where`, a dictionary-like object which maps fields to required values within a query. To help demonstrate how to use `where`, we can create a new function (in this example named `species_query`) to print the number of results as well as the first result, adding on to our previous `fetch_results` function which utilized the `executeQuery` function.

[23]:
```
def species_query(query = {}, timeout = 180):
    """
    Function which utilizes the `fetch_results` (and thereby `executeQuery`) function␣
↪and prints the number of results
    as well as the first result.
    """
    # set `data` to results of query
    data = fetch_results(query = query, timeout = timeout)
    # get the number of results within `data`
    num_results = len(data)
    # if `data` is not null and contains at least one result, the number of results␣
↪and the first result are printed
    if((data is not None) and (num_results > 0)):
        first_result = data[0]
        print(f"Number of results: {num_results}")
        print(f"First result: {json.dumps(first_result, indent=2)}")
    # else prints "No result"
    else:
        print(num_results)
        print("No result")
```

Let's call the `species_query` function. We enter the same bounding box values as in the previous example. This time around, we enter `where` in our query and set the `project` as `gabon_mondah` and the `species` as `Coelocaryon sp.`. We can set a list of fields to return in query response using `fields`. For this example, we can choose to return only the project, family, species, latitude, and longitude values. After completing our query, we can manually set the timeout value (in this example '200').

[24]:
```
# call `species_query` function with bounding box values, where the project is "gabon_
↪mondah",
# the species is "Aucoumea klaineana", fields include "project", "family", "species",
↪"latitude", and "longitude",
# and the timeout value is 200 (use the scrollbar to see the entire function call)
species_query({"bbox": [9.315, 0.535, 9.32, 0.54], "where": {"project": "gabon_mondah
↪", "species": "Coelocaryon sp."}, "fields": ["project", "family", "species",
↪"latitude", "longitude"]}, 200)
```

```
Number of results: 10
First result: {
  "project": "gabon_mondah",
  "family": "Myristicaceae",
  "species": "Coelocaryon sp.",
  "latitude": 0.538705025207016,
  "longitude": 9.31982893597376
}
```

We now see that there are many results and the latitude and longitude coordinates for the first result. To see this information for Aucoumea klaineana, we can copy the code from the above cell, changing the `species` to `Aucoumea klaineana` within the function argument.

[25]:
```
# call `species_query` function with bounding box values, where the project is "gabon_
↪mondah",
# the species is "Coelocaryon sp.", fields include "project", "family", "species",
↪"latitude", and "longitude",
# and the timeout value is 200 (use the scrollbar to see the entire function call)
species_query({"bbox": [9.315, 0.535, 9.32, 0.54], "where": {"project": "gabon_mondah
↪", "species": "Aucoumea klaineana"}, "fields": ["project", "family", "species",
↪"latitude", "longitude"]}, 200)
```

```
Number of results: 49
First result: {
  "project": "gabon_mondah",
  "family": "Burseraceae",
  "species": "Aucoumea klaineana",
  "latitude": 0.538705025207016,
  "longitude": 9.31982893597376
}
```

Using `executeQuery` to Vizualize Plots and Trees

Now that we know how to print the query results of the collection, let's look at examples of vizualizing the information from query results. The first time you run this, you will need to install `folium`. `folium` allows us to visualize data on an interactive map. With `folium` installed, you will then import the `folium` library. Also import `matplotlib.pyplot`, which allows us to generate interactive plots

[26]:
```
# installs folium
!pip install folium
```

```
Requirement already satisfied: folium in /Users/philvarner/code/devseed/maap-
↪documentation/.venv/lib/python3.7/site-packages (0.12.1.post1)
Requirement already satisfied: numpy in /Users/philvarner/code/devseed/maap-
↪documentation/.venv/lib/python3.7/site-packages (from folium) (1.21.5)
Requirement already satisfied: jinja2>=2.9 in /Users/philvarner/code/devseed/maap-
↪documentation/.venv/lib/python3.7/site-packages (from folium) (3.0.3)
Requirement already satisfied: requests in /Users/philvarner/code/devseed/maap-
↪documentation/.venv/lib/python3.7/site-packages (from folium) (2.27.1)
Requirement already satisfied: branca>=0.3.0 in /Users/philvarner/code/devseed/maap-
↪documentation/.venv/lib/python3.7/site-packages (from folium) (0.4.2)
Requirement already satisfied: MarkupSafe>=2.0 in /Users/philvarner/code/devseed/maap-
↪documentation/.venv/lib/python3.7/site-packages (from jinja2>=2.9->folium) (2.0.1)
Requirement already satisfied: certifi>=2017.4.17 in /Users/philvarner/code/devseed/
↪maap-documentation/.venv/lib/python3.7/site-packages (from requests->folium) (2021.
↪10.8)
Requirement already satisfied: idna<4,>=2.5 in /Users/philvarner/code/devseed/maap-
↪documentation/.venv/lib/python3.7/site-packages (from requests->folium) (3.3)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /Users/philvarner/code/
↪devseed/maap-documentation/.venv/lib/python3.7/site-packages (from requests->
↪folium) (1.26.8)
Requirement already satisfied: charset-normalizer~=2.0.0 in /Users/philvarner/code/
↪devseed/maap-documentation/.venv/lib/python3.7/site-packages (from requests->
↪folium) (2.0.11)
```

[27]:
```
# import folium library and matplotlib.pyplot
import folium
import matplotlib.pyplot as plt
```

Plot Project Plots

Now we can begin plotting all the plots for a given project. For this example, we'll check out the `australia_ausplotsforests` project. We can query the plot table by using our `fetch_results` helper function. This time, we set the `table` parameter. `table` is the name of the table to query. The GEDI Cal/Val database has tables for "tree" and "plot" so let's set `table` to "plot". We can then preview our results.

[28]:
```
# set project and query for all the plots in that project
project = 'australia_ausplotsforests'
results = fetch_results({
    "bbox": [-180, 90, 180, -90],
    "where": {
        "project": project
    },
    "fields": ["latitude", "longitude", "plot"],
    "table":"plot"
}, 1000)
# print first 10 results
results[0:10]
```

[28]:
```
[{'latitude': -31.2421, 'longitude': 152.4609, 'plot': 'NSFNNC002'},
 {'latitude': -31.2421, 'longitude': 152.4609, 'plot': 'NSFNNC002'},
 {'latitude': -31.2421, 'longitude': 152.4609, 'plot': 'NSFNNC002'},
 {'latitude': -31.2421, 'longitude': 152.4609, 'plot': 'NSFNNC002'},
 {'latitude': -31.2421, 'longitude': 152.4609, 'plot': 'NSFNNC002'},
 {'latitude': -31.2421, 'longitude': 152.4609, 'plot': 'NSFNNC002'},
 {'latitude': -31.2421, 'longitude': 152.4609, 'plot': 'NSFNNC002'},
 {'latitude': -31.2421, 'longitude': 152.4609, 'plot': 'NSFNNC002'},
 {'latitude': -31.2421, 'longitude': 152.4609, 'plot': 'NSFNNC002'},
 {'latitude': -31.2421, 'longitude': 152.4609, 'plot': 'NSFNNC002'}]
```

Create Project Plots Dictionary

Now we can create a dictionary for the project plots from our results.

[29]:
```
# create dictionary for project plots, where each plot has lat/lon info
project_plots = {}
keys = [ 'latitude', 'longitude' ]
for result in results:
    project_plots[result['plot']] = { key: result[key] for key in keys }
```

To center our map more easily and plot trees later on, we can set a variable to the first project plot to be used for centering the map.

[30]:
```
# Select the first plot, just to center the map easily
first_plot = list(project_plots.keys())[0]
first_plot
```

[30]:
```
'NSFNNC002'
```

Map the Plots

Now we can create a map and place the plot points onto it. First we will set variables for the map's location center and starting zoom level. Then we will use the `folium.Map` function to create a map which displays "Stamen Terrain" tiles with our location center and starting zoom level. Next, we will create a loop using the `folium.Marker` function

to add markers to the map at the latitude and longitude for each plot and have popup text displaying each plot's name. Lastly, we display the map in order to interact with it.

[31]:
```
# set location center for map
center = [ project_plots[first_plot]['latitude'], project_plots[first_plot]['longitude
↪'] ]
# set zoom level, note that depending on the project,
# you may wish to increase the zoom level
zoom = 3
# create a map with Folium and Leaflet.js
m = folium.Map(location=center, tiles="Stamen Terrain", zoom_start = zoom)
# Add markers to map for each plot in `project_plots`
for plot in project_plots.items():
    folium.Marker(
        [plot[1]['latitude'], plot[1]['longitude']],
        popup = f"plot: {plot[0]}"
    ).add_to(m)
# display map
m
```

[31]:
```
<folium.folium.Map at 0x125d8ccd0>
```

The map has zoom in and zoom out buttons, can be dragged with a mouse, and displays markers which can be clicked on to display their popup text.

Plot Trees

Now let's look at plotting trees for the first plot of the designated project. We construct another query with our `fetch_results` helper function. This time, we query for fields containing plant height, UTM coordinates, and elevation as well as set the `table` parameter to "tree".

[32]:
```
# query for trees for the first plot of the project
results = fetch_results({
    "bbox": [-180, 90, 180, -90],
    "where": {
        "project": project,
        "plot": first_plot
    },
    # h.t -> total height of plant from ground to highest leaf
    # x -> easting UTM coordinate
    # y -> northing UTM coordinate
    # z -> elevation relative to geoid coordinate
    "fields": [ "h.t", "x", "y", "z"],
    "table": "tree"
}, 1000)
```

Determine the Number and Height of Trees

To see the number of trees in the list of results, we use the `len` function within a formatted print statement. Since not all of the results contain heights, lets check how many trees with height data exist by creating a new variable set to the results with heights and use the `len` function within a formatted print statement again. Also, we can set variables to contain the heights, x and y values contained within our results to be used when plotting the trees.

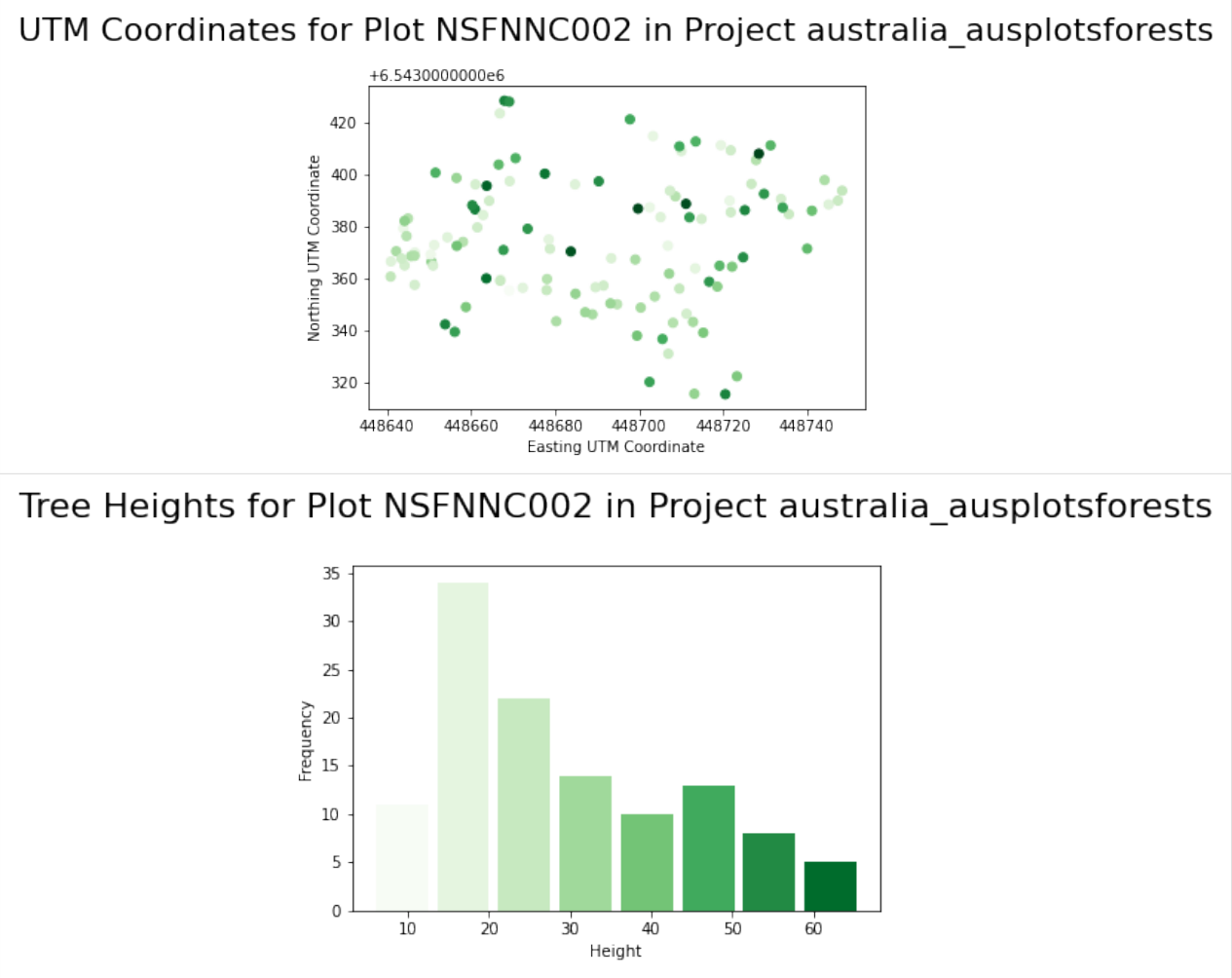
[33]:
```
# print number of trees in results
print(f"Number of trees: {len(results)}")
# print number of trees with heights
heights = [r["h.t"] for r in results if r["h.t"] is not None]
print(f"Number of trees with heights: {len(heights)}")
# set variables to the heights, x and y values contained in the results
hts = [r['h.t'] for r in results]
xs = [r['x'] for r in results]
ys = [r['y'] for r in results]
```

```
Number of trees: 520
Number of trees with heights: 117
```

Plot the Trees

With our variables for tree heights, x and y coordinate values, let's use the `plt.scatter` function to create a scatter plot with a title and labels for the UTM coordinates of the trees. Lastly, let's also create a histogram using `plt.hist` to visualize the distribution of tree heights. Here we also include some code to adjust the display colors and add labels and a title.

[34]:
```
# create scatter plot with x and y values
plt.scatter(xs, ys, c=hts, cmap="Greens")
# add labels and title
plt.xlabel('Easting UTM Coordinate')
plt.ylabel('Northing UTM Coordinate')
plt.title(f"UTM Coordinates for Plot {first_plot} in Project {project}\n",␣
↪fontsize=22)
plt.show()
# adjust display colors
cm = plt.cm.Greens
nbins = 8
# create histogram for tree heights
n, bins, patches = plt.hist(x=heights, bins=nbins, rwidth=0.85)
for i, p in enumerate(patches):
    plt.setp(p, 'facecolor', cm(i/nbins))
# add labels and title
plt.xlabel('Height')
plt.ylabel('Frequency')
plt.title(f"Tree Heights for Plot {first_plot} in Project {project}\n", fontsize=22)
plt.show()
```



3.4.2 Fields Within the Gedi Cal/Val Data

Last updated: 2020-06-30

The Global Ecosystem Dynamics Investigation (GEDI) Forest Structure and Biomass Database (FSBD) is a collection of field and LiDAR datasets developed to serve a central repository for calibration and validation of the GEDI mission data. The GEDI Cal/Val Field Data collection contains the field data contributions to the FSBD. The database has contributions across the globe for a variety of vegetation types from projects dating back to 2003.

The table below describes the variables found in the files:

variable	units	description
plot	NA	name of plot
subplot	NA	subplot identifier
survey	NA	name of survey event
private	NA	data privacy (1=private, 0=public)
date	NA	date of survey event
region	NA	region of plot location: Eu=Europe; As=Asia; Au=Australia/New Zealand; Af=Africa; SEAsia=Sout
vegetation	NA	vegetation type where sampled: Sav = Savannah; TropRF = Tropical rainforest; TempRF = Temperat

Table 1

variable	units	description
map	mm	mean annual rainfall
mat	deg	mean annual temperature
pft.modis	NA	plant functional type from the nearest MCD12Q1 product date
pft.name	NA	plant functional type from field descriptions or other ancillary data
wwf.ecoregion	NA	Terrestrial ecoregion from the WWF Olson et al. (2004) terrestrial ecoregions of the globe
latitude	deg	latitude of location where sampled (-90 to 90 deg South to North)
longitude	deg	longitude of location where sampled (-180 to 180 deg West to East)
p.sample	NA	subplot type: 0=independent fixed area plots; 1=variable area plots; 2=multiple fixed area plots for ag
p.stemmap	NA	stem mapped plot (1=TRUE,0=FALSE)
p.origin	NA	origin of the plot, SW has to be interpreted as the lower left corner relative to the orientation
p.orientation	deg	orientation of plot degrees clockwise from UTM map grid north
p.shape	NA	plot shape (E=ellipse, R=rectangle)
p.majoraxis	m	major axis of plot
p.minoraxis	m	minor axis of plot
p.geom	m	plot geometry WKT string
p.epsg	NA	epsg code of the UTM zone used for the tree coordinates
p.area	m2	plot polygon area
p.mindiam	m	minimum tree diameter measured across all subplots
sp.geom	m	subplot geometry WKT string
sp.ix	NA	subplot track index
sp.iy	NA	subplot pulse index
sp.shape	NA	subplot shape (E=ellipse, R=rectangle)
sp.area	m2	subplot area
sp.mindiam	m	minimum tree diameter measured for the subplot
pai	m2/m2	plant area index of vegetation
lai	m2/m2	leaf area index of vegetation
cover	NA	vertically projected canopy cover (1-Pgap)
dft	NA	dominant plant functional type: EA = evergreen angiosperm; DA = deciduous angiosperm; EG = eve
agb	kg	mass of above-ground standing trees and shrubs
agb.valid	NA	valid above ground mass prediction (1=TRUE,0=FALSE)
agb.lower	kg	lower limit of standard error of mass of above-ground standing trees and shrubs
agb.upper	kg	upper limit of standard error of mass of above-ground standing trees and shrubs
agbd.ha	Mg/ha	mass density of above-ground standing trees and shrubs
agbd.ha.lower	Mg/ha	lower limit of standard error of mass density of above-ground standing trees and shrubs
agbd.ha.upper	Mg/ha	upper limit of standard error of mass density of above-ground standing trees and shrubs
sn	n	plot stem number
snd.ha	n/ha	plot stem number density
sba	m2	plot stand basal area
sba.ha	m2/ha	plot stand basal area per hectare
swsg.ba	g/cm3/m2	basal-area-weighted wood specific gravity
h.t.max	m	maximum total height of plants from ground to highest leaf
sp.agb	kg	subplot mass of above-ground standing trees and shrubs
sp.agb.valid	NA	valid subplot above ground mass prediction (1=TRUE,0=FALSE)
sp.agbd.ha	Mg/ha	subplot mass density of above-ground standing trees and shrubs
sp.agbd.ha.lower	Mg/ha	subplot lower limit of standard error of mass density of above-ground standing trees and shrubs
sp.agbd.ha.upper	Mg/ha	subplot upper limit of standard error of mass density of above-ground standing trees and shrubs
sp.sba.ha	m2/ha	subplot stand basal area per hectare
sp.swsg.ba	g/cm3/m2	subplot basal-area-weighted wood specific gravity
sp.h.t.max	m	subplot maximum total height of plants from ground to highest leaf

Table 1

variable	units	description
l.project	NA	name of the lidar project dataset on UMD servers
l.instr	NA	lidar instrument manufacturer and model
l.epsg	NA	epsg code of the corresponding lidar data
l.date	NA	date of the corresponding lidar acquisition
g.fp	NA	geometry collection WKT string of GEDI lidar footprint center locations
tree.date	NA	date of tree measurement
family	NA	latin name of botanical family
species	NA	latin name of species (genus species)
pft	NA	plant functional type: EA = evergreen angiosperm; DA = deciduous angiosperm; EG = evergreen gy
wsg	g/cm3	the ratio of wood density to water
wsg.sd	g/cm3	standard deviation of WSG sample values
tree	NA	tree identifier
stem	NA	stem identifier
x	m	easting UTM coordinate
y	m	northing UTM coordinate
z	m	elevation relative to geoid coordinate
status	NA	live tree (1=TRUE,0=FALSE)
allom.key	NA	key to allometric LUT
a.stem	m2	area of total stem cross-section at measurement height
h.t	m	total height of plant from ground to highest leaf
h.t.mod	NA	tree heights modelled using local or regional DBH-Ht relationship
d.stem	m	diameter of stem at measurement height
d.stem.valid	NA	valid tree diameter to use in AGB calculation (1=TRUE,0=FALSE)
d.ht	m	height at which stem diameter was measured
c.w	m	diameter or width of crown
m.agb	kg	mass of above-ground components of tree

3.5 Adding Cloud-Optimized GeoTIFFs to the MAAP Dashboard

The following notebook steps through how to add a dataset to the MAAP Dashboard.

Note, there are 2 scenarios:

1. Adding a single Cloud-Optimized GeoTIFF (COG) and,
2. Adding many distinct COGs as a "mosaic" with mosaicJSON.

High-level, the steps are:

1. Inspect your COG(s) to understand the best rescale and colormap name parameters. Optionally create a mosaic.
2. Define a colormap. Colormaps provide mappings of data values to RGB values.
3. Create a PR to the datasets repo to add or update your dataset.

The MAAP dashboard has 3 environments:

1. Developer-in-test (DIT): https://biomass.dit.maap-project.org
2. Staging: https://uat.earthdata.nasa.gov/maap-biomass
3. Production: https://earthdata.nasa.gov/maap-biomass

These instructions will guide you towards adding your dataset to `biomass.dit.maap-project.org`. The MAAP Dashboard team will "promote" changes to staging and production periodically (release schedule forthcoming).

3.5.1 Setup

[ ]:
```
%%capture
!pip install rasterio rio-cogeo supermercado
```

[1]:
```
# import the MAAP package
import ipycmc
from maap.maap import MAAP

import glob
import json
import os
import matplotlib
import urllib

# create MAAP class
maap = MAAP(maap_host='api.ops.maap-project.org')
```

[2]:
```
project_dir = "/projects/shared-buckets/<your_name>/<project_dir>"
# e.g.
project_dir = "/projects/shared-buckets/alexdevseed/landsat8/viz/"
```

[3]:
```
# Search for files to include, use recursive if nested folders (common in DPS output)
files = glob.glob(os.path.join(project_dir, "*.tif"), recursive=False)
files = [os.path.basename(f) for f in files]
#files
```

[4]:
```
my_tif = files[1]
```

[5]:
```
%%bash -s "$project_dir" "$my_tif"
rio cogeo validate $1/$2
```

```
/projects/shared-buckets/alexdevseed/landsat8/viz/Copernicus_30439_covars_cog_topo_
↪stack.tif is a valid cloud optimized GeoTIFF
```

[6]:
```
%%bash -s "$project_dir" "$my_tif"
gdalinfo $1/$2 -stats
```

```
Driver: GTiff/GeoTIFF
Files: /projects/shared-buckets/alexdevseed/landsat8/viz//Copernicus_30439_covars_cog_
↪topo_stack.tif
       /projects/shared-buckets/alexdevseed/landsat8/viz//Copernicus_30439_covars_cog_
↪topo_stack.tif.aux.xml
Size is 1000, 1000
Coordinate System is:
PROJCRS["unknown",
    BASEGEOGCRS["NAD83",
        DATUM["North American Datum 1983",
            ELLIPSOID["GRS 1980",6378137,298.257222101004,
                LENGTHUNIT["metre",1]]],
        PRIMEM["Greenwich",0,
```

(continues on next page)

(continued from previous page)

```
            ANGLEUNIT["degree",0.0174532925199433]],
        ID["EPSG",4269]],
    CONVERSION["Albers Equal Area",
        METHOD["Albers Equal Area",
            ID["EPSG",9822]],
        PARAMETER["Latitude of false origin",40,
            ANGLEUNIT["degree",0.0174532925199433],
            ID["EPSG",8821]],
        PARAMETER["Longitude of false origin",180,
            ANGLEUNIT["degree",0.0174532925199433],
            ID["EPSG",8822]],
        PARAMETER["Latitude of 1st standard parallel",50,
            ANGLEUNIT["degree",0.0174532925199433],
            ID["EPSG",8823]],
        PARAMETER["Latitude of 2nd standard parallel",70,
            ANGLEUNIT["degree",0.0174532925199433],
            ID["EPSG",8824]],
        PARAMETER["Easting at false origin",0,
            LENGTHUNIT["metre",1],
            ID["EPSG",8826]],
        PARAMETER["Northing at false origin",0,
            LENGTHUNIT["metre",1],
            ID["EPSG",8827]]],
    CS[Cartesian,2],
        AXIS["easting",east,
            ORDER[1],
            LENGTHUNIT["metre",1,
                ID["EPSG",9001]]],
        AXIS["northing",north,
            ORDER[2],
            LENGTHUNIT["metre",1,
                ID["EPSG",9001]]]]
Data axis to CRS axis mapping: 1,2
Origin = (698522.000000000000000,3153304.000000000000000)
Pixel Size = (30.000000000000000,-30.000000000000000)
Metadata:
  AREA_OR_POINT=Area
  OVR_RESAMPLING_ALG=NEAREST
Image Structure Metadata:
  COMPRESSION=DEFLATE
  INTERLEAVE=PIXEL
  LAYOUT=COG
Corner Coordinates:
Upper Left  (  698522.000, 3153304.000) (163d20'28.79"W, 67d29'44.89"N)
Lower Left  (  698522.000, 3123304.000) (163d30'48.19"W, 67d14'14.17"N)
Upper Right (  728522.000, 3153304.000) (162d39'23.92"W, 67d25'44.51"N)
Lower Right (  728522.000, 3123304.000) (162d50' 6.57"W, 67d10'16.36"N)
Center      (  713522.000, 3138304.000) (163d 5'11.87"W, 67d20' 1.17"N)
Band 1 Block=256x256 Type=Float32, ColorInterp=Gray
  Description = elevation
  Min=-1.830 Max=557.863
  Minimum=-1.830, Maximum=557.863, Mean=74.140, StdDev=84.128
  Overviews: 500x500, 250x250
  Metadata:
    STATISTICS_MAXIMUM=557.86291503906
    STATISTICS_MEAN=74.140305286651
    STATISTICS_MINIMUM=-1.8297636508942
```

(continues on next page)

(continued from previous page)

```
    STATISTICS_STDDEV=84.128259469301
    STATISTICS_VALID_PERCENT=100
Band 2 Block=256x256 Type=Float32, ColorInterp=Undefined
  Description = slope
  Min=0.000 Max=40.566
  Minimum=0.000, Maximum=40.566, Mean=2.972, StdDev=4.108
  Overviews: 500x500, 250x250
  Metadata:
    STATISTICS_MAXIMUM=40.566223144531
    STATISTICS_MEAN=2.9720626801483
    STATISTICS_MINIMUM=0
    STATISTICS_STDDEV=4.1079777350157
    STATISTICS_VALID_PERCENT=100
Band 3 Block=256x256 Type=Float32, ColorInterp=Undefined
  Description = tsri
  Min=0.000 Max=1.000
  Minimum=0.000, Maximum=1.000, Mean=0.466, StdDev=0.333
  Overviews: 500x500, 250x250
  Metadata:
    STATISTICS_MAXIMUM=0.99999368190765
    STATISTICS_MEAN=0.46556238474922
    STATISTICS_MINIMUM=4.3117461245856e-06
    STATISTICS_STDDEV=0.33263731194436
    STATISTICS_VALID_PERCENT=100
Band 4 Block=256x256 Type=Float32, ColorInterp=Undefined
  Description = tpi
  Min=-4.774 Max=7.687
  Minimum=-4.774, Maximum=7.687, Mean=-0.000, StdDev=0.349
  Overviews: 500x500, 250x250
  Metadata:
    STATISTICS_MAXIMUM=7.6866292953491
    STATISTICS_MEAN=-4.5027400747145e-05
    STATISTICS_MINIMUM=-4.7744207382202
    STATISTICS_STDDEV=0.34907900787421
    STATISTICS_VALID_PERCENT=100
Band 5 Block=256x256 Type=Float32, ColorInterp=Undefined
  Description = slopemask
  Min=0.000 Max=1.000
  Minimum=0.000, Maximum=1.000, Mean=0.982, StdDev=0.128
  Overviews: 500x500, 250x250
  Metadata:
    STATISTICS_MAXIMUM=1
    STATISTICS_MEAN=0.98238805740667
    STATISTICS_MINIMUM=0
    STATISTICS_STDDEV=0.12799686533701
    STATISTICS_VALID_PERCENT=100
```

3.5.2 Create some parameters for the dynamic tiler.

These parameters will be pased to `titiler_url` for visualization.

Note the values below: `band_min`, `band_max` and `colormap_name` are set as the current defaults for biomass in Mg/hectare for the dashboard. For other datasets, users should inspect the output of the `gdalinfo` for `band_min` and `band_max` and modify the `colormap_name` as makes sense for the current dataset. This notebook includes section on what colormaps are available and how to configure different types of colormaps and legends.

[7]:
```
titiler_url = f"https://titiler.maap-project.org"
band_min = 0
band_max = 400
rescale = f"{band_min},{band_max}"
bidx = 1
colormap_name = 'gist_earth_r'
```

3.5.3 Helper functions

3.5.4 Step 1, Scenario 1: Adding a single COG

1. Upload the file

Only use the following steps if you only have one COG to share to the dashboard. If you want to create a mosaic from multiple COGs, skip to Step 1 Option 2.

Upload the file to S3 and make note of the location.

[8]:
```
def local_to_s3(url):
    ''' A Function to convert local paths to s3 urls'''
    return url.replace("/projects/shared-buckets", "s3://maap-ops-workspace/shared")

location = local_to_s3(f"{project_dir}{my_tif}")
location
```

[8]:
```
's3://maap-ops-workspace/shared/alexdevseed/landsat8/viz/Copernicus_30439_covars_cog_
↪topo_stack.tif'
```

3.5.5 2. Test the COG with the titiler and ipyCMC

[9]:
```
## Build a WMTS call
wmts_url = f"{titiler_url}/cog/WMTSCapabilities.xml"
params = {
    "tile_format": "png",
    "tile_scale": "1",
    "pixel_selection": "first",
    "TileMatrixSetId": "WebMercatorQuad",
    "url": location,
    "bidx": bidx, # Select which band to use
    "resampling_method":"nearest",
    "rescale": rescale,
    "return_mask": "true",
    "colormap_name": colormap_name # Any colormap from matplotlib will work
}

wmts_call = "?".join([wmts_url, urllib.parse.urlencode(params)])

# Note Jupyter bug add amp; incorrectly when printing the url
wmts_call
```

[9]:
```
'https://titiler.maap-project.org/cog/WMTSCapabilities.xml?tile_format=png&tile_
↪scale=1&pixel_selection=first&TileMatrixSetId=WebMercatorQuad&url=s3%3A%2F%2Fmaap-
↪ops-workspace%2Fshared%2Falexdevseed%2Flandsat8%2Fviz%2FCopernicus_30439_covars_cog_
↪topo_stack.tif&bidx=1&resampling_method=nearest&rescale=0%2C400&return_mask=true&
↪colormap_name=gist_earth_r'
```

[10]:
```
w = ipycmc.MapCMC()
w
```

```
MapCMC()
```

[11]:
```
w.load_layer_config(wmts_call, "wmts/xml")
```

3.5.6 Step 1, Scenario 2: Adding data from multiple COGs by creating a mosaic

Many datasets are comprised of many tiles distributed spatially over the globe. In order to visualize them all together, we can use mosaicJSON to create a mosaic for the dynamic tiler API. The dynamic tiler API knows how to read this mosaicJSON and select which tiles to render based on the current zoom, x and y coordinates across spatially distinct COGs.

[12]:
```
%%capture
!pip install cogeo-mosaic
```

[13]:
```
import glob
import os
import urllib

from cogeo_mosaic.mosaic import MosaicJSON
from cogeo_mosaic.backends import MosaicBackend
```

[14]:
```
# Skipping this step since we don't want to upload these large files more than once!
full_path_files = [f'{project_dir}{file}' for file in files]
```

[15]:
```
def local_to_s3(url):
    ''' A Function to convert local paths to s3 urls'''
    return url.replace("/projects/shared-buckets", "s3://maap-ops-workspace/shared")

tiles = [local_to_s3(file) for file in full_path_files]
print(tiles)
```

```
['s3://maap-ops-workspace/shared/alexdevseed/landsat8/viz/Copernicus_30438_covars_cog_
↪topo_stack.tif', 's3://maap-ops-workspace/shared/alexdevseed/landsat8/viz/
↪Copernicus_30439_covars_cog_topo_stack.tif', 's3://maap-ops-workspace/shared/
↪alexdevseed/landsat8/viz/Copernicus_30440_covars_cog_topo_stack.tif', 's3://maap-
↪ops-workspace/shared/alexdevseed/landsat8/viz/Copernicus_30542_covars_cog_topo_
↪stack.tif', 's3://maap-ops-workspace/shared/alexdevseed/landsat8/viz/Copernicus_
↪30543_covars_cog_topo_stack.tif', 's3://maap-ops-workspace/shared/alexdevseed/
↪landsat8/viz/Copernicus_30821_covars_cog_topo_stack.tif', 's3://maap-ops-workspace/
↪shared/alexdevseed/landsat8/viz/Copernicus_30822_covars_cog_topo_stack.tif', 's3://
↪maap-ops-workspace/shared/alexdevseed/landsat8/viz/Copernicus_30823_covars_cog_topo_
↪stack.tif', 's3://maap-ops-workspace/shared/alexdevseed/landsat8/viz/Landsat8_30542_
↪comp_cog_2015-2020_dps.tif', 's3://maap-ops-workspace/shared/alexdevseed/landsat8/
↪viz/Landsat8_30543_comp_cog_2015-2020_dps.tif', 's3://maap-ops-workspace/shared/
↪alexdevseed/landsat8/viz/Landsat8_30822_comp_cog_2015-2020_dps.tif', 's3://maap-ops-
↪workspace/shared/alexdevseed/landsat8/viz/Landsat8_30823_comp_cog_2015-2020_dps.tif
↪']
```

[16]:
```
mosaicdata = MosaicJSON.from_urls(tiles, minzoom=1, maxzoom=16)
```

3.5.7 Using mosaicJSON with titiler

There are 2 options for using mosaicJSON with titiler:

1. (Preferred) Post mosaicJSON to titiler `mosaics` endpoint and use the `mosaicjson/mosaics` endpoint for dynamic tiling.
2. Upload mosaicJSON to S3 and pass the S3 url to the titiler `mosaicjson/tiles` endpoint.

3.5.8 Post mosaicJSON to titiler

[17]:
```
import requests
from pprint import pprint

r = requests.post(
    url=f"{titiler_url}/mosaics",
    headers={
        "Content-Type": "application/vnd.titiler.mosaicjson+json",
    },
    json=mosaicdata.dict(exclude_none=True)).json()

pprint(r)
```

```
{'id': 'edf63f99-84dc-447c-afec-90b7109feab4',
 'links': [{'href': 'https://titiler.maap-project.org/mosaics/edf63f99-84dc-447c-afec-
↪90b7109feab4',
            'rel': 'self',
            'title': 'Self',
            'type': 'application/json'},
           {'href': 'https://titiler.maap-project.org/mosaics/edf63f99-84dc-447c-afec-
↪90b7109feab4/mosaicjson',
            'rel': 'mosaicjson',
            'title': 'MosaicJSON',
            'type': 'application/json'},
           {'href': 'https://titiler.maap-project.org/mosaics/edf63f99-84dc-447c-afec-
↪90b7109feab4/tilejson.json',
            'rel': 'tilejson',
            'title': 'TileJSON',
            'type': 'application/json'},
           {'href': 'https://titiler.maap-project.org/mosaics/edf63f99-84dc-447c-afec-
↪90b7109feab4/tiles/{z}/{x}/{y}',
            'rel': 'tiles',
            'title': 'Tiles',
            'type': 'application/json'},
           {'href': 'https://titiler.maap-project.org/mosaics/edf63f99-84dc-447c-afec-
↪90b7109feab4/WMTSCapabilities.xml',
            'rel': 'wmts',
            'title': 'WMTS',
            'type': 'application/json'}]}
```

[18]:
```
mosaic_link = list(filter(lambda x: x['rel'] == 'tiles', r['links']))[0]['href']
wmts_mosaic_link = list(filter(lambda x: x['rel'] == 'wmts', r['links']))[0]['href']
```

3.5.9 Test your mosaic

[19]:
```
params = {
    "tile_format": "png",
    "bidx": bidx, # Select which band to use
    "resampling_method":"nearest",
    "rescale": rescale,
    "return_mask": "true",
    "colormap_name": colormap_name # Any colormap from matplotlib will work
}

wmts_call = "?".join([wmts_mosaic_link, urllib.parse.urlencode(params)])

w.load_layer_config(wmts_call, "wmts/xml")
```

By default, the image will be displayed in greyscale if no `colormap_name` parameter is passed to the titiler API. Guidance below is provided to help determine what a valid colormap_name might be and how to create a legend for the dashboard.

Dashboard ColorRamps & Legends

When using the dashboard, there 2 components for implementing a color scheme for your map. There is the map render and there is the legend.

> Titiler used for Cloud Optimized Geotiff (COG) rendering accepts any color scheme from the python matplotlib library, and custom color formulas.

- Rio Tiler Colors
- Matplotlib Colors

Available `colormap_name` values for titiler: `above, accent, accent_r, afmhot, afmhot_r, autumn, autumn_r, binary, binary_r, blues, blues_r, bone, bone_r, brbg, brbg_r, brg, brg_r, bugn, bugn_r, bupu, bupu_r, bwr, bwr_r, cfastie, cividis, cividis_r, cmrmap, cmrmap_r, cool, cool_r, coolwarm, coolwarm_r, copper, copper_r, cubehelix, cubehelix_r, dark2, dark2_r, flag, flag_r, gist_earth, gist_earth_r, gist_gray, gist_gray_r, gist_heat, gist_heat_r, gist_ncar, gist_ncar_r, gist_rainbow, gist_rainbow_r, gist_stern, gist_stern_r, gist_yarg, gist_yarg_r, gnbu, gnbu_r, gnuplot, gnuplot2, gnuplot2_r, gnuplot_r, gray, gray_r, greens, greens_r, greys, greys_r, hot, hot_r, hsv, hsv_r, inferno, inferno_r, jet, jet_r, magma, magma_r, nipy_spectral, nipy_spectral_r, ocean, ocean_r, oranges, oranges_r, orrd, orrd_r, paired, paired_r, pastel1, pastel1_r, pastel2, pastel2_r, pink, pink_r, piyg, piyg_r, plasma, plasma_r, prgn, prgn_r, prism, prism_r, pubu, pubu_r, pubugn, pubugn_r, puor, puor_r, purd, purd_r, purples, purples_r, rainbow, rainbow_r, rdbu, rdbu_r, rdgy, rdgy_r, rdpu, rdpu_r, rdylbu, rdylbu_r, rdylgn, rdylgn_r, reds, reds_r, rplumbo, schwarzwald, seismic, seismic_r, set1, set1_r, set2, set2_r, set3, set3_r, spectral, spectral_r, spring, spring_r, summer, summer_r, tab10, tab10_r, tab20, tab20_r, tab20b, tab20b_r, tab20c, tab20c_r, terrain, terrain_r, twilight, twilight_r, twilight_shifted, twilight_shifted_r, viridis, viridis_r, winter, winter_r, wistia, wistia_r, ylgn, ylgn_r, ylgnbu, ylgnbu_r, ylorbr, ylorbr_r, ylorrd, ylorrd_r`

Example 1: Class based known colors

In this example, the raster represents classes of forest with 11 possible values. There are specific colors selected to correspond to each class. We combine the list of colors and the list of classes and format them for the legend parameter the dashboard needs.

https://github.com/MAAP-Project/dashboard-datasets-maap/blob/main/datasets/taiga-forest-classification.json

```
[ ]: colors = [
        '#5255A3','#1796A3','#FDBF6F','#FF7F00', '#FFFFBF','#D9EF8B','#91CF60','#1A9850',
     ↪'#C4C4C4','#FF0000', '#0000FF'
     ]
     labels = [
         'Sparse & Uniform',
         'Sparse & Diffuse-gradual',
         'Sparse & Diffuse-rapid',
         'Sparse & Abrupt ',
         'Open & Uniform ',
         'Open & Diffuse-gradual',
         'Open & Diffuse-rapid',
         'Open & Abrupt',
         'Intermediate & Closed',
         'Non-forest edge (dry)',
         'Non-forest edge (wet)'
     ]

     legend = [dict(color=colors[i],label=labels[i]) for i in range(0, len(colors))]
     print(json.dumps(legend, indent=2))

     # Copy and Paste the output below to your dashboard config.
```

Example 2: Discrete ColorRamp

In this example, the range of values is known, but the color scale has many non-sequential colors. Starting with the premade color list, we create a continuous color ramp that uses the known colors as stops points. Arbitrarly 12 breaks looked decent in the dashboard legend so we split it into 12 discrete colors. Then combine the list of values and colors into the correct json syntax.

https://github.com/MAAP-Project/dashboard-datasets-maap/blob/main/datasets/ATL08.json

```
[ ]: forest_ht = matplotlib.colors.LinearSegmentedColormap.from_list('forest_ht', ['#636363
     ↪','#FC8D59','#FEE08B','#FFFFBF','#D9EF8B','#91CF60','#1A9850','#005A32'], 12)
     cols = [matplotlib.colors.to_hex(forest_ht(i)) for i in range(forest_ht.N)]

     cats = range(0,25, (25//len(cols)))
     legend = [[cats[i],cols[i]] for i in range(0, len(cols))]
     text = (json.dumps(legend, separators=(',', ': ') ))

     print(text.replace('],[',']\n['))

     # Copy and Paste the output below to your dashboard config.
```

Example 3: Continuous ColorRamp

In this example, we are using a built in ColorRamp from matplotlib. So we just need to extract enough colors to fill the legend adequately, and convert the colors to hex codes.

https://github.com/MAAP-Project/dashboard-datasets-maap/blob/main/datasets/topo.json

```
[ ]: cmap_name = 'gist_earth_r'
cmap = matplotlib.cm.get_cmap(cmap_name, 12)
cols = [matplotlib.colors.to_hex(cmap(i)) for i in range(cmap.N)]
print(cols)

# Copy and Paste the output below to your dashboard config.
```

3.5.10 Step 3: Create and submit your dashboard dataset json

```
[ ]: # This example is for a continuous color ramps
dataset_id = "paraguay-estimated-biomass"
dataset_name = "Estimated Biomass in Paraguay"
dataset_type = "raster"
legend_type = "gradient-adjustable"
info = "Estimated biomass within 6km grids."
stops = cols
parameters = f"colormap_name={cmap_name}&rescale={band_min},{band_max}&bidx={bidx}"
```

```
[ ]: # Single COG
tiles_link = f"{{titiler_server_url}}/cog/tiles/{{z}}/{{x}}/{{y}}.png?url={location}&
↪{parameters}"
```

```
[ ]: tiles_link
```

```
[ ]: dataset_dict = {
    "id": dataset_id,
    "name": dataset_name,
    "type": dataset_type,
    "swatch": {
      "color": "#6976d7",
      "name": "Moody Blue"
    },
    "source": {
        "type": dataset_type,
        "tiles": [ tiles_link ]
    },
    "legend": {
      "type": legend_type,
       "min": band_min,
       "max": band_max,
      "stops": stops
    },
    "info": info
}
print(json.dumps(dataset_dict, indent=4))
```

3.5.11 Create a PR to the datasets repo

```
git clone git@github.com:MAAP-Project/biomass-dashboard-datasets.git
cd biomass-dashboard-datasets
git checkout -b ab/add-dataset
```

(continues on next page)

(continued from previous page)

```
# select and copy json above
echo <copied_json> >> datasets/paraguay-estimated-biomass.json
```

Add to the datasets list in config.yml

In `config.yml`:

```
DATASETS:
- paraguay-estimated-biomass.json
```

Add to the product or country pilot

In `country_pilots/paraguay/country_pilot.json`:

```
{
    "id": "paraguay",
    "label": "Paraguay",
    //...
    "datasets": [
        {
            "id": "paraguay-forest-mask"
        },
        {
            "id": "paraguay-tree-cover"
        },
        {
            "id": "paraguay-estimated-biomass"
        }
    ]
}
```

3.5.12 Add content to the summary

There should be a `summary.html` file corresponding to the product or country pilot you are working on, for example: `country_pilots/paraguay/summary.html`. Add or modify content in that file as appropriate.

3.5.13 Submit a PR

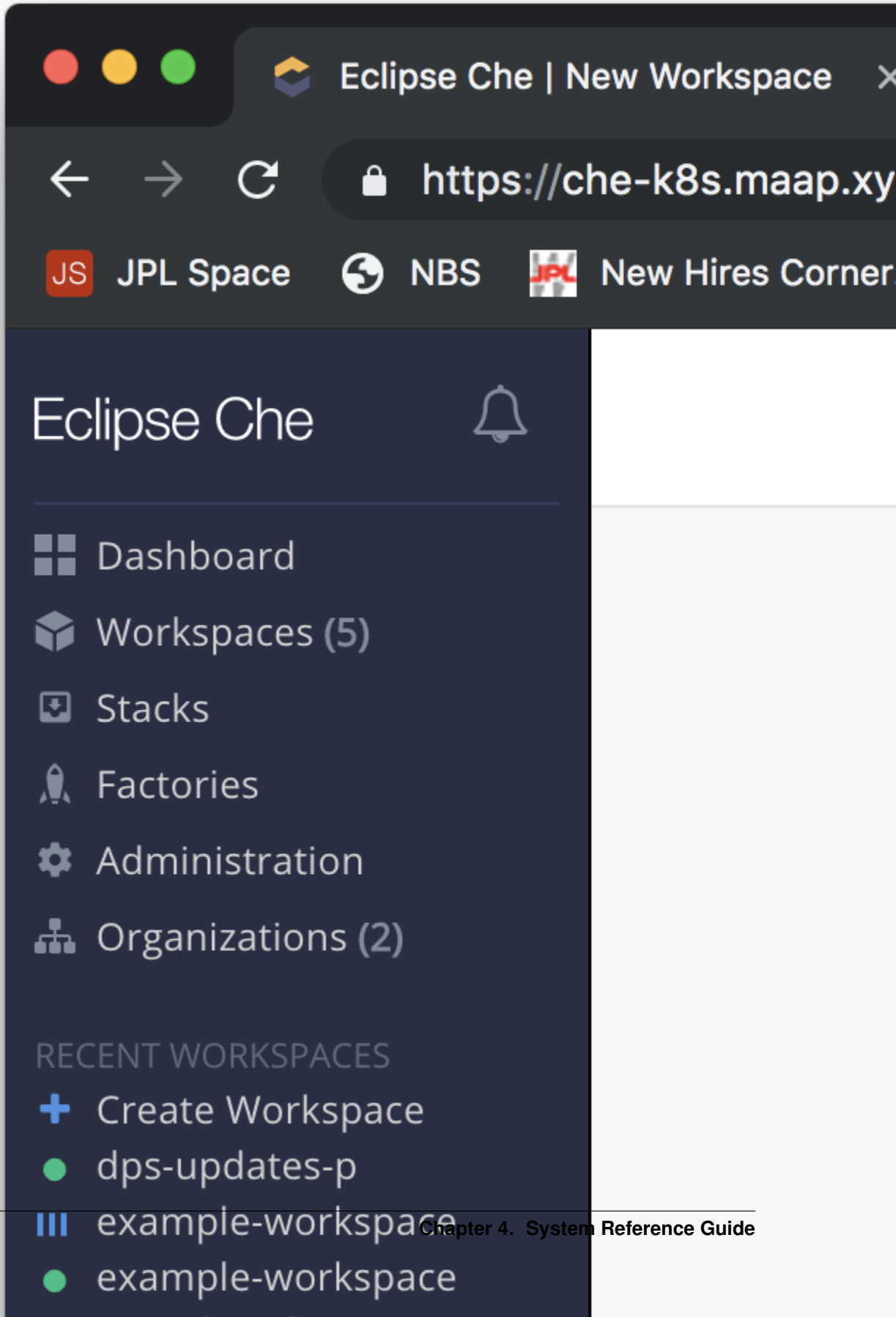
Once you have added the dataset json file and summary content, submit a PR to https://github.com/MAAP-Project/biomass-dashboard-datasets. A member of the data team will review the PR and when it is merged your content will appear in biomass.dit.maap-project.org.

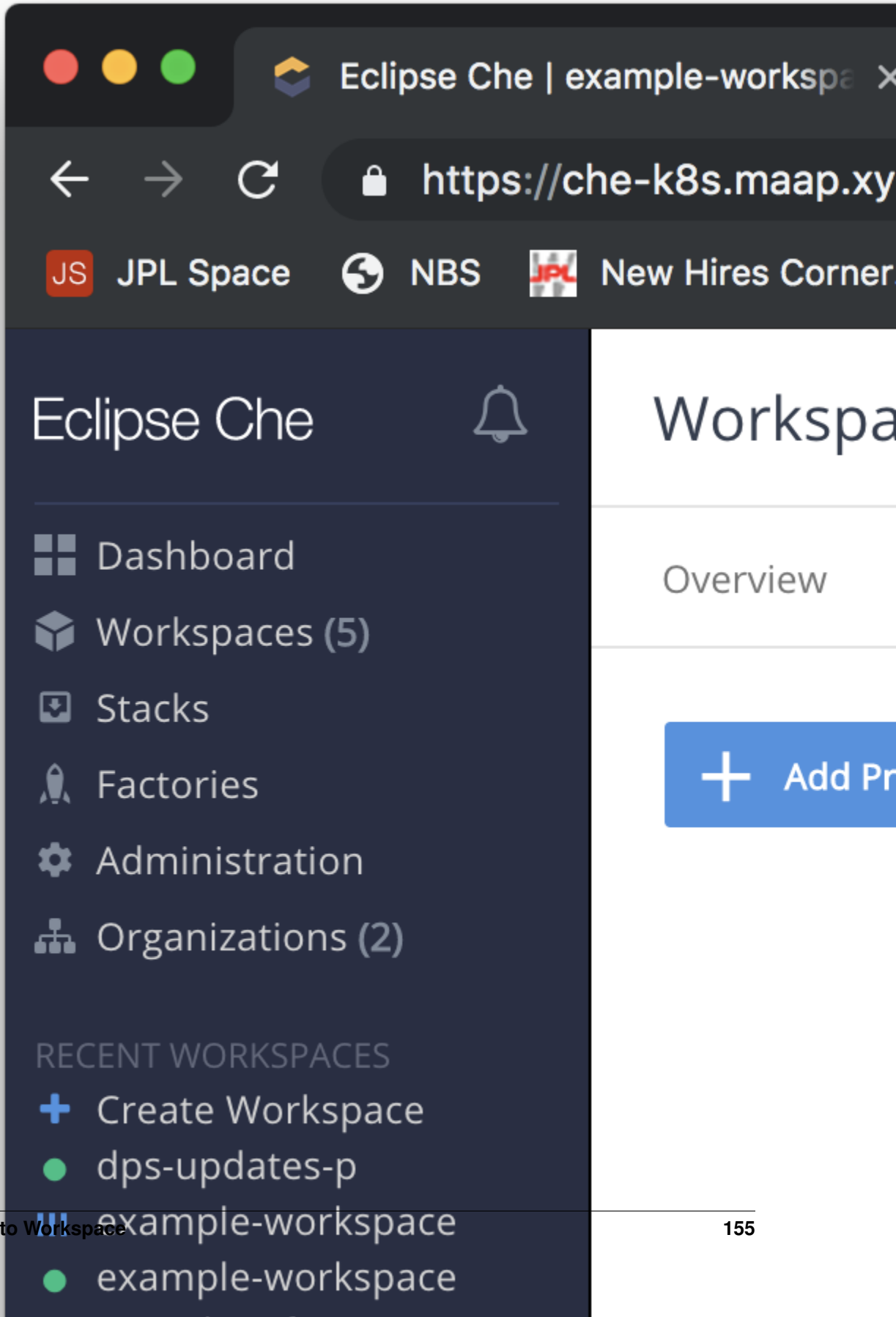
CHAPTER 4

System Reference Guide

4.1 Add Project to Workspace

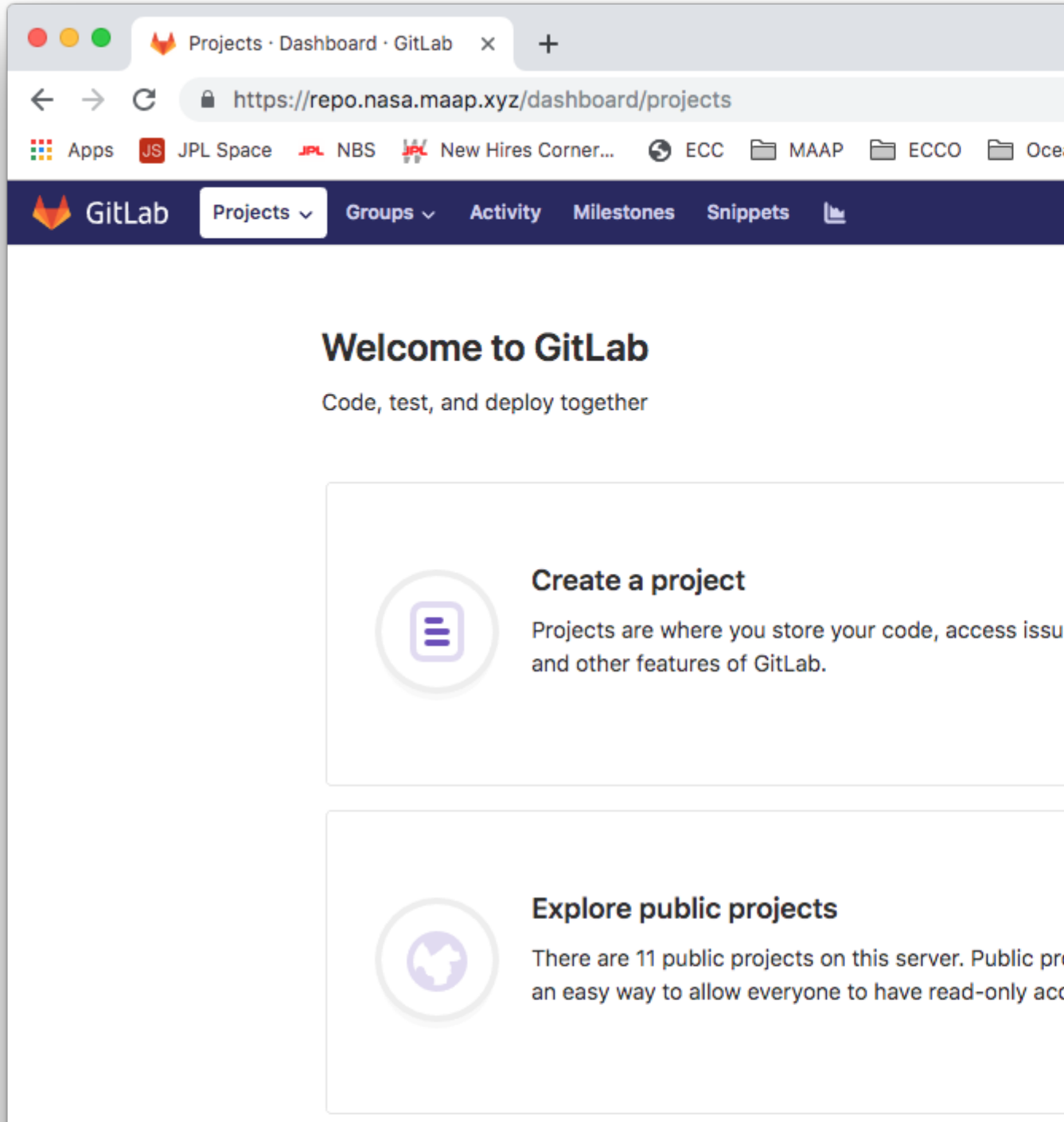
Projects can be added to a workspace during or after workspace creation. If a project is added after a workspace exists, the workspace must be restarted when the projects are added. After restart, your projects will automatically appear in your workspace.


Eclipse Che | New Workspace
https://che-k8s.maap.xy
JS JPL Space
NBS
New Hires Corner
Eclipse Che
Dashboard
Workspaces (5)
Stacks
Factories
Administration
Organizations (2)
RECENT WORKSPACES
Create Workspace
dps-updates-p
example-workspace
example-workspace


Eclipse Che | example-workspa
https://che-k8s.maap.xy
JS JPL Space
NBS
New Hires Corner
Eclipse Che
Dashboard
Workspaces (5)
Stacks
Factories
Administration
Organizations (2)
RECENT WORKSPACES
Create Workspace
dps-updates-p
example-workspace
example-workspace
Workspa
Overview
Add Pr

4.2 Create a Project

The first step in getting started with projects in MAAP is going to MAAP's GitLab and creating a new project. When you create a MAAP account, an account is automatically configured for you on MAAP's GitLab instance. To sign in, click on `Sign in with CAS`, then use your URS or ESA login. This will direct you to your projects.

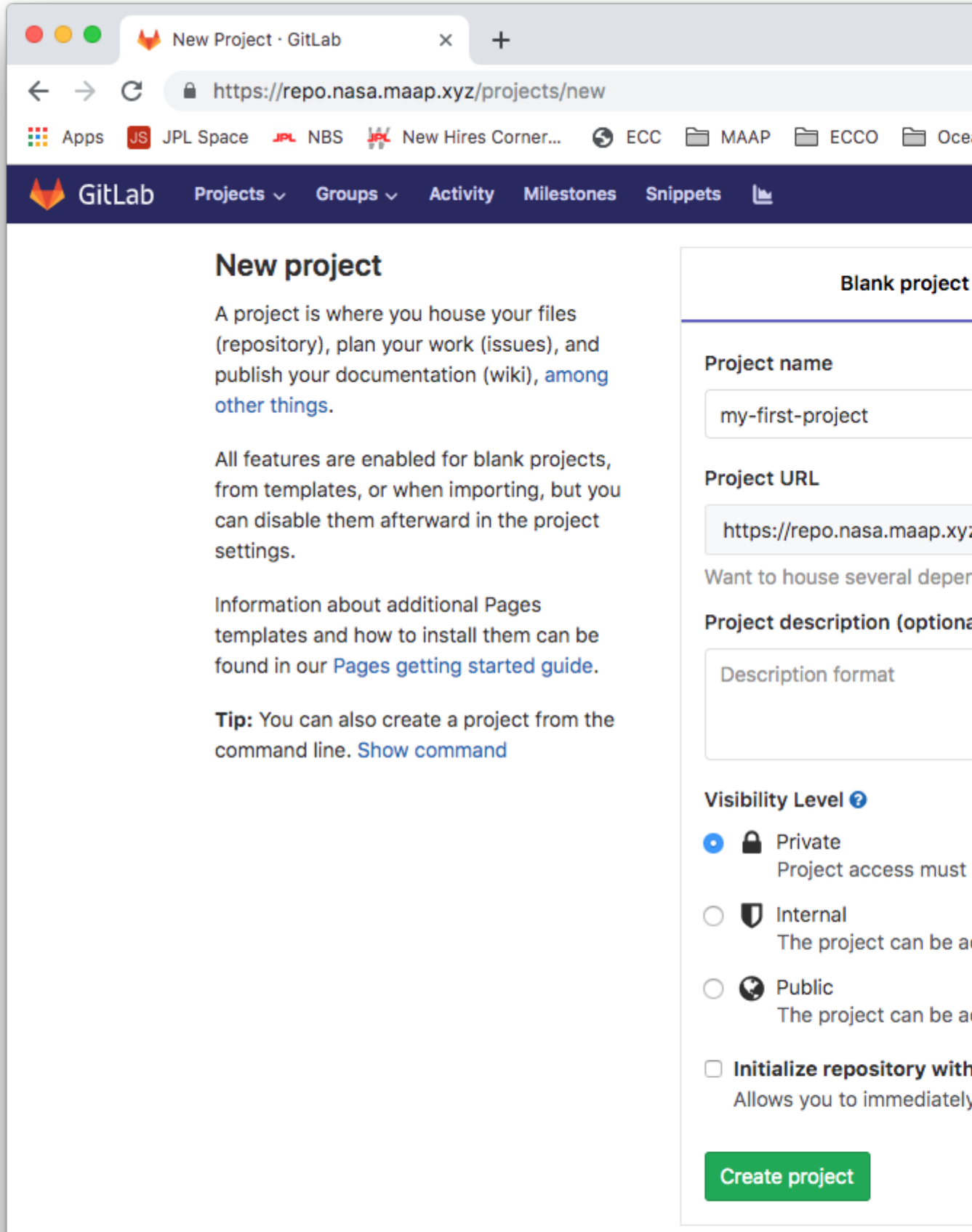

Projects · Dashboard · GitLab
https://repo.nasa.maap.xyz/dashboard/projects
Apps
JPL Space
NBS
New Hires Corner...
ECC
MAAP
ECCO
GitLab
Projects
Groups
Activity
Milestones
Snippets
Welcome to GitLab
Code, test, and deploy together
Create a project
Projects are where you store your code, access issu
and other features of GitLab.
Explore public projects
There are 11 public projects on this server. Public pr
an easy way to allow everyone to have read-only acc

Click on `Create a Project`. Name your project and select its visibility level.

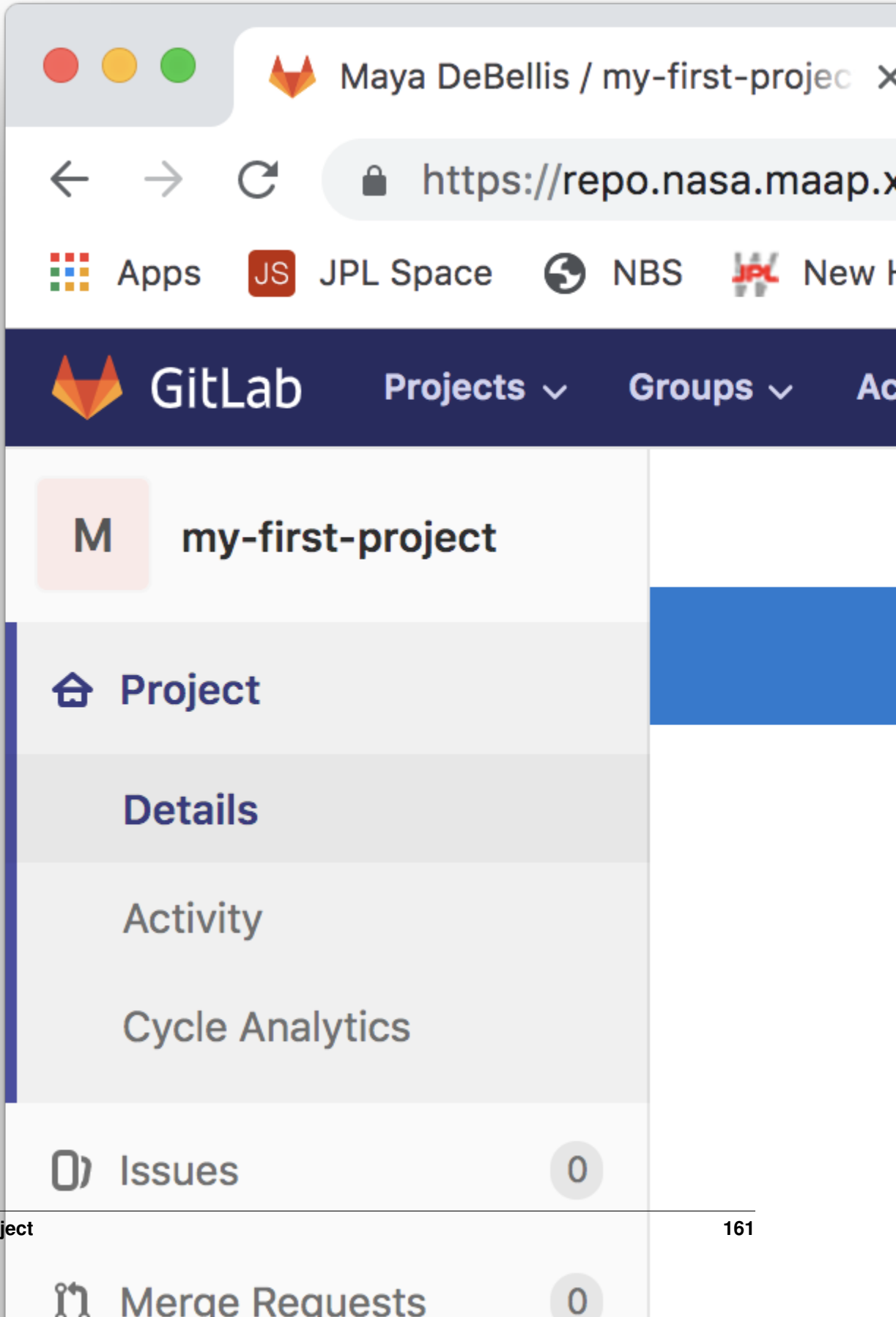
NOTE: Projects will default to private, meaning only you can see it, and if you want to allow another user to contribute, you will have to explicitly add them in the GitLab interface. Adding a user to a workspace that has a project in it does not automatically give them access to your GitLab project, you must also add them to the private project if you want to allow them to make changes.

Projects set to internal can be seen by any authenticated MAAP user. Projects set to public can be seen by anyone, regardless of whether they are a MAAP user or not.

Settings for your new project:

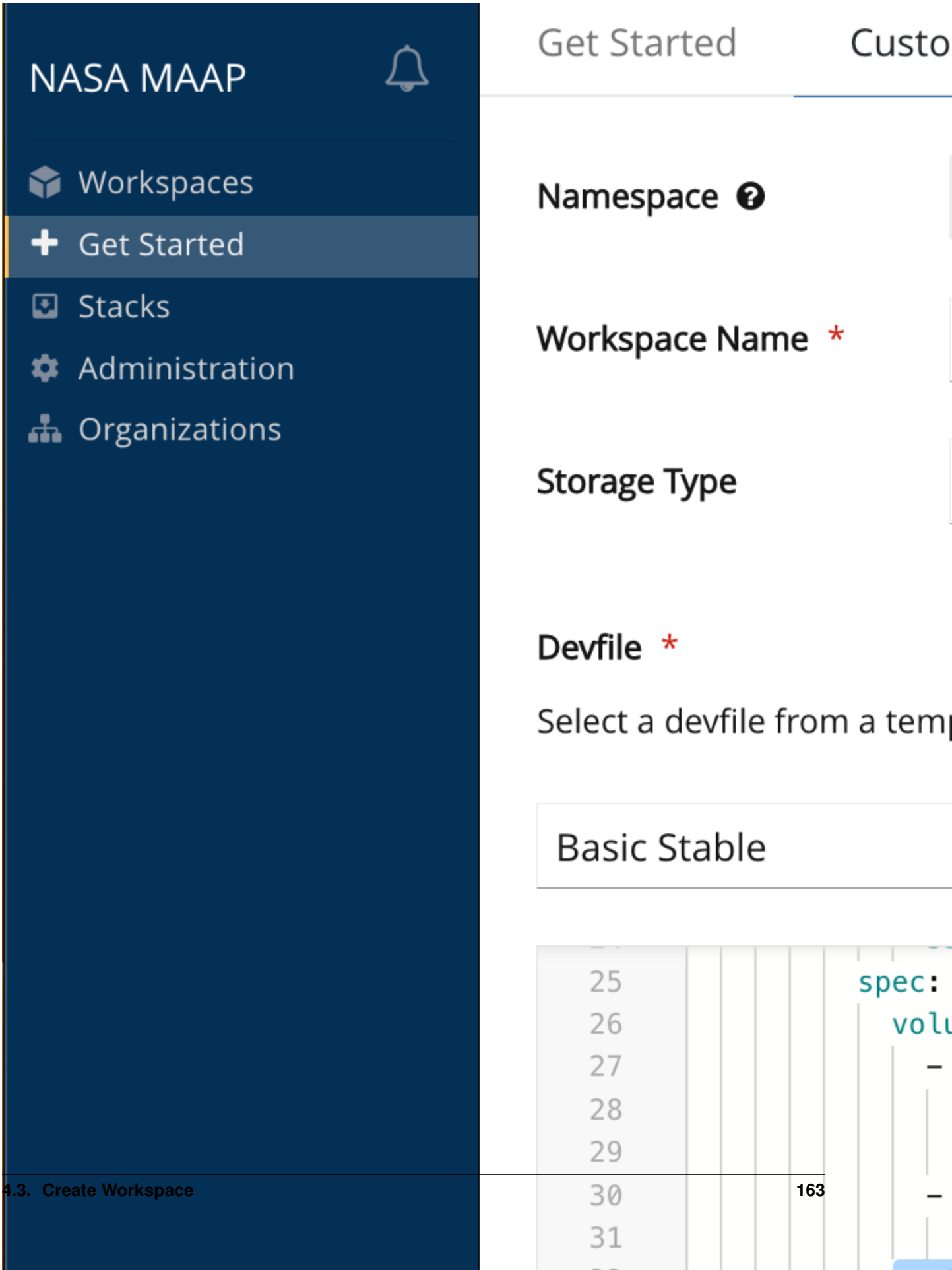

New Project · GitLab
https://repo.nasa.maap.xyz/projects/new
Apps
JPL Space
NBS
New Hires Corner...
ECC
MAAP
ECCO
GitLab
Projects
Groups
Activity
Milestones
Snippets
New project
A project is where you house your files (repository), plan your work (issues), and publish your documentation (wiki), among other things.
All features are enabled for blank projects, from templates, or when importing, but you can disable them afterward in the project settings.
Information about additional Pages templates and how to install them can be found in our Pages getting started guide.
Tip: You can also create a project from the command line. Show command
Blank project
Project name
my-first-project
Project URL
https://repo.nasa.maap.xyz
Want to house several deper
Project description (optiona
Description format
Visibility Level
Private
Project access must
Internal
The project can be a
Public
The project can be a
Initialize repository with
Allows you to immediately
Create project

Once you have created a project, copy the https url for the project that can be found under the clone button's dropdown. You will use this to add your project to your Che workspace.


Maya DeBellis / my-first-projec
https://repo.nasa.maap.x
Apps
JS
JPL Space
NBS
New
GitLab
Projects
Groups
Ac
M
my-first-project
Project
Details
Activity
Cycle Analytics
Issues
0
Merge Requests
0

4.3 Create Workspace

Navigate to the side panel tab `Workspaces` and click `Add Workspace`. On the Get Started tab, you can select a stack to get started quickly. If you would like a more customized approach, on the Custom Workspace tab you can name your workspace, select the storage type, and select a devfile template from available stacks or upload your own devfile. Here, you can edit the devfile under `containers/jupyter/resources/limits/memory` to alter the memory limit for your workspace.


NASA MAAP
Workspaces
Get Started
Stacks
Administration
Organizations
Get Started
Custo
Namespace
Workspace Name *
Storage Type
Devfile *
Select a devfile from a temp
Basic Stable
25
26
27
28
29
30
31
spec:
volu

If you want to edit the devfile or your workspace's memory limit after its creation, just click on the `Workspaces` side panel tab, then the workspace, then the Devfile tab at the top of the page.

4.4 In-line Notebook Magics

4.4.1 Load the inline magics in notebook

WARNING: This will not work for a python script. This is for use in notebooks only.

You must enable the extension in the notebook: `%load_ext dps_magic` You can run a command by entering `%` followed by a command and its arguments into a notebook cell.

4.4.2 Available Magics

You can list the available magics by running `%help` Output:

```
%capabilities
get information about MAAP API services

%list
list algorithms registered in MAS

%describe
describe the selected algorithm

%execute
submit a job to DPS using an algorithm registered in MAS

%status
check the status of a submitted job

%result
get the results for a completed job

%delete_algorithm
remove a registered algorithm from MAS

%delete_job
remove a completed job from DPS

%s3_url
get a presigned s3 url for an object

%help
print this help info
```

Pass the argument `help` to a command for more specific instructions on how to use it:

```
%execute help
```

Output: execute_help_menu

4.4.3 List

Cell: `%list`

Output: List algorithms

```
Algorithms:
    example_plot_ubuntu:1.0
    gdal_service:master
    george_demo_ubuntu:master
    george_esdis_demo_ubuntu:1.0
    liz-plot2_ubuntu:1.0
    plant_algo:master
    plant_test:master
    plot_algo:master
    plot_demo3:master
    plot_demo4:master
    plot_demo_ubuntu:master
    plot_example_ubuntu:1.0
    plot_test2_ubuntu:master
    plot_test3_ubuntu:master
    plot_test5_ubuntu:master
    plot_test_ubuntu:master
    plot_ubuntu:master
```

4.4.4 Describe

Cell: `%describe help`

Output: Describe Algorithm Help

Check the inputs required for an algorithm stored in MAS. You need to know your algorithm name and version.

Example Describe Call: `%describe plot_algo:master` Example Dictionary Call: `d = {'algo_id': 'plot_algo','version':'master'} %describe $d`

Cell: `%describe george_demo_ubuntu:master` OR

Cell:

`d = {'algo_id':'george_demo_ubuntu','version':'master'} %describe $d`

Output:

```
Algorithm: george_demo_ubuntu
Version: master
Identifier: george_demo_ubuntu:master
Input
    Title:  pass_number
    Identifier: pass_number
    DataType:   string

Input
    Title:  username
    Identifier: username
    DataType:   string
```

4.4.5 Execute

Cell: `%execute help`

Output: Execute Job Help

Execute a job through DPS. You need to know the parameters of your algorithm. Use %describe to check if you are unsure.

Example String Call: `%execute plot_algo:master(pass_number=3,username=liz)` Example Dictionary Call: `d = {'pass_number':'6','username':'liz','algo_id':'plot_algo', 'version':'master'} %execute $d`

Cell: `%execute plot_algo:master(pass_number=3,username=eyam)` OR

```
d = {'pass_number':'6','username':'eyam','algo_id':'plot_algo','version':
'master'} %execute $d
```

Output: `JobID is cd36b6ff-5477-4cbf-80b8-772a77a44c58`

Make sure you use your username, or else the Job ID will not be saved anywhere else and will not be displayed in the jobs table

4.4.6 Status

Cell: `%status help`

Output: Job Status Help

Check the status of a job in DPS. You need to know your job ID. Example Status Call: `%status ef6fde9e-0975-4556-b8a7-ee52e91d8e61`

Cell: `%status cd36b6ff-5477-4cbf-80b8-772a77a44c58`

Output: `JobID is cd36b6ff-5477-4cbf-80b8-772a77a44c58 Status: Succeeded`

4.4.7 Result

Cell: `%result help`

Output: Job Result Help

Check the result of a completed job in DPS. You need to know your job ID. Example Result Call:

```
`%result ef6fde9e-0975-4556-b8a7-ee52e91d8e61`
```

Cell: `%result cd36b6ff-5477-4cbf-80b8-772a77a44c58`

Output: Job Results

```
JobID:  cd36b6ff-5477-4cbf-80b8-772a77a44c58
ProductName:    output-2019-09-13_00-21-53
Locations:  http://maapdev.s3.amazonaws.com/products/plot/v1.0/2019/09/13/output-2019-
↪09-13_00-21-53
• s3://s3.amazonaws.com:80/maapdev/products/plot/v1.0/2019/09/13/output-2019-09-13_00-
↪21-53
• https://s3.console.aws.amazon.com/s3/buckets/maapdev/products/plot/v1.0/2019/09/13/
↪output-2019-09-13_00-21-53/?region=us-east-1&tab=overview
```

the call will return an error if the job was not successfully completed (includes started, revoked, failed, and deleted)

4.4.8 Delete

Cell: `%delete_job help`

Output: Delete Job Help

Delete a finished (completed or failed) job or queued job stored in DPS. You need to know your Job ID. Example Delete Call: `%delete_job ef6fde9e-0975-4556-b8a7-ee52e91d8e61`

Cell: `%delete_job 5bcac3c8-9958-4c54-bb73-99d5eba09879`

Output: `JobID is 5bcac3c8-9958-4c54-bb73-99d5eba09879 Status: Deleted`

4.4.9 Dismiss

Cell: `%dismiss help`

Output: Dismiss Job Help

Dismiss a running (started, NOT queued) job on DPS. You need to know your Job ID. Example Dismiss call: `%dismiss_job ef6fde9e-0975-4556-b8a7-ee52e91d8e61'`

4.4.10 s3 Url

Cell: `%s3_url help`

Output: Presigned S3 Url Help

Get a presigned s3 url for an object. You need to know the path of the file relative to the bucket or your /projects directory. Example s3_url call: `%s3_url eyam/file-on-bucket`

Cell: `%s3_url eyam/file-on-bucket`

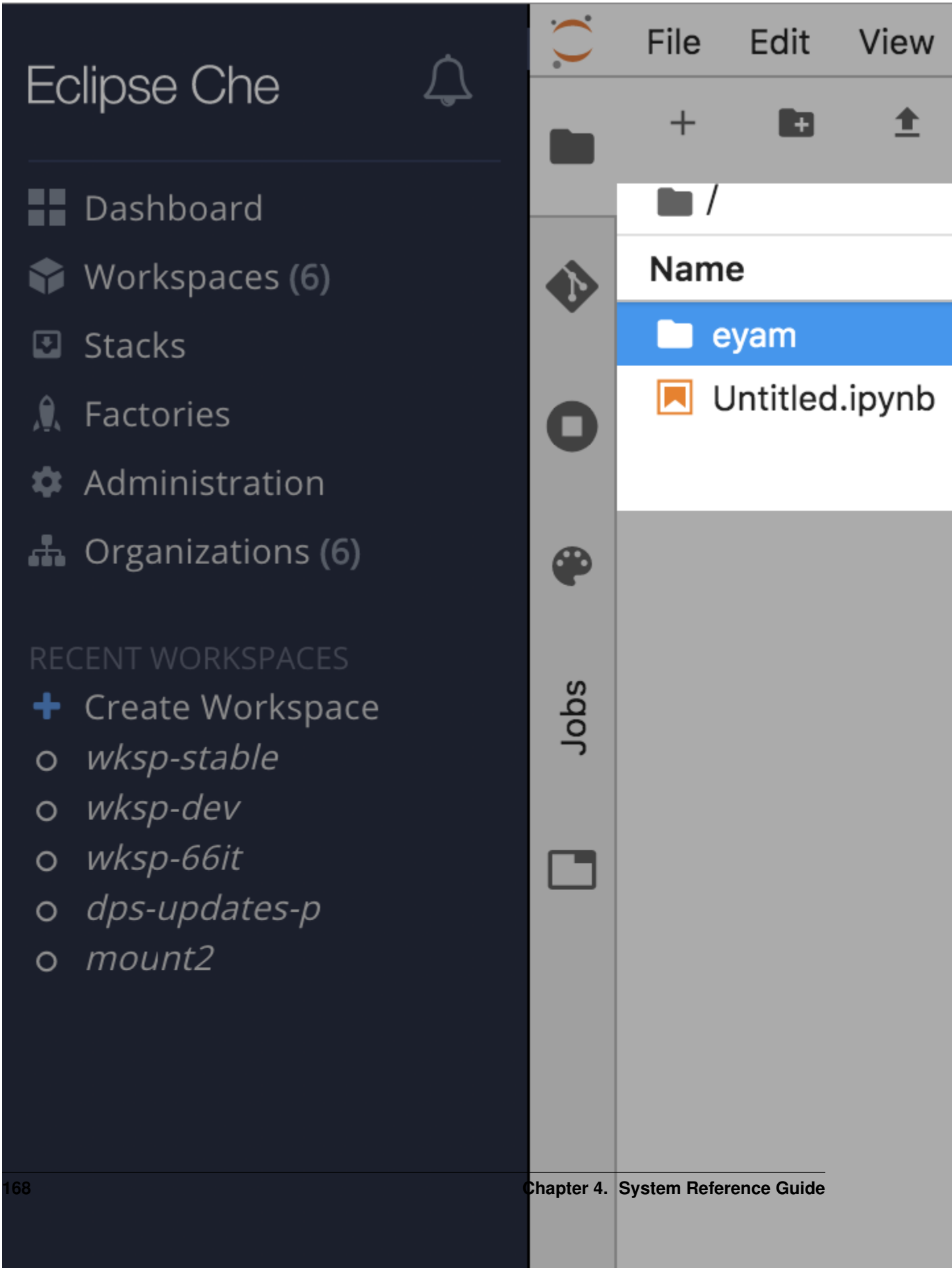
Output: `https://maap-mount-dev.s3.amazonaws.com/eyam/file-on-bucket?AWSAccessKeyId=AKIAJTSO23RLALBN45NQ&Signature=vgK%2FvwuSSYc4WUf39o4KWtbdVtw%3D&Expires=156`

4.5 Share Data

Users who have access to a workspace have access to all the files contained in that workspace.

All users have their own personal s3-bucket folder mounted in the FileBrowser home In `/projects`, each user has a personal s3-hosted folder with the same name as their CAS username. Files in this folder are automatically uploaded to s3 and are accessible from any workspace a user signs into.

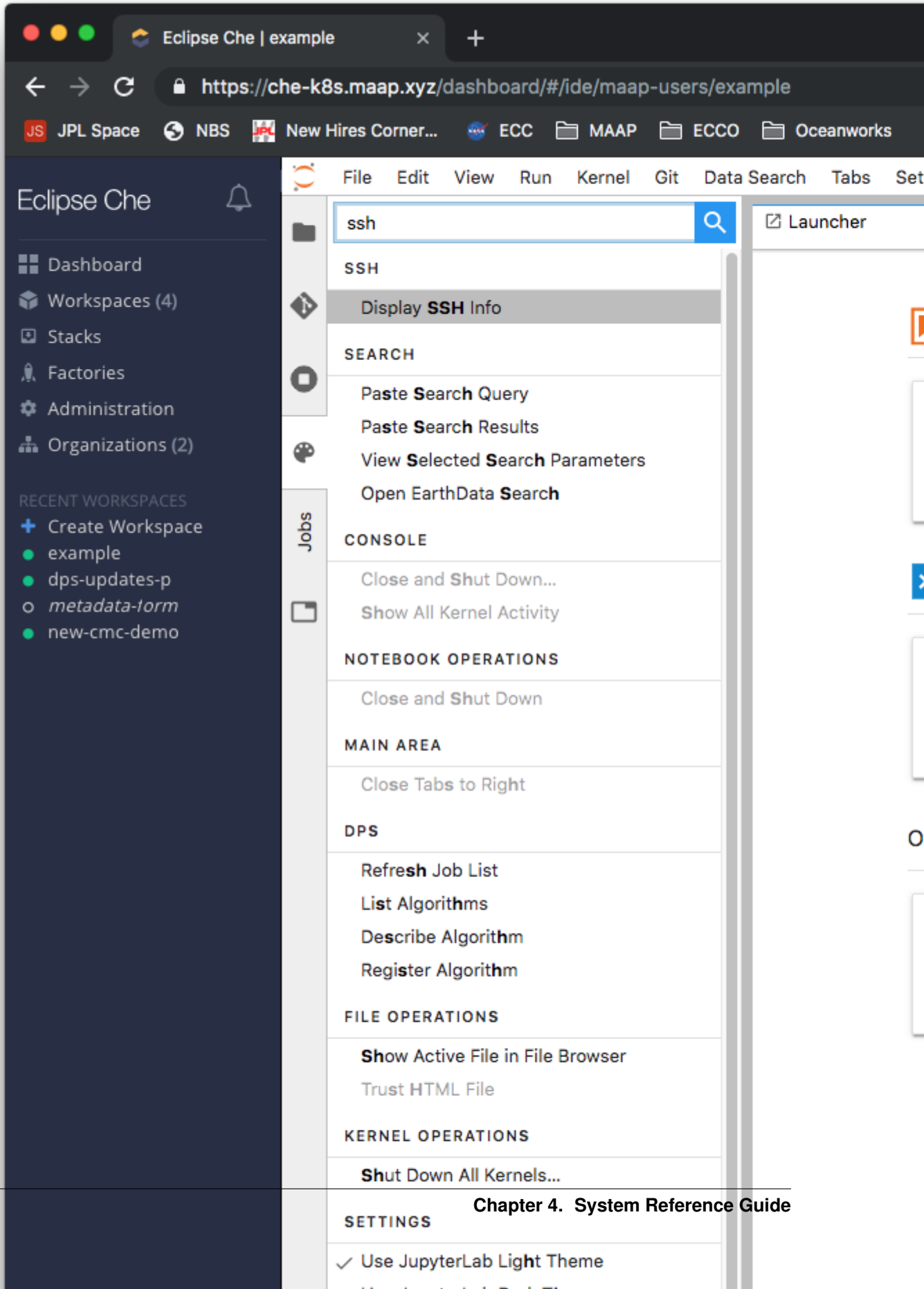
The intention of this mounted folder is that you can use this to share data with others, and also store files you want to access across different workspaces. It is not intended that you do all of your work in this directory. Because this directory is mounted to s3, you will notice that processes are slower when working in this directory.

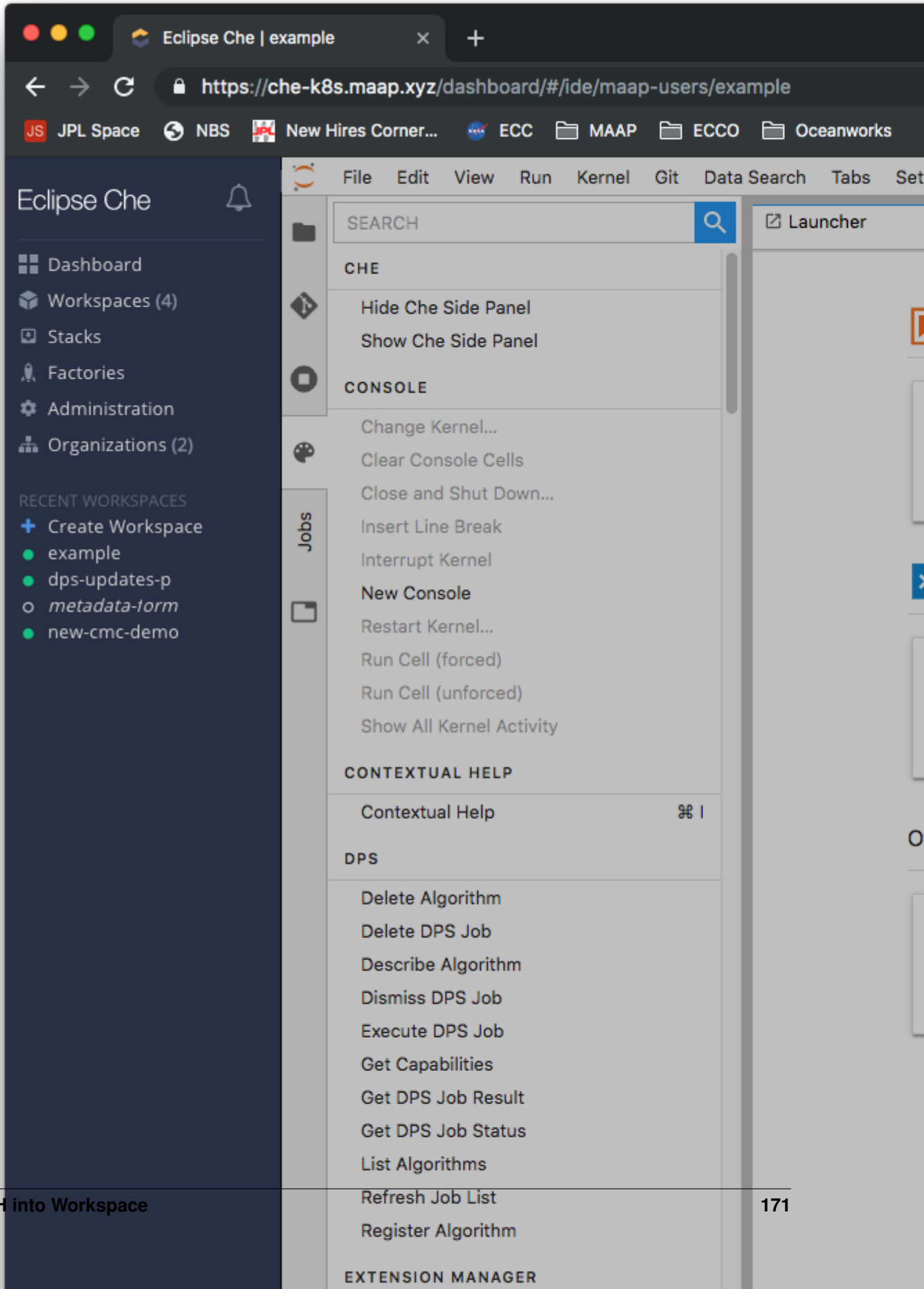

Eclipse Che
Dashboard
Workspaces (6)
Stacks
Factories
Administration
Organizations (6)
RECENT WORKSPACES
Create Workspace
wksp-stable
wksp-dev
wksp-66it
dps-updates-p
mount2
File
Edit
View
/
Name
eyam
Untitled.ipynb
Jobs

Users can create a shareable link for any files in their folder that is hosted on s3. To do this, go to `Command Palette -> User -> Get Presigned S3 URL` and enter the relative path to the file you want to share. The link will expire after 12 hours.

4.6 SSH into Workspace

As an alternative to using the jupyter interface, you can SSH directly into the container that your workspace set up. In order to get the IP and port information, navigate to the command palette of the jupyter interface. Find the command `Display SSH Info`, which will display the information you need (you can easily search for SSH). Your public SSH key that you added to your account will be added to any workspace you create. If you did not upload an SSH key to your profile, you will not be able to SSH in and must use the jupyter interface.


Eclipse Che | example
https://che-k8s.maap.xyz/dashboard/#/ide/maap-users/example
JPL Space
NBS
New Hires Corner...
ECC
MAAP
ECCO
Oceanworks
Eclipse Che
Dashboard
Workspaces (4)
Stacks
Factories
Administration
Organizations (2)
RECENT WORKSPACES
Create Workspace
example
dps-updates-p
metadata-torm
new-cmc-demo
File
Edit
View
Run
Kernel
Git
Data Search
Tabs
ssh
Launcher
Jobs
SSH
Display SSH Info
SEARCH
Paste Search Query
Paste Search Results
View Selected Search Parameters
Open EarthData Search
CONSOLE
Close and Shut Down...
Show All Kernel Activity
NOTEBOOK OPERATIONS
Close and Shut Down
MAIN AREA
Close Tabs to Right
DPS
Refresh Job List
List Algorithms
Describe Algorithm
Register Algorithm
FILE OPERATIONS
Show Active File in File Browser
Trust HTML File
KERNEL OPERATIONS
Shut Down All Kernels...
SETTINGS
Use JupyterLab Light Theme


Eclipse Che | example
https://che-k8s.maap.xyz/dashboard/#/ide/maap-users/example
JPL Space
NBS
New Hires Corner...
ECC
MAAP
ECCO
Oceanworks
Eclipse Che
Dashboard
Workspaces (4)
Stacks
Factories
Administration
Organizations (2)
RECENT WORKSPACES
Create Workspace
example
dps-updates-p
metadata-form
new-cmc-demo
File
Edit
View
Run
Kernel
Git
Data Search
Tabs
SEARCH
Launcher
CHE
Hide Che Side Panel
Show Che Side Panel
CONSOLE
Change Kernel...
Clear Console Cells
Close and Shut Down...
Insert Line Break
Interrupt Kernel
New Console
Restart Kernel...
Run Cell (forced)
Run Cell (unforced)
Show All Kernel Activity
CONTEXTUAL HELP
Contextual Help
DPS
Delete Algorithm
Delete DPS Job
Describe Algorithm
Dismiss DPS Job
Execute DPS Job
Get Capabilities
Get DPS Job Result
Get DPS Job Status
List Algorithms
Refresh Job List
Register Algorithm
EXTENSION MANAGER
Jobs

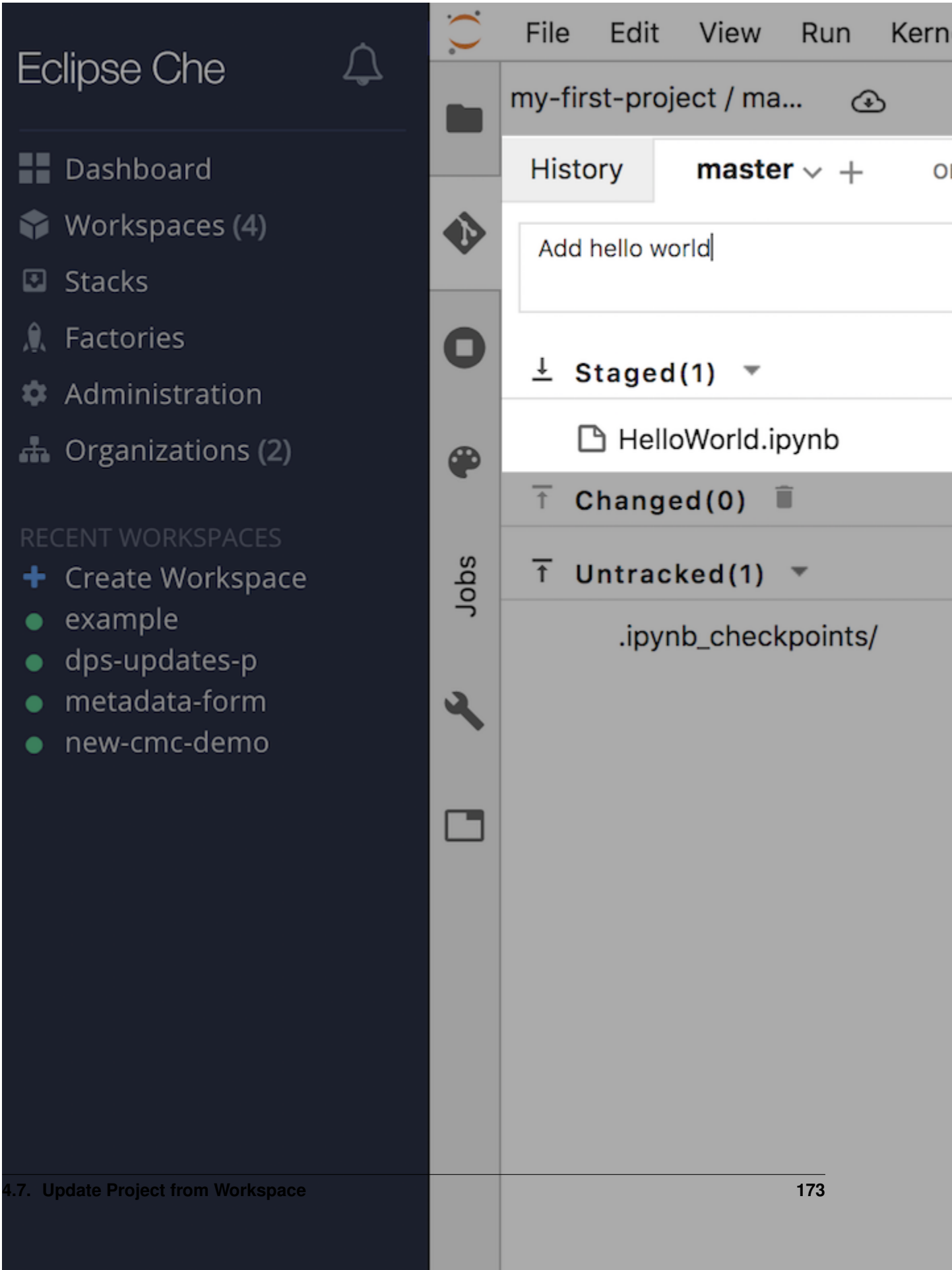
4.7 Update Project from Workspace

All projects imported from GitLab can be found in `/projects`.

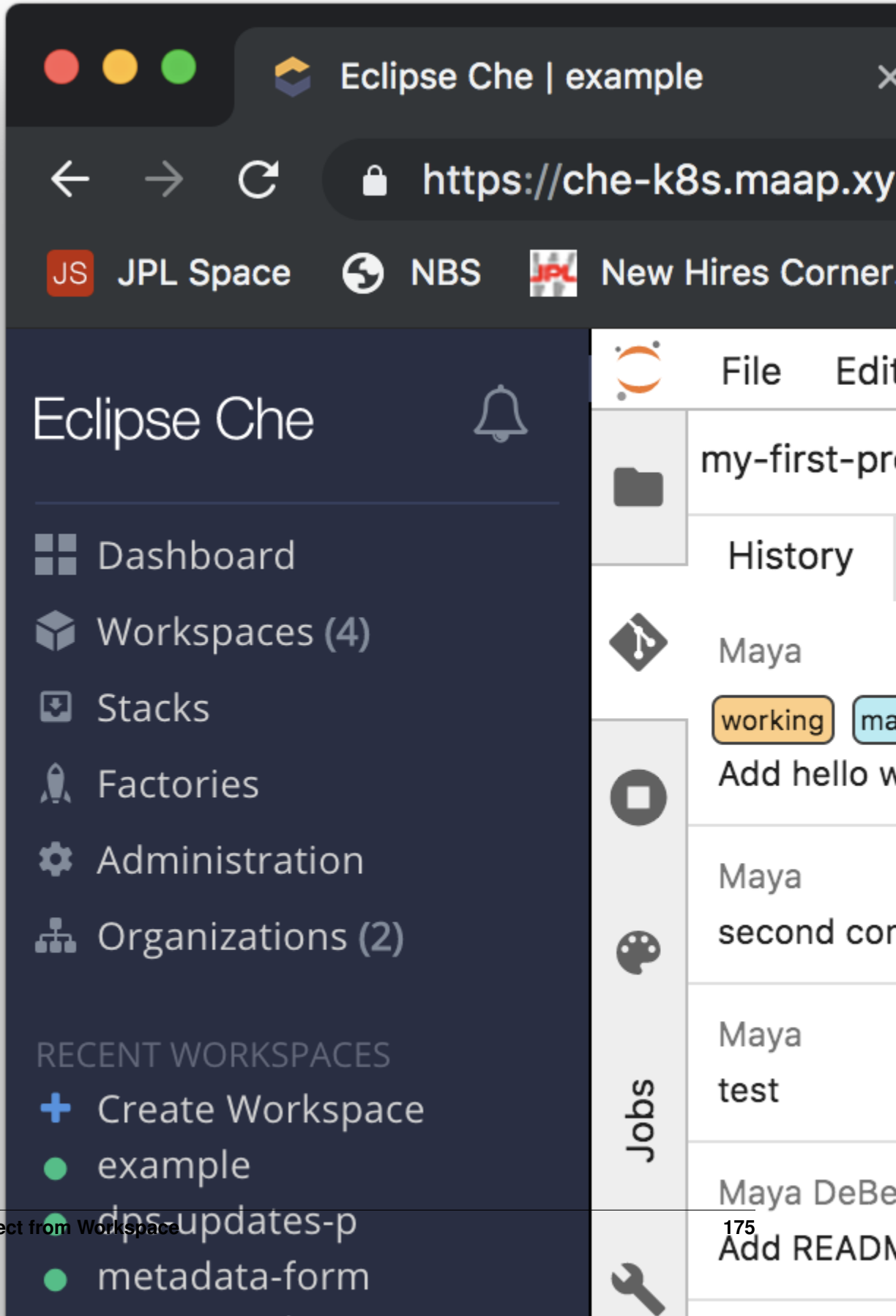
Using the left side panel in the jupyter interface, you can push changes to your gitlab project.

If you are more comfortable using the command line to interact with git, you do not need to use the side panel. It will work the same way in the terminal, once you navigate to the project's filepath.

When you are ready to update your project with your changes, navigate to the git panel. Add the files you want to change to the list of staged changes. Then write a commit message, and click the check. Now you need to push your changes by selecting the push changes button on the toolbar.


Eclipse Che
Dashboard
Workspaces (4)
Stacks
Factories
Administration
Organizations (2)
RECENT WORKSPACES
Create Workspace
example
dps-updates-p
metadata-form
new-cmc-demo
File
Edit
View
Run
Kern
my-first-project / ma...
History
master
Add hello world
Staged(1)
HelloWorld.ipynb
Changed(0)
Untracked(1)
.ipynb_checkpoints/
Jobs

If you want to check your commit history, look at branches, and confirm your updates have been pushed, you can see this on the history tab.

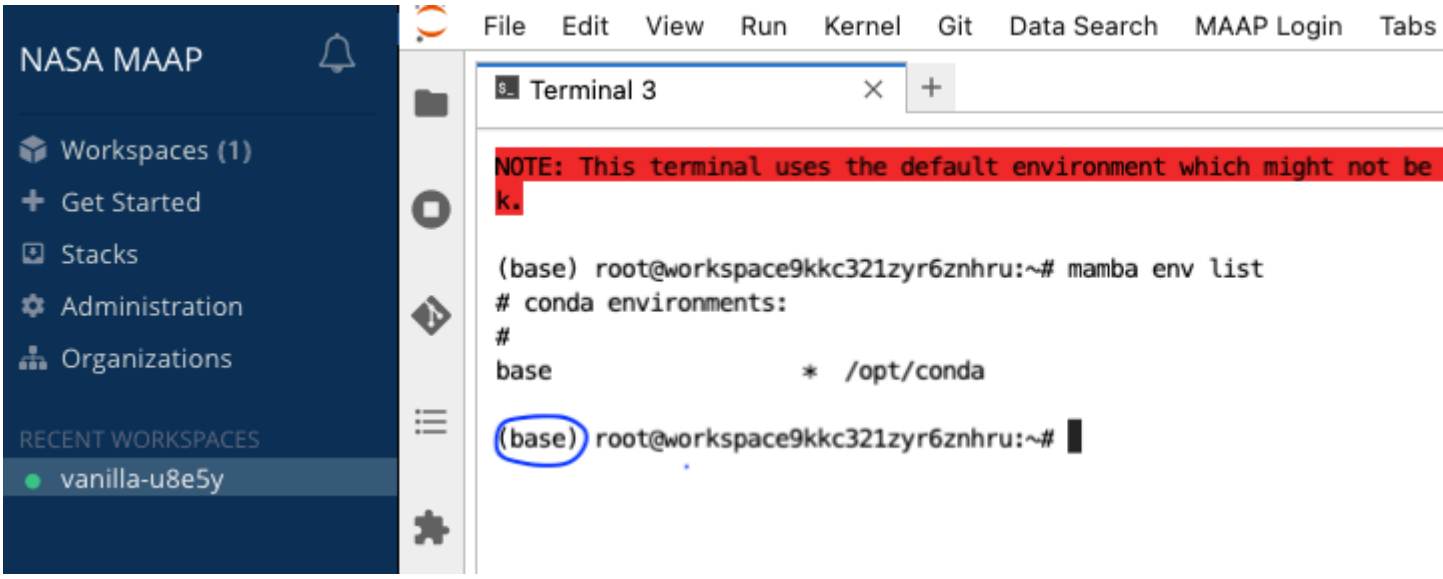

Eclipse Che | example
https://che-k8s.maap.xy
JS JPL Space
NBS
New Hires Corner
Eclipse Che
Dashboard
Workspaces (4)
Stacks
Factories
Administration
Organizations (2)
RECENT WORKSPACES
Create Workspace
example
dps-updates-p
metadata-form
File
Edit
my-first-pr
History
Maya
working
Add hello w
Maya
second cor
Maya
test
Maya DeBe
Add READM
Jobs

4.8 Environments

This document guides MAAP users in the process of selecting, extending existing environments (the set of libraries availables for analysis) or creating custom environments.

4.8.1 Workspaces

The MAAP ADE offers various workspace options, each workspace coming with its own environment that has pre-installed essential libraries for computing and geospatial analysis. At the time of writing this guide, here are the options :



For example, the `MAAP RGEDI Stable` and `MAAP R Stable` workspace options come with various pre-installed R packages.

For more information : Each of these options rely on Docker images that were build off from Dockerfiles that are publicly available in the MAAP workspace repository. If you want to learn more about what libraries each image contains, check out this repository.

4.8.2 Extending environments

Users may need libraries for their specific analysis purposes that are not present in the environments of the different workspace options offered. In this case, ideally, the steps should be the following :

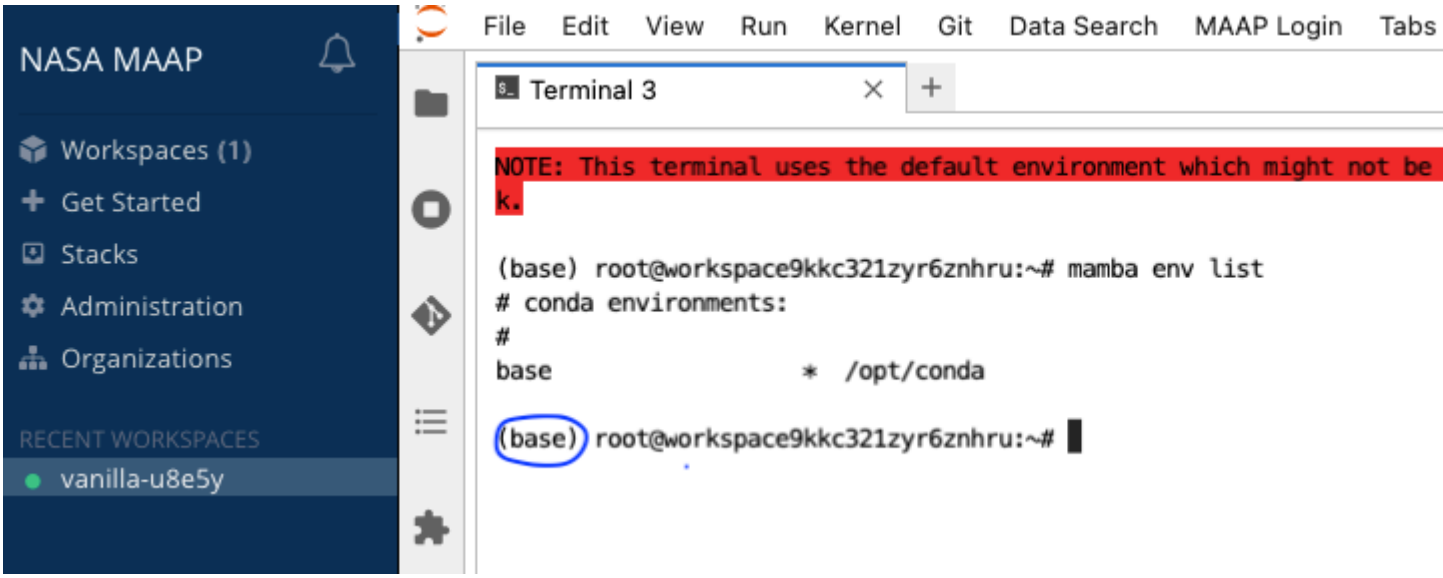
1. The user explores her/his environment need by extending the environment of an existing workspace or creating her/his custom environment in an existing workspace (see next sections).
2. Once that is done, the user submits a ticket/coordinates with the platform team to create a new workspace option with the requested, finalized environment.

The above approach is ideal because modifications to the pre-defined workspace environment do not survive a workspace restart (see next sections), and because sharing new experimented environments is valuable.

The next sections explain how to extend environments or create custom environments, and for this, introduces information regarding which environment management solution we are using.

Package manager

We use `mamba` (a fast `conda` drop-in replacement) as a package manager to install, update or remove packages (libraries). `mamba` works with 'environments' that are directories in your local file system containing a set of packages. When you work 'in a given environment', it means you that your programs will look for dependencies in that environment's `mamba` directory. All workspaces launch with a environment called `base`, which is a `mamba` environment that has all the pre-installed libraries. If you open a terminal launcher after creating a `Basic Stable` workspace :



You can notice that a `base` `mamba` environment is activated, and its libraries are located in `/opt/conda`.

Extending the `base` environment in a given workspace session.

Note : any modification to the ''base'' environment does not survive a workspace restart. In other words, modifications to ''/opt/conda'' disappear after a workspace restart.

Extending an existing `mamba` environment means adding packages on top of what it contains, which works provided there are no dependency conflicts. You can use a configuration file specifying all the new libraries to add. It is recommended for reproducibility and shareability. See the below sections for configuration file usage.

Alternatively, you can install libraries using the `mamba install` command to install additional packages in your current environment (run `mamba --help` to learn more about how to use `mamba` commands). For example :

```
mamba install xarray
```

4.8.3 Custom environments

You can use the `mamba` CLI to create a new, custom environment.

The parameters (the list of libraries, the location where to search for them, etc...) can be passed either from a configuration YAML file or directly on the console. We recommend using the first option (a YAML file is easier to share and modify).

Basic custom environment

Here is an example configuration file that we name `env` :

```
# env.yml
name: env
channels:
    - conda-forge
dependencies:
    - python=3.8
    - pandas=1.5.3
    - geopandas=0.12.2
```

It installs specific versions `python`, `pandas` and `geopandas` from either `conda-forge` or `defaults`. If versions aren't specified, the latest is installed. We recommend to always specify the version for reproducibility. The basic command to create this environment would be :

```
mamba env create -f env.yml
```

However, this stores this environment files in `/opt/conda`, which a directory that is reset when the workspace restarts, and so custom environments are lost. Therefore, you want to specify a storage location in your user directory with the `--prefix` parameter

```
mamba env create -f env.yml --prefix /projects/env
```

and to activate it :

```
mamba activate env
```

Updating an existing environment with a configuration file

For this section and the next, you can find example configuration files in `example_conda_configuration_files` in the same folder as this notebook.

You can update an existing environment with a configuration file as well. For example, let's assume you have a `mamba` environment with a set of packages already installed in it (for example the `base` environment), but it doesn't have `xarray` and `geopandas`. You can create a file like `base.yml`, that specifies updates to the `base` environment. Then, running `mamba env update -f base.yml` will update `base` by adding `xarray` and `geopandas`, provided it does not cause conflicts with the existing libraries.

Using `pip` for python packages

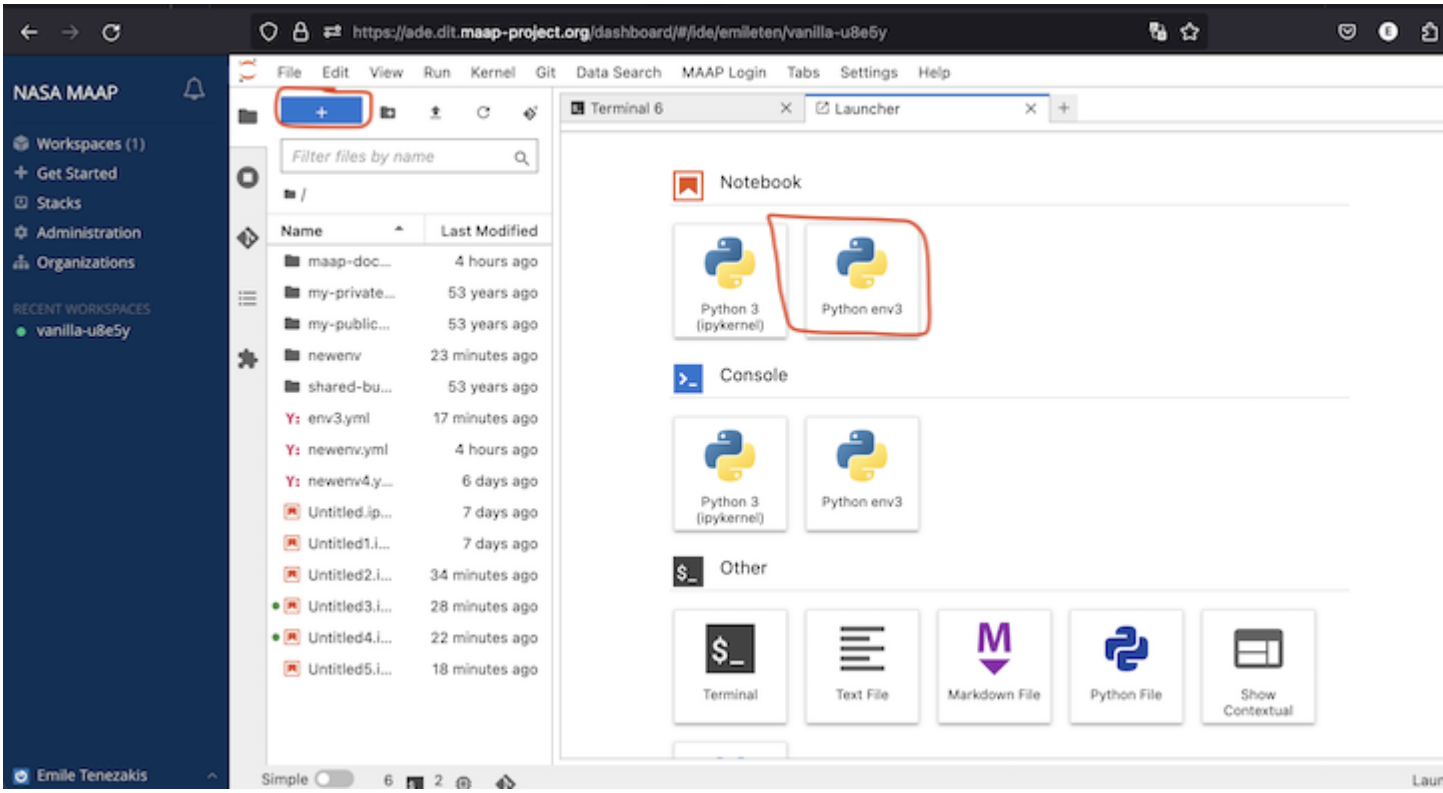
Some python packages might not be availabe in the channel you are using, or in any `mamba` channel. If that package however is in `PyPI` (the official python package repository), one can use `pip` within a `mamba` environment to download packages. The recommended way is to specify this in the configuration file. You can find an example in `env2.yml`, that creates an environment named `env2`. In that example, we add `stackstac` as a dependency to install from `PyPI` because it is not available in the `conda-forge` channel.

Using custom environments in jupyter notebooks

To make your environment accessible in a Jupyter notebook, you need to register a kernel that has your environment. You can use the `ipykernel install` command for this, which means you must list `ipykernel` as a dependency in your configuration file. An example is in `env3.yml`, that creates an environment named `env3`. After creating it, you run the kernel registration command this way :

```
python -m ipykernel install --user --name env3 --display-name "Python env3"
```

Once this is done, wait around 30 seconds for the registration to propagate. Then, click on a new launcher button. Among the notebook options, you should see "Python env3". This will spin up a notebook with your environment. Below you can find a screenshot showing the commands on the notebook :



Suggested packages for custom environment

MAAP users typically use the python `maap-py`. It's pre-installed in all workspaces, in the `base` mamba environment, but any custom environment should specify it, otherwise it is not going to be accessible from that environment. However, `maap-py` is not packaged in a public package repository, like `PyPI` or `conda-forge`. It is possible to install it directly from its github repository with `pip` though. An example configuration file can be found at `env4.yml`. You can note that inside of it `maap-py` is 'versioned' using a commit hash (at the end of the github URL).

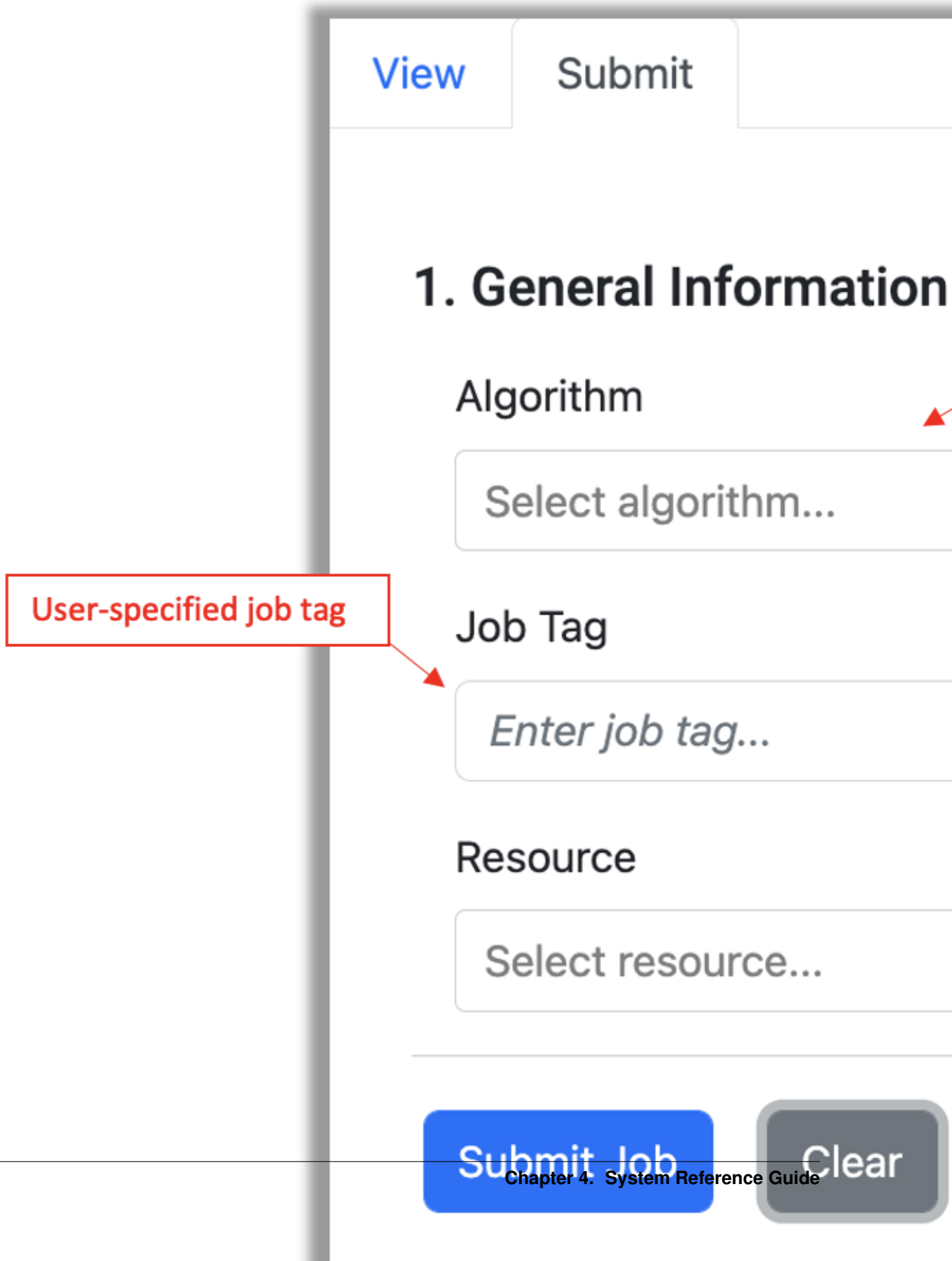
4.9 ADE Custom Extensions

4.9.1 Jobs UI

The Jobs UI allows users to submit and view DPS jobs from their Jupyter workspace. Users can monitor job status, access generated products, view errors, and view other job metadata.

Access

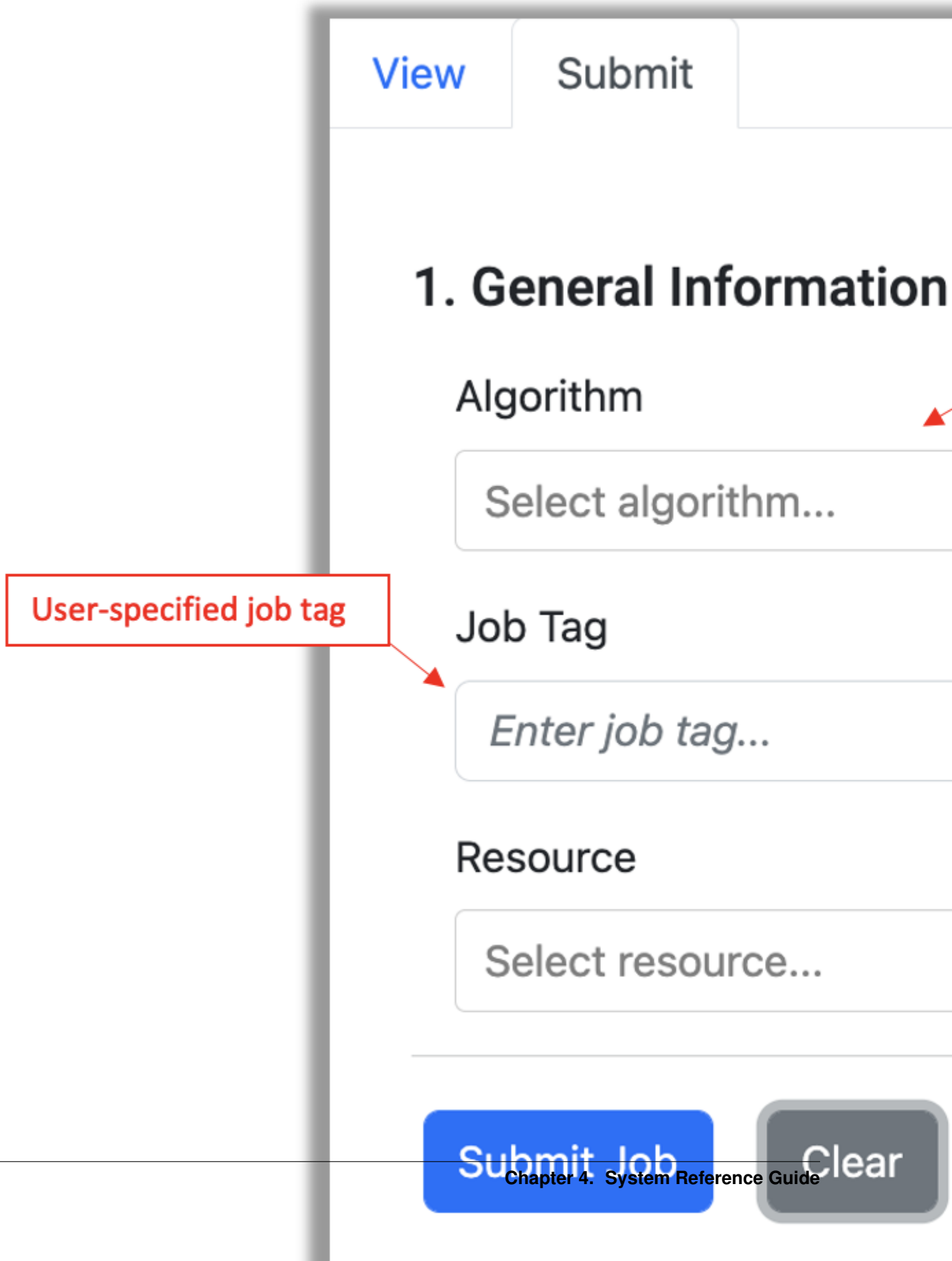
1. From your workspace click on the **View & Submit Jobs** card on the launcher tab.


View
Submit
1. General Information
Algorithm
Select algorithm...
User-specified job tag
Job Tag
Enter job tag...
Resource
Select resource...
Submit Job
Clear

View Jobs

The **View** pane lists all the jobs submitted by the user. The top table shows only a few key fields. Users can click on any row to view detailed data for the selected job such as inputs, outputs generated, and errors produced if the job failed to complete successfully.

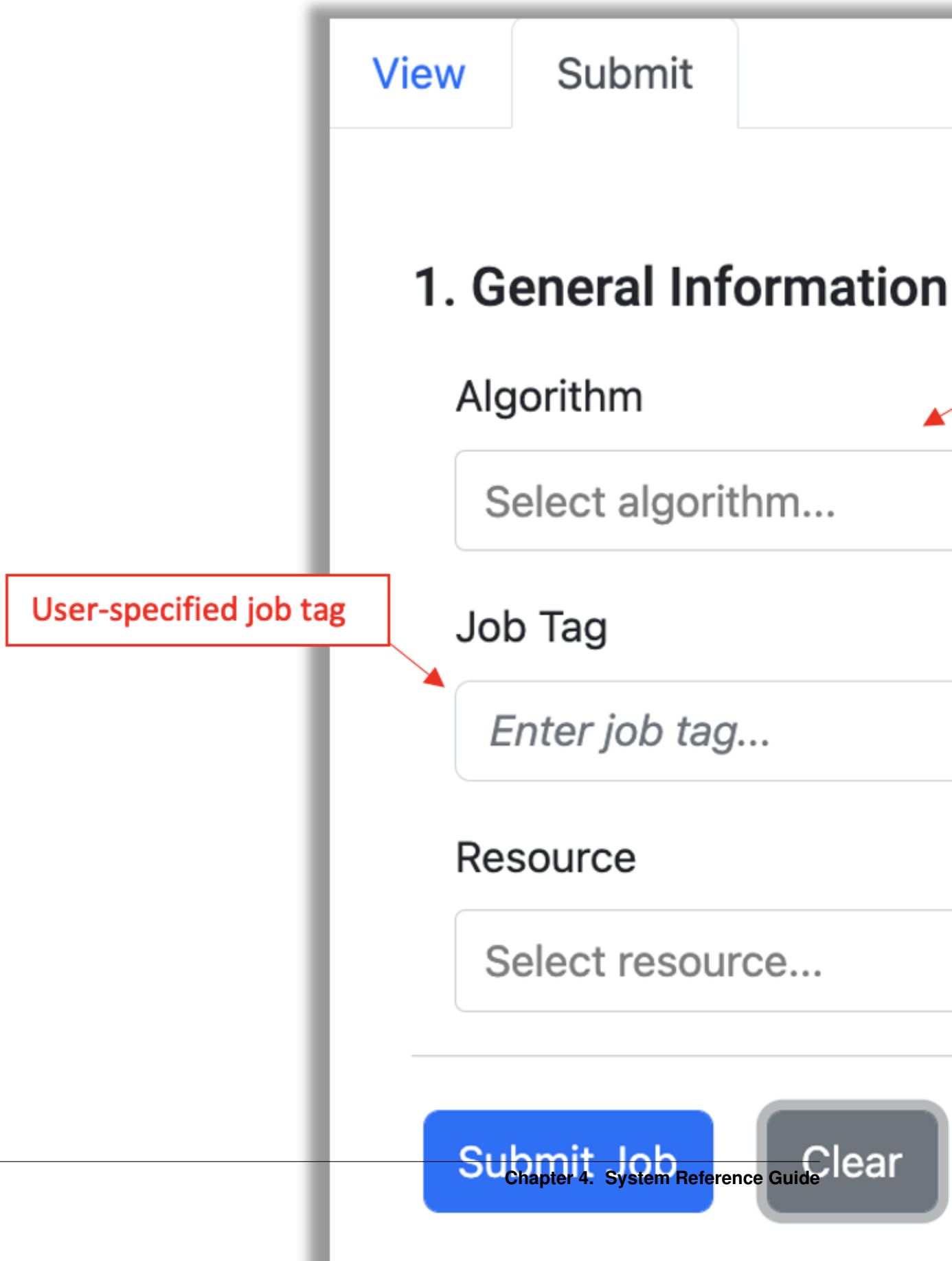
Users can sort jobs by queued, start, and end time in ascending/descending order. Users may use the search bar to filter the job list down to jobs containing the user-provided string in any of the fields shown.


View
Submit
1. General Information
Algorithm
Select algorithm...
User-specified job tag
Job Tag
Enter job tag...
Resource
Select resource...
Submit Job
Clear

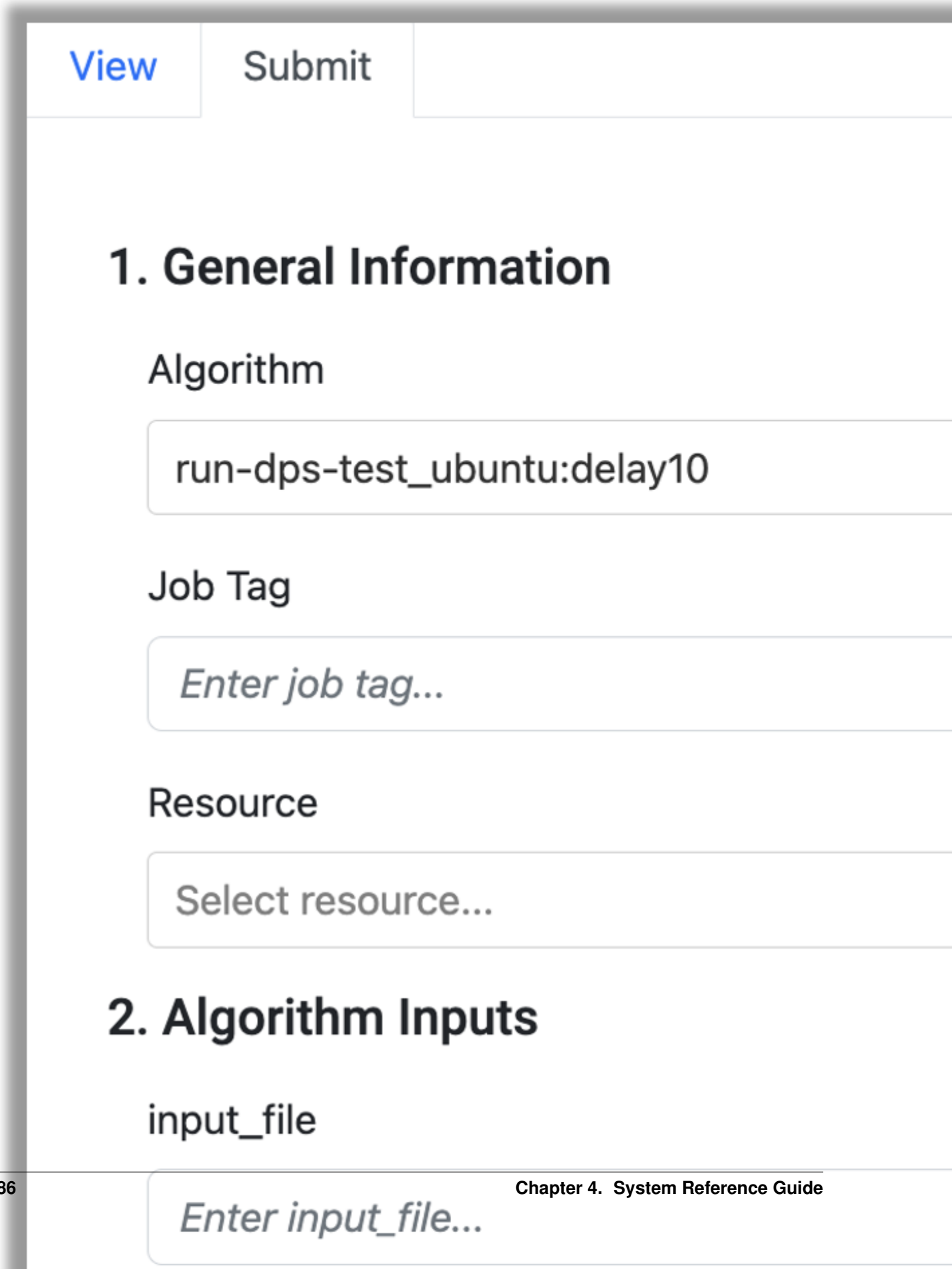
Submit Jobs

Users can submit individual jobs from the **Submit** pane. The following are the minimum requirements for submitting a job:

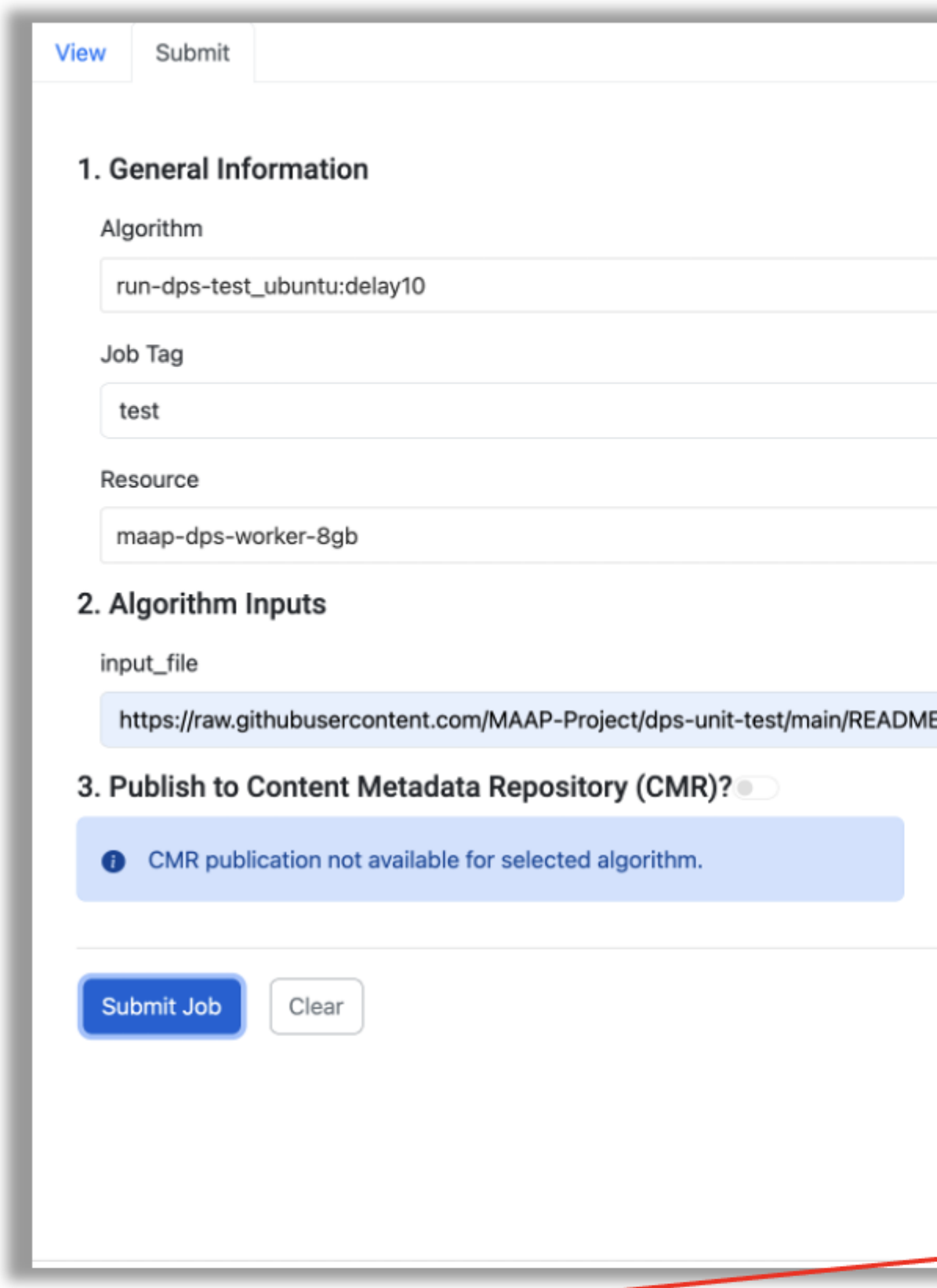
1. Select an algorithm from the dropdown menu. Note: only registered algorithms will be shown.
2. Provide a tag that may then be used to easily search for and identify the submitted job.
3. Select the resource to use for algorithm execution.


View
Submit
1. General Information
Algorithm
Select algorithm...
User-specified job tag
Job Tag
Enter job tag...
Resource
Select resource...
Submit Job
Clear

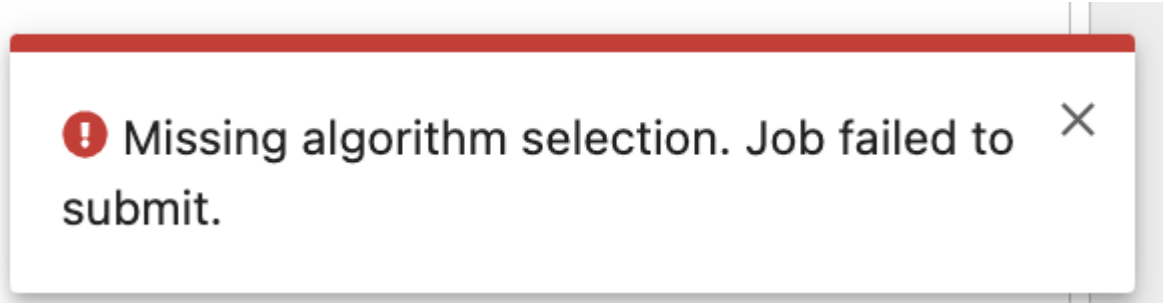
Algorithms may contain additional inputs that users may have to provide.


View
Submit
1. General Information
Algorithm
run-dps-test_ubuntu:delay10
Job Tag
Enter job tag...
Resource
Select resource...
2. Algorithm Inputs
input_file
Enter input_file...

Once all inputs have been provided, the user may click **Submit Job** to submit the job. If the job was submitted successfully, a toast will appear in the bottom right corner containing the unique job id.


View
Submit
1. General Information
Algorithm
run-dps-test_ubuntu:delay10
Job Tag
test
Resource
maap-dps-worker-8gb
2. Algorithm Inputs
input_file
https://raw.githubusercontent.com/MAAP-Project/dps-unit-test/main/README
3. Publish to Content Metadata Repository (CMR)?
CMR publication not available for selected algorithm.
Submit Job
Clear

If the job failed to submit, a toast will appear indicating the job failed to submit.

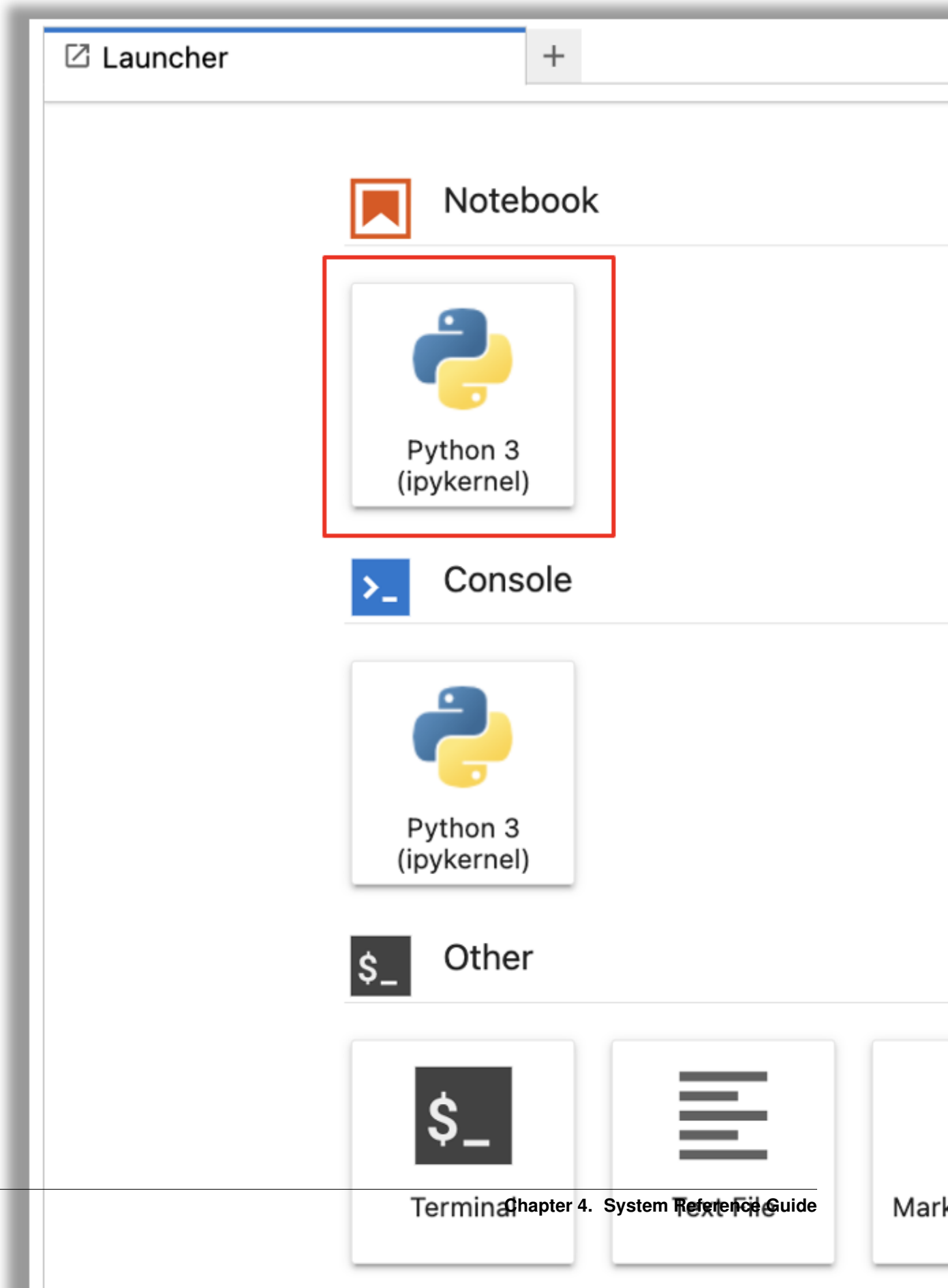


4.9.2 MAAP Libraries

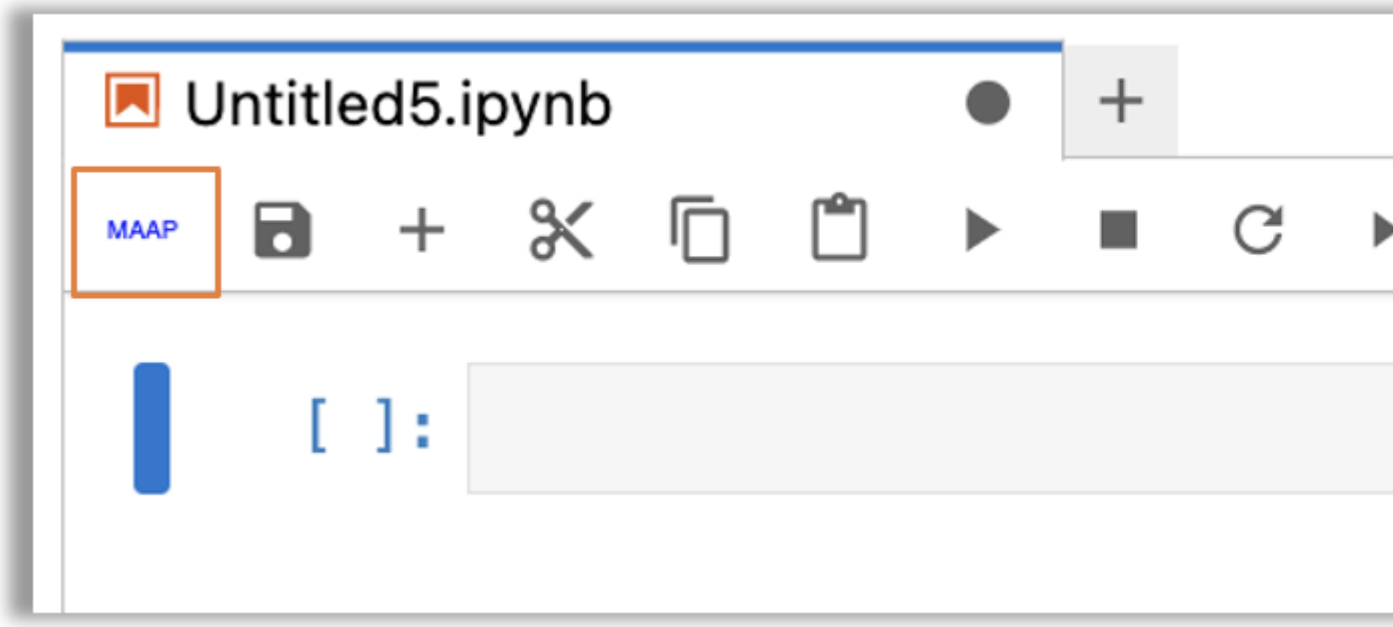
This Jupyter extension generates boilerplate code that imports and initializes MAAP libraries for usage in Jupyter notebooks. It currently supports the maap-py client library and ipycmc.

Access

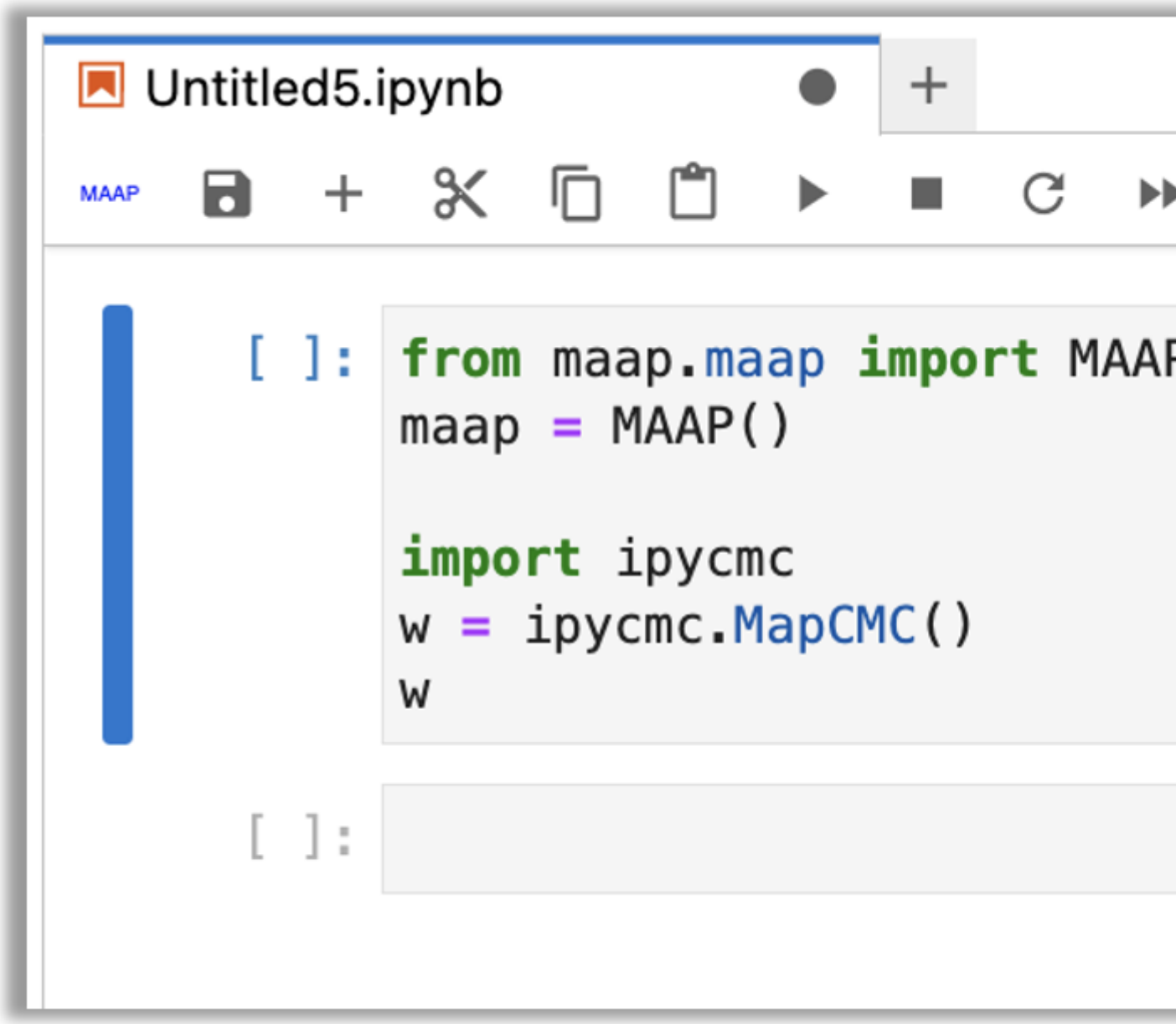
1. From your workspace, create a Jupyter notebook from the launcher pane.


Launcher
Notebook
Python 3
(ipykernel)
Console
Python 3
(ipykernel)
Other
Terminal
Text File
Mark

2. Click on the **MAAP** icon.



3. The Jupyter notebook will be populated with the following code block.

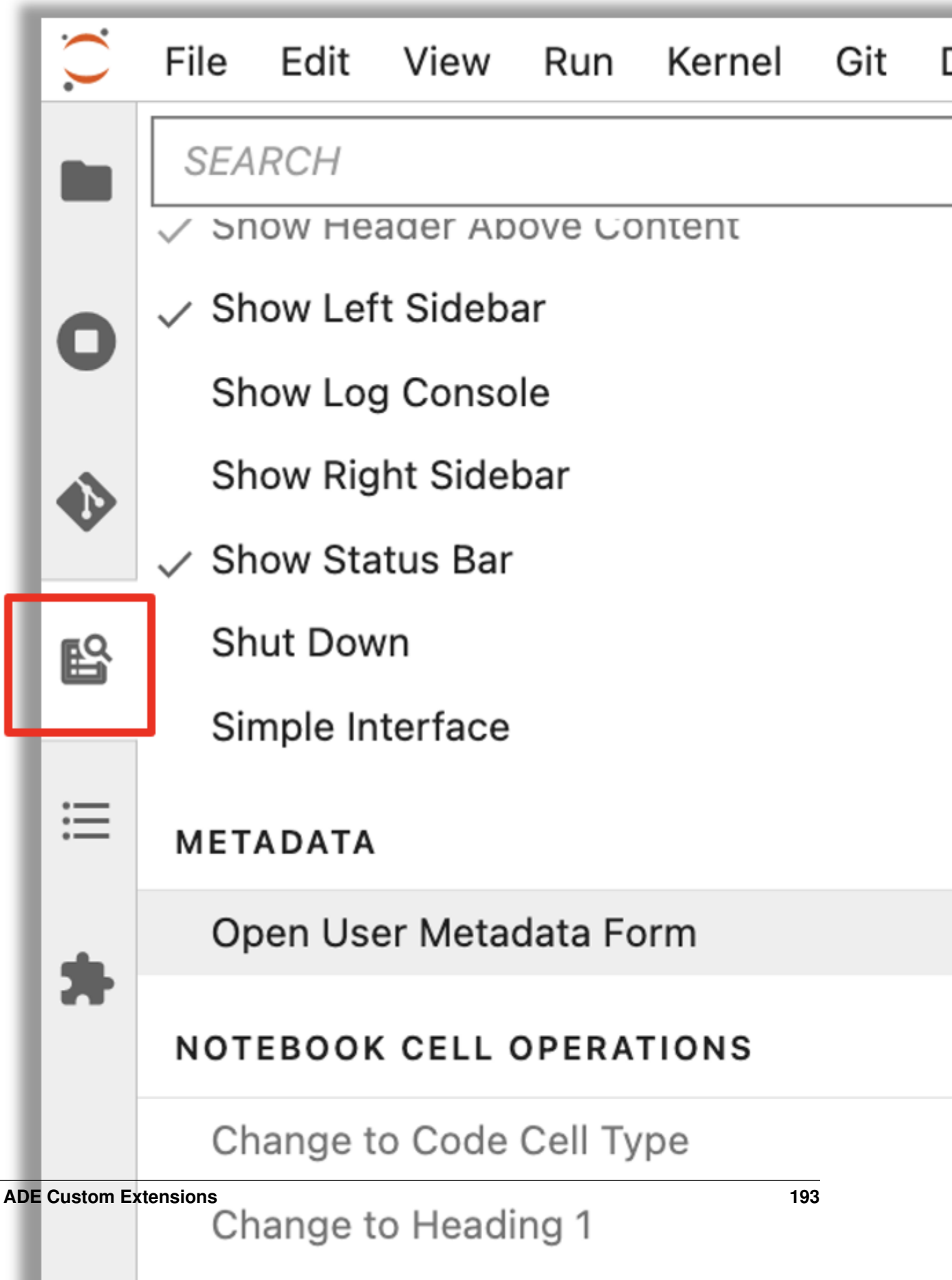


4.9.3 User Metadata Form

This extension allows users to easily access the **MAAP User Shared Data Questionnaire**.

Access

1. From your workspace, click on the command icon on the left ribbon menu and scroll down to the Metadata section.


File
Edit
View
Run
Kernel
Git
SEARCH
Show Header Above Content
Show Left Sidebar
Show Log Console
Show Right Sidebar
Show Status Bar
Shut Down
Simple Interface
METADATA
Open User Metadata Form
NOTEBOOK CELL OPERATIONS
Change to Code Cell Type
Change to Heading 1

2. Click on **Open User Metadata Form** to be redirected to the shared-data questionnaire in a new window.

4.9.4 Jupyter Server Extension

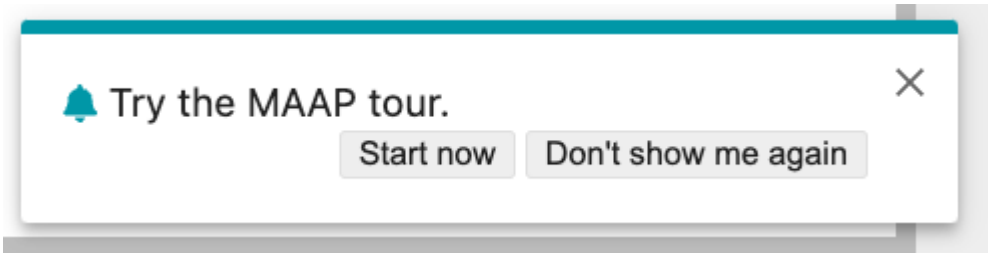
This is the backend extension for the MAAP common collection of custom Jupyter extensions. This backend is accessible to users and other Jupyter extensions through a RESTful interface. The supported endpoints interact with the MAAP API and user workspace. For a list of the endpoints, refer to the repo documentation.

4.9.5 Maap Help Jupyter Extension

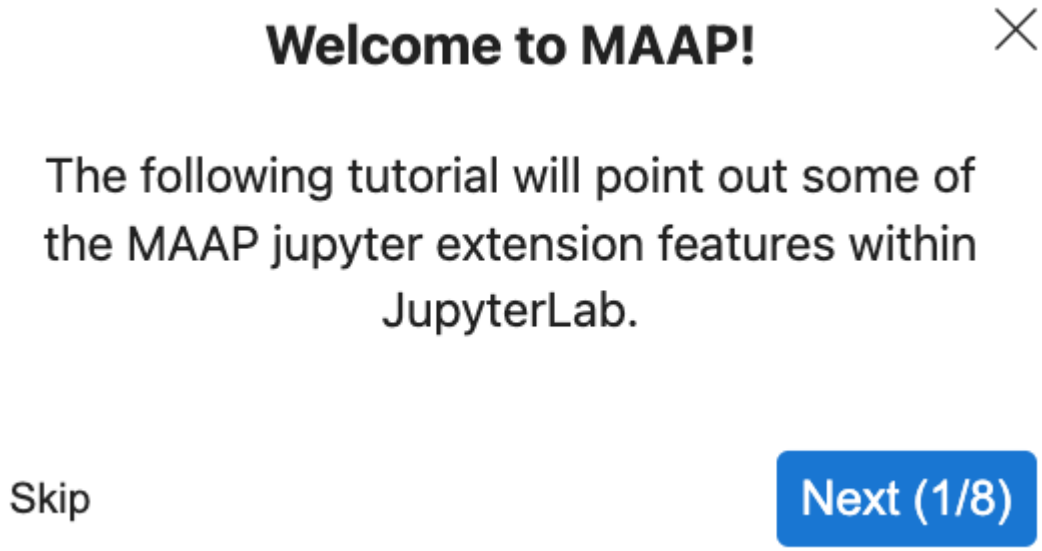
This Jupyter extension provides an interactive tutorial on a new user's first launch of the ADE and adds MAAP specific information to the help tab. The interactive tutorial can be accessed again from the help menu.

Interactive Tour

1. Launch the ADE for your first time as a new user, and see the MAAP tour notification



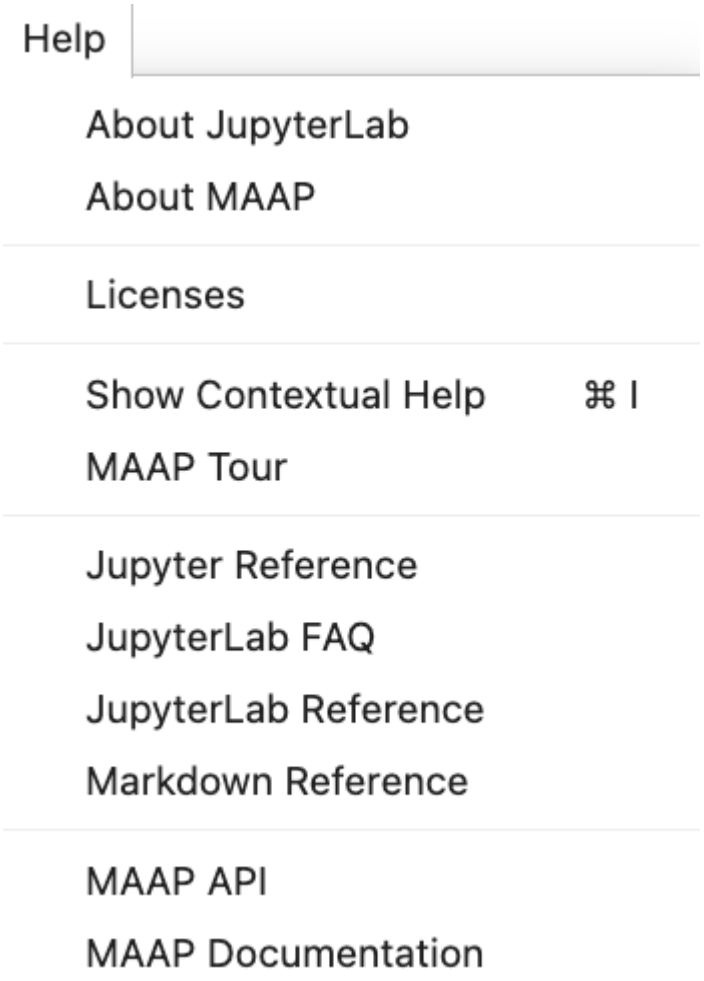
2. Step through the tour with the "Next" button.



3. The MAAP tour will no longer appear on launch if you a) finish the tour, b) press skip during the tour, or c) press "Don't show me again." You can revisit the MAAP tour by selecting it from the help menu.

MAAP Help Menu

Options show MAAP specific help information along with default JupyterLab help information.

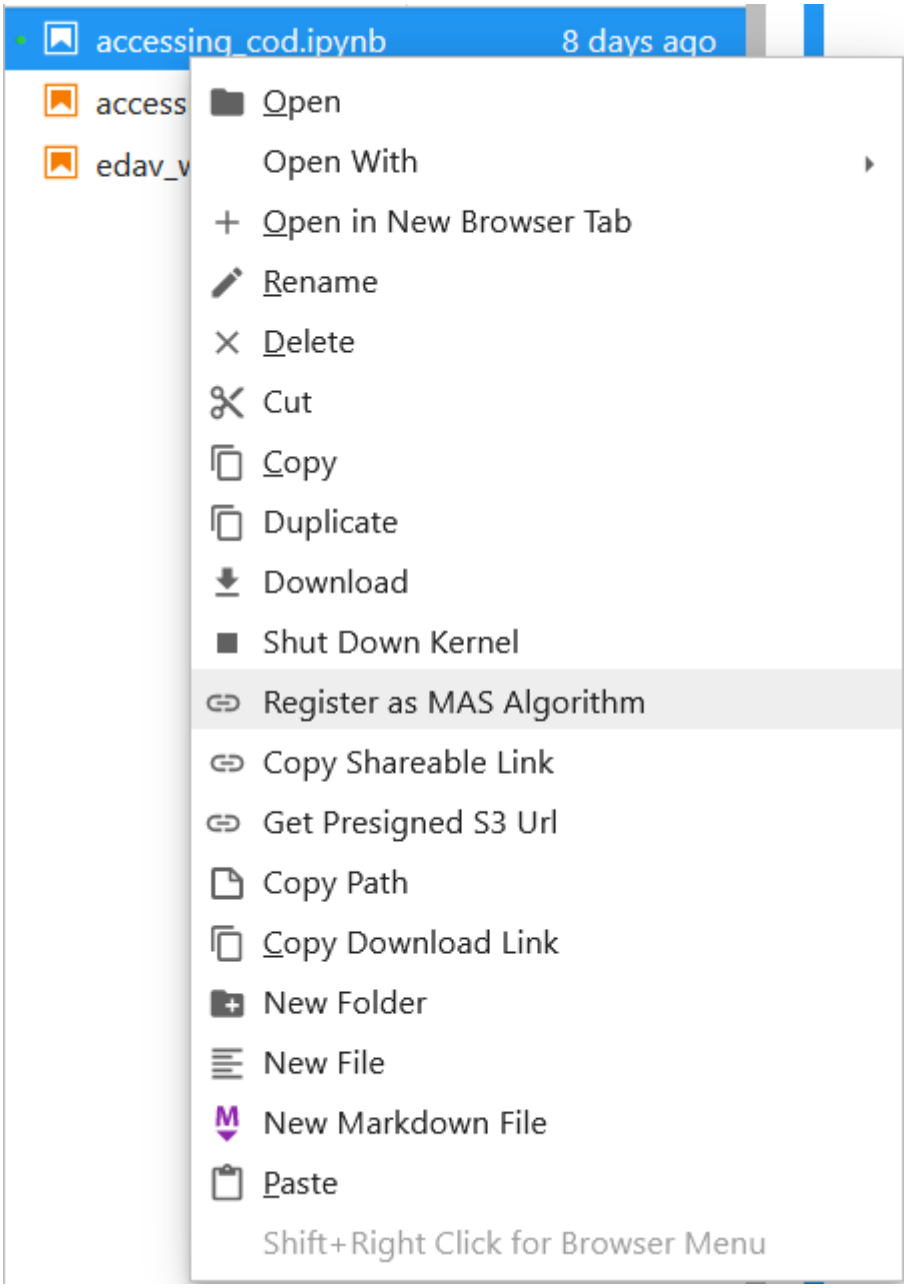


4.10 FAQ

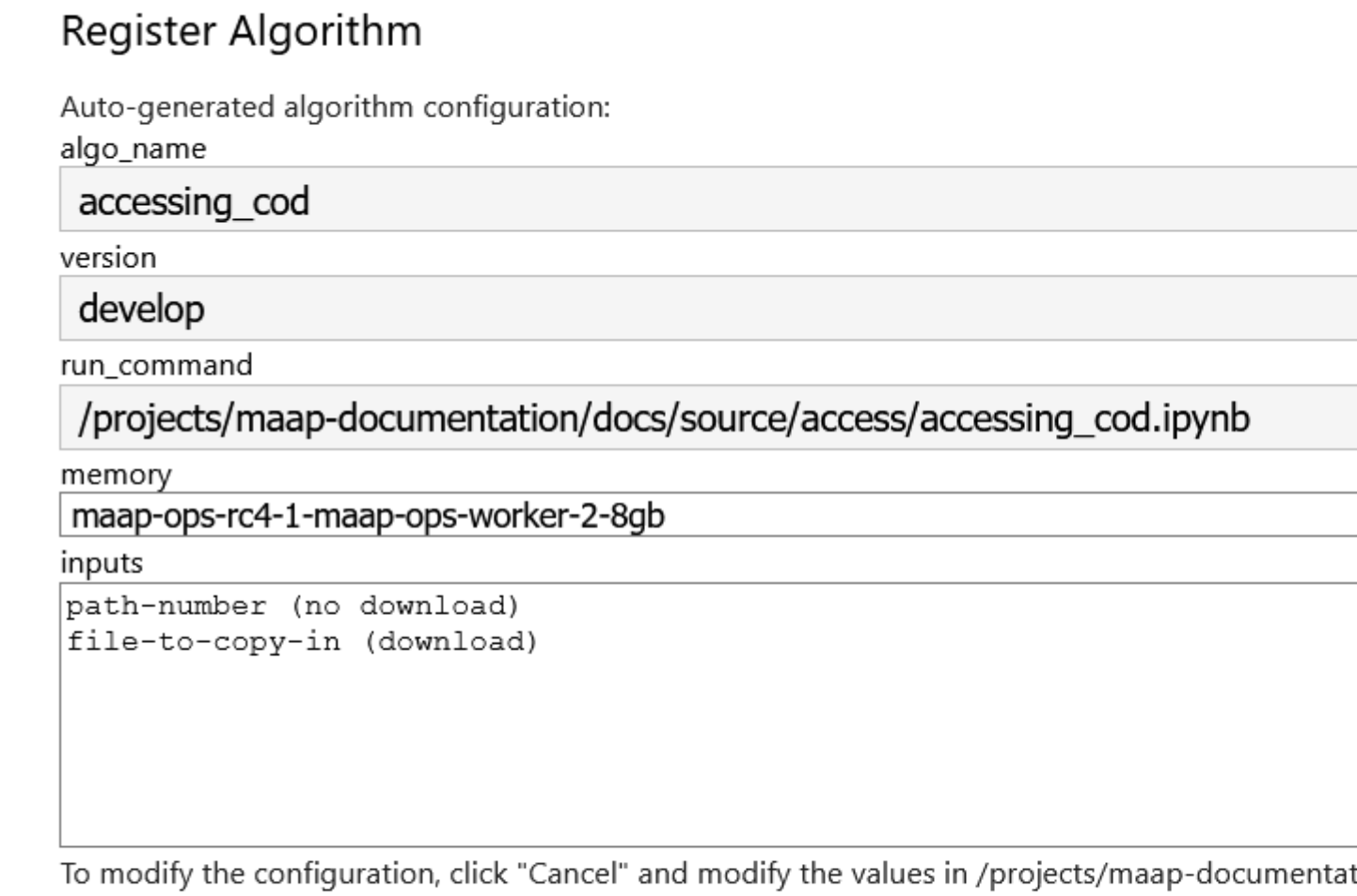
4.10.1 How Do I Register An Algorithm To MAS?

Before starting, make sure your algorithm can run in your environment and all changes have been committed/pushed to GitLab or your external repository.

When you are ready, in the file browser, right click the file which you wish to register as an algorithm and select *Register as MAS Algorithm*.



If your changes have not been committed, there will be a popup asking you to commit your changes first. The fields will be summarized for you in a form.



These are saved in the automatically generated file *algorithm_config.yaml*, in the same directory as the script to be run. Don't register the algorithm if any fields are missing or incorrect.

Click cancel and open the file *algorithm_config.yaml* to edit those fields.

```yaml
# DO NOT DELETE
# THIS CONFIG IS AUTO-GENERATED BY ADE UI
algo_name: accessing_cod
version: develop
environment: ubuntu
repository_url: https://github.com/MAAP-Project/maap-documentation.git
docker_url: mas.maap-project.org:5000/root/ade_base_images/vanilla:${versio
# queue chosen when registering
memory: 15GB

# fill out these fields
# explain what this algorithm does
description:
# path to the wrapper script for running the algorithm
run_command: /projects/maap-documentation/docs/source/access/accessing_cod.
# set a storage value in GB or MB or KB, e.g. "100GB", "20MB", "10KB"
disk_space: 10GB
inputs:
# remove below this line if no inputs
# rename and set algorithm input names accordingly
  - name: path-number
    download: False
  - name: file-to-copy-in
    download: True
```

When everything is correct, save your config changes and right-click the file again to register your algorithm.

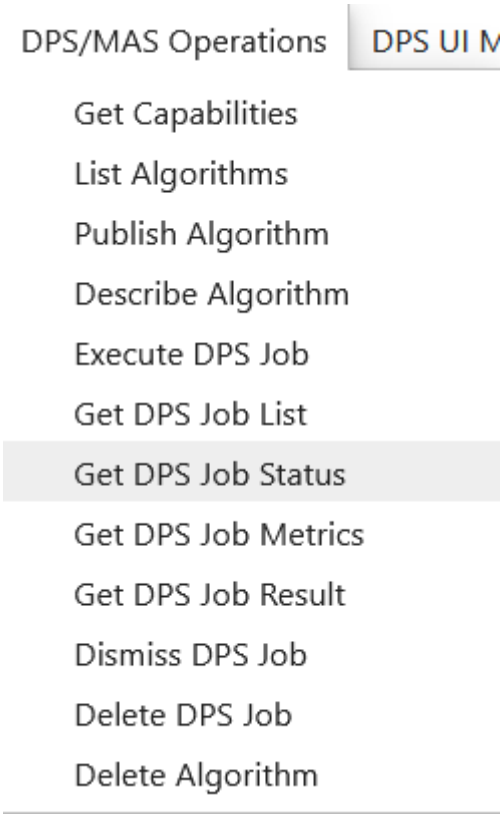
Notes:

There will only be one *algorithm_config.yaml* file for any directory. If you have multiple scripts to be registered as algorithms, consider separating them into different folders.The outputs should be written to a folder named outputs.

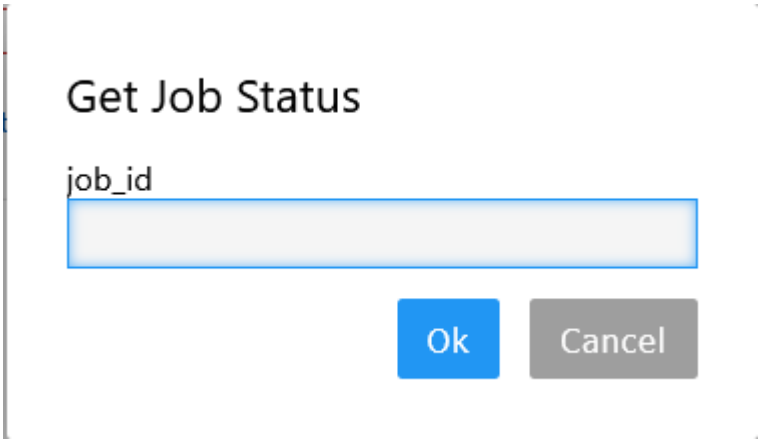
4.10.2 Check DPS Job Status

Option 1: Menu Popup

Open the *DPS/MAS Operations* menu and select *Get DPS Job Status*.



Enter the job ID of the job you are checking and click OK.



You will see the job's status (queued, started, complete, failed, or dismissed).

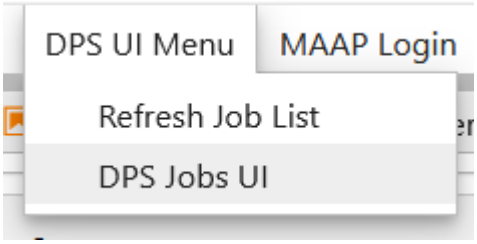
Option 2: Jobs UI – Side Panel

From the *Jobs* side panel, click *DPS UI -> Refresh Job List* to update the list of jobs.

They will be sorted in order of most recently submitted. The job statuses are listed next to their corresponding job IDs.

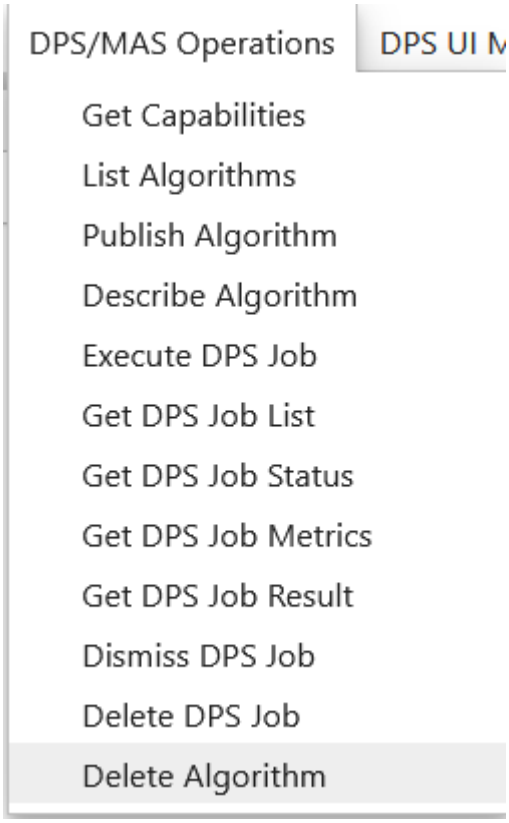
Option 3: Jobs UI – Main Widget

You can also view job status in the *DPS UI* (open the *DPS Jobs UI* option on the *DPS UI Menu*), in the list of submitted jobs at the bottom (same list as in the side panel).

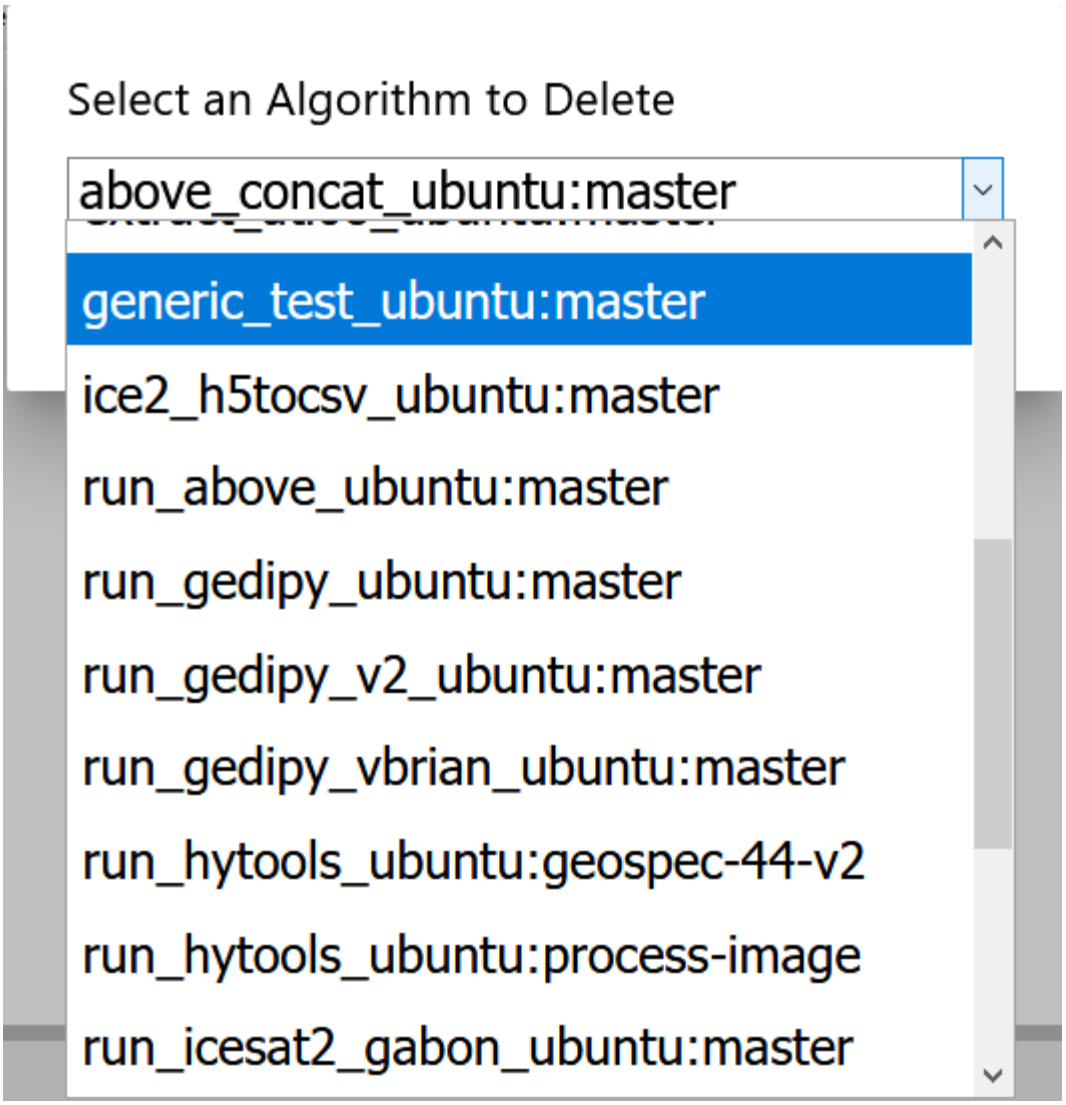


4.10.3 How Do I Delete My Algorithm From MAS?

Open the *DPS/MAS Operations* menu and select *Delete Algorithm*.



A dropdown list of the currently available algorithms will show up.

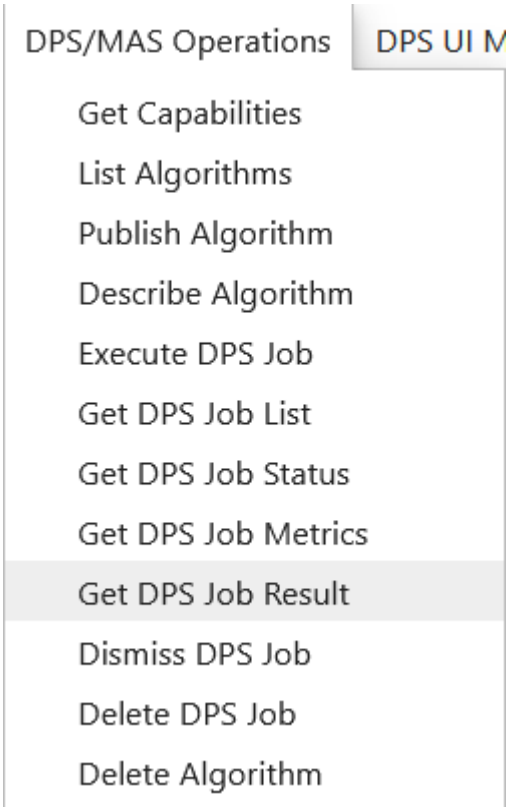


Select the algorithm you wish to delete and press OK.

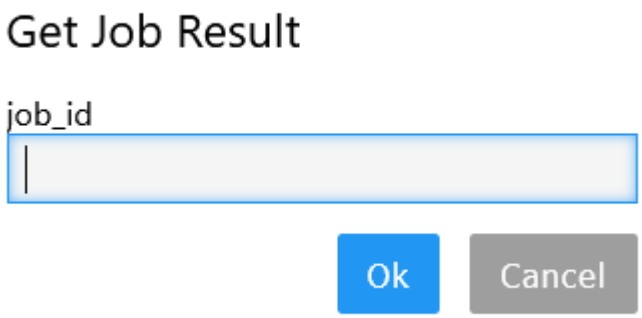
4.10.4 How Do I Get Results From My DPS Job?

Option 1: Menu Popup

Open the *DPS/MAS Operations* menu and select *Get DPS Job Result*.



Enter the job ID of the job you are checking and click OK.



If the outputs have been written to the correct folder, you will be given a list of product names and links. The clickable link goes to the AWS S3 location where your outputs have been exported.

If there were no outputs detected, there will be a blank popup.

If the job is incomplete (any state other than complete), there will be an error and you should wait until the job has completed before checking for results.

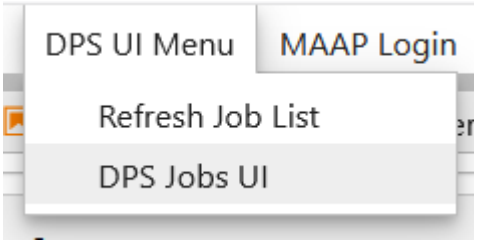
Option 2: Jobs UI – Side Panel

From the *Jobs* side panel, click *DPS UI -> Refresh Job List* to update the list of jobs.

They will be sorted in order of most recently submitted. Click on a job ID in the side panel to review more information about the run and results if the job has been completed.

Option 3: Jobs UI – Main Widget

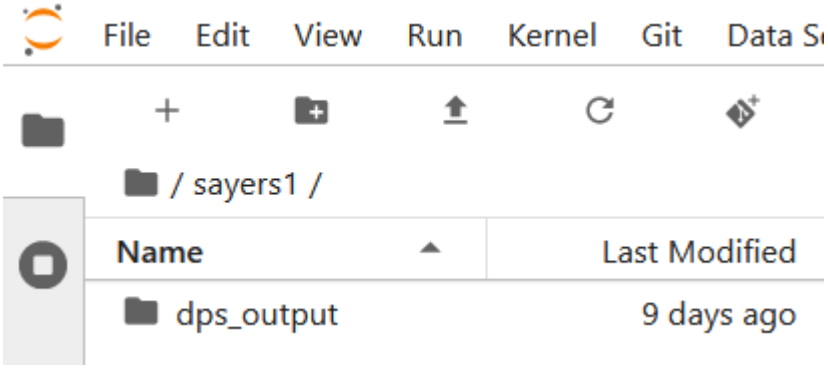
You can also view job status in the *DPS UI* (open the *DPS Jobs UI* option on the *DPS UI Menu*).



Click on a job ID under the *Submitted Jobs* list, and the job information and results (if completed) will be shown under the *Job Info* tab.

Option 4: Workspace File Directory

In the latest ADE images (v2.0), your personal s3-backed folder is automatically populated with a dps_output folder, where the outputs of completed jobs are automatically uploaded and accessible from within your workspace.



The hierarchy is as follows:

username/dps_output/algo_name/algo_verison/run_year/run_month/run_day/run_hr/run_min/run_sec/run_ms

For example: eyam/dps_output/hello_world/master/2020/03/10/00/44/15/865562

4.10.5 Which Files In My Workspace Are Persistent?

When you first start your workspace, there should be at least three folders already in the file browser: che, maap-users, and <username>.

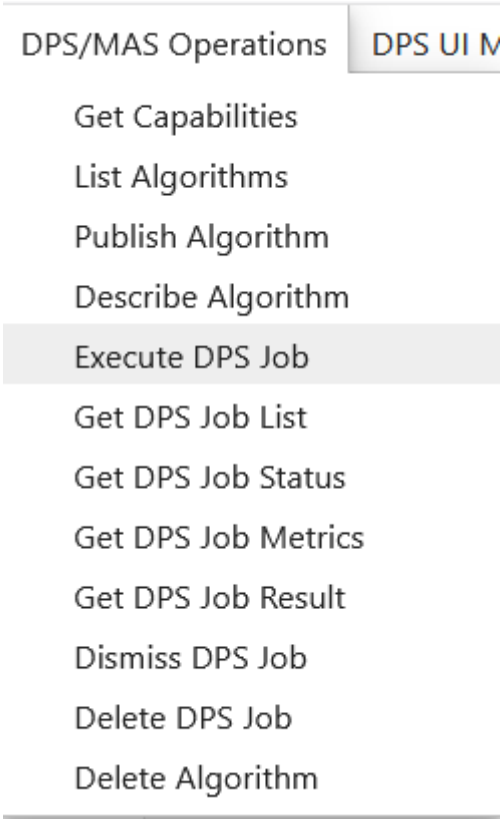
- che contains config files for your ADE. You don't need to worry about it.
- maap-users is an s3-backed folder shared among all members of the maap-users organization (every registered MAAP user). If you are in other MAAP organizations, each organization will have its own shared s3-backed folder.
- Finally, <username> is also s3-backed, but only shown to you. However, when you share your workspace with another user, both users' personal folders will be mounted and accessible from the shared workspace. Anything placed within a s3-backed folder will be added to s3 and be persistent. Anything outside those folders will be ephemeral. Please clone your git repositories (e.g., dps_plot) outside the s3-backed folders, as they should already be version controlled elsewhere.

4.10.6 How Do I Run A DPS Job?

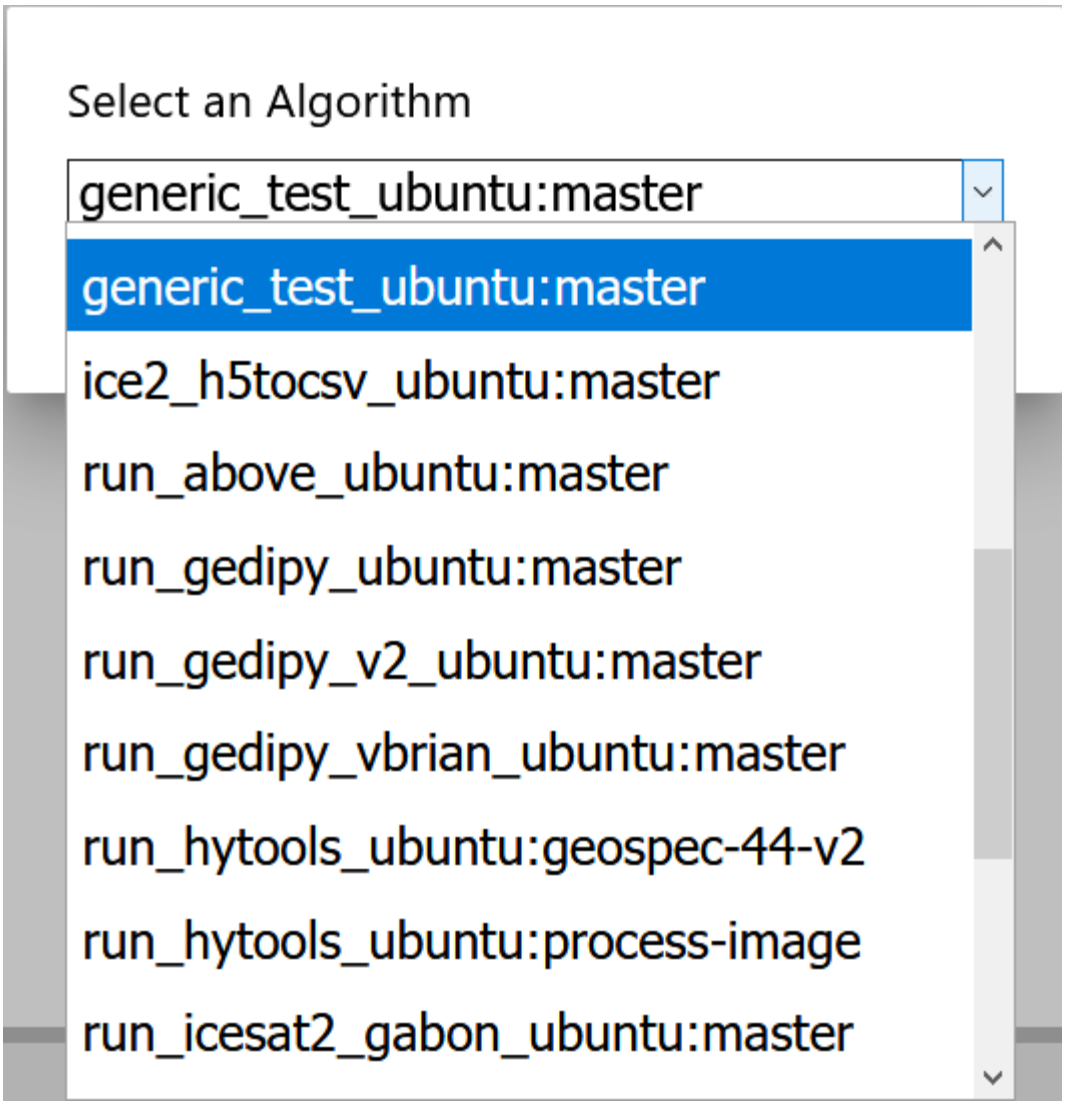
There are 3 ways to do this: guided through popups, via the *Jobs Main Widget UI*, and via a call to the maap-py Python library.

Option 1: Guided Popups

Open the *DPS/MAS Operations* menu and select *Execute DPS Job*.



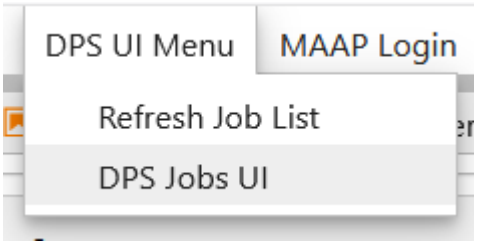
Then select the algorithm you wish to run from the dropdown of what is currently available and click OK.



If the chosen algorithm requires inputs, a new popup will ask for inputs to each field the algorithm takes. If it doesn't take inputs, the popup will say so. Click OK again to view the ID for the job you just submitted. If you didn't get to save it, just click *DPS UI -> Refresh Job List*, and the latest job will appear at the top of the Jobs side panel.

Option 2: Jobs UI

Open the UI if you haven't already done so (*DPS UI Menu -> DPS Jobs UI*).



Under the *Run Jobs* tab and under *Algorithms List*, click on the algorithm you wish to run. To the side, you will see the stored algorithm description, and the Execute Job column will automatically update for required inputs. Fill out

those fields and click the *Execute Job* button. Your submitted job's ID will appear below the button.

Option 3: Python Call

From within a *Jupyter Notebook* running Python, or inside a Python script, you can use the maap-py library. This is already installed on all ADE workspaces. To import/set it up, you can click the small *MAAP* button at the top left of the *Jupyter* toolbar, or you can add these lines before your script:

```
[ ]: from maap.maap import MAAP
     maap = MAAP()
```

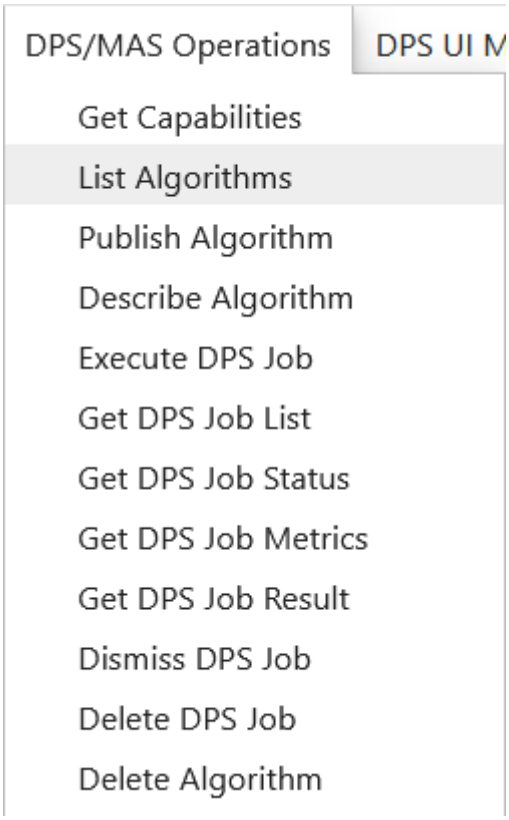
You are then required to pass your algorithm's name, version, required inputs, and username to the function maap.submitJob (identifier is job-algo_name:algo_version):

```
[ ]: # Add some search results to create a list of granules
     granules = []
     mit_result = maap.submitJob(
         identifier="jobplot_analysis_demo_ubuntu:master",
         algo_id="plot_analysis_demo_ubuntu",
         version="master",
         username="bsatoriu",
         granules=[]
     )
```

4.10.7 How Do I View Algorithms Already Registered In MAS?

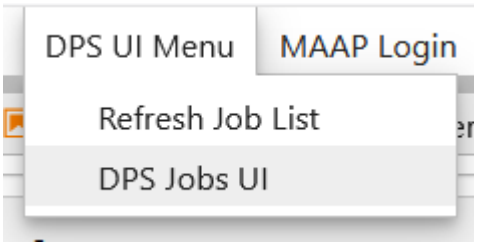
Option 1: Menu

Open the *DPS/MAS* Operations menu and select *List Algorithms*.



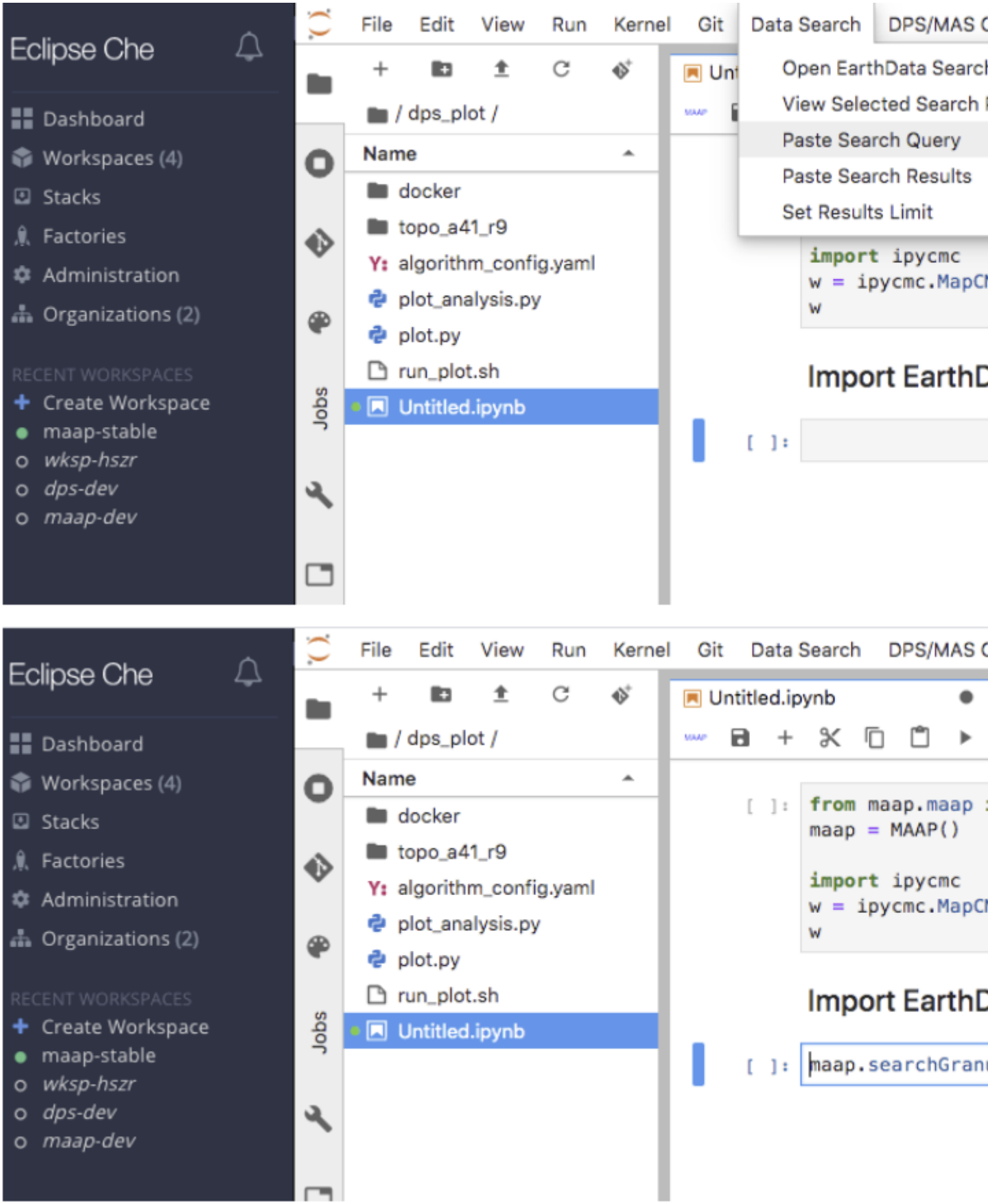
Option 2: Jobs UI

Another way would be via the *DPS UI* (open the *DPS Jobs UI* option on the *DPS UI Menu*), which automatically displays the list of algorithms under the *Run Jobs* tab.



4.10.8 How Do I Copy My EARTHDATA Search Into My Jupyter Notebook?

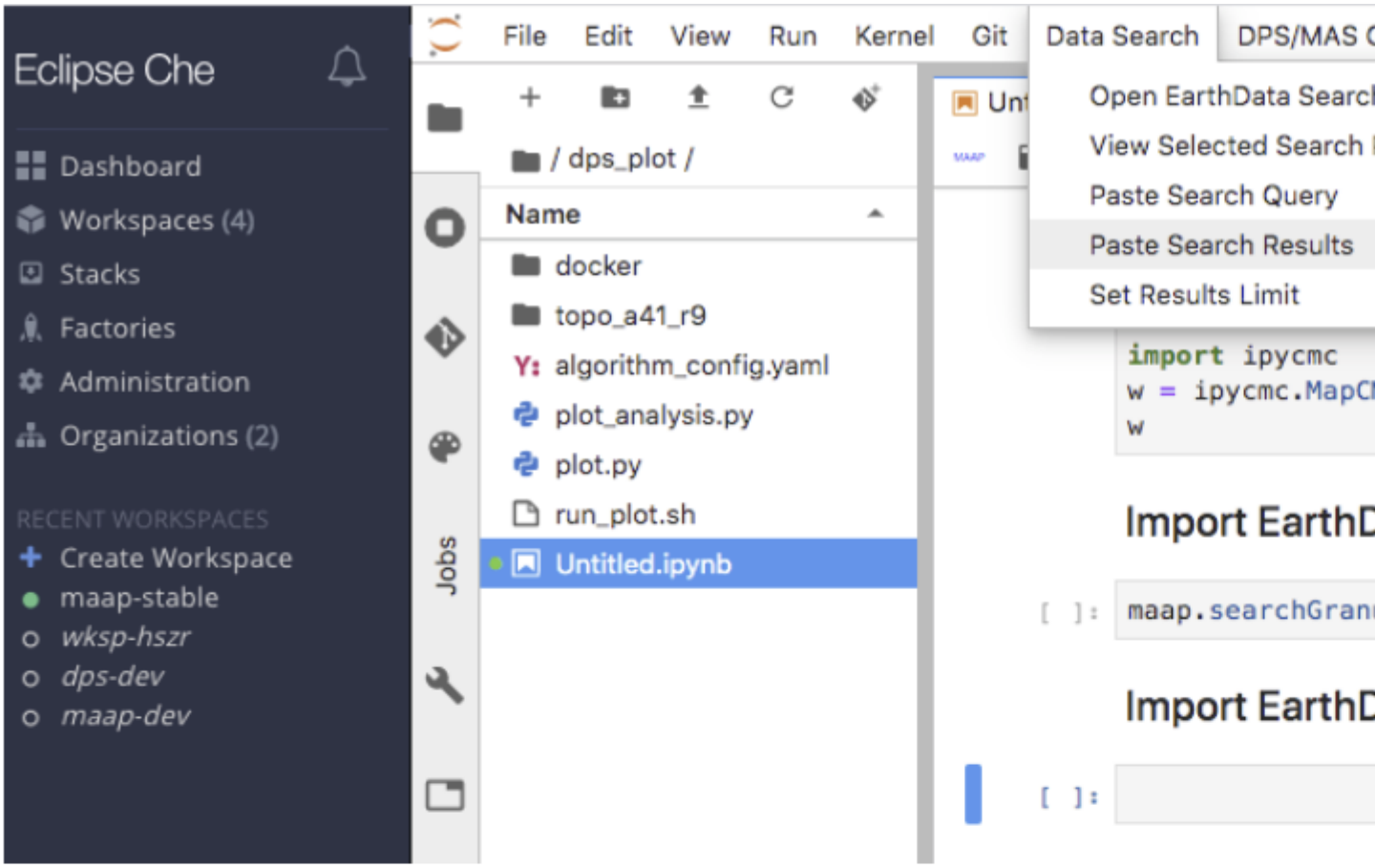
After setting your search parameters, switch tabs back to your Jupyter notebook. At the top, open the Data Search menu, and select *Paste Search Query*.

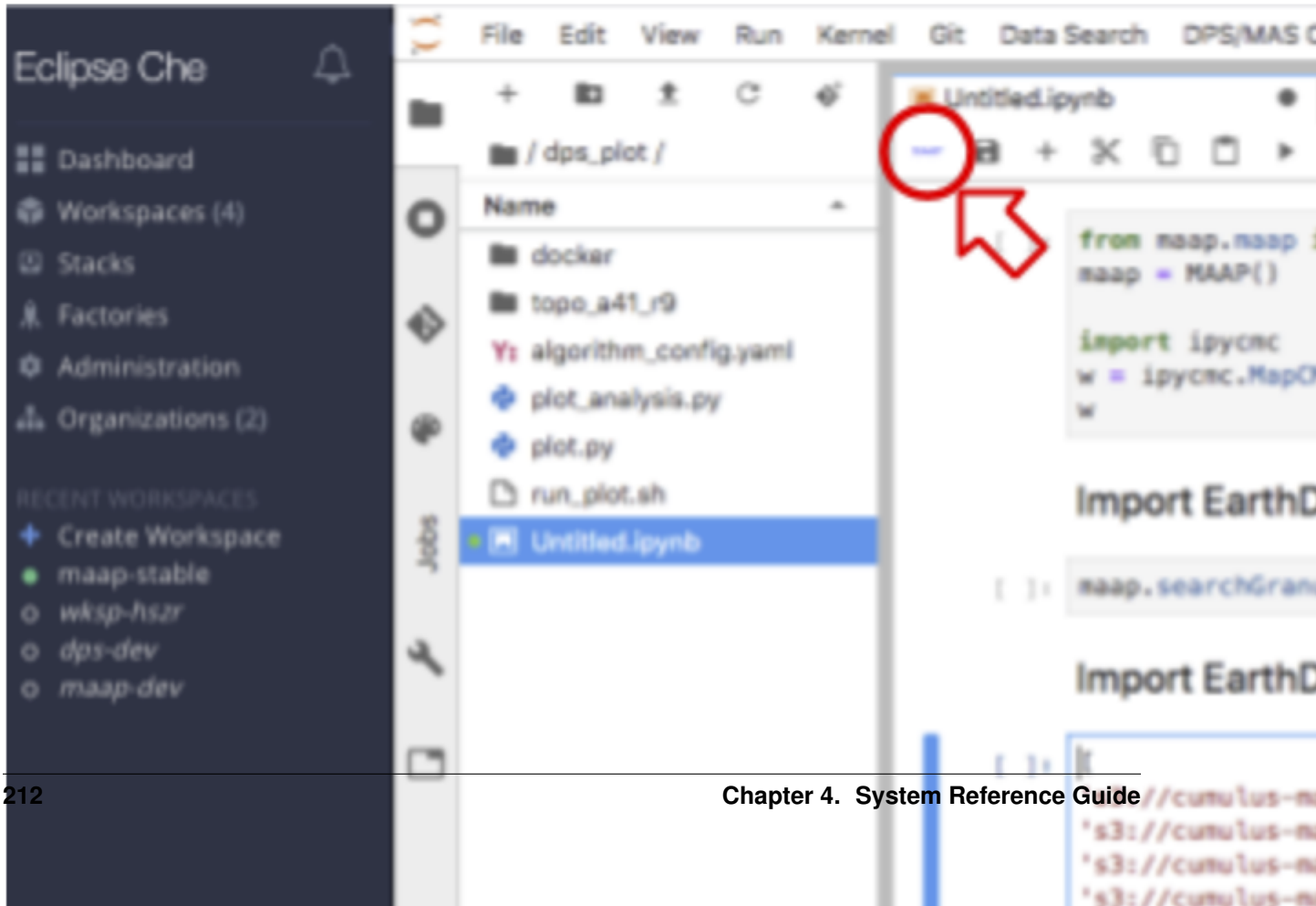

Eclipse Che
Dashboard
Workspaces (4)
Stacks
Factories
Administration
Organizations (2)
RECENT WORKSPACES
Create Workspace
maap-stable
wksp-hszr
dps-dev
maap-dev
File Edit View Run Kernel Git Data Search DPS/MAS
/ dps_plot /
Name
docker
topo_a41_r9
algorithm_config.yaml
plot_analysis.py
plot.py
run_plot.sh
Untitled.ipynb
Jobs
Open EarthData Search
View Selected Search
Paste Search Query
Paste Search Results
Set Results Limit
import ipycmc
w = ipycmc.MapC
w
Import EarthD
[ ]:
Eclipse Che
Dashboard
Workspaces (4)
Stacks
Factories
Administration
Organizations (2)
RECENT WORKSPACES
Create Workspace
maap-stable
wksp-hszr
dps-dev
maap-dev
File Edit View Run Kernel Git Data Search DPS/MAS
Untitled.ipynb
/ dps_plot /
Name
docker
topo_a41_r9
algorithm_config.yaml
plot_analysis.py
plot.py
run_plot.sh
Untitled.ipynb
Jobs
[ ]:
from maap.maap
maap = MAAP()
import ipycmc
w = ipycmc.MapC
w
Import EarthD
[ ]:
maap.searchGran

Caveat: This call uses the MAAP Python library. Make sure you import it before running the inserted code. You can do this by clicking on the blue "MAAP" text just below your notebook name.

4.10.9 How Do I Import Granules Over From My EARTHDATA Search Into My Jupyter Notebook?

After setting your search parameters, switch tabs back to your Jupyter notebook. At the top, open the Data Search menu, and select *Paste Search Results*.


Eclipse Che
Dashboard
Workspaces (4)
Stacks
Factories
Administration
Organizations (2)
RECENT WORKSPACES
Create Workspace
maap-stable
wksp-hszr
dps-dev
maap-dev
File Edit View Run Kernel Git
Data Search
Open EarthData Search
View Selected Search
Paste Search Query
Paste Search Results
Set Results Limit
/ dps_plot /
Name
docker
topo_a41_r9
algorithm_config.yaml
plot_analysis.py
plot.py
run_plot.sh
Untitled.ipynb
Jobs
import ipycmc
Import EarthD
maap.searchGran

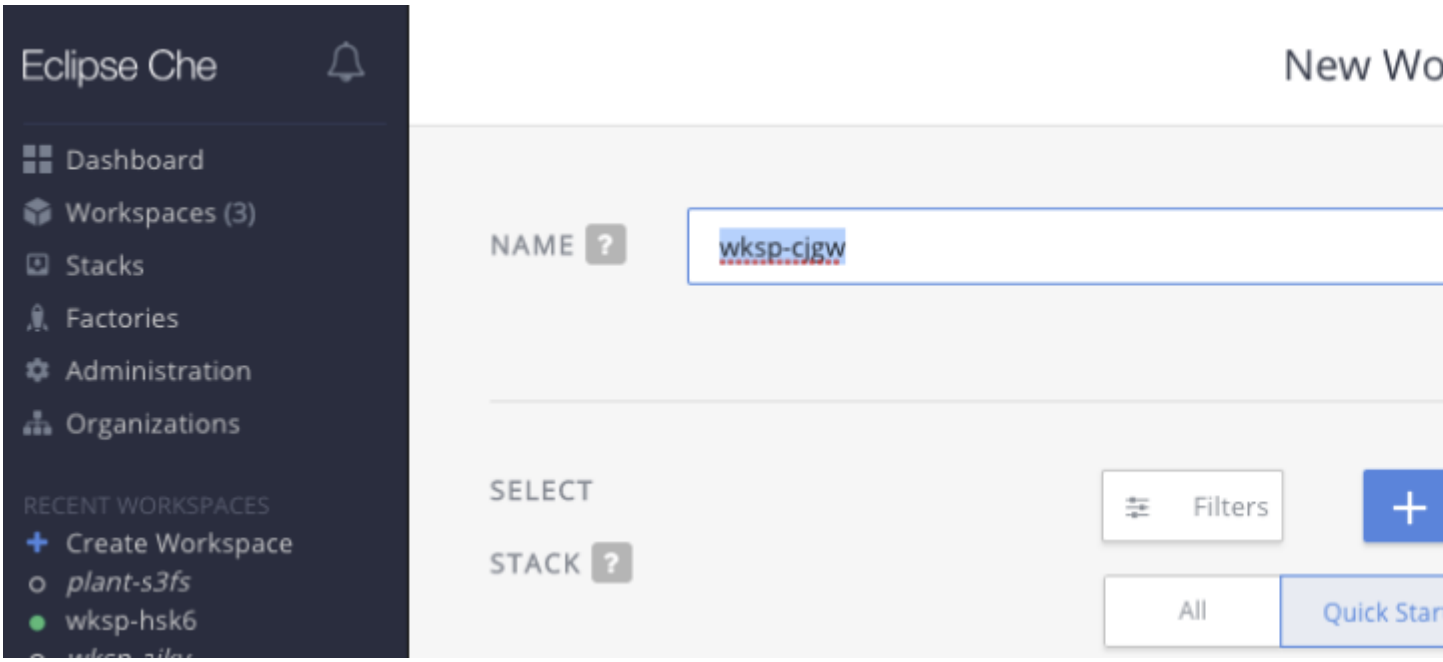

Eclipse Che
Dashboard
Workspaces (4)
Stacks
Factories
Administration
Organizations (2)
Create Workspace
maap-stable
wksp-hszr
dps-dev
maap-dev
File Edit View Run Kernel Git Data Search
/ dps_plot /
Name
docker
topo_a41_r9
algorithm_config.yaml
plot_analysis.py
plot.py
run_plot.sh
Untitled.ipynb
Jobs
from maap.maap
maap = MAAP()
import ipycmc
Import EarthD

Caveat: This call uses the MAAP Python library. Make sure you import it before running the inserted code. You can do this by clicking on the blue "MAAP" text just below your notebook name (circled in red).

4.10.10 How Do I Rename My Workspace?

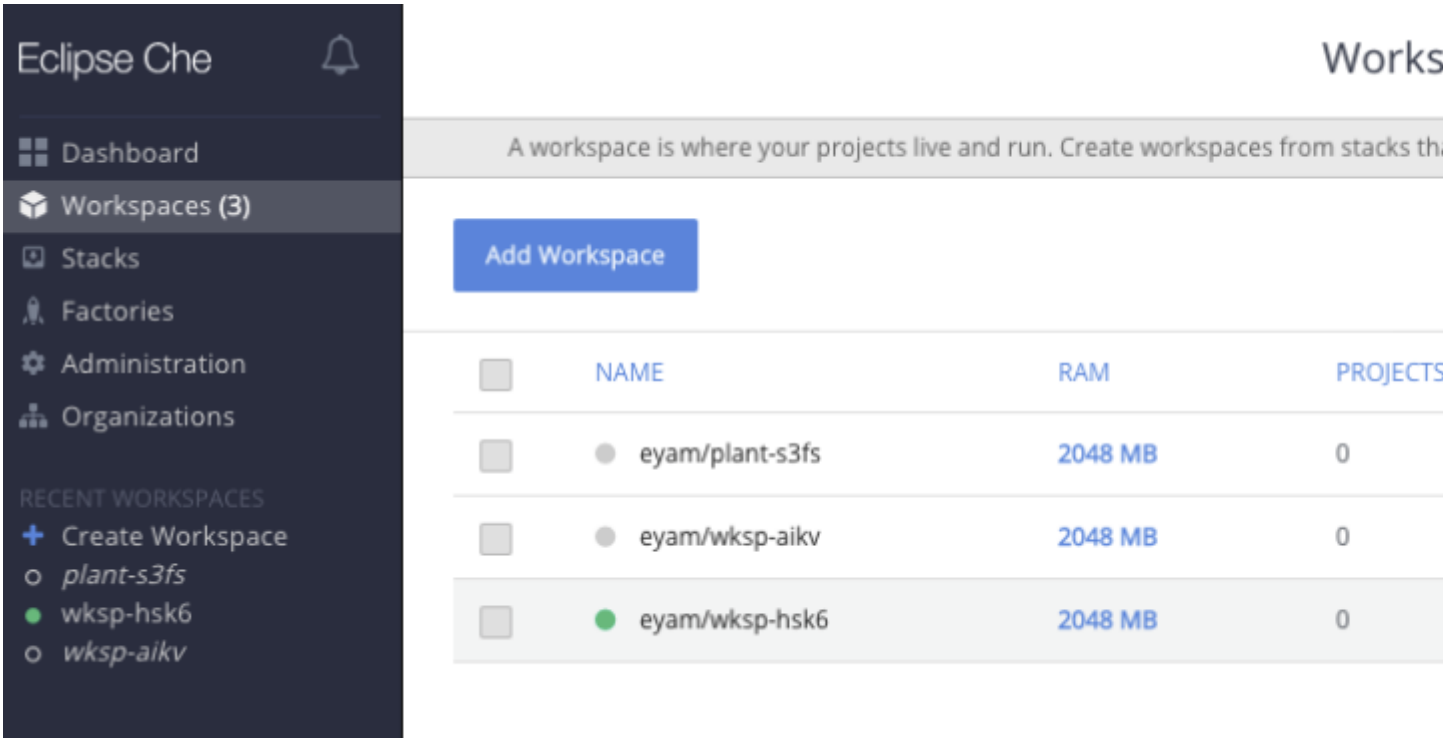
Option 1: At Workspace Creation

You can replace the auto-generated workspace name during creation.

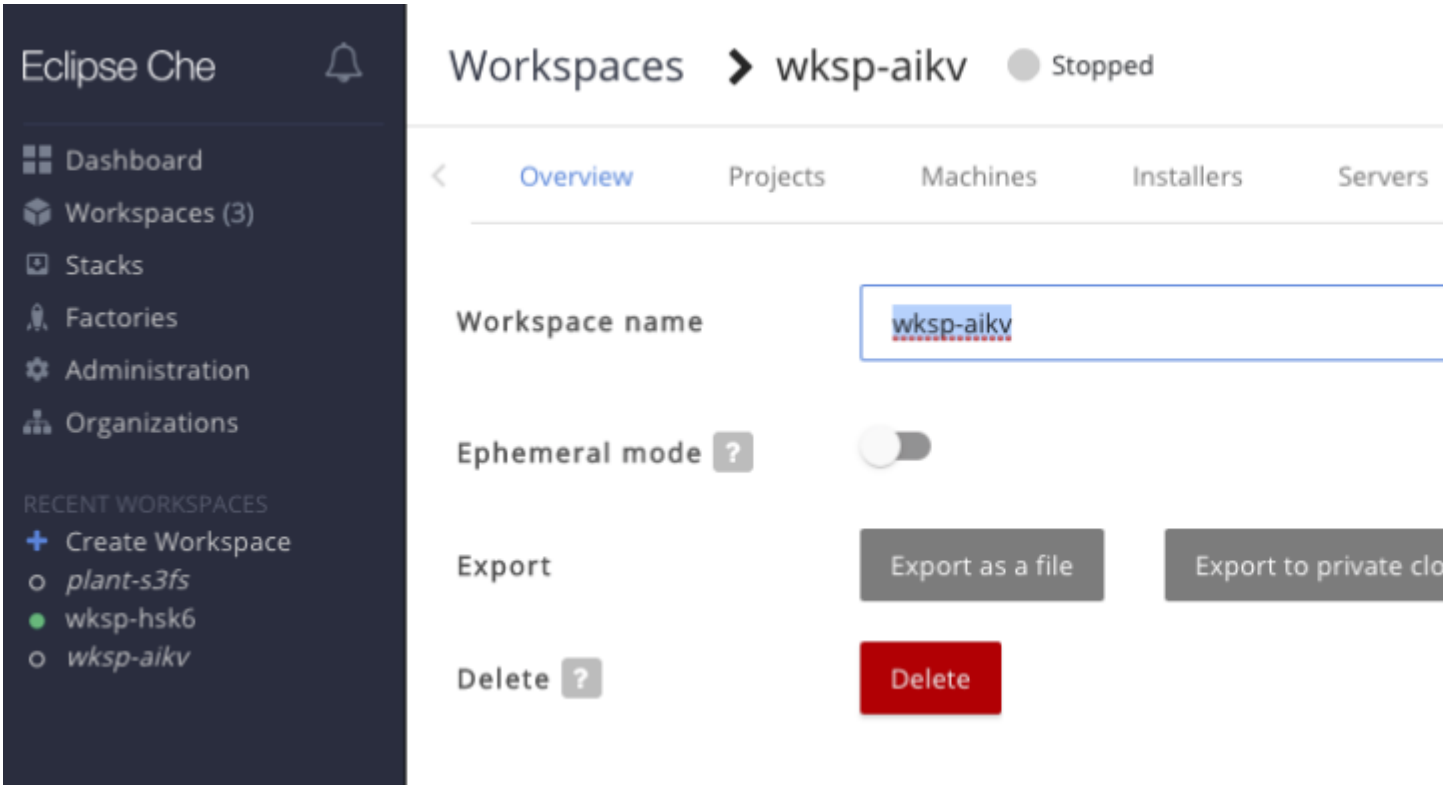


Option 2: Edit Existing Workspace

1. In the workspaces tab under the Che side panel, select the workspace you want to rename.



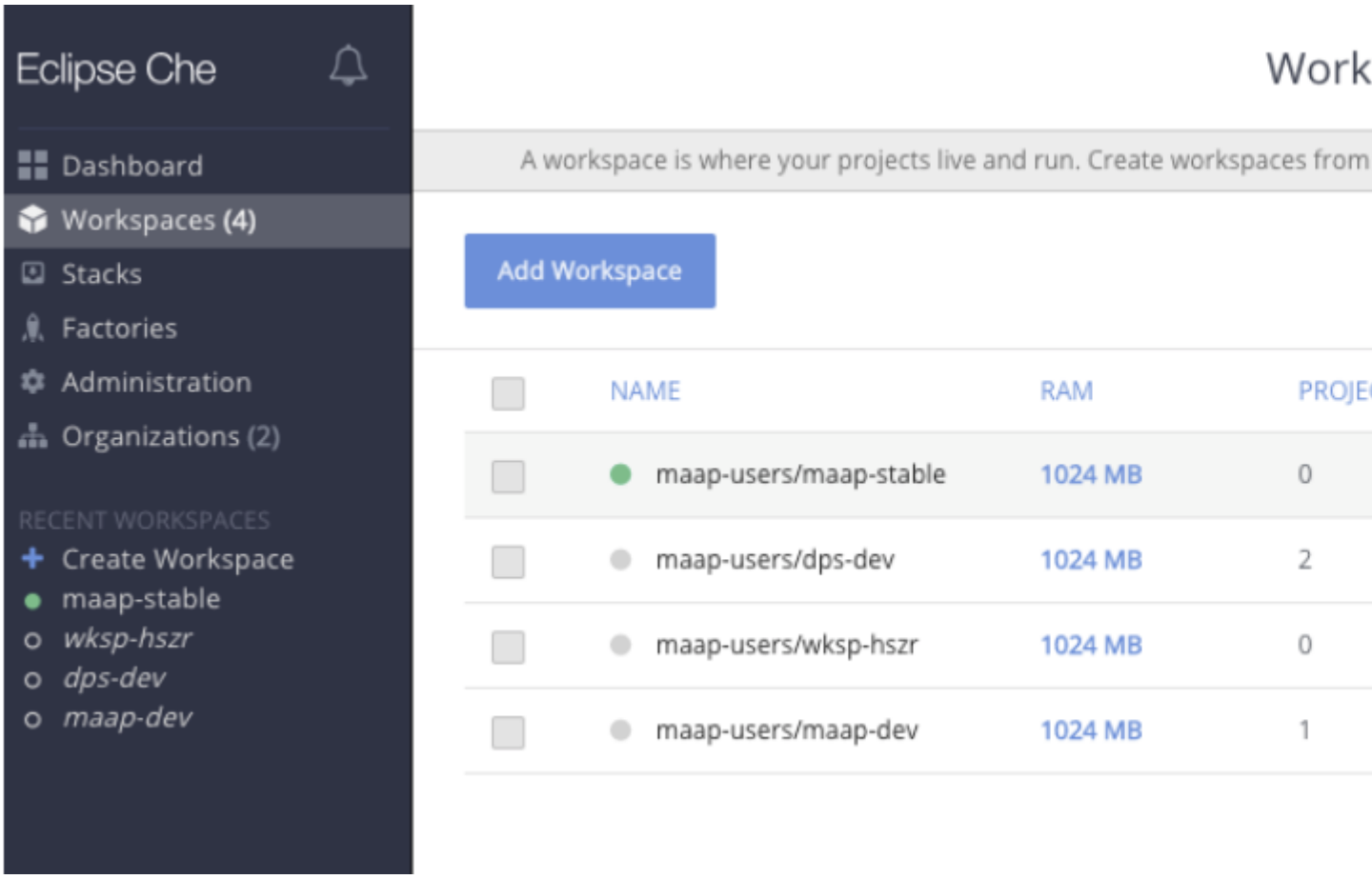
2. Under the Overview section, you can replace the Workspace name field with whatever you wish to name it.



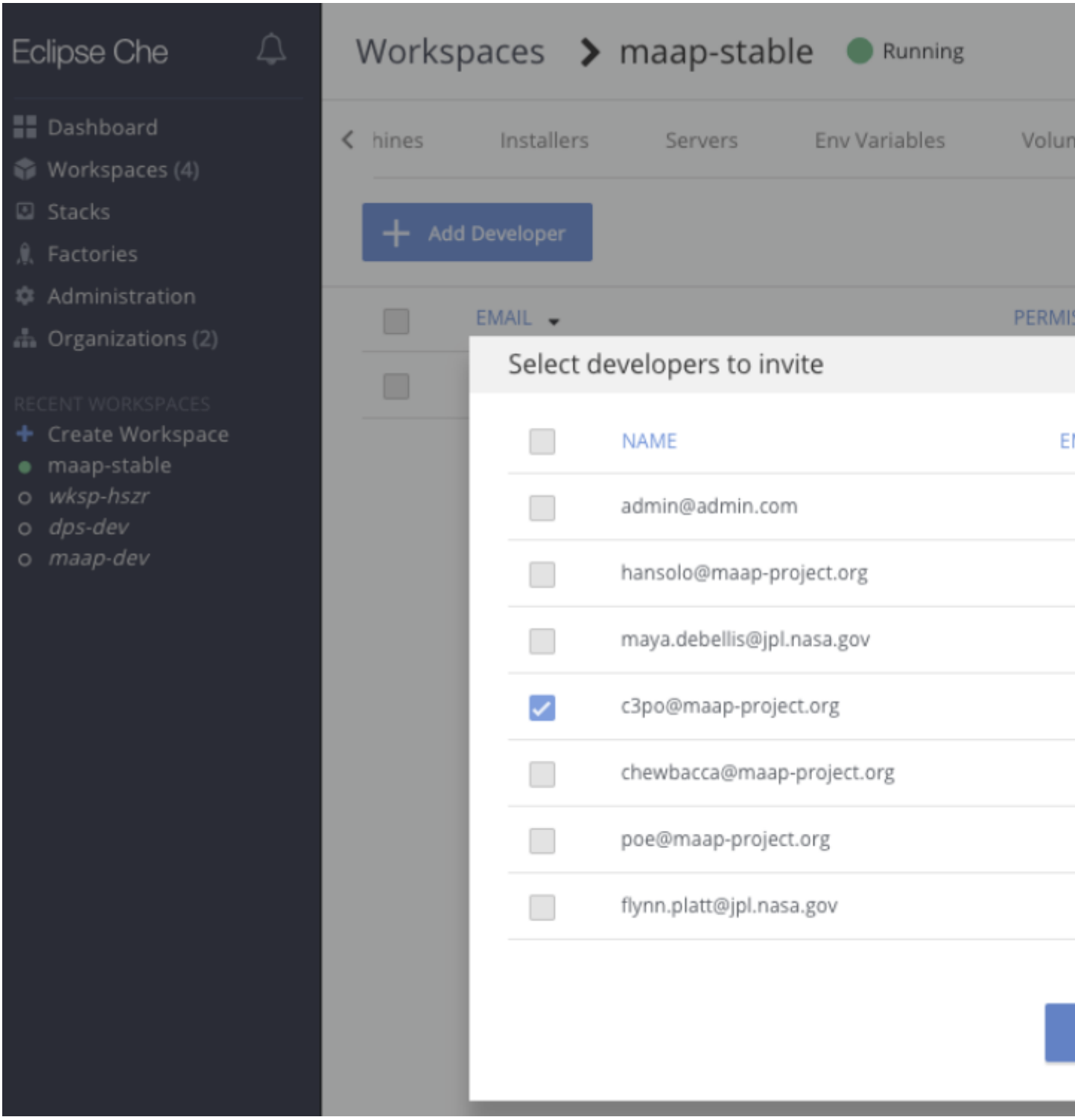
Caveat: no special characters like space, dollar sign, etc; name should start/end only with digits, Latin letters, or underscores

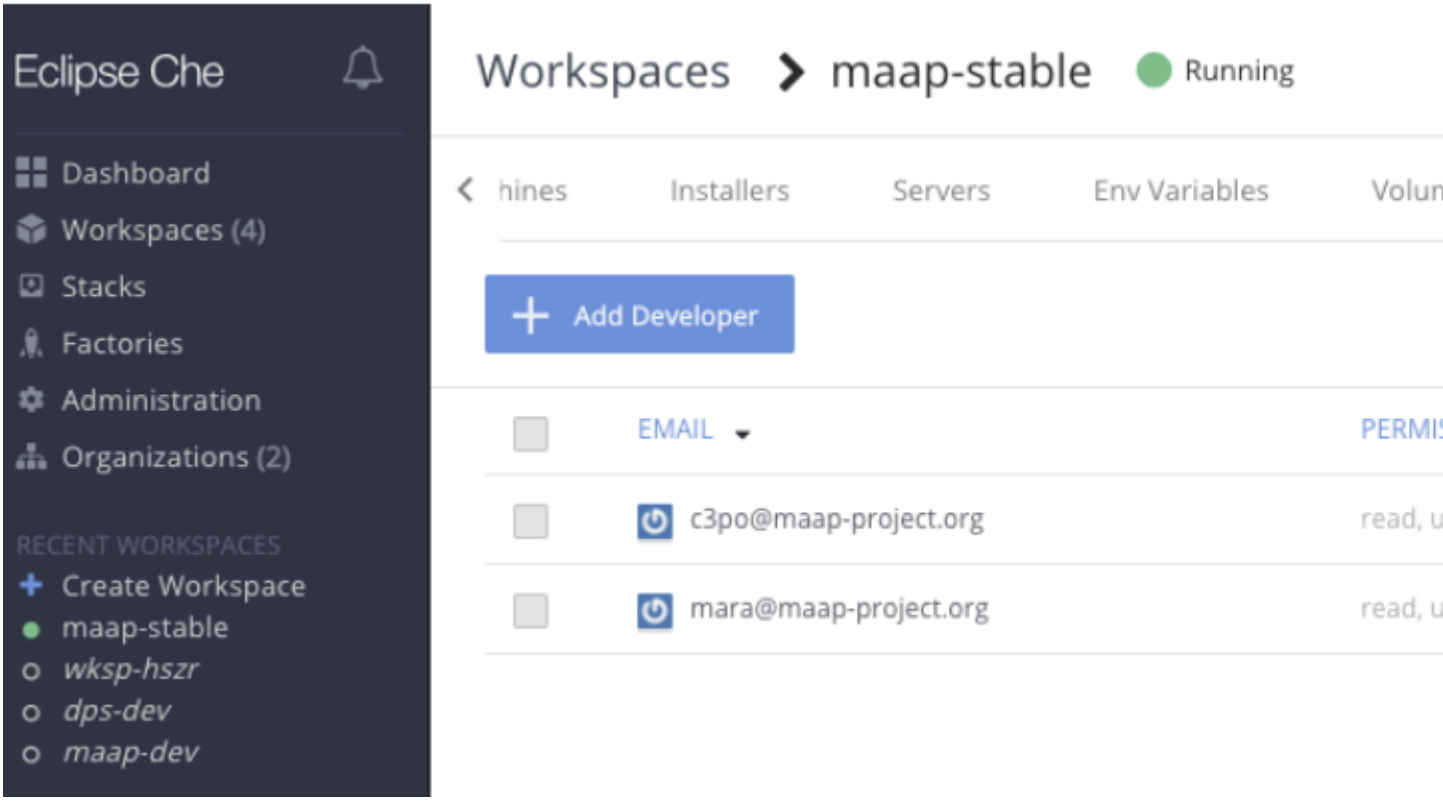
4.10.11 How Do I Share My Workspace With Another MAAP User?

1. In the workspaces tab under the Che side panel, select the workspace you want to share.



2. Under the Share section, click the button + Add Developer. Then check the box next to the email of the person/people you wish to share the workspace with and click *Share*. Only emails registered with MAAP can be used to share workspaces.


Eclipse Che
Dashboard
Workspaces (4)
Stacks
Factories
Administration
Organizations (2)
RECENT WORKSPACES
Create Workspace
maap-stable
wksp-hszr
dps-dev
maap-dev
Workspaces
maap-stable
Running
hines
Installers
Servers
Env Variables
Add Developer
EMAIL
Select developers to invite
NAME
admin@admin.com
hansolo@maap-project.org
maya.debellis@jpl.nasa.gov
c3po@maap-project.org
chewbacca@maap-project.org
poe@maap-project.org
flynn.platt@jpl.nasa.gov



3. Note: If the workspace was opened under an organization, the workspace can only be shared with members of that organization and not any other MAAP user.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search