Exploring GeoCODES

These pages introduce some work related to exploring the GeoCODES graph and document store.

The document store is an AWS S3 API compliant store leveraging the Minio open source project. It could leverage any such system such as AWS S3, Ceph, GCS, Wasbi, etc. The graph database is an RDF based triples accessed via SPARQL. The document store is synchronized to the graph and acts as the source of truth for the setup.

Both serve different functions and compliment each other. At this time most of these examples are using the document store but that will change. Mostly this is due to me exploring a few concepts such as:

- S3Select calls for data inspection, validation and sub-setting
- Dask based concurrent object access for updates and validation
    - SHACL
    - JSON inspection

Content in Jupyter Book

There are many ways to write content in Jupyter Book. This short section covers a few tips for how to do so.

Markdown Files

Whether you write your book's content in Jupyter Notebooks (`.ipynb`) or in regular markdown files (`.md`), you'll write in the same flavor of markdown called **MyST Markdown**.

What is MyST?

MyST stands for "Markedly Structured Text". It is a slight variation on a flavor of markdown called "CommonMark" markdown, with small syntax extensions to allow you to write **roles** and **directives** in the Sphinx ecosystem.

What are roles and directives?

Roles and directives are two of the most powerful tools in Jupyter Book. They are kind of like functions, but written in a markup language. They both serve a similar purpose, but **roles are written in one line**, whereas **directives span many lines**. They both accept different kinds of inputs, and what they do with those inputs depends on the specific role or directive that is being called.

Using a directive

At its simplest, you can insert a directive into your book's content like so:

```
```{mydirectivename}
My directive content
```
```

This will only work if a directive with name `mydirectivename` already exists (which it doesn't). There are many pre-defined directives associated with Jupyter Book. For example, to insert a note box into your content, you can use the following directive:

```
```{note}
Here is a note
```
```

This results in:

> **Note**
>
> Here is a note

In your built book.

For more information on writing directives, see the MyST documentation.

Using a role

Roles are very similar to directives, but they are less-complex and written entirely on one line. You can insert a role into your book's content with this pattern:

```
Some content {rolename}`and here is my role's content!`
```

Again, roles will only work if `rolename` is a valid role's name. For example, the `doc` role can be used to refer to another page in your book. You can refer directly to another page by its relative path. For example, the role syntax `{doc}`intro`` will result in: Exploring GeoCODES.

For more information on writing roles, see the MyST documentation.

Adding a citation

You can also cite references that are stored in a `bibtex` file. For example, the following syntax: `{cite}`holdgraf_evidence_2014`` will render like this: [HdHPK14].

Moreoever, you can insert a bibliography into your page with this syntax: The `{bibliography}` directive must be used for all the `{cite}` roles to render properly. For example, if the references for your book are stored in `references.bib`, then the bibliography is inserted with:

````
```{bibliography}
```
````

Resulting in a rendered bibliography that looks like:

[HdHPK14]
Christopher Ramsay Holdgraf, Wendy de Heer, Brian N. Pasley, and Robert T. Knight. Evidence for Predictive Coding in Human Auditory Cortex. In *International Conference on Cognitive Neuroscience*. Brisbane, Australia, Australia, 2014. Frontiers in Neuroscience.

Executing code in your markdown files

If you'd like to include computational content inside these markdown files, you can use MyST Markdown to define cells that will be executed when your book is built. Jupyter Book uses *jupytext* to do this.

First, add Jupytext metadata to the file. For example, to add Jupytext metadata to this markdown page, run this command:

```
jupyter-book myst init markdown.md
```

Once a markdown file has Jupytext metadata in it, you can add the following directive to run the code at build time:

````
```{code-cell}
print("Here is some code to execute")
```
````

When your book is built, the contents of any `{code-cell}` blocks will be executed with your default Jupyter kernel, and their outputs will be displayed in-line with the rest of your content.

For more information about executing computational content with Jupyter Book, see The MyST-NB documentation.

Content with notebooks

You can also create content with Jupyter Notebooks. This means that you can include code blocks and their outputs in your book.

Markdown + notebooks

As it is markdown, you can embed images, HTML, etc into your posts!

You an also \(add_{math}\) and

\[math^{blocks} \]
or

\[\begin{split} \begin{aligned} \mbox{mean} la_{tex} \\ \\ math blocks \end{aligned} \end{split}\]
But make sure you $Escape $your $dollar signs $you want to keep!

MyST markdown

MyST markdown works in Jupyter Notebooks as well. For more information about MyST markdown, check out the MyST guide in Jupyter Book, or see the MyST markdown documentation.

Code blocks and outputs

Jupyter Book will also embed your code blocks and output in your book. For example, here's some sample Matplotlib code:

```
from matplotlib import rcParams, cycler
import matplotlib.pyplot as plt
import numpy as np
plt.ion()
```

```
# Fixing random state for reproducibility
np.random.seed(19680801)

N = 10
data = [np.logspace(0, 1, 100) + np.random.randn(100) + ii for ii in range(N)]
data = np.array(data).T
cmap = plt.cm.coolwarm
rcParams['axes.prop_cycle'] = cycler(color=cmap(np.linspace(0, 1, N)))


from matplotlib.lines import Line2D
custom_lines = [Line2D([0], [0], color=cmap(0.), lw=4),
                Line2D([0], [0], color=cmap(.5), lw=4),
                Line2D([0], [0], color=cmap(1.), lw=4)]

fig, ax = plt.subplots(figsize=(10, 5))
lines = ax.plot(data)
ax.legend(custom_lines, ['Cold', 'Medium', 'Hot']);
```

There is a lot more that you can do with outputs (such as including interactive outputs) with your book. For more information about this, see the Jupyter Book documentation

Data Unit Testing, ranking, clustering

A section on some efforts to leverage the concept of data unit testing for projects. There is also some material below related to document clustering too. Which needs to be resolved.

About

Some material about what this page is about

- https://github.com/capitalone/DataProfiler
- https://github.com/awslabs/deequ

- https://github.com/dylan-profiler/visions
- https://github.com/pandas-profiling/pandas-profiling
- https://sherlock.media.mit.edu/
- http://brandonrose.org/clustering
- https://pypi.org/project/pytextrank/

References

Here is a reference to the intro. Here is a reference to .

Notebooks

About

The following notebooks are exploring both document store access but also approaches to semantic search, clustering and semantic similarity.

> **Note to the reader**
>
> The following are not good notebooks to learn from.. as I am learning too. ;)

SciKit K-means Clustering

Exploring SciKit-Learn (https://scikit-learn.org/stable/) for semantic search. Here I am looking at the k-means approach (https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_assumptions.html#sphx-glr-auto-examples-cluster-plot-kmeans-assumptions-py). Specifically the Mini-Batch K-Means clustering (https://scikit-learn.org/stable/modules/generated/sklearn.cluster.MiniBatchKMeans.html).

There are MANY approaches (https://scikit-learn.org/stable/auto_examples/index.html#cluster-examples) and it would be nice to get some guidance on what might make a good approach for building a similarity matrix of descriptive abstracts for datasets.

Gensim

This is an exploration of Gensim as a potential to create the “node set”, V, results from a semantic search. That would would be fed into a graph database and used to start the path searches and or analysis to create the desired results set for an interface.

> **Note**
>
> I’ve not been able to get Gensim to produce the final output I expect due to a error from one of the library dependencies. I’ve tried clean environments and even a clean VM and not been able to get 64 bit linux to work.

This V_semsearch might be intersected with a V_spatial and or others to form a node set for the graph. This is essentially a search “preprocessor”. Another potential set might be V_text that uses a more classical full text index approaches.

TXTAI

Exploring TXTAI (https://github.com/neuml/txtai) as yet another candidate in generating a set of nodes (V) that could be fed into a graph as the initial node set. Essentially looking at semantic search for the initial full text index search and then moving on to a graph database (triplestore in my case) fort he graph search / analysis portion.

This is the “search broker” concept I’ve been trying to resolve.

EarthCube Graph Analytics Exploration

This is the start of learning a bit about leveraging graph analytics to assess the EarthCube graph and explore both the relationships but also look for methods to better search the graph for relevant connections.

SciKit K-means Clustering

Exploring SciKit-Learn (https://scikit-learn.org/stable/) for semantic search. Here I am looking at the k-means approach (https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_assumptions.html#sphx-glr-auto-examples-cluster-plot-kmeans-assumptions-py). Specifically the Mini-Batch K-Means clustering (https://scikit-learn.org/stable/modules/generated/sklearn.cluster.MiniBatchKMeans.html).

There are MANY approaches (https://scikit-learn.org/stable/auto_examples/index.html#cluster-examples) and it would be nice to get some guidance on what might make a good approach for building a similarity matrix of descriptive abstracts for datasets.

Notes:

Some search concepts are:

- Geolocation: “Gulf of Mexico”, “Bay of Fundy”, “Georges Bank”
- Parameter: “chlorophyll”, “abundance”, “dissolved organic carbon”
- Species: “calanus finmarchicus”
- Instrument”: “CTD”, “bongo net”
- Project: “C-DEBI”

The worry is these by themselves just become “frequncy searches”. What we want are search phrases that we can pull semantics from.

```
!apt-get install libproj-dev proj-data proj-bin -qq
!apt-get install libgeos-dev -qq
```

Imports and Inits

```
%%capture
!pip install -q PyLD
!pip install -q boto3
!pip install -q s3fs
!pip install -q minio
!pip install -q rdflib==4.2.2
!pip install -q cython
!pip install -q cartopy
!pip install -q SPARQLWrapper
!pip install -q geopandas
!pip install -q contextily==1.0rc2
!pip install -q rdflib-jsonld==0.5.0
!pip install -q sklearn
```

```
import requests
import json
import rdflib
import pandas as pd
from pandas.io.json import json_normalize
import concurrent.futures
import urllib.request
import dask, boto3
from SPARQLWrapper import SPARQLWrapper, JSON
import numpy as np
import geopandas
import matplotlib.pyplot as plt
import shapely

from sklearn.cluster import MiniBatchKMeans
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import PCA


dbsparql = "http://dbpedia.org/sparql"
okn = "http://graph.openknowledge.network/blazegraph/namespace/samplesearth/sparql"
whoi = "https://lod.bco-dmo.org/sparql"

# Pandas options
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
pd.set_option('display.width', None)
pd.set_option('display.max_colwidth', None)
```

Gleaner Data

First lets load up some of the data Gleaner has collected. This is just simple data graph objects and not any graphs or other processed products from Gleaner.

```
# Set up our S3FileSystem object
import s3fs
oss = s3fs.S3FileSystem(
      anon=True,
      client_kwargs = {"endpoint_url":"https://oss.geodex.org"}
   )
# oss.ls('gleaner/summoned')
```

```
# # A simple example of grabbing one item...
# import json

# jld = ""
# with
oss.open('gleaner/summoned/opentopo/231f7fa996be8bd5c28b64ed42907b65cca5ee30.jsonld',
'rb') as f:
#   #print(f.read())
#    jld = f.read().decode("utf-8", "ignore").replace('\n',' ')
#    json = json.loads(jld)

# document = json['description']
# print(document)
```

```
import json

@dask.delayed()
def read_a_file(fn):
    # or preferably open in text mode and json.load from the file
    with oss.open(fn, 'rb') as f:
        #return json.loads(f.read().replace('\n',' '))
        return json.loads(f.read().decode("utf-8", "ignore").replace('\n',' '))

# buckets = ['gleaner/summoned/dataucaredu', 'gleaner/summoned/getiedadataorg',
'gleaner/summoned/iris', 'gleaner/summoned/opentopo', 'gleaner/summoned/ssdb',
'gleaner/summoned/wikilinkedearth', 'gleaner/summoned/wwwbco-dmoorg',
'gleaner/summoned/wwwhydroshareorg', 'gleaner/summoned/wwwunavcoorg']

buckets = ['gleaner/summoned/opentopo']

filenames = []

for d in range(len(buckets)):
  print("indexing {}".format(buckets[d]))
  f = oss.ls(buckets[d])
  filenames += f

#filenames = oss.cat('gleaner/summoned/opentopo', recursive=True)
output = [read_a_file(f) for f in filenames]
print(len(filenames))
```

```
indexing gleaner/summoned/opentopo
654
```

```
%%time

gldf = pd.DataFrame(columns=['name', 'url', "keywords", "description", "object"])

#for key in filenames:

for doc in range(len(output)):
#for doc in range(10):
#for key in filenames:
  #if ".jsonld" in key:
  if "/.jsonld" not in filenames[doc] :
    try:
      jld = output[doc].compute()
    except:
      print(filenames[doc])
      print("Doc has bad encoding")

    # TODO  Really need to flatten and or frame this
    try:
      desc = jld["description"]
    except:
      desc = "NA"
      continue
    kws = "keywords" #jld["keywords"]
    name = jld["name"]
    url = "NA" #jld["url"]
    object = filenames[doc]

    gldf = gldf.append({'name':name, 'url':url, 'keywords':kws, 'description': desc,
'object': object}, ignore_index=True)
```

```
CPU times: user 17.9 s, sys: 384 ms, total: 18.3 s
Wall time: 1min 1s
```

```
gldf.info()
# gldf.to_parquet('index.parquet.gzip',  compression='gzip') # optional save state here
...  one master parquet for Geodex?
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 654 entries, 0 to 653
Data columns (total 5 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   name         654 non-null    object
 1   url          654 non-null    object
 2   keywords     654 non-null    object
 3   description  654 non-null    object
 4   object       654 non-null    object
dtypes: object(5)
memory usage: 25.7+ KB
```

Feature extraction

Let's just worry about the description section for now.

```
vec = TfidfVectorizer(stop_words="english")
vec.fit(gldf.description.values)
features = vec.transform(gldf.description.values)
```

Kmeans clusters

Guess at the number a few times since we don't have a prior idea who many natural clusterings we might expect

```
random_state = 0

cls = MiniBatchKMeans(n_clusters=20, random_state=random_state)
cls.fit(features)
```

```
MiniBatchKMeans(n_clusters=20, random_state=0)
```

```
# predict cluster labels for new dataset
cls.predict(features)

# to get cluster labels for the dataset used while
# training the model (used for models that does not
# support prediction on new dataset).
cls.labels_
```

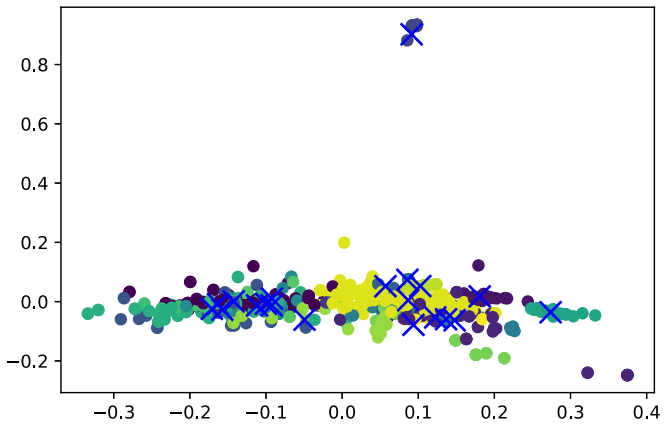
```
array([18,  9, 18, 18, 18,  5, 18, 12, 18, 12, 15, 16,  0,  0, 15, 18, 15,
        1,  0,  5, 18, 12, 18,  0, 14,  8, 18, 13, 18, 12,  8, 18, 12, 18,
       16, 12,  2, 18,  0,  0, 17, 18,  6,  0,  0,  5,  2, 12,  2, 18, 12,
       15,  2, 17,  9, 11,  0, 12,  9,  1,  2,  0, 18,  2,  5, 12, 13, 15,
       17,  2,  2, 18, 18, 16,  0,  2, 18,  0, 13,  5, 18,  2,  0, 11, 18,
        0, 12, 14,  0,  0,  9,  8,  9, 11,  5,  9, 12, 17,  1, 12,  7,  0,
        9,  5,  2, 12, 13, 18, 18, 12,  2,  0,  0, 18, 18,  0,  0, 18, 18,
        8,  2,  2,  0, 18, 13,  1,  2,  9, 13,  0,  8, 18, 13, 12,  1,  2,
        2,  0, 16,  5,  0,  9,  0, 12,  0, 13, 12, 18, 18,  1,  2,  8, 13,
        1,  9,  0,  0,  2, 13, 15,  4,  0,  2, 12,  2, 18,  9,  1, 13,  5,
        0,  0, 18, 18,  0,  9, 15, 14, 18, 19,  3, 18, 15, 18,  2, 18,  0,
        8, 16, 18, 18,  0,  9, 13, 18, 17,  2,  2,  2, 18, 18, 17,  5, 11,
       18, 18, 18, 18, 13,  2, 18, 18, 18, 18, 18, 18, 19,  0, 18,  5,  2,
        5,  2,  0,  0,  2,  9,  4,  1, 18,  0,  4,  3,  9, 16,  1, 16,  0,
        5, 13, 12,  9,  2, 11, 12, 12,  2, 18, 18,  2,  9, 12,  3, 12, 18,
        2, 18,  7, 18,  0,  3,  0,  2,  0,  5,  9,  1,  2, 16,  3, 18, 13,
        9, 14, 18, 12, 15, 11,  2, 18,  5,  9,  0,  0,  9,  5, 18,  2,  2,
       18, 10,  2, 12,  8, 12, 15,  0, 18,  2, 18, 10,  2,  9, 18, 16,  0,
       16,  8,  5, 11, 11,  2,  5, 12,  3, 18, 16,  5,  2,  0, 18,  8,  8,
        0, 15, 17, 12, 15,  2,  0,  2, 12, 18, 15, 12,  5, 18,  5, 18,  5,
       11,  8, 18, 14, 18, 15,  5,  0, 12, 18, 18,  2,  0,  9,  1,  0,  4,
       17, 17,  0,  5, 18,  2, 12,  5, 17,  2, 18, 15,  4, 12, 12, 12,  3,
       18,  2, 17, 18,  0,  5, 18, 12,  2, 15, 18, 18,  2,  5,  0, 16,  0,
       15,  0,  5, 18, 18, 12, 11,  3,  2, 18,  8,  5,  0,  1,  8,  8,  5,
        2,  0,  2, 18, 18, 12, 18, 12, 12,  4,  1,  1, 15,  8, 18,  8,  5,
        6, 18,  2,  0, 18, 17, 14,  2,  9, 18, 18, 18, 12,  0,  2,  4,  0,
       18,  0,  9,  0,  2,  3, 13, 18,  8,  4, 13, 18, 18,  0, 18,  0,  6,
       16, 13, 18,  7,  2, 17,  0,  5, 17,  2,  6,  1,  2, 18, 13,  9, 18,
       18, 12,  8,  4, 18, 13,  2, 12, 18, 16,  5, 18, 13,  0,  9,  0, 18,
        2, 13, 16, 11, 12,  3, 13, 18, 18,  5, 18, 12, 18,  2,  9, 18,  0,
        2, 18,  9,  0, 18, 19, 18, 18,  0, 19, 18, 16, 18,  1,  7, 12,  2,
       18,  9, 11, 18, 18,  2,  5, 15,  2,  4,  3,  2, 13,  0, 13, 18,  0,
        0,  0, 11, 18, 18,  9, 12, 13,  7,  0,  0,  2, 15,  5,  5,  9,  8,
        0, 13, 18, 15, 18, 12, 18,  7,  2,  0, 18, 18, 12,  4,  3, 16,  0,
       18, 12, 12, 18, 18,  5,  1, 11, 13,  9, 13,  0, 18, 18,  0, 18, 18,
        0, 18, 12, 15, 18, 18,  0,  2, 12,  0,  0, 17, 13, 12, 16,  2,  4,
       13, 12, 18, 18, 18, 18, 18, 18, 11, 12, 11, 11,  2, 12,  2, 12, 12,
        9, 18, 16, 13,  2,  2, 12, 11, 12, 15, 17, 12, 18, 16, 18, 18,  0,
       18, 18,  1,  2,  0,  5,  0,  2], dtype=int32)
```

```
# reduce the features to 2D
pca = PCA(n_components=2, random_state=random_state)
reduced_features = pca.fit_transform(features.toarray())

# reduce the cluster centers to 2D
reduced_cluster_centers = pca.transform(cls.cluster_centers_)
```

```
plt.scatter(reduced_features[:,0], reduced_features[:,1], c=cls.predict(features))
plt.scatter(reduced_cluster_centers[:, 0], reduced_cluster_centers[:,1], marker='x',
s=150, c='b')
```

```
<matplotlib.collections.PathCollection at 0x7f4cbaa66340>
```



Nearest Neighbor testing

```
from sklearn.neighbors import NearestNeighbors
knn = NearestNeighbors(n_neighbors=10, metric='cosine')
knn.fit(features)
```

```
NearestNeighbors(metric='cosine', n_neighbors=10)
```

```
knn.kneighbors(features[0:1], return_distance=False)
```

```
array([[  0, 520, 395, 248, 355, 276, 176, 327, 598, 638]])
```

```
knn.kneighbors(features[0:1], return_distance=True)
```

```
(array([[0.        , 0.05126516, 0.37921939, 0.4165376 , 0.6984332 ,
         0.87318168, 0.87318168, 0.87318168, 0.87318168, 0.88349159]]),
 array([[  0, 520, 395, 248, 355, 276, 176, 327, 598, 638]]))
```

Search testing

run a few test searches and then plot the first n (4) results

```
input_texts = ["New Zeland lidar data", "California housing", "new madrid seismic
zone"]
input_features = vec.transform(input_texts)

D, N = knn.kneighbors(input_features, n_neighbors=4, return_distance=True)

for input_text, distances, neighbors in zip(input_texts, D, N):
    print("Input text = ", input_text[:200], "\n")
    for dist, neighbor_idx in zip(distances, neighbors):
        print("Distance = ", dist, "Neighbor idx = ", neighbor_idx)
        print(gldf.name[neighbor_idx])
        print(gldf.description[neighbor_idx][:400])
        print("-"*200)
    print("="*200)
    print()
```

```
Input text =  New Zeland lidar data

Distance =  0.6759095922425897 Neighbor idx =  59
Bay of Plenty, New Zealand 2018-2019
Lidar was captured for BOPLASS Limited by AAM New Zealand between December 2018 and
April 2019. The dataset was generated by AAM New Zealand and their subcontractors.
The survey area includes Tauranga, Rotorua and Whakatane and the surrounding area.
Data management and distribution is by Land Information New Zealand.
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
------------------------------
Distance =  0.6833093201645597 Neighbor idx =  470
Auckland South, New Zealand 2016-2017
Lidar was captured for Auckland Council by AAM New Zealand between September 2016
through to June 2017. The original dataset was generated by AAM New Zealand and their
subcontractors. The survey area covers the southern Auckland suburbs and regions.
Data management and distribution is by Land Information New Zealand.
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
------------------------------
Distance =  0.7069674707157929 Neighbor idx =  125
Marsden Point, Northland, New Zealand 2016
Lidar was captured for Land Information New Zealand by Aerial Surveys in November
2016. The dataset was generated by Aerial Surveys and their subcontractors. The
survey area includes Marsden Point and the surrounding area. Data management and
distribution is by Land Information New Zealand.
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
------------------------------
Distance =  0.7311078986510137 Neighbor idx =  134
Canterbury, New Zealand 2018-2019
Lidar was captured for Environment Canterbury Regional Council by Aerial Surveys New
Zealand in March 2018 through to May 2019. The dataset was generated by Aerial
Surveys New Zealand and their subcontractors. The survey area includes Amuri Plain,
Ashburton, Fairlie Foothills, Pegasus, Motunau, Selwyn North, Selwyn South and
Waimate. Data management and distribution is by Land Information New Zeal
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
```

```
------------------------------
================================================================================
================================================================================
==============================

Input text =  California housing

Distance =  0.7222151549783433 Neighbor idx =  129
Merced, CA: Origin and Evolution of the Mima Mounds
NCALM Seed Project. PI: Sarah Reed, University of California, Berkeley. This lidar
survey was conducted on Sunday, September 17, 2006 on a 33 square kilometer polygon
northeast of Merced in Merced County, California.  These data were collected to
investigate the origin and evolution of Mima mounds in California's Central Valley.
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
------------------------------
Distance =  0.7222151549783433 Neighbor idx =  359
Merced, CA: Origin and Evolution of the Mima Mounds
NCALM Seed Project. PI: Sarah Reed, University of California, Berkeley. This lidar
survey was conducted on Sunday, September 17, 2006 on a 33 square kilometer polygon
northeast of Merced in Merced County, California.  These data were collected to
investigate the origin and evolution of Mima mounds in California's Central Valley.
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
------------------------------
Distance =  0.7804319924949502 Neighbor idx =  314
South Fork Eel River, CA Watershed Morphology
NCALM Project. Mary Power, University of California Berkeley. The survey area
consisted of a polygon located 20 kilometers east of Dos Rios, California. The survey
took place over eight flights from 6/27/2004 to 6/30/2004.
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
------------------------------
Distance =  0.7804319924949502 Neighbor idx =  232
South Fork Eel River, CA Watershed Morphology
NCALM Project. Mary Power, University of California Berkeley. The survey area
consisted of a polygon located 20 kilometers east of Dos Rios, California. The survey
took place over eight flights from 6/27/2004 to 6/30/2004.
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
------------------------------
================================================================================
================================================================================
==============================

Input text =  new madrid seismic zone

Distance =  0.6004996071077049 Neighbor idx =  286
Central U.S. ARRA Lidar, New Madrid Seismic Zone
This dataset is intended for researchers interested in active tectonics and
earthquake hazards research in the New Madrid Seismic Zone. The target areas were
developed and defined by USGS scientists in collaboration with colleagues working in
this region. The target areas are: the Blytheville Arch, the Meeman-Shelby lineament,
the Reelfoot scarp and the northern New Madrid area. The data collectio
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
------------------------------
Distance =  0.6004996071077049 Neighbor idx =  400
Central U.S. ARRA Lidar, New Madrid Seismic Zone
This dataset is intended for researchers interested in active tectonics and
earthquake hazards research in the New Madrid Seismic Zone. The target areas were
developed and defined by USGS scientists in collaboration with colleagues working in
this region. The target areas are: the Blytheville Arch, the Meeman-Shelby lineament,
the Reelfoot scarp and the northern New Madrid area. The data collectio
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
------------------------------
Distance =  0.8295725353504988 Neighbor idx =  487
New Madrid Seismic Zone
The New Madrid Seismic Zone (NMSZ) has been responsible for producing some of the
largest intraplate earthquakes on record (Tuttle et al., 2002). Paleoseismologic
studies of sand blows and the Reelfoot fault show that earthquakes occurred in the
last 4000 years at intervals of approximately 400-600 years (Kelson et al., 1995;
Tuttle et al., 2002; Holbrook et al., 2006). The 1811-1812 NMSZ sequence
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
------------------------------
Distance =  0.8766571303555153 Neighbor idx =  550
Durham, NH: Hyporheic Zone Extent and Exchange in a Coastal Stream
NCALM Seed Project PI: Danna Truslow, University of New Hampshire. The survey area is
an irregular polygon approximately 16 km West of Portsmouth, NH and enclosing 42.5
square kilometers. The data were collected to study hyporheic zone extent and
exchange in a coastal New Hampshire stream using heat as a tracer.
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
------------------------------
================================================================================
```

```
================================================================================
==============================
```

Gensim

This is an exploration of Gensim as a potential to create the “node set”, V, results from a semantic search. That would wouild be fed into a graph database and used to start the path searches and or analysis to create the desired results set for an interface.

This V_semsearch might be intersected with a V_spatial and or others to form a node set for the graph. This is essentially a search “preprocessor”. Another potential set might be V_text that usses more classical full text index approaches.

References

- https://github.com/topics/document-similarity
- https://radimrehurek.com/gensim/auto_examples/core/run_core_concepts.html

```
# %%capture
!pip install -q --upgrade gensim
!pip install -q  dask[dataframe] --upgrade
!pip install -q s3fs
!pip install -q boto3
!pip install -q python-Levenshtein
```

```
ERROR: pip's dependency resolver does not currently take into account all the
packages that are installed. This behaviour is the source of the following dependency
conflicts.
aiobotocore 1.3.0 requires botocore<1.20.50,>=1.20.49, but you have botocore 1.20.57
which is incompatible.
```

```
import pprint
import spacy
from spacy import displacy
import pandas as pd
import dask, boto3
import dask.dataframe as dd
```

Gleaner Data

First lets load up some of the data Gleaner has collected. This is just simple data graph objects and not any graphs or other processed products from Gleaner.

```
# Set up our S3FileSystem object
import s3fs
oss = s3fs.S3FileSystem(
      anon=True,
      client_kwargs = {"endpoint_url":"https://oss.geodex.org"}
   )
```

Further examples

```
import json

@dask.delayed()
def read_a_file(fn):
    # or preferably open in text mode and json.load from the file
    with oss.open(fn, 'rb') as f:
        #return json.loads(f.read().replace('\n',' '))
        return json.loads(f.read().decode("utf-8", "ignore").replace('\n',' '))

filenames = oss.ls('gleaner/summoned/opentopo')
output = [read_a_file(f) for f in filenames]
```

```
gldf = pd.DataFrame(columns=['name', 'url', "keywords", "description"])

for doc in range(len(output)):
#for doc in range(10):
  try:
    jld = output[doc].compute()
  except:
    print("Doc has bad encoding")

  # TODO  Really need to flatten and or frame this

  desc = jld["description"]
  kws = jld["keywords"]
  name = jld["name"]
  url = jld["url"]
  gldf = gldf.append({'name':name, 'url':url, 'keywords':kws, 'description': desc},
ignore_index=True)
```

```
# gldf.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 654 entries, 0 to 653
Data columns (total 4 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   name         654 non-null    object
 1   url          654 non-null    object
 2   keywords     654 non-null    object
 3   description  654 non-null    object
dtypes: object(4)
memory usage: 20.6+ KB
```

```
import re

# document = "Human machine interface for lab abc computer applications"

# text_corpus = [
#     "Human machine interface for lab abc computer applications",
#     "A survey of user opinion of computer system response time",
#     "The EPS user interface management system",
#     "System and human system engineering testing of EPS",
#     "Relation of user perceived response time to error measurement",
#     "The generation of random binary unordered trees",
#     "The intersection graph of paths in trees",
#     "Graph minors IV Widths of trees and well quasi ordering",
#     "Graph minors A survey",
# ]

text_corpus = []

# for i in range(len(gldf)):
#   text_corpus += gldf.at[i,'description']

for i in range(len(gldf)):
# for i in range(10):
  d = gldf.at[i,'description']
  # d.replace('(', '').replace(')', '').replace('\"', '')
  dp = re.sub(r'[^A-Za-z0-9 ]+', '', str(d))
  text_corpus.append(str(dp))

  # if not "http" in d:
  #   if not "(" in d:
  #     if not "<" in d:
  #       text_corpus.append(str(d))
```

```
# for x in range(len(text_corpus)):
#   print(text_corpus[x])
```

```
# Create a set of frequent words
stoplist = set('for a of the and to in'.split(' '))
# Lowercase each document, split it by white space and filter out stopwords
texts = [[word for word in document.lower().split() if word not in stoplist]
         for document in text_corpus]

# Count word frequencies
from collections import defaultdict
frequency = defaultdict(int)
for text in texts:
    for token in text:
        frequency[token] += 1

# Only keep words that appear more than once
processed_corpus = [[token for token in text if frequency[token] > 1] for text in
texts]
# pprint.pprint(processed_corpus)
```

```
from gensim import corpora

dictionary = corpora.Dictionary(processed_corpus)
print(dictionary)
```

```
Dictionary(4443 unique tokens: ['2010', '2016066280', 'aa', 'affonso',
'airborne']...)
```

```
# pprint.pprint(dictionary.token2id)
```

```
# Side demo
new_doc = "Human computer interaction"
new_vec = dictionary.doc2bow(new_doc.lower().split())
print(new_vec)
```

```
[(1304, 1)]
```

```
bow_corpus = [dictionary.doc2bow(text) for text in processed_corpus]
# pprint.pprint(bow_corpus)
```

```
from gensim import models

# train the model
tfidf = models.TfidfModel(bow_corpus)

# transform the "system minors" string
words = "system minors".lower().split()
print(tfidf[dictionary.doc2bow(words)])
```

```
[(212, 1.0)]
```

```
from gensim import similarities

index = similarities.SparseMatrixSimilarity(tfidf[bow_corpus], num_features=12)
```

```
query_document = 'Airborne Laser Mapping'.split()
query_bow = dictionary.doc2bow(query_document)
sims = index[tfidf[query_bow]]
print(list(enumerate(sims)))
```

```
for document_number, score in sorted(enumerate(sims), key=lambda x: x[1],
reverse=True):
    print(document_number, score)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-7663932f4991> in <module>()
----> 1 for document_number, score in sorted(enumerate(sims), key=lambda x: x[1],
reverse=True):
      2     print(document_number, score)

NameError: name 'sims' is not defined
```

Gleaner & txtai

About

Exploring TXTAI (https://github.com/neuml/txtai) as yet another canidate in generating a set of nodes (V) that could be fed into a graph as the initial node set. Essentially looking at semantic search for the initial full text index search and then moving on to a graph database (triplestore in my case) fort he graph search / analysis portion.

This is the “search broker” concept I’ve been trying to resolve.

References

- https://github.com/neuml/txtai

Imports and Installs

```
# %%capture
!pip install -q git+https://github.com/neuml/txtai
!pip install -q  'fsspec>=0.3.3'
!pip install -q  s3fs
!pip install -q  boto3
!pip install -q  spacy
!pip install -q  pyarrow
!pip install -q  fastparquet
```

```
ERROR: pip's dependency resolver does not currently take into account all the
packages that are installed. This behaviour is the source of the following dependency
conflicts.
boto3 1.17.55 requires botocore<1.21.0,>=1.20.55, but you have botocore 1.20.49 which
is incompatible.
ERROR: pip's dependency resolver does not currently take into account all the
packages that are installed. This behaviour is the source of the following dependency
conflicts.
aiobotocore 1.3.0 requires botocore<1.20.50,>=1.20.49, but you have botocore 1.20.56
which is incompatible.
```

```
import pprint
import spacy
from spacy import displacy
import pandas as pd
import dask, boto3
import dask.dataframe as dd
from txtai.embeddings import Embeddings

# Create embeddings model, backed by sentence-transformers & transformers
embeddings = Embeddings({"method": "transformers", "path": "sentence-transformers/bert-
base-nli-mean-tokens"})
```

Gleaner Data

First lets load up some of the data Gleaner has collected. This is just simple data graph objects and not any graphs or other processed products from Gleaner.

```
# Set up our S3FileSystem object
import s3fs
oss = s3fs.S3FileSystem(
      anon=True,
      client_kwargs = {"endpoint_url":"https://oss.geodex.org"}
   )
# oss.ls('gleaner/summoned')
```

```
# # A simple example of grabbing one item...
# import json

# jld = ""
# with
oss.open('gleaner/summoned/opentopo/231f7fa996be8bd5c28b64ed42907b65cca5ee30.jsonld',
'rb') as f:
#   #print(f.read())
#    jld = f.read().decode("utf-8", "ignore").replace('\n',' ')
#    json = json.loads(jld)

# document = json['description']
# print(document)
```

```
import json

@dask.delayed()
def read_a_file(fn):
    # or preferably open in text mode and json.load from the file
    with oss.open(fn, 'rb') as f:
        #return json.loads(f.read().replace('\n',' '))
        return json.loads(f.read().decode("utf-8", "ignore").replace('\n',' '))

# buckets = ['gleaner/summoned/dataucaredu', 'gleaner/summoned/getiedadataorg',
'gleaner/summoned/iris', 'gleaner/summoned/opentopo', 'gleaner/summoned/ssdb',
'gleaner/summoned/wikilinkedearth', 'gleaner/summoned/wwwbco-dmoorg',
'gleaner/summoned/wwwhydroshareorg', 'gleaner/summoned/wwwunavcoorg']

buckets = ['gleaner/summoned/opentopo']

filenames = []

for d in range(len(buckets)):
  print("indexing {}".format(buckets[d]))
  f = oss.ls(buckets[d])
  filenames += f

#filenames = oss.cat('gleaner/summoned/opentopo', recursive=True)
output = [read_a_file(f) for f in filenames]
print(len(filenames))
```

```
indexing gleaner/summoned/opentopo
654
```

```
%%time

gldf = pd.DataFrame(columns=['name', 'url', "keywords", "description", "object"])

#for key in filenames:

for doc in range(len(output)):
#for doc in range(10):
#for key in filenames:
  #if ".jsonld" in key:
  if "/.jsonld" not in filenames[doc] :
    try:
      jld = output[doc].compute()
    except:
      print(filenames[doc])
      print("Doc has bad encoding")

    # TODO  Really need to flatten and or frame this
    try:
      desc = jld["description"]
    except:
      desc = "NA"
      continue
    kws = "keywords" #jld["keywords"]
    name = jld["name"]
    url = "NA" #jld["url"]
    object = filenames[doc]

    gldf = gldf.append({'name':name, 'url':url, 'keywords':kws, 'description': desc,
'object': object}, ignore_index=True)
```

```
CPU times: user 12.3 s, sys: 830 ms, total: 13.1 s
Wall time: 59 s
```

```
gldf.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 654 entries, 0 to 653
Data columns (total 5 columns):
 #   Column       Non-Null Count  Dtype 
---  ------       --------------  ----- 
 0   name         654 non-null    object
 1   url          654 non-null    object
 2   keywords     654 non-null    object
 3   description  654 non-null    object
 4   object       654 non-null    object
dtypes: object(5)
memory usage: 25.7+ KB
```

```
gldf.to_parquet('index.parquet.gzip',  compression='gzip')
```

Erratta

```
import re

text_corpus = []

# for i in range(len(gldf)):
#   text_corpus += gldf.at[i,'description']

# for i in range(len(gldf)):
for i in range(10):
  d = gldf.at[i,'description']
  # d.replace('(', '').replace(')', '').replace('\"', '')
  dp = re.sub(r'[^A-Za-z0-9 ]+', '', str(d))
  text_corpus.append(str(dp))

  # if not "http" in d:
  #   if not "(" in d:
  #     if not "<" in d:
  #       text_corpus.append(str(d))

# for x in range(len(text_corpus)):
#   print(text_corpus[x])
```

```
# Not needed for textai

# Create a set of frequent words
stoplist = set('for a of the and to in'.split(' '))
# Lowercase each document, split it by white space and filter out stopwords
texts = [[word for word in document.lower().split() if word not in stoplist]
         for document in text_corpus]

# Count word frequencies
from collections import defaultdict
frequency = defaultdict(int)
for text in texts:
    for token in text:
        frequency[token] += 1

# Only keep words that appear more than once
processed_corpus = [[token for token in text if frequency[token] > 1] for text in
texts]
# pprint.pprint(processed_corpus)
```

txtai section

```
import numpy as np

# Create an index for the list of text_corpus
embeddings.index([(gldf.at[uid,'object'], text, None) for uid, text in
enumerate(text_corpus)])
embeddings.save("index")
embeddings = Embeddings()
embeddings.load("index")

results = embeddings.search("lidar data ", 3)
for r in results:
  uid = r[0]
  score = r[1]
  print('score:{} -- {}\n\n'.format(score, uid)) #text_corpus[uid]))
  #print(gldf.at[uid,'object'])
```

```
score:0.3274398148059845 --
gleaner/summoned/opentopo/04d01beb4b6be2ea15309823124e8029a8547f82.jsonld


score:0.263794869184494 --
gleaner/summoned/opentopo/008b91b98f92c4b6110bb40ec1dae10240ec28f0.jsonld


score:0.2295398861169815 --
gleaner/summoned/opentopo/04324ac3558c70ed30fbafe4ad62637fd9d2975b.jsonld
```

EarthCube Graph Analytics Exploration

About

This is the start of learning a bit about leveraging graph analytics to assess the EarthCube graph and explore both the relationships but also look for methods to better search the graph for relevant connections.

Thoughts

It seems we don't care about the triples with literal objects. Only the triples that represent connections between types. Could we use a CONSTRUCT call to remove the unwanted triples? Filter on only IRI to IRI.

References

- RDFLib
- NetworkX
- iGraph
- NetworkX link analysis
- https://faculty.math.illinois.edu/~riveraq2/teaching/simcamp16/PageRankwithPython.html
- https://docs.dask.org/en/latest/
- https://examples.dask.org/bag.html
- https://s3fs.readthedocs.io/en/latest/
- https://docs.dask.org/en/latest/remote-data-services.html

Installs

```
!pip -q install mimesis
!pip -q install minio
!pip -q install s3fs
!pip -q install SPARQLWrapper
!pip -q install boto3
!pip -q install 'fsspec>=0.3.3'
!pip -q install rdflib
!pip -q install rdflib-jsonld
!pip -q install PyLD==2.0.2
!pip -q install networkx
```

```
ERROR: After October 2020 you may experience errors when installing or updating
packages. This is because pip will change the way that it resolves dependency
conflicts.

We recommend you use --use-feature=2020-resolver to test your packages with the new
resolver before it becomes the default.

boto3 1.17.46 requires botocore<1.21.0,>=1.20.46, but you'll have botocore 1.19.52
which is incompatible.
ERROR: After October 2020 you may experience errors when installing or updating
packages. This is because pip will change the way that it resolves dependency
conflicts.

We recommend you use --use-feature=2020-resolver to test your packages with the new
resolver before it becomes the default.

aiobotocore 1.2.2 requires botocore<1.19.53,>=1.19.52, but you'll have botocore
1.20.59 which is incompatible.
```

```
import sys
sys.path.append("../lib/")  # path contains python_file.py

import sparqlPandas
```

Imports

```
import dask, boto3
import dask.dataframe as dd
import pandas as pd
import json

from SPARQLWrapper import SPARQLWrapper, JSON

sweet = "http://cor.esipfed.org/sparql"
dbsparql = "http://dbpedia.org/sparql"
ufokn = "http://graph.ufokn.org/blazegraph/namespace/ufokn-dev/sparql"
```

Code inits

Helper function(s)

The following block is a SPARQL to Pandas feature. You may need to run it to load the function per standard notebook actions.

```
#@title
def get_sparql_dataframe(service, query):
    """
    Helper function to convert SPARQL results into a Pandas data frame.
    """
    sparql = SPARQLWrapper(service)
    sparql.setQuery(query)
    sparql.setReturnFormat(JSON)
    result = sparql.query()

    processed_results = json.load(result.response)
    cols = processed_results['head']['vars']

    out = []
    for row in processed_results['results']['bindings']:
        item = []
        for c in cols:
            item.append(row.get(c, {}).get('value'))
        out.append(item)

    return pd.DataFrame(out, columns=cols)
```

Set up some Pandas Dataframe options

```
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
pd.set_option('display.width', None)
pd.set_option('display.max_colwidth', None)
```

Set up the connection to the object store to access the graph objects from

```
import s3fs

oss = s3fs.S3FileSystem(
      anon=True,
      key="",
      secret="",
      client_kwargs = {"endpoint_url":"https://oss.geodex.org"}
   )
```

```
# Simple command to list objects in the current results bucket prefix
oss.ls('gleaner/results/cdfv3')
```

```
['gleaner/results/cdfv3/bcodmo_graph.nq',
 'gleaner/results/cdfv3/cchdo_graph.nq',
 'gleaner/results/cdfv3/earthchem_graph.nq',
 'gleaner/results/cdfv3/hydroshare_graph.nq',
 'gleaner/results/cdfv3/ieda_graph.nq',
 'gleaner/results/cdfv3/iris_graph.nq',
 'gleaner/results/cdfv3/lipdverse_graph.nq',
 'gleaner/results/cdfv3/ocd_graph.nq',
 'gleaner/results/cdfv3/opentopo_graph.nq',
 'gleaner/results/cdfv3/ssdb_graph.nq',
 'gleaner/results/cdfv3/unavco_graph.nq']
```

Pull a graph and load

Let's pull back an example graph and load it up into an RDFLib graph so we can test out a SPARQL call on it.

```
import rdflib
import gzip

with oss.open('gleaner/results/cdfv3/opentopo_graph.nq', 'rb') as f:
  #print(f.read())
  file_content = f.read()  #.decode("utf-8", "ignore").replace('\n',' ')

# print(file_content)
# with gzip.open('./oceanexperts_graph.nq.gz', 'rb') as f:
#     file_content = f.read()
```

```
g = rdflib.Graph()
parsed = g.parse(data = file_content, format="nquads")
```

Note

When we start to do the network analysis we don't really care about the links to literal strings. Rather, we want to see connections between various types. More specidically, types connecting to types.

Note, the isIRI filter removes the blank nodes since the rdf:type can point to both IRI and blank nodes.

```
      BIND("Nca34627a4b6d4272be7e2d22bab3becd" as ?s)
```

```
prefix schema: <http://schema.org/>
SELECT DISTINCT ?s ?o
   WHERE {
       ?s  a schema:Dataset.
       ?s ?p ?o .
       ?o a ?type
       FILTER isIRI(?o)
   }
 LIMIT 1000
```

```
qres = g.query(
    """prefix schema: <https://schema.org/>
    SELECT DISTINCT ?s ?o
       WHERE {
           ?s  a schema:Dataset.
           ?s ?p ?o .
           ?o a ?type
           FILTER isIRI(?o)
       }
       LIMIT 1000
       """)

qrdf = pd.DataFrame(columns=['s', 'o'])

for row in qres:
    qrdf = qrdf.append({'s': row[0], 'o': row[1]}, ignore_index=True)
#     print("%s : %s " % row)
```

```
qrdf.head()
```

	s	o
0	https://portal.opentopography.org/lidarDataset?opentopoID=OTLAS.072018.6340.1	https://portal.opentopography.org/lidarDataset?opentopoID=OTLAS.072018.6340.1
1	https://portal.opentopography.org/lidarDataset?opentopoID=OTLAS.052018.2444.2	https://portal.opentopography.org/lidarDataset?opentopoID=OTLAS.052018.2444.2
2	https://portal.opentopography.org/lidarDataset?opentopoID=OTLAS.122016.2193.6	https://portal.opentopography.org/lidarDataset?opentopoID=OTLAS.122016.2193.6
3	https://portal.opentopography.org/raster?opentopoID=OTSDEM.082019.26912.1	https://portal.opentopography.org/raster?opentopoID=OTSDEM.082019.26912.1
4	https://portal.opentopography.org/raster?opentopoID=OTSDEM.012012.26915.1	https://portal.opentopography.org/raster?opentopoID=OTSDEM.012012.26915.1

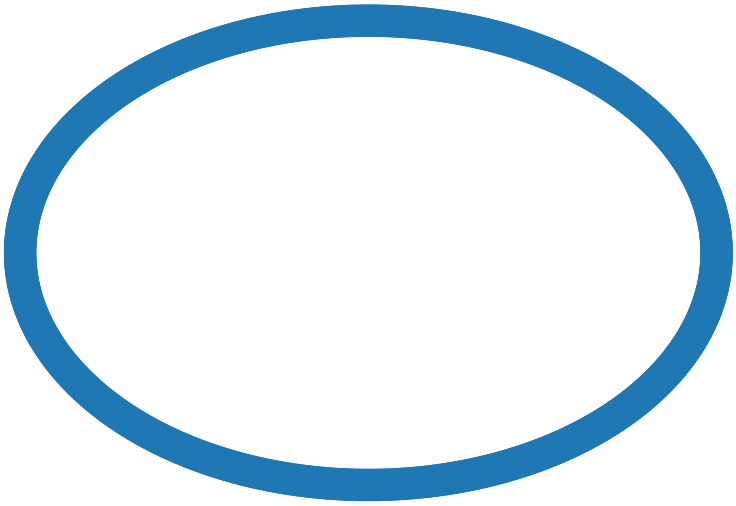
Convert to NetworkX

Convert this to a networkx graph so we can explore some analytics calls

```
import rdflib
from rdflib.extras.external_graph_libs import rdflib_to_networkx_multidigraph
from rdflib.extras.external_graph_libs import rdflib_to_networkx_digraph
import networkx as nx
import matplotlib.pyplot as plt

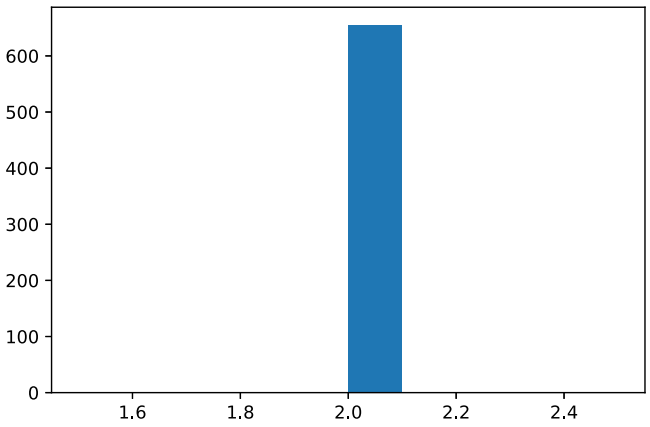
G = rdflib_to_networkx_digraph(parsed)
G=nx.from_pandas_edgelist(qrdf, source='s', target='o')
```

```
# Pointless to draw for a graph this big..   just a black ball
nx.draw_circular(G, with_labels = False)
plt.show() # display
```



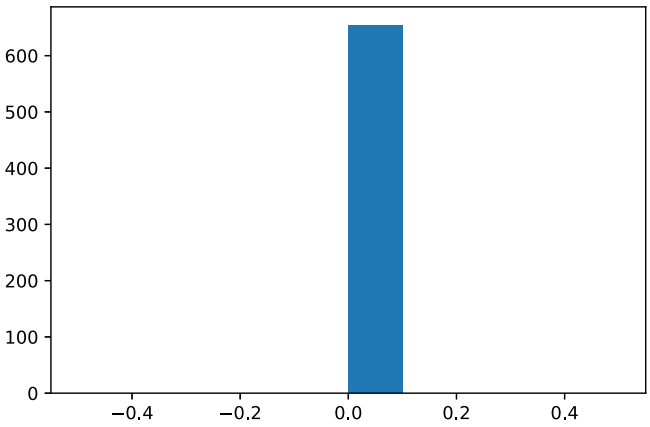
```
plt.hist([v for k,v in nx.degree(G)])
```

```
(array([  0.,   0.,   0.,   0.,   0., 654.,   0.,   0.,   0.,   0.]),
 array([1.5, 1.6, 1.7, 1.8, 1.9, 2. , 2.1, 2.2, 2.3, 2.4, 2.5]),
 <BarContainer object of 10 artists>)
```




```
plt.hist(nx.centrality.betweenness_centrality(G).values())
```

```
(array([  0.,   0.,   0.,   0.,   0., 654.,   0.,   0.,   0.,   0.]),
 array([-0.5, -0.4, -0.3, -0.2, -0.1,  0. ,  0.1,  0.2,  0.3,  0.4,  0.5]),
 <BarContainer object of 10 artists>)
```




```
#  nx.diameter(G)  # found infinite
```

```
nx.cluster.average_clustering(G)
```

```
0.0
```

Pagerank

Test a page rank call and see if we can load the results into Pandas and sort.

```
import pandas as pd

pr = nx.pagerank(G,alpha=0.9)

prdf = pd.DataFrame.from_dict(pr, orient='index')
prdf.sort_values(by=0,ascending=False, inplace=True,)
prdf.head(10)
```

	0
https://portal.opentopography.org/lidarDataset?opentopoID=OTLAS.072018.6340.1	0.001529
https://portal.opentopography.org/dataspace/dataset?opentopoID=OTDS.122019.4326.5	0.001529
https://portal.opentopography.org/dataspace/dataset?opentopoID=OTDS.022019.32637.5	0.001529
https://portal.opentopography.org/raster?opentopoID=OTSDEM.102012.26916.1	0.001529
https://portal.opentopography.org/lidarDataset?opentopoID=OTLAS.022012.26910.2	0.001529
https://portal.opentopography.org/raster?opentopoID=OTSDEM.072013.2965.1	0.001529
https://portal.opentopography.org/dataspace/dataset?opentopoID=OTDS.112018.32635.1	0.001529
https://portal.opentopography.org/raster?opentopoID=OTSDEM.062017.26910.1	0.001529
https://portal.opentopography.org/raster?opentopoID=OTSDEM.032013.26910.2	0.001529
https://portal.opentopography.org/dataspace/dataset?opentopoID=OTDS.042020.32632.1	0.001529

NetworkX hits

```
hits = nx.hits(G)  # can also be done with a nodelist (which is interesting.   provide with SPARQL call?)
```

```
hitsdf = pd.DataFrame.from_dict(hits)
hitsdf.head()
```

	https://portal.opentopography.org/lidarDataset?opentopoID=OTLAS.072018.6340.1	https://portal.opentopography.org/lidarDat opentopoID=OTLAS.052018.2
0	0.001529	0.0(
1	0.001529	0.0(

```
bc = nx.betweenness_centrality(G)
```

```
bcdf = pd.DataFrame.from_dict(bc, orient='index')
bcdf.sort_values(by=0,ascending=False, inplace=True,)
bcdf.head(10)
```

	0
https://portal.opentopography.org/lidarDataset?opentopoID=OTLAS.072018.6340.1	0.0
https://portal.opentopography.org/dataspace/dataset?opentopoID=OTDS.122019.4326.5	0.0
https://portal.opentopography.org/dataspace/dataset?opentopoID=OTDS.022019.32637.5	0.0
https://portal.opentopography.org/raster?opentopoID=OTSDEM.102012.26916.1	0.0
https://portal.opentopography.org/lidarDataset?opentopoID=OTLAS.022012.26910.2	0.0
https://portal.opentopography.org/raster?opentopoID=OTSDEM.072013.2965.1	0.0
https://portal.opentopography.org/dataspace/dataset?opentopoID=OTDS.112018.32635.1	0.0
https://portal.opentopography.org/raster?opentopoID=OTSDEM.062017.26910.1	0.0
https://portal.opentopography.org/raster?opentopoID=OTSDEM.032013.26910.2	0.0
https://portal.opentopography.org/dataspace/dataset?opentopoID=OTDS.042020.32632.1	0.0

EC2020 examples

Hack

quick hack to test ec2020 notebooks… will clean this up later

Processing digital elevation data for deep learning models using Keras Spatial

Aiman Soliman and Jeffrey Terstriep

National Center for Supercomputing Applications (NCSA)

University of Illinois at Urbana-Champaign, Urbana, Illinois

1. How to avoid ad hoc preprocessing with Keras Spatial

There are endless possibilities for developing deep learning applications to extract geospatial insights from large remote sensing archives such as NASA LP DAAC and NSIDC DAAC. However, if you have tried to develop your GeoAI application, you must have already stumbled upon the hurdle of getting remote sensing data ready for your deep learning model. Unlike regular images, remote sensing data includes important information about their location, map projection, and coordinate system. Further, rasters tend to be stored in a single file much larger than a single training sample. A common solution around this problem is to 'chop' the large remote sensing image in many equal-sized samples. This ad hoc process becomes even more tedious if you would like to repeat the experiment with your GeoAI model as each change of the model input dimensions (i.e., width and height) would require repeating the entire preprocessing steps and duplicating your data.

In this notebook, you will learn how to avoid these problems by using Keras Spatial, a new python library for preprocessing geospatial data. This library will help you feed your remote sensing data as batches with predefined dimensions, without worrying too much about preprocessing your data in advance. The key point here is that Keras Spatial will handle the projection and the spatial resolution of your remote sensing data and let you specify your input dimensions as if you are handling a regular image.

2. What is Keras Spatial?

Keras Spatial is a python package designed to be part of Keras' preprocessing library. It provides capabilities to generate samples and extracting tensor data derived on-the-fly from a raster data source. The most significant design principle for Keras Spatial is the separation of the sample boundary and array size passed to the DL model. The researcher can define a coordinate reference system and sample size in spatial coordinates that are most appropriate to the problem being solved. Following, multiple raster sources can be easily merged together, while re-projection and re-sampling will be done automatically to match the dimensions of the model's input layer.

Following we will discuss with examples the three main components of Keras Spatial: (1) a SpatialDataGenerator (SDG) class that handles access to raster data, (2) Sample definition utilities to aid in the definition of sample boundaries, and (3) Sample Iterator.

```
import geopandas as gpd
import numpy as np
import matplotlib.pyplot as plt
from rasterio.plot import show
```

2.1 Spatial Data Generator

If you are familiar with the deep learning framework Keras, then the SpatialDataGenerator (SDG) resembles the standard Keras ImageDataGenerator class. The main difference is that SDG extracts sample data from raster files on-the-fly. This approach is more convenient for remote sensing applications, where the dimensions of an input raster file are not equal to the dimensions of a single sample and it may be desirable to extract hundreds or thousands of samples from a single raster. SDG also understands coordinate reference systems and transparently handles reprojection when required.

```
from keras_spatial import SpatialDataGenerator

raster_dem = './data/raster.tiff'
sdg = SpatialDataGenerator(source=raster_dem)
print(sdg.src.meta)
```

```
{'driver': 'GTiff', 'dtype': 'float32', 'nodata': -9999.0, 'width': 2000, 'height':
2000, 'count': 1, 'crs': CRS.from_epsg(3443), 'transform': Affine(48.96, 0.0,
801795.0,
       0.0, -42.5675, 1460734.99)}
```

As can be seen in the final print statement above, the RasterIO src instance is available from the SDG and is useful when inspecting the source raster. The SDG class wraps the RasterIO file class and provides many of the same attributes and methods. In general, the SDG attributes and methods should be preferred as the RasterIO src may be opened and closed automatically during program execution.

Using default parameters, the SDG will return samples in the same coordinate reference system as the source raster. An alternative system can be chosen using the *crs* parameter or *crs* attribute. Similarly *indexes* can be used to specify one or more bands from the source raster. The SDG also supports standard interpolation methods (nearest neighbor, bilinear, cubic, etc) as provided by the GDAL library and the Rasterio package. Resampling becomes significant when transforming from raster pixel size to sample size as will be shown in more detail later.

2.2 Sample Definition Utilities

In addition to the source raster data, a GeoAI model will require a set of sample boundaries. Samples can be any vector source containing polygons that can be read into a GeoPandas GeoDataframe. If pre-defined boundaries are not available, the Keras Spatial grid module provides two functions for generating GeoDataFrames that define the samples, regular_grid, and random_grid. Both require the spatial extent of the study area, the sample size in the source coordinate units, and the coordinate reference system (CRS). The regular_grid may also include a percentage overlap that increases the number of samples available to model.

The SDG class includes convenience methods that provide a shortcut in accessing these functions where the spatial extent and CRS are determined directly from the raster source. By default, the sample width and height are specified in the native coordinate system units (in this case, feet). Alternatively, samples can be specified in pixels by using the *units='pixels'* parameter.

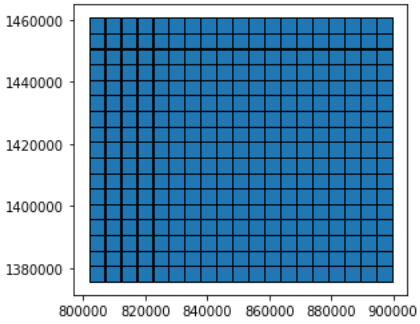
Regardless of the approach used, the sample boundaries will be defined in spatial coordinates and are **unrelated** to the final array size that will be passed to the model. The sample size should be based on the geometry or physical attributes of the feature being studied. For instance, detecting buildings with sample sizes of 3 by 3 meters may be inappropriate whereas the same sample size may be appropriate for detecting small gullies.

```
df = sdg.regular_grid(5000, 5000)
print(f'Created {len(df)} samples')

df.plot(color=None, edgecolor='k')
df.head()
```

```
Created 323 samples
```

	geometry
0	POLYGON ((806795 1375599.99, 806795 1380599.99...
1	POLYGON ((811957.2222222222 1375599.99, 811957...
2	POLYGON ((817119.4444444445 1375599.99, 817119...
3	POLYGON ((822281.6666666666 1375599.99, 822281...
4	POLYGON ((827443.8888888889 1375599.99, 827443...



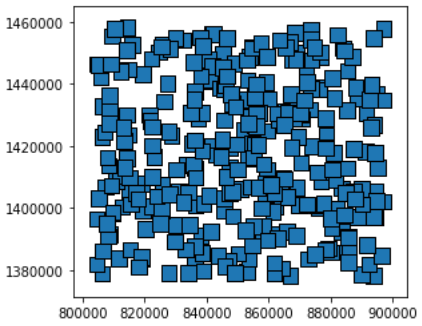
Similarly, you could create a dataframe with random samples. In this case, you will need to specify the number of samples in addition to the width and height dimensions of each sample in the native resolution of the SDG source raster. For example, here we created 323 samples as in the regular grid with equal dimensions of 5000 feet.

```
df_rand = sdg.random_grid(5000, 5000, 323)

print(f'Created {len(df_rand)} samples')
df_rand.plot(color=None, edgecolor='k')
```

```
Created 323 samples
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff10e4f2128>
```



2.3 Sample Iterator

Developers familiar with ImageDataGenerator may be familiar with the *flow_from_dataframe* method that returns an iterator that reads images from the file system based on the path contained in the dataframe. The SDG *flow_from_dataframe* method performs similarly returning an iterator that performs the following steps:

- Extract sample data from raster source, reprojecting if necessary
- Resample data to match model input size
- Invoke any requested callback functions on each sample
- Stack samples into desired batch sizes

The most important task of the *flow_from_dataframe* generator is to return arrays that match the input size expected by the model. The iterator always returns a NumPy array of shape *(batch_size, height, width, layers)*. *Batch_size, height,* and *width* are parameters passed to the *flow_from_dataframe* method and should be self-explanatory. The *layers* may be indexes (also known as bands) read from the source raster, data created using the SDG callback mechanism, or a combination of the two.

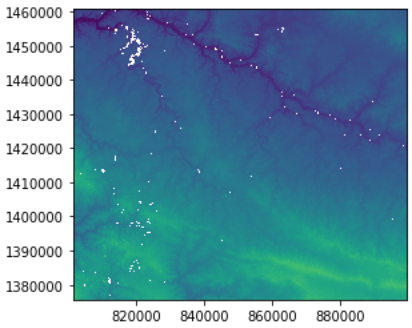
The SDG callback mechanism enables on-the-fly data transformation or augmentation of each sample. A callback can be any function that accepts a NumPy array of shape (1, width, height, n) and returns a NumPy array of shape (1, width, height, m) so the callback may produce new layers. Callbacks are invoked as a pipeline with the results of the earlier callbacks being passed to the subsequent callback. The final callback should always produce an array that matches the model input size.

3. A Practical Example: Preprocessing Digital Elevation Models (DEM)

We will start with a simple example of preparing Digital Elevation Models (DEM) for a landscape morphological classification using a deep learning model. Here we have two input rasters (1) an elevation raster that we want to segment and (2) a label raster that identifies the locations grass waterways, which are commonly found surface hydrology features in agriculture fields. The model produces a binary classification of positions on the landscape that are associated with waterways.

The DEM raster, below, shows the height of every pixel over the Lake Bloomington Watershed area in central Illinois, where the dark pixels are regions with lower elevation.

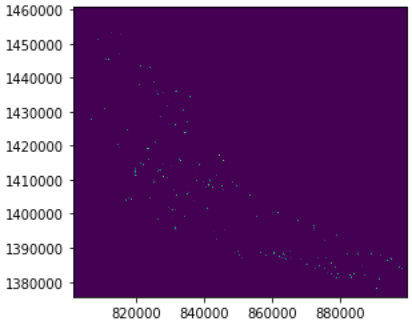
```
dem_path = './data/raster.tiff'
dem = SpatialDataGenerator(source=dem_path)
show(dem.src)
```




```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff10b2cbda0>
```

The label is a binary raster where light color pixels indicate the locations of the feature of interest, in this case, grass waterways. You should notice that the features are concentrated within the boundaries of the Lake Bloomington Watershed. Areas outside of the watershed were not manually labeled, therefore we can not use them to train the model, and these areas must be excluded from the training and evaluation samples set. It should be also noticed that the geographic projection and spatial resolution of DEM and label rasters are different, but both rasters' units are in feet.

```
label_path = './data/label.tiff'
labels = SpatialDataGenerator(source=label_path)
show(labels.src)
```




```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff10b1037f0>
```

3.1 Managing sample space

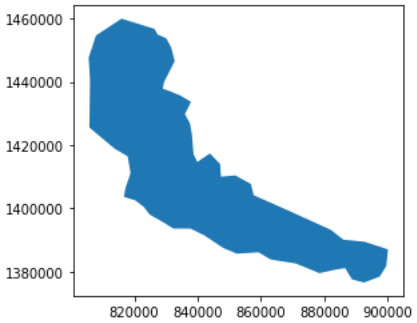
One of the advantages of using a GeoDataFrame to store the boundaries of the samples is easy filtering and selecting samples to feed to the deep learning model. You can select a subset of samples either by applying a spatial selection criterion (intersection, within, etc.) with an external vector file or by applying a database query on the sample attributes columns. We will discuss how to create sample attributes columns later, but let us start with selecting samples based on spatial relationships.

Selecting samples within an area of interest
Selecting samples based on spatial relationships is useful in the case that the study area has an irregular boundary, or as we mentioned earlier, some areas might have not been included in the manual labeling. Keras Spatial inherited the spatial queries from geopandas, therefore Keras Spatial supports any complicated spatial selection criteria on the samples objects.

We will use the Lake Bloomington watershed as a mask and select only the samples that intersect the watershed boundaries. Notice that you could use a more complicated spatial relationship as a selection criterion, such as selecting samples that fall within, touch, or do not intersect with the boundaries of a vector object.

```
watershed_path = './data/watershed.geojson'
mask = gpd.read_file(watershed_path)
mask.plot()
```

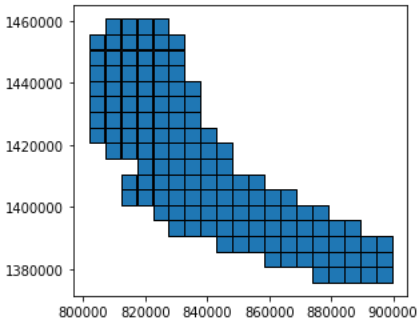
```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff10b0e40f0>
```



Here we select all the samples that intersects our watershed.

```
samples = df[df.intersects(mask.unary_union)].copy()
samples.plot(color=None, edgecolor='k')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff10b089c50>
```



Calculating attributes for each sample

Upon creating a GeoDataFrame for samples, it will contain a single column with the bounding box (geometry) of each sample. A powerful concept in Keras Spatial is setting selection criteria based on sample attributes. This capability comes handy when quantifying class imbalance in training samples, or the geographic bias of samples towards an area with specific attributes, such as specific soil type, average elevation, etc. Once these attributes are added to your GeoDataFrame, selecting samples will be a matter of applying conditional query based on column values as in slicing any DataFrame. The question is how to add the samples attributes to the samples GeoDataFrame?

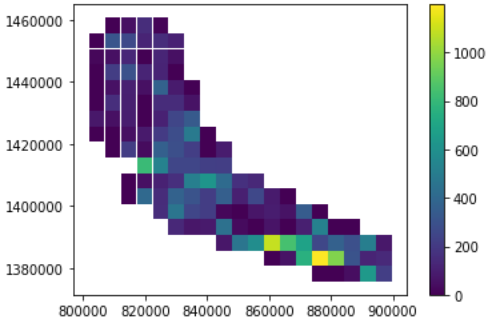
An easy way to solve this problem is to use the SpatialDataGenerator's iterator to extract samples from the source raster, calculate the attribute for each sample, and store it as a column in the GeoDataFrame. In the following example, the count of labels pixels will be extracted from the label raster and added as a second column named features in the samples' GeoDataFrame. We defined a batch size of one to guarantee a one to one correspondence with every single sample.

```
# define label raster as a source raster
label_path = './data/label.tiff'
lsdg = SpatialDataGenerator(source=label_path)

width, height = 128, 128
samples['features'] = [(np.sum(np.where(arr==1, 0, arr) ))
                       for arr in lsdg.flow_from_dataframe (samples, width, height,
batch_size=1)]

samples.plot(column='features', legend=True)
samples.head(5)
```

	geometry	features
14	POLYGON ((879066.1111111111 1375599.99, 879066...	0.000000
15	POLYGON ((884228.3333333334 1375599.99, 884228...	0.000000
16	POLYGON ((889390.5555555555 1375599.99, 889390...	46.056988
17	POLYGON ((894552.7777777778 1375599.99, 894552...	642.023926
18	POLYGON ((899715 1375599.99, 899715 1380599.99...	218.907867

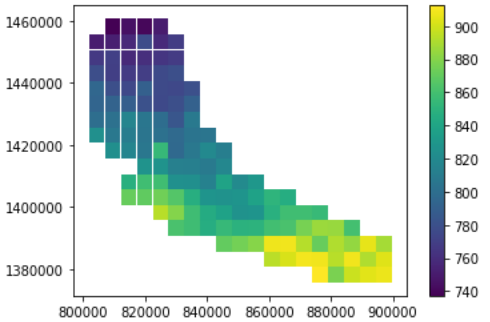


The same process could be repeated to add more attributes from other raster files. In the following cell, we calculate the average elevation of each sample. Remarkably, we can add attributes from vector layers as well. For example, a land cover layer could be used to estimate the dominant land cover type of each sample.

```
dem_path = './data/raster.tiff'
rsdg = SpatialDataGenerator(dem_path)

samples['elevation'] = [(np.max(arr)) for arr in rsdg.flow_from_dataframe (samples,
128, 128, batch_size=1)]
samples.plot(column='elevation', legend=True)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff10b2bc978>
```



3.2 Estimating global statistics

One of the major challenges that you could face when using data generators is estimating global statistics for the entire samples set. For example, in the case that you want to normalize each sample using the global maximum and minimum of the entire data set. This process is difficult because of the piecewise strategy of the data generator to load a batch of samples at a time. Although this strategy uses memory efficiently, it makes the retrieval of global statistics difficult.

We adopted a two-step solution to solve this problem with Keras Spatial. In the first step, we estimate the local statistical attributes for each sample. In the second step, the global attribute is passed as a parameter to the SDG callback function.

In the following example, we estimate the global maximum and minimum elevation by first adding two columns with the maximum and minimum elevation for each sample (local statistics) as we demonstrated before. Once the minimum and maximum columns are added to the dataframe, we can estimate the global maximum and minimum directly from the dataframe.

```
samples['maxelv'] = [(np.max(arr)) for arr in rsdg.flow_from_dataframe (samples, 128,
128, batch_size=1)]
samples['minelv'] = [(np.min(np.where(arr>0, arr, np.nan))) for arr in
rsdg.flow_from_dataframe (samples, 128, 128, batch_size=1)]

print(samples.maxelv.max(), samples.minelv.min())
```

```
912.4614868164062 724.9473266601562
```

3.3 Sample normalization using a callback function

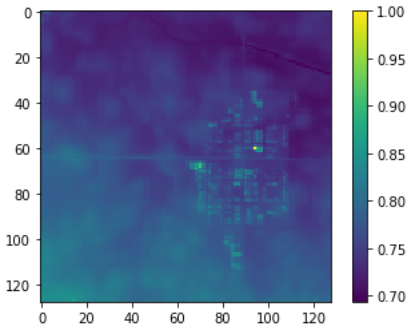
Here we define a normalization function and use it as a callback. The first sample is plotted for demonstration purposes.

```
def normalize(arr, gmin, gmax):
    return  (arr - gmin) / (gmax - gmin)

sdg.add_preprocess_callback('elvnorm', normalize, samples.minelv.min(),
samples.maxelv.max())
gen = sdg.flow_from_dataframe(samples, 128, 128, batch_size=1)
arr = next(gen)

imgplot = plt.imshow(arr[0, :, :, 0], cmap='viridis')
plt.colorbar()
```

```
<matplotlib.colorbar.Colorbar at 0x7ff1095a7278>
```



3.4 Feeding data directly to train a deep learning model

After the sample data frame is filtered and all the relevant samples are selected, then it can be used to feed data in a stepwise fashion to the deep learning model. Our final challenge is to provide a tuple containing our label and source data. We use Python's zip function to create a new iterator that is passed to the TensorFlow model.

```
# example of Feeding samples to a DL model
width, height = 128, 128
train_gen = zip(labels.flow_from_dataframe(df, width, height),
dem.flow_from_dataframe(df, width, height))
```

4. Conclusion

We introduced Keras Spatial that provides a preprocessing module tailored for remote sensing and geospatial data. Keras Spatial reduces the friction in handling geospatial data by providing loosely coupled preprocessing capabilities to retrieve and transform data on-the-fly before feeding them to deep learning models. Through this notebook, you learned the advantages of using Keras Spatial over more traditional Ad-hoc pipelines, particularly in (1) preparing your samples set in a reproducible way and (2) controlling the sample space and hence avoiding issues such as bias and class imbalance during training. Keras Spatial could also contribute to solving the Geo-AI model bias problem, by providing means to quantify the samples' statistical distribution and estimate the degree of concordance between the statistical distribution of training and prediction datasets to ensure the quality of prediction. Please refer to (Soliman and Terstriep, 2019) for more information about Keras Spatial.

Acknowledgment

This research is based in part upon work supported by the Illinois Natural Resources Conservation Service, Illinois State Geological Survey, and the Illinois State Water Survey.

BALTO Graphical User Interface (A Prototype)

Scott Dale Peckham[1], Maria Stoica[1], D. Sarah Stamps[2], James Gallagher[3], Nathan Potter[3], David Fulker[3]
[1]University of Colorado, Boulder, United States of America; [2]Virginia Tech; [1]OPeNDAP, Inc.
Corresponding author: Scott.Peckham@colorado.edu

Table of Contents

Introduction

This Jupyter notebook creates a GUI (graphical user interface) for the BALTO (Brokered Alignment of Long-Tail Observations) project. BALTO is funded by the NSF EarthCube program. The GUI aims to provide a simplified and customizable method for users to access data sets of interest on servers that support the OpenDAP data access protocol. This interactive GUI runs within the Jupyter notebook and uses the Python packages: **ipywidgets** (for widget controls), **ipyleaflet** (for interactive maps) and **pydap** (an OpenDAP client).

The Python source code to create the GUI and to process events is in a Python module called **balto_gui.py** that must be copied into the same directory as this Jupyter notebook. A Python module to create plots and color images from data, called **balto_plot.py**, is included as well.

This GUI can be toggled between an **accordion style** and a **tab style**. Both styles allow you to switch between GUI panels without scrolling in the notebook.

You can run this notebook in a browser window without installing anything on your computer, using something called **Binder**. On GitHub, on the README page, you will see a Binder icon and a link labeled “Launch Binder”.

To run this Jupyter notebook without Binder, it is recommended to install Python 3.7 from an Anaconda distribution and to then create a **conda environment** called **balto**. Instructions for how to create a conda environment are given in Appendix 1.

This GUI makes it much easier to browse, download and analyze data from servers that support the OpenDAP protocol. Under the hood (in balto_gui.py), it performs many calculations automatically, by making use of any metadata that is available for a chosen dataset. Keep in mind, however, that this is a work in progress that will continue to be improved.

The default **OpenDAP URL Dir** in the GUI is a test server with a wide variety of datasets for testing common, unusual and edge cases.

Import and Start the BALTO GUI

```
import balto_gui as bg
balto = bg.balto_gui()
# balto.show_gui(ACC_STYLE=True)    # Use accordion style
balto.show_gui()   # Use tab style
```

Download and Explore Some Data

First open the **Browse Data** panel. If you changed the “OpenDAP URL Dir”, click the Reset button at the bottom of the Data panel to restore the defaults. The OpenDAP URL Dir should again be: **http://test.opendap.org/dap/data/nc/**.

Click on the Go button.
From the **Filename droplist**, choose **sst.mnmean.nc.gz**.
From the **Variable** droplist, choose **sst**.

Notice that the shape of this dataset is **(1857, 89, 180)**, with dimensions: **(‘time’, ‘lat’, ‘lon’)**.

Next, open the **Spatial Extent** panel and use the interactive map to choose a geographic bounding box for some region you are interested in. Try choosing different basemaps from the Basemap droplist above the map. You can also click on the Full Screen icon in the upper right corner to expand the map to the full screen of your computer.

You can also type new bounding lats and lons into the text boxes and click the **Update** button. The map will then zoom to show a bounding box that contains your bounding box. Note, however, that the aspect ratio of the GUI’s map window may not match the shape of your bounding box. (Sometimes you have to click the **Reset** button and retry to get it to work

properly.)

Now let's zoom into the Caribbean island of Puerto Rico. In the text boxes, enter **-68.2** for the West edge, **-64.7** for the East edge, **18.9** for the North edge and **17.6** for the South edge, then click the **Update** button. This should zoom in to a view in which the whole island is visible in the window, with water visible all around. You can use the "minus" button (upper left) to zoom out by one click to download a larger array.

Let's now check how the monthly mean sea surface temperature has changed over the years spanned by the SST dataset we downloaded. To see the date and time range spanned by this dataset, open the **Date Range** panel. If you want, you can change the dates and times to restrict the data to a shorter time period. For example, you could change the Start Date to 1908-01-01 to restrict to a 100-year time period.

Next, open the **Download Data** panel and click on the **Download** button. This causes the SST dataset — restricted to this spatial extent and date range — to be downloaded to your computer and stored in the **balto** object as **user_var**. This may take a minute or so, before you can execute the next cell.

You can now begin to work with the data in this notebook. First, we print out some of the basic features of the variable "sst".

```
sst = balto.user_var
print('type(sst) =', type(sst))
print('sst.dtype =', sst.dtype)
print('sst.shape =', sst.shape)
if (hasattr(sst, 'min')):
    print('sst.min() =', sst.min())
    print('sst.max() =', sst.max())
```

```
type(sst) = <class 'numpy.ndarray'>
sst.dtype = float64
sst.shape = (608, 4, 8)
sst.min() = 22.12999951314
sst.max() = 30.75999932328
```

```
# How many nodata values?
w1  = (sst == 32767)
n1 = w1.sum()
print(n1)
w2 = (sst[0] == 32767)
n2 = w2.sum()
print(n2)
```

```
0
0
```

Recall from the Browse Data panel that the SST dataset has dimensions **('time', 'lat', 'lon')**. When you chose a new bounding box and time interval in the **Spatial Extent** and **Date Range** panels of the GUI, you restricted the time, lat and lon indices, resulting in a much smaller array, as seen by the new shape. This also resulted in a much faster download.

As can be seen in the **Browse Data** panel, the original SST data values were stored as 2-byte, signed integers. However, looking at the Attributes in that panel, we see that there is a **scale_factor** attribute that we are supposed to multiply these integer values by in order to convert them to actual temperatures, with units of **degrees C**. Some data sets also have an **add_offset** attribute. The GUI looks for these, and if found, applies them for you. This converts the data type from 2-byte, signed integer to a floating-point number. If the dataset also has a **missing_value** attribute, these values are restored after applying the scale factor and offset.

The GUI also retrieves and stores the restricted range of times, lats and lons associated with this variable (sea surface temperature). You can get these as follows.

```
times = balto.user_var_times
lats  = balto.user_var_lats
lons  = balto.user_var_lons
print('min(times), max(times) =', times.min(), times.max())
print('min(lats),  max(lats)  =', lats.min(), lats.max())
print('min(lons),  max(lons)  =', lons.min(), lons.max())
# print('lons([0:3] = ', lons[0:3])
# print('lats([0:3] = ', lats[0:3])
```

```
min(times), max(times) = 57708.0 76183.0
min(lats),  max(lats)  = -20.0 -14.0
min(lons),  max(lons)  = -76.0 -62.0
```

Note that the latitudes have the wrong sign, but that is how they were stored in the original dataset. If you choose **lat** from the **Variable** droplist, and then look at its attributes in the **Attributes** droplist, you see that the **actual_range** is **[88.0, -88.0]**, where min and max are reversed from what you would expect.

You can do a variant of this exercise for comparison by choosing **coads_climatology2.nc** from the **Filename** droplist, choosing **SST** from the **Variable** droplist and repeating all of the other steps. You can do another variant by choosing the filename from the **Filename** droplist that begins with **20070917-MODIS** and choosing **sea_surface_temperature** from the **Variable** droplist.

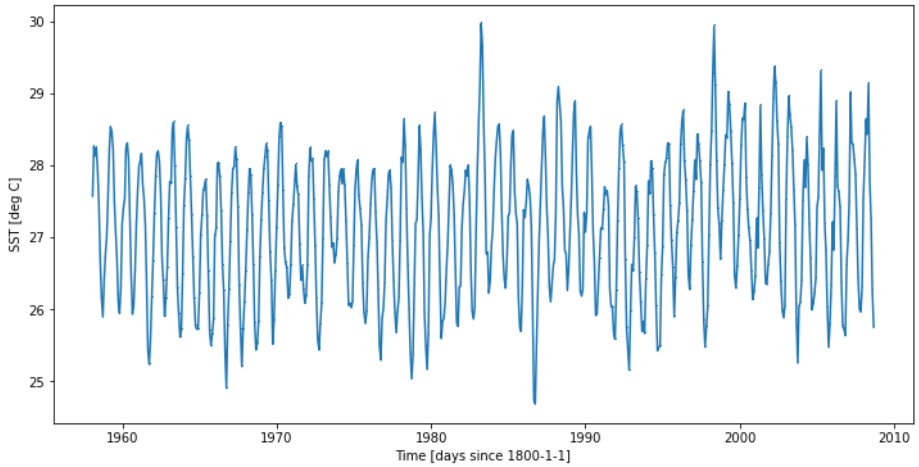
Plot Some Data

We can now examine how the sea surface temperatures around Puerto Rico have changed over the years. Remember that the first dimension of the SST variable is time, and temperature values are monthly averages over a given grid cell.

First, we will plot the monthly mean SST for a single grid cell, over the full range of times. Here we are using the **plot_data** function defined in **balto_plot.py**.

Plot Monthly Mean Sea Surface Temperature vs. Time

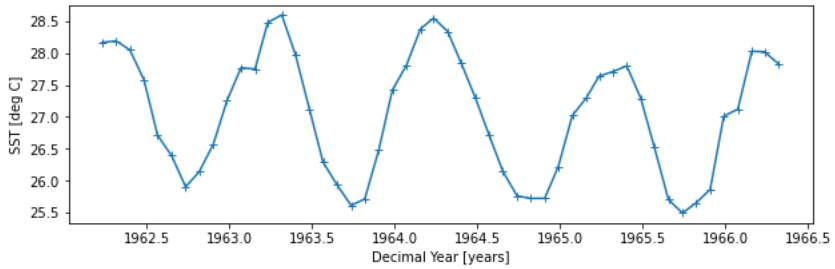
```
import balto_plot as bp
t = balto.get_years_from_time_since( times )
y = sst[:,0,0]
bp.plot_data( t, y, xmin=None, xmax=None, ymin=None, ymax=None,
              x_name='Time', x_units='days since 1800-1-1', marker=',',
              y_name='SST', y_units='deg C',
              x_size=12,   y_size=6)
print('First year =', t[0])
print('Last year  =', t[t.size - 1])
```




```
First year = 1958.0684931506848
Last year  = 2008.6849315068494
```

Based on this plot, it looks like sea surface temperatures in this area have been trending upward over this time period. The oscillations are likely associated with the 12-month solar cycle. To check this, we plot a more restricted range of values and change the marker to a plus sign. Remember, these are monthly mean values for sea surface temperature. Sure enough, there are 12 data points between one trough and the next.

```
t2 = t[50:100]
y2 = y[50:100]
bp.plot_data( t2, y2, xmin=None, xmax=None, ymin=None, ymax=None,
              x_name='Decimal Year', x_units='years', marker='+',
              y_name='SST', y_units='deg C', x_size=10, y_size=3)
print('First year =', t2[0])
print('Last year  =', t2[t2.size - 1])
```




```
First year = 1962.2328767123288
Last year  = 1966.3205479452054
```

Note: These warm temperatures are consistent with current sea surface temperatures around Puerto Rico, as given on **this website**.

Show Color Image of Monthly Mean Sea Surface Temperature for a Given Time

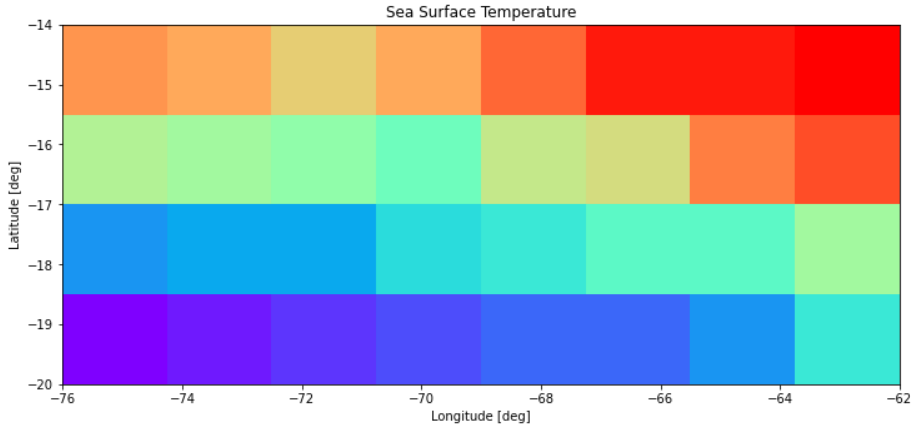
Next, we will create a color image that shows the monthly mean SST grid for a single time. In this case we are using the **show_grid_as_image()** function defined in **balto_plot.py**. Try changing the **time_index** or stretch parameters in the next cell.

```
# Avail. color maps:  rainbow, hsv, jet, gist_rainbow, gist_ncar, gist_stern

# Available stretches:
# hist_equal, power_stretch1, power_stretch2, power_stretch3, log_stretch.
# The last 4 have parameters.  See balto_plot.py.

extent = [lons.min(), lons.max(), lats.min(), lats.max()]
time_index = 20
grid = sst[time_index,:,:]

bp.show_grid_as_image( grid, 'Sea Surface Temperature', extent=extent,
                       cmap='rainbow', stretch_name='hist_equal',
                       nodata_value=32767.0, xsize=12.0, ysize=8.0,
                       stretch_a = 1.0, stretch_b = 1.0, stretch_p = 1.0 )
```



Matching Variable Names to Standardized Names

The **Scientific Variables Ontology** (SVO), developed by Peckham and Stoica provides a machine-readable mechanism for the standardized, unambiguous representation of scientific variable concepts. Developed over many years, and spanning variables across the sciences, it can be used as the “hub” in a hub-and-spoke system to map the variable names used internally by different computational models and datasets to a common representation. For more information, see Peckham (2014) and Stoica and Peckham (2018, 2019a, 2019b).

As you have seen from browsing datasets in the GUI, the names of variables used in scientific datasets are not standardized, and may simply be abbreviations. This can make it difficult and time-consuming to find variables of interest in a collection of datasets.

We have developed an experimental **variable name matching** service for SVO, that is available at: http://34.73.227.230:8000/match_phrase/. To use it, you simply append a variable name phrase, with separate words separated by underscores, to this URL in a browser. This service is also embedded in the BALTO GUI object as a method

called **get_possible_svo_names()**.

```
balto.get_possible_svo_names('sea surface temperature', SHOW_IRI=False)
```

```
Working...
Finished.

label = sea_surface_water__temperature
rank  = 0.9

label = sea_surface_air__temperature
rank  = 0.9

label = sea_surface_water__anomaly_of_temperature
rank  = 0.875

label = sea_surface_air__reference_temperature
rank  = 0.875

label = sea_ice_surface_air__temperature
rank  = 0.857

label = sea_surface_air-vs-water__difference_of_temperature
rank  = 0.812

label = sea_surface__slope
rank  = 0.667

label = sea_surface__longitude
rank  = 0.667

label = sea_surface__latitude
rank  = 0.667

label = sea_surface__elevation
rank  = 0.667

label = sea_water__temperature
rank  = 0.625

label = sea_bottom_surface__slope
rank  = 0.625

label = sea_bottom_surface__longitude
rank  = 0.625

label = sea_bottom_surface__latitude
rank  = 0.625

label = sea_bottom_surface__elevation
rank  = 0.625
```

```
balto.get_possible_svo_names('monthly mean', SHOW_IRI=False)
```

```
Working...
Finished.

label = snowpack__mean_of_temperature
rank  = 0.183

label = delta_front__mean_of_slope
rank  = 0.175

label = basin~drainage__mean_of_elevation
rank  = 0.175

label = snowpack__z_mean_of_mass-per-volume_density
rank  = 0.17

label = glacier_top_surface__mean_of_elevation
rank  = 0.17

label = earth__mean_of_orbital_speed
rank  = 0.17

label = delta_front_toe__mean_of_elevation
rank  = 0.17

label = delta_beds~foreset__mean_of_slope
rank  = 0.17

label = snowpack_grains__mean_of_diameter
rank  = 0.167

label = land_surface__one-month_daily_mean_of_temperature
rank  = 0.167

label = delta_plain~upper__mean_of_slope
rank  = 0.167

label = delta_plain~subaqueous__mean_of_slope
rank  = 0.167

label = delta_plain~lower__mean_of_slope
rank  = 0.167

label = sea_bottom_sediment_grain__mean_of_diameter
rank  = 0.164

label = land_surface__third_dekad_daily_mean_of_temperature
rank  = 0.164
```

Low-Level Access to the GUI Settings

If you have some familiarity with the Python programming language, you can browse the 3000 or so lines of source code in **balto_gui.py** and see how to access dataset information, GUI settings and class methods directly for command-line use. Here a few simple examples.

```
print(balto.var_short_names)
time = balto.dataset.time
print(time.actual_range)
print(time.units)
print(balto.map_minlon.value)
extent = balto.get_map_bounds(FROM_MAP=True, style='pyplot_imshow')
print('extent =', extent)
```

```
['lat', 'lon', 'time', 'time_bnds', 'sst']
[19723.0, 76214.0]
days since 1800-1-1 00:00:00
-73.4765625
extent = [-73.47656250000001, -59.41406250000001, 15.834535741221565,
20.632784250388028]
```

References for More Info

Anaconda Python Distribution
https://www.anaconda.com/products/individual

appmode, a Jupyter plugin (not used here)
https://github.com/oschuett/appmode
https://github.com/binder-examples/appmode

BALTO, an EarthCube Project
https://www.earthcube.org/group/brokered-alignment-long-tail-observations-balto
https://www.nsf.gov/awardsearch/showAward?AWD_ID=1740704
http://balto.opendap.org/opendap/

Binder Project
https://mybinder.org/
https://github.com/binder-examples

conda package manager for Anaconda
https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html

ipywidgets Python package
https://ipywidgets.readthedocs.io/en/latest/index.html
https://ipywidgets.readthedocs.io/en/latest/user_guide.html
https://ipywidgets.readthedocs.io/en/latest/user_install.html#installing-the-jupyterlab-extension

ipyleaflet Python package
https://ipyleaflet.readthedocs.io/en/latest/

Jupyter Lab Project
https://jupyterlab.readthedocs.io/en/stable/index.html
https://jupyterlab.readthedocs.io/en/stable/user/extensions.html

Jupyter Project
(Jupyter notebook and Jupyter lab)
https://jupyter.org/
https://docs.continuum.io/anaconda/user-guide/tasks/use-jupyter-notebook-extensions/

NetCDF CF Conventions
http://cfconventions.org/cf-conventions/cf-conventions.html

OpenDAP Data Access Protocol
https://www.opendap.org/
https://opendap.github.io/documentation/QuickStart.html
https://www.opendap.org/support/user-documentation

Peckham, S.D. (2014) The CSDMS Standard Names: Cross-domain naming conventions for describing process models, data sets and their associated variables, Proceedings of the 7th Intl. Congress on Env. Modelling and Software, International Environmental Modelling and Software Society (iEMSs), San Diego, CA. (Eds. D.P. Ames, N.W.T. Quinn, A.E. Rizzoli), Paper 12. http://scholarsarchive.byu.edu/iemssconference/2014/Stream-A/12/.image.png

pydap Python package
https://www.pydap.org/en/latest/
https://github.com/pydap/pydap

Scientific Variables Ontology (SVO)
http://www.geoscienceontology.org/

Stoica, M. and S.D. Peckham (2019a) The Scientific Variables Ontology: A blueprint for custom manual and automated creation and alignment of machine-interpretable qualitative and quantitative variable concepts, MWS 2019: Modeling the World's Systems, http://pittmodelingconference.sci.pitt.edu.image.png

Stoica, M. and S.D. Peckham (2019b) Incorporating new concepts into the Scientific Variables Ontology, Workshop on Advanced Knowledge Technologies for Science in a FAIR World, San Diego, CA, (Sept. 24, 2019) http://mint-project.info/assets/publications/stoica-peckham_escience19_abstract.pdf, https://www.isi.edu/ikcap/akts/akts2019/.
image.png

Stoica, M. and S.D. Peckham (2018) An ontology blueprint for constructing qualitative and quantitative scientific variables, ISWC 2018: The 17th International Semantic Web Conference, Monterey, CA (Oct. 8-12), http://ceur-ws.org/Vol-2180/paper-64.pdfimage.png

traitlets Python package (used by ipywidgets)
https://traitlets.readthedocs.io/en/stable/

Appendix 1: Set up a conda environment called “balto”

To run this Jupyter notebook, it is recommended to use Python 3.7 from an Anaconda distribution and to install the required Python packages in a conda environment called **balto**. This prevents conflicts with other Python packages you may have installed. The Anaconda distribution includes many packages from the **Python Standard Library**. The BALTO GUI requires the additional packages: **nb_conda**, **ipywidgets**, **ipyleaflet**, **pydap**, and **matplotlib**. Simply type the following commands at an OS prompt after installing Anaconda.

```
% conda update -n base -c defaults conda
% conda create --name balto
% conda activate balto
% conda list
% conda install -c conda-forge nb_conda
% conda install -c conda-forge ipywidgets
% conda install -c conda-forge ipyleaflet
% conda install -c conda-forge pydap
% conda install -c conda-forge matplotlib
```

Conda Environments

Note that **conda** is the name of the package manager for the popular Anaconda Python distribution. One feature of conda is support for multiple environments, which are isolated from one another. When you install Anaconda, an environment called **base** is created for you and a base set of commonly-used Python packages are installed there. However, you can (and should!) create additional, named environments and install different sets of Python packages into them without worrying about potential conflicts with packages in other environments. You can switch to one of your other environments using the command **conda activate envname**. (Replace “envname” with the name of an environment.) You can switch back to the base environment with the command **conda deactivate**. It is better not to install new packages into the base environment. See the online conda documentation on **Managing Environments** for more information.

It is always a good idea to update conda itself before creating new environments and installing packages in them. The “-n” flag is followed by the name of the environment to update, and the “-c” flag is followed by the name of the **channel** from which to get packages. A channel is a collection of Python packages that are provided and maintained by some group. The word “defaults” refers to **Anaconda’s own collection**, while **conda-forge** refers to another popular collection and the GitHub organization that maintains it. Many Python packages are available from both of these channels. (However, the ipyleaflet and pydap packages are currently not available in the Anaconda collection.) When you are installing several packages into an environment, the potential for installation problems seems to be less if you get them all from the same channel. Keep in mind that packages you install will likely depend on many other Python packages, so there is a potential for conflicts, usually related to different package versions. Using conda environments helps to mitigate against this and helps with **reproducibility**.

Once you’ve switched to an environment with **conda activate envname**, you can type **conda list** to see a list of packages. If you do this right after you create a new environment you will see that it contains no packages. If you do this right after installing each package above you will see that:

- Installing **nb_conda** triggers installation of **nb_conda_kernels**, **ipykernel** (5.3.0), **notebook** (6.0.3) and **traitlets** (4.3.3), among many others.
- Installing **ipywidgets** triggers installation of **widgetsnbextension** (3.5.1).
- Installing **ipyleaflet** triggers installation of **branca** and **traittypes**.
- Installing **pydap**, triggers installation of **numpy** (1.18.4), **requests** (2.23.0) and **urllib3** (1.25.9), and many others.

Note: Many packages depend on the **numpy** package, so it often gets installed as a dependency. Installing either pydap or matplotlib causes numpy to get installed.

Jupyter Notebook Extensions

Note that **nb_conda** is installed first above, and triggers installation of **nb_conda_kernels** along with **notebook**. This is important as it makes your Jupyter notebook app aware of your conda environments and available in the app as “kernels”. Anaconda provides a helpful page on the **Jupyter Notebook Extensions**. That page also explains how you can enable or disable these extensions individually. The command **jupyter nbextension list** shows you the extensions that are installed and whether they are enabled. If you run the **jupyter notebook** or **jupyter lab** command in an environment that has **nb_conda_kernels** installed (see below), you will have the ability to associate one of your available conda environments with any new notebook you create. Different environments give rise to different **kernels** in Jupyter, and the kernel name includes the environment name, e.g. **Python [conda env:balto]**. The kernel name is displayed in

the upper right corner. Notebooks typically open with the “environment kernel” they were created with. However, there is a **Change Kernel** option in the **Kernel** menu in the Jupyter app menu bar. (After changing the kernel, you may need to choose **Restart** from the **Kernel** menu.

Cloning a conda Environment

If your notebook is working but then you want to import additional packages (possibly with many dependencies, and potential for problems), you can keep the first environment but clone it with ***conda create –name clonename –copy –clone envname***, and then install the additional packages in the clone. This way, you can switch to the new environment’s kernel and try to run your notebook, but if you run into any problems you can easily revert back to the original environment and functionality.

Note: Setting the “–copy” flag installs all packages using copies instead of hard or soft links. This is necessary to avoid problems when using **pip** together with **conda** as described **on this page**.

Running Notebooks in the Jupyter Notebook App

When you want to run the notebook, type **conda activate balto** (at an OS command prompt) to activate this environment. Then change to the directory that contains this notebook and type **jupyter notebook**. By default, this folder is called **Jupyter** and is in your home directory. In the app, choose this notebook by name, “BALTO_GUI.ipynb”, and make sure to choose the kernel called: **Python [conda env:balto]**. See the References section at the end for more info.

Running Notebooks in the JupyterLab App

The **JupyterLab** app is a cool, new successor to the Notebook app and offers many additional features. If you want to use this notebook in JupyterLab, you need to install one more Python package and 2 extensions, as follows.

```
% conda activate balto
% conda install -c conda-forge jupyterlab
% jupyter labextension install jupyter-leaflet
% jupyter labextension install @jupyter-widgets/jupyterlab-manager
```

The two extensions here provide support for **ipyleaflet** and **ipywidgets**.

You launch the JupyterLab app by typing **jupyter lab** instead of **jupyter notebook**. To quit, choose **Logout** or **Shutdown** from the app’s **File** menu.

Note: At one point it was also necessary to install **nodejs** with: “% conda install -c conda-forge nodejs”, but that is no longer the case.

CMIP6 without the interpolation: Grid-native analysis with Pangeo in the cloud

Julius Busecke[1] and *Ryan Abernathey*[2]

([1] Princeton University [2] Columbia University)

The coupled model intercomparison project (CMIP; currently in its 6th iteration) involves contribution from countries around the world, constituting a combined dataset of of 20+ Petabytes.

Many years of planning go into the inception of the different scenarios, various model intercomparison projects (MIPs) and the delivery of the output data from the modeling community around the globe. A major challenge for the earth science community now, is to effectively analyze this dataset and answer science question that improve our understanding of the earth system for the coming years.

The Pangeo project recently introduced large parts of the CMIP6 data archive into the cloud. This enables, for the first time, centralized, reproducible science of state-of-the-art climate simulations without the need to own large storage or a supercomputer as a user.

The data itself however, still presents significant challenges for analysis, one of which is applying operations across many models. Two of the major hurdles are different naming/metadata between modeling centers and complex grid layouts, particularly for the ocean components of climate models. Common workflows in the past often included interpolating/remapping desired variables and creating new files, creating organizational burden, and increasing storage requirements.

We will demonstrate two pangeo tools which enable seamless calculation of common operations like vector calculus operators (grad, curl, div) and weighted averages/integrals across a wide range of CMIP6 models directly on the data stored in the cloud. `cmip6_preprocessing` provides numerous tools to unify naming conventions and reconstruct grid information and metrics (like distances). This information is used by `xgcm` to enable finite volume analysis on the native model grids. The combination of both tools facilitates fast analysis while ensuring a reproducible and accurate workflow.

```
from dask.distributed import Client, progress
from dask_gateway import Gateway

gateway = Gateway()
cluster = gateway.new_cluster()
cluster.scale(60)
cluster
```

```
client = Client(cluster)
client
```

Client

Scheduler: gateway://traefik-ocean-prod-dask-gateway.ocean-prod:80/ocean-prod.b8379c8640314c0ea8614c0921483fab
Dashboard: https://ocean.pangeo.io/services/dask-gateway/clusters/ocean-prod.b8379c8640314c0ea8614c0921483fab/status

Cluster

Workers: 0
Cores: 0
Memory: 0 B

Loading the CMIP6 data from the cloud

In order to load the cmip data from the cloud storage, we use `intake-esm`, and choose to load all monthly ocean model output (`table_id='Omon'`) of sea surface temperature (`variable_id='tos'`) on the native grid (`grid_label='gn'`) and for the 'historical' run (`experiment_id='historical'`). For more details on how to use intake-esm we refer to this Earth Cube presentation.

```
import intake
import warnings
warnings.filterwarnings("ignore")

url = "https://raw.githubusercontent.com/NCAR/intake-esm-
datastore/master/catalogs/pangeo-cmip6.json"
col = intake.open_esm_datastore(url)
models = col.df['source_id'].unique()
# we will have to eliminate some models that do not *yet* play nice with
cmip6_preprocessing
# ignore_models = ['FGOALS-f3-L','EC-Earth3','EC-Earth3-Veg-LR', 'CESM2-FV2',
#                  'GFDL-ESM4', 'MPI-ESM-1-2-HAM', 'MPI-ESM1-2-LR', 'NorCPM1',
#                  'GISS-E2-1-H', 'IPSL-CM6A-LR', 'MRI-ESM2-0', 'CESM2-WACCM',
#                  'GISS-E2-1-G', 'FIO-ESM-2-0', 'MCM-UA-1-0', 'FGOALS-g3']
# models = [m for m in models if 'AWI' not in m and m not in ignore_models]

# These might change in the future so it is better to explicitly allow the models that
work

models = ['TaiESM1','BCC-ESM1','CNRM-ESM2-1','MIROC6','UKESM1-0-LL','NorESM2-LM',
          'BCC-CSM2-MR','CanESM5','ACCESS-ESM1-5','E3SM-1-1-ECA','E3SM-1-1',
          'MIROC-ES2L','CESM2','CNRM-CM6-1','GFDL-CM4','CAMS-CSM1-0','CAS-ESM2-0',
          'CanESM5-CanOE','IITM-ESM','CNRM-CM6-1-HR','ACCESS-CM2','E3SM-1-0',
          'EC-Earth3-LR','EC-Earth3-Veg','INM-CM4-8','INM-CM5-0','HadGEM3-GC31-LL',
          'HadGEM3-GC31-MM','MPI-ESM1-2-HR','GISS-E2-1-G-CC','GISS-E2-2-G','CESM2-
WACCM-FV2',
          'NorESM1-F','NorESM2-MM','KACE-1-0-G','GFDL-AM4','NESM3',
          'SAM0-UNICON','CIESM','CESM1-1-CAM5-CMIP5','CMCC-CM2-HR4','CMCC-CM2-VHR4',
          'EC-Earth3P-HR','EC-Earth3P','ECMWF-IFS-HR','ECMWF-IFS-LR','INM-CM5-H',
          'IPSL-CM6A-ATM-HR','HadGEM3-GC31-HM','HadGEM3-GC31-LM','MPI-ESM1-2-XR','MRI-
AGCM3-2-H',
          'MRI-AGCM3-2-S','GFDL-CM4C192','GFDL-OM4p5B']

cat = col.search(table_id='Omon', grid_label='gn', experiment_id='historical',
variable_id='tos', source_id=models)
```

```
/srv/conda/envs/notebook/lib/python3.7/site-packages/intake/source/discovery.py:138:
FutureWarning: The drivers ['geojson', 'postgis', 'shapefile', 'spatialite'] do not
specify entry_points and were only discovered via a package scan. This may break in a
future release of intake. The packages should be updated.
  FutureWarning)
```

We have to eliminate some models from this analysis: The `AWI` models, due to their unstructured native grid, as well as others for various subtleties that have yet to be resolved. This will be addressed in a future update of `cmip6_preprocessing`. If you do not see your favorite model, please consider raising an issue on github.

```
cat.df['source_id'].nunique() # This represents the number of models as of 6/14/2020.
                              # More models might be added in the future. See commented
parts in previous code cell.
```

```
27
```

This gives us 27 different models ('source_ids') to work with. Lets load them into a dictionary and inspect them closer.

```
# Fix described here: https://github.com/intake/filesystem_spec/issues/317
# Would cause `Name (gs) already in the registry and clobber is False` error somewhere
in `intake-esm`
import fsspec
fsspec.filesystem('gs')

ddict = cat.to_dataset_dict(zarr_kwargs={'consolidated':True, 'decode_times':False})
```

```
--> The keys in the returned dictionary of datasets are constructed as follows:
        'activity_id.institution_id.source_id.experiment_id.table_id.grid_label'
```

100.00% [27/27 00:09<00:00]

What we ultimately want is to apply an analysis or just a visualization across all these models. So before we jump into that, lets inspect a few of the datasets:

```
ddict['CMIP.NCAR.CESM2.historical.Omon.gn']
```

xarray.Dataset

▸ Dimensions: (d2: 2, **member_id**: 11, **nlat**: 384, **nlon**: 320, **time**: 3960, vertices: 4)

▾ Coordinates:

lon	(nlat, nlon)	float64	dask.array<chunksize=(384, 320…
time_bnds	(time, d2)	float64	dask.array<chunksize=(3960, 2),…
lon_bnds	(nlat, nlon, vertices)	float32	dask.array<chunksize=(384, 320…
lat	(nlat, nlon)	float64	dask.array<chunksize=(384, 320…
lat_bnds	(nlat, nlon, vertices)	float32	dask.array<chunksize=(384, 320…
time	(time)	float64	0.0 707.0 ... 1.444e+06 1.445e+06
nlat	(nlat)	int32	1 2 3 4 5 6 ... 380 381 382 383 384
nlon	(nlon)	int32	1 2 3 4 5 6 ... 316 317 318 319 320
member_id	(member_id)	<U9	'r10i1p1f1' ... 'r9i1p1f1'

▾ Data variables:

tos	(member_id, time, nlat, nlon)	float32	dask.array<chunksize=(1, 400, 3…

▸ Attributes: (48)

```
ddict['CMIP.CNRM-CERFACS.CNRM-CM6-1.historical.Omon.gn']
```

xarray.Dataset

▸ Dimensions: (axis_nbounds: 2, **member_id**: 30, nvertex: 4, **time**: 3959, x: 362, y: 294)

▾ Coordinates:

bounds_lat	(y, x, nvertex)	float64	dask.array<chunksize=(294, 362, 4), …
lon	(y, x)	float64	dask.array<chunksize=(294, 362), met…
time_bounds	(time, axis_nbounds)	float64	dask.array<chunksize=(3959, 2), meta…
lat	(y, x)	float64	dask.array<chunksize=(294, 362), met…
bounds_lon	(y, x, nvertex)	float64	dask.array<chunksize=(294, 362, 4), …
time	(time)	float64	0.0 15.5 ... 1.445e+06 1.446e+06
member_id	(member_id)	<U9	'r10i1p1f2' ... 'r9i1p1f2'

▾ Data variables:

tos	(member_id, time, y, x)	float32	dask.array<chunksize=(1, 126, 294, 3…

▸ Attributes: (51)

```
ddict['CMIP.CSIRO.ACCESS-ESM1-5.historical.Omon.gn']
```

xarray.Dataset

▸ Dimensions: (bnds: 2, **i**: 360, **j**: 300, **member_id**: 3, **time**: 1980, vertices: 4)

▾ Coordinates:

time_bnds	(time, bnds)	float64	dask.array<chunksize=(1980, 2), meta=…
latitude	(j, i)	float64	dask.array<chunksize=(300, 360), meta…
longitude	(j, i)	float64	dask.array<chunksize=(300, 360), meta…
time	(time)	int64	0 708 1416 ... 1444884 1445616
i	(i)	int32	0 1 2 3 4 5 ... 355 356 357 358 359
j	(j)	int32	0 1 2 3 4 5 ... 295 296 297 298 299
member_id	(member_id)	<U8	'r1i1p1f1' 'r2i1p1f1' 'r3i1p1f1'

▾ Data variables:

vertices_longit…	(j, i, vertices)	float64	dask.array<chunksize=(300, 360, 4), m…
vertices_latitude	(j, i, vertices)	float64	dask.array<chunksize=(300, 360, 4), m…
tos	(member_id, time, j, i)	float32	dask.array<chunksize=(1, 201, 300, 36…

▸ Attributes: (50)

We can immediately spot problems:

- There is no consistent convention for the labeling of dimensions (note the 'logical 1D dimension in the x-direction is called `x`, `nlon`, `i`)
- Some models (here: `CMIP.CNRM-CERFACS.CNRM-CM6-1.historical.Omon.gn`) are missing the 'vertex'/'vertices' dimension, due to the fact that the cell geometry is given by longitude and latitude 'bounds' centered on the cell face compared to the corners of the cell.

There are more issues that both make working with the data cumbersome and can quickly introduce errors. For instance, some models give depth in units of cm, not m, and many other issues. Instead of focusing on problems, here we would like to illustrate how easy analysis across models with different grids can be, using `cmip6_preprocessing`.

Use `cmip6_preprocessing` to harmonize different model output

`cmip6_preprocessing` was born out of the cmip6-hackathon and aims to provide a central package through which these conventions can be harmonized. For convenience we will make use of the `preprocess` functionality and apply the 'catch-all' function `combined_preprocessing` to the data before it gets aggregated. For a more detailed description of the corrections applied we refer to the documentation and examples therein.

```
from cmip6_preprocessing.preprocessing import combined_preprocessing

ddict_pp = cat.to_dataset_dict(
    zarr_kwargs={'consolidated':True, 'decode_times':False},
    preprocess=combined_preprocessing
)
```

```
--> The keys in the returned dictionary of datasets are constructed as follows:
        'activity_id.institution_id.source_id.experiment_id.table_id.grid_label'
```

100.00% [27/27 00:50<00:00]

> Note that all functions in the `cmip6_preprocessing.preprocessing` modules can also be used with 'raw' datasets, like e.g. a local netcdf file, as long as the metadata is intact.

Now lets look at the preprocessed version of the same datasets from above:

```
ddict_pp['CMIP.NCAR.CESM2.historical.Omon.gn']
```

xarray.Dataset

▸ Dimensions: (**bnds**: 2, **member_id**: 11, **time**: 3960, **vertex**: 4, **x**: 320, **y**: 384)

▾ Coordinates:

lon	(y, x)	float64	dask.array<chunksize=(384, 320), met…
lat_verticies	(y, x, vertex)	float32	dask.array<chunksize=(384, 320, 4), …
lon_verticies	(y, x, vertex)	float32	dask.array<chunksize=(384, 320, 4), …
time_bounds	(time, bnds)	float64	dask.array<chunksize=(3960, 2), meta…
lat	(y, x)	float64	dask.array<chunksize=(384, 320), met…
lat_bounds	(bnds, y, x)	float32	dask.array<chunksize=(1, 384, 320), …
lon_bounds	(bnds, y, x)	float32	dask.array<chunksize=(1, 384, 320), …
time	(time)	float64	0.0 707.0 ... 1.444e+06 1.445e+06
vertex	(vertex)	int64	0 1 2 3
bnds	(bnds)	int64	0 1
x	(x)	float64	1.062 2.187 3.312 ... 358.8 359.9
y	(y)	float64	-79.22 -78.69 ... 89.66 89.71
member_id	(member_id)	<U9	'r10i1p1f1' ... 'r9i1p1f1'

▾ Data variables:

tos	(member_id, time, y, x)	float32	dask.array<chunksize=(1, 400, 384, 3…

▸ Attributes: (48)

```
ddict_pp['CMIP.CNRM-CERFACS.CNRM-CM6-1.historical.Omon.gn']
```

xarray.Dataset

▸ Dimensions: (**bnds**: 2, **member_id**: 30, **time**: 3959, **vertex**: 4, **x**: 362, **y**: 294)

▾ Coordinates:

lon	(y, x)	float64	dask.array<chunksize=(294, 362), met…
lat_verticies	(y, x, vertex)	float64	dask.array<chunksize=(294, 362, 4), …
lon_verticies	(y, x, vertex)	float64	dask.array<chunksize=(294, 362, 4), …
time_bounds	(time, bnds)	float64	dask.array<chunksize=(3959, 2), meta…
lat	(y, x)	float64	dask.array<chunksize=(294, 362), met…
lat_bounds	(bnds, y, x)	float64	dask.array<chunksize=(1, 294, 362), …
lon_bounds	(bnds, y, x)	float64	dask.array<chunksize=(1, 294, 362), …
time	(time)	float64	0.0 15.5 ... 1.445e+06 1.446e+06
vertex	(vertex)	int64	0 1 2 3
bnds	(bnds)	int64	0 1
x	(x)	float64	0.5 1.5 2.5 ... 357.5 358.5 359.5
y	(y)	float64	-78.36 -78.17 -77.98 ... 89.74 90.0
member_id	(member_id)	<U9	'r10i1p1f2' ... 'r9i1p1f2'

▾ Data variables:

tos	(member_id, time, y, x)	float32	dask.array<chunksize=(1, 126, 294, 3…

▸ Attributes: (51)

```
ddict_pp['CMIP.CSIRO.ACCESS-ESM1-5.historical.Omon.gn']
```

xarray.Dataset

▸ Dimensions: (**bnds**: 2, **member_id**: 3, **time**: 1980, **vertex**: 4, **x**: 360, **y**: 300)

▾ Coordinates:

lon	(y, x)	float64	dask.array<chunksize=(300, 360), met…
lat_verticies	(y, x, vertex)	float64	dask.array<chunksize=(300, 360, 4), …
lon_verticies	(y, x, vertex)	float64	dask.array<chunksize=(300, 360, 4), …
time_bounds	(time, bnds)	float64	dask.array<chunksize=(1980, 2), meta…
lat	(y, x)	float64	dask.array<chunksize=(300, 360), met…
lat_bounds	(bnds, y, x)	float64	dask.array<chunksize=(1, 300, 360), …
lon_bounds	(bnds, y, x)	float64	dask.array<chunksize=(1, 300, 360), …
time	(time)	int64	0 708 1416 ... 1444884 1445616
vertex	(vertex)	int64	0 1 2 3
bnds	(bnds)	int64	0 1
x	(x)	float64	0.5 1.5 2.5 ... 357.5 358.5 359.5
y	(y)	float64	-77.88 -77.63 ... 89.31 89.75
member_id	(member_id)	<U8	'r1i1p1f1' 'r2i1p1f1' 'r3i1p1f1'

▾ Data variables:

tos	(member_id, time, y, x)	float32	dask.array<chunksize=(1, 201, 300, 3…

▸ Attributes: (50)

Much better!

As you can see, they all have consistent dimension names and coordinates. Time to see if this works as advertised, by plotting the sea surface temperature (SST) for all models

```
import matplotlib.pyplot as plt
import cartopy.crs as ccrs
import cartopy.feature as cfeature

fig, axarr = plt.subplots(
    ncols=5, nrows=6, figsize=[15, 15], subplot_kw={"projection": ccrs.Robinson(190)}
)
for ax, (k, ds) in zip(axarr.flat, ddict_pp.items()):
    # Select a single member for each model
    if 'member_id' in ds.dims:
        ds = ds.isel(member_id=-1)

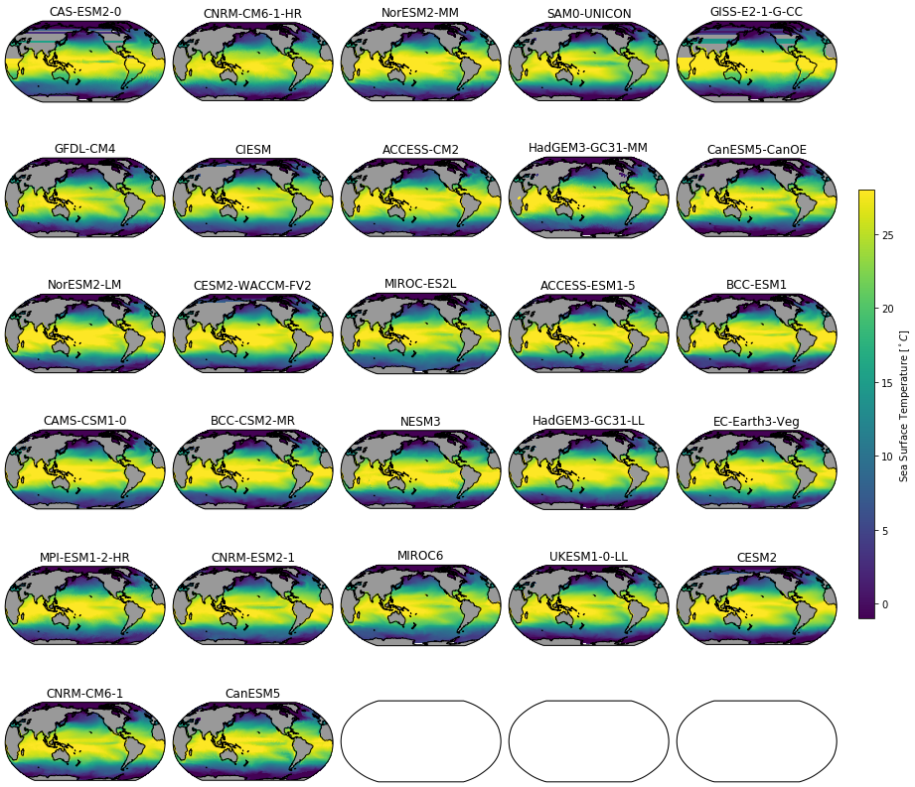
    # select the first time step
    da = ds.tos.isel(time=0)

    # some models have large values instead of nans, so we mask unresonable values
    da = da.where(da < 1e30)

    # and plot the resulting 2d array using xarray
    pp = da.plot(
        ax=ax,
        x="lon",
        y="lat",
        vmin=-1,
        vmax=28,
        transform=ccrs.PlateCarree(),
        infer_intervals=False,
        add_colorbar=False,
    )

    ax.set_title(ds.attrs['source_id'])
    ax.add_feature(cfeature.LAND, facecolor='0.6')
    ax.coastlines()
fig.subplots_adjust(hspace=0.05, wspace=0.05)
fig.subplots_adjust(right=0.9)
cbar_ax = fig.add_axes([0.92, 0.3, 0.015, 0.4])
fig.colorbar(pp, cax=cbar_ax, label="Sea Surface Temperature [$^\circ C$]")
```

```
<matplotlib.colorbar.Colorbar at 0x7f907aa4edd8>
```



That was very little code to plot such a comprehensive figure, even if the models all have different resolution and grid architectures.

`cmip6_preprocessing` helps keeping the logic needed for simple analysis and visualization to a minimum, but in the newest version is also helps to facilitate the use of `xgcm` with the native model grids, making it a powerful tool for more complex analysis across the models.

Calculations on native model grids using `xgcm`.

Most modern circulation models discretize the partial differential equations needed to simulate the earth system on a logically rectangular grid. This means the grid for a single time step can be represented as a 3-dimensional array of cells. Operations like e.g., a derivative are then approximated by a finite difference between neighboring cells, divided by the appropriate distance.

Calculating operators like *gradient*, *curl* or *divergence* is usually associated with a lot of bookkeeping to make sure that the difference is taken between the correct grid points, etc. This often leads to users preferring interpolated grids (on regular lon/lat grids), which have to be processed and stored next to the native grid data, both increasing storage requirements and potentially losing some of the detailed structure in the high-resolution model fields.

The combination of `cmip6_preprocessing` and `xgcm` enables the user to quickly calculate operators like *gradient* or *curl* (shown in detail below) on the native model grids, preserving the maximum detail the models provide and preventing unnecessary storage burden. `cmip6_preprocessing` parses the detailed grid information for the different models so that `xgcm` can carry out the computations, without requiring the user to dive into the details of each model grid.

As an example, let us compute and plot the gradient magnitude of the SST.

$$\overline{\nabla(SST)} = \sqrt{\frac{\partial SST}{\partial x}^2 + \frac{\partial SST}{\partial y}^2}$$

We will start with an example of just computing the zonal gradient $\frac{\partial SST}{\partial x}$:

First, we need to create a suitable grid, with dimensions both for the cell center (where our tracer is located) and the cell faces. These are needed for xgcm to calculate the finite distance version of the above equation.

The function `combine_staggered_grid` parses a dataset so that it is compatible with xgcm. It also provides a 'ready-to-use' xgm Grid object for convenience.

```
# we pick one of the many models
ds = ddict_pp['CMIP.CNRM-CERFACS.CNRM-CM6-1-HR.historical.Omon.gn']
ds
```

xarray.Dataset

▸ Dimensions: (**bnds**: 2, **member_id**: 1, **time**: 1980, **vertex**: 4, **x**: 1442, **y**: 1050)

▾ Coordinates:

lon	(y, x)	float32	dask.array<chunksize=(1050, 1442), …
lat_verticies	(y, x, vertex)	float32	dask.array<chunksize=(1050, 1442, 4)…
lon_verticies	(y, x, vertex)	float32	dask.array<chunksize=(1050, 1442, 4)…
time_bounds	(time, bnds)	float64	dask.array<chunksize=(1980, 2), meta…
lat	(y, x)	float32	dask.array<chunksize=(1050, 1442), …
lat_bounds	(bnds, y, x)	float32	dask.array<chunksize=(1, 1050, 1442)…
lon_bounds	(bnds, y, x)	float32	dask.array<chunksize=(1, 1050, 1442)…
time	(time)	int64	0 708 1416 ... 1444884 1445616
vertex	(vertex)	int64	0 1 2 3
bnds	(bnds)	int64	0 1
x	(x)	float32	0.25 0.5 0.75 ... 359.75 360.0
y	(y)	float32	-78.107124 -78.05977 ... 90.0
member_id	(member_id)	<U8	'r1i1p1f2'

▾ Data variables:

tos	(member_id, time, y, x)	float32	dask.array<chunksize=(1, 20, 1050, 1…

▸ Attributes: (55)

```
from cmip6_preprocessing.grids import combine_staggered_grid

# Now we parse the necessary additional dimensions
grid,ds = combine_staggered_grid(ds)
ds
```

xarray.Dataset

▸ Dimensions: (**bnds**: 2, **member_id**: 1, **time**: 1980, **vertex**: 4, **x**: 1442, **x_left**: 1442, **y**: 1050, **y_right**: 1050)

▾ Coordinates:

lon	(y, x)	float32	dask.array<chunksize=(1050, 1442), …
lat_verticies	(y, x, vertex)	float32	dask.array<chunksize=(1050, 1442, 4)…
lon_verticies	(y, x, vertex)	float32	dask.array<chunksize=(1050, 1442, 4)…
time_bounds	(time, bnds)	float64	dask.array<chunksize=(1980, 2), meta…
lat	(y, x)	float32	dask.array<chunksize=(1050, 1442), …
lat_bounds	(bnds, y, x)	float32	dask.array<chunksize=(1, 1050, 1442)…
lon_bounds	(bnds, y, x)	float32	dask.array<chunksize=(1, 1050, 1442)…
time	(time)	int64	0 708 1416 ... 1444884 1445616
vertex	(vertex)	int64	0 1 2 3
bnds	(bnds)	int64	0 1
x	(x)	float32	0.25 0.5 0.75 ... 359.75 360.0
y	(y)	float32	-78.107124 -78.05977 ... 90.0
member_id	(member_id)	<U8	'r1i1p1f2'
x_left	(x_left)	float32	0.125 0.375 ... 359.625 359.875
y_right	(y_right)	float32	-78.08345 -78.03601 ... 90.02606

▾ Data variables:

tos	(member_id, time, y, x)	float32	dask.array<chunksize=(1, 20, 1050, 1…

▸ Attributes: (55)

Note how the new dimensions `x_left` and `y_right` are added, which are associated with the ‘eastern’ and ‘northern’ cell faces. We can now easily calculate the finite difference in the logical `X` direction:

```
delta_t_x = grid.diff(ds.tos, 'X')
delta_t_x
```

xarray.DataArray 'sub-831cd583ca34002d4e8742a331286368'
(**member_id**: 1, **time**: 1980, **y**: 1050, **x_left**: 1442)

dask.array<chunksize=(1, 20, 1050, 1), meta=np.ndarray>

▾ Coordinates:

member_id	(member_id)	<U8	'r1i1p1f2'
time	(time)	int64	0 708 1416 ... 1444884 1445616
y	(y)	float32	-78.107124 -78.05977 ... 90.0
x_left	(x_left)	float32	0.125 0.375 ... 359.625 359.875

▸ Attributes: (0)

This array is now located on the center of the eastern cell face. But in order to recreate a derivative, we need to divide this difference by an appropriate distance. Usually these distances are provided with model output, but currently there is no such data publicly available for CMIP models.

To solve this problem, `cmip6_preprocessing` can recalculate grid distances.

> Be aware that this can lead to rather large biases when the grid is strongly warped, usually around the Arctic. More on that at the end of the notebook.

```
grid, ds = combine_staggered_grid(ds, recalculate_metrics=True)
```

```
ds
```

xarray.Dataset

▸ Dimensions: (**bnds**: 2, **member_id**: 1, **time**: 1980, **vertex**: 4, **x**: 1442, **x_left**: 1442, **y**: 1050, **y_right**: 1050)

▾ Coordinates:

lon	(y, x)	float32	1.4255313 1.672195 ... 73.59864
lat_verticies	(y, x, vertex)	float32	-78.02738 -78.02738 ... 80.10339
lon_verticies	(y, x, vertex)	float32	1.4255313 1.4255313 ... 73.29744
time_bounds	(bnds, time)	float64	dask.array<chunksize=(2, 1980), meta…
lat	(y, x)	float32	-78.02738 -78.02127 ... 80.15926
lat_bounds	(y, x, bnds)	float32	dask.array<chunksize=(1050, 1442, 1)…
lon_bounds	(y, x, bnds)	float32	dask.array<chunksize=(1050, 1442, 1)…
time	(time)	int64	0 708 1416 ... 1444884 1445616
vertex	(vertex)	int64	0 1 2 3
bnds	(bnds)	int64	0 1
x	(x)	float32	0.25 0.5 0.75 ... 359.75 360.0
y	(y)	float32	-78.107124 -78.05977 ... 90.0
member_id	(member_id)	<U8	'r1i1p1f2'
x_left	(x_left)	float32	0.125 0.375 ... 359.625 359.875
y_right	(y_right)	float32	-78.08345 -78.03601 ... 90.02606
dx_t	(y, x)	float32	0.0 0.0 0.0 ... 12526.91 12527.76
dy_t	(y, x)	float32	0.0 0.0 0.0 ... 11413.054 11409.701
dx_gy	(y_right, x)	float32	0.0 0.0 0.0 ... 12526.062 12527.76
dy_gx	(y, x_left)	float32	0.0 0.0 0.0 ... 11414.309 11411.036
dx_gx	(y, x_left)	float32	5695.537 5695.9785 ... 12526.91
dy_gy	(y_right, x)	float32	0.0 0.0 ... 18432876.0 18439940.0
dx_gxgy	(y_right, x_left)	float32	5695.537 5695.9785 ... 12526.91
dy_gxgy	(y_right, x_left)	float32	2923.577 2926.2566 ... 18436744.0

▾ Data variables:

tos	(y, x, member_id, time)	float32	dask.array<chunksize=(1050, 1442, 1,…

▸ Attributes: (55)

Now there are additional coordinates, representing distances in the x and y direction at different grid locations. We can now calculate the derivative with respect to x.

```
dt_dx = grid.diff(ds.tos, 'X') / ds.dx_gx
dt_dx
```

xarray.DataArray (**y**: 1050, **x_left**: 1442, **member_id**: 1, **time**: 1980)

dask.array<chunksize=(1050, 1, 1, 20), meta=np.ndarray>

▾ Coordinates:

y	(y)	float32	-78.107124 -78.05977 ... 90.0
x_left	(x_left)	float32	0.125 0.375 ... 359.625 359.875
member_id	(member_id)	<U8	'r1i1p1f2'
time	(time)	int64	0 708 1416 ... 1444884 1445616
dy_gx	(y, x_left)	float32	0.0 0.0 0.0 ... 11414.309 11411.036
dx_gx	(y, x_left)	float32	5695.537 5695.9785 ... 12526.91

▸ Attributes: (0)

`xgcm` can handle these cases even smarter, by automatically picking the right grid metric, in this case the zonal distance.

```
dt_dx_auto = grid.derivative(ds.tos, 'X')
dt_dx_auto
```

xarray.DataArray (**y**: 1050, **x_left**: 1442, **member_id**: 1, **time**: 1980)

dask.array<chunksize=(1050, 1, 1, 20), meta=np.ndarray>

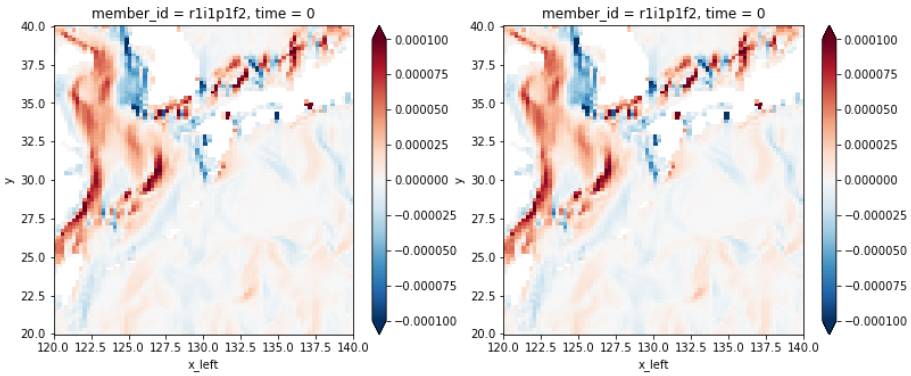
▾ Coordinates:

y	(y)	float32	-78.107124 -78.05977 ... 90.0
x_left	(x_left)	float32	0.125 0.375 ... 359.625 359.875
member_id	(member_id)	<U8	'r1i1p1f2'
time	(time)	int64	0 708 1416 ... 1444884 1445616

▸ Attributes: (0)

```
fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=[13,5])
dt_dx.isel(time=0).sel(x_left=slice(120, 140), y=slice(20,40)).plot(ax=ax1, vmax=1e-4)
dt_dx_auto.isel(time=0).sel(x_left=slice(120, 140), y=slice(20,40)).plot(ax=ax2,
vmax=1e-4)
```

```
<matplotlib.collections.QuadMesh at 0x7f90718e3208>
```



The results are identical, and the user does not have to remember the names of the distance coordinates, `xgcm` takes care of it. The derivative in the y-direction can be calculated similarly.

Then, let's compute the full gradient amplitude for **all** the models in one loop, averaging the values over all members and 5 years at the beginning of the historical period.

```
fig, axarr = plt.subplots(
    ncols=4, nrows=7, figsize=[15, 15], subplot_kw={"projection": ccrs.Robinson(190)}
)
for ax, (k, ds) in zip(axarr.flat, ddict_pp.items()):

    # the cmip6_preprocessing magic:
    # create an xgcm grid object and dataset with reconstructed grid metrics
    grid, ds = combine_staggered_grid(ds, recalculate_metrics=True)

    da = ds.tos

    # some models have large values instead of nans, so we mask unresonable values
    da = da.where(da < 1e30)

    # calculate the zonal temperature gradient
    dt_dx = grid.derivative(da, 'X')
    dt_dy = grid.derivative(da, 'Y', boundary='extend')
    # these values are now situated on the cell faces, we need to
    # interpolate them back to the center to combine them

    dt_dx = grid.interp(dt_dx, 'X')
    dt_dy = grid.interp(dt_dy, 'Y', boundary='extend')
    grad = (dt_dx**2 + dt_dy**2)**0.5

    ds['grad'] = grad

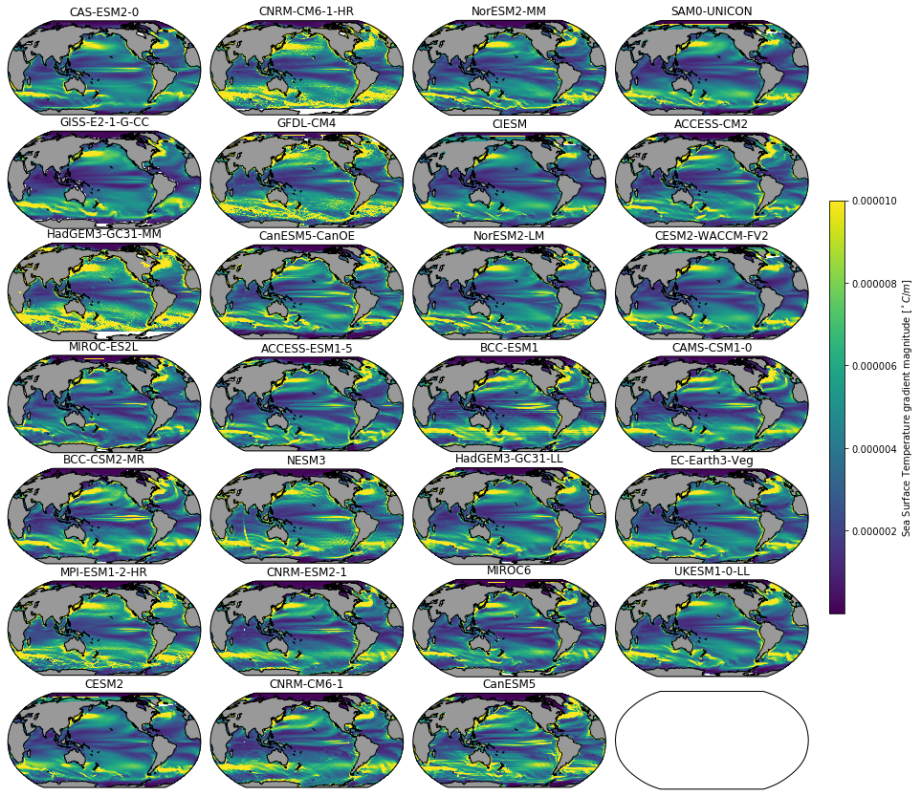
    # take an average over the first 5 years of the run
    ds = ds.isel(time=slice(0,12*5)).mean('time', keep_attrs=True)

    # take the average over all available model members
    if "member_id" in ds.dims:
        ds = ds.mean('member_id', keep_attrs=True)

    # and plot the resulting 2d array using xarray
    pp = ds.grad.plot(
        ax=ax,
        x="lon",
        y="lat",
        vmin=1e-9,
        vmax=1e-5,
        transform=ccrs.PlateCarree(),
        infer_intervals=False,
        add_colorbar=False,
    )
    ax.set_title(ds.attrs['source_id'])
    ax.add_feature(cfeature.LAND, facecolor='0.6')
    ax.coastlines()

fig.subplots_adjust(hspace=0.05, wspace=0.05)
fig.subplots_adjust(right=0.9)
cbar_ax = fig.add_axes([0.92, 0.3, 0.015, 0.4])
fig.colorbar(pp, cax=cbar_ax, label="Sea Surface Temperature gradient magnitude
[$^\circ C/m$]")
```

```
<matplotlib.colorbar.Colorbar at 0x7f907a5a0be0>
```



Not bad for five additional lines of code!

Features like the western boundary currents and the fronts across along the Antarctic Circumpolar Current and the equatorial upwelling zones are clearly visible and present in most models.

Calculate surface vorticity in the North Atlantic

But this was still just operating on a tracer field. What about velocity fields? We can also combine different variables into an xgcm compatible dataset!

The recommended workflow is to first load datasets seperately for each desired variable:

```
variables = ['thetao', 'uo', 'vo']
ddict_multi = {var:{} for var in variables}
for var in variables:
    cat_var = col.search(table_id='Omon',
                         grid_label='gn',
                         experiment_id='historical',
                         variable_id=var,
                         source_id=models)

    ddict_multi[var] = cat_var.to_dataset_dict(
        zarr_kwargs={'consolidated':True, 'decode_times':False},
        preprocess=combined_preprocessing
    )
```

```
--> The keys in the returned dictionary of datasets are constructed as follows:
        'activity_id.institution_id.source_id.experiment_id.table_id.grid_label'
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
MIROC6: No units found
MIROC-ES2L: No units found
CESM2-WACCM-FV2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
```


100.00% [27/27 00:50<00:00]

```
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
MIROC-ES2L: No units found
MIROC6: No units found
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
MIROC-ES2L: No units found
MIROC6: No units found
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
MIROC6: No units found
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
MIROC6: No units found
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
MIROC6: No units found
MIROC6: No units found
MIROC6: No units found
MIROC6: No units found
MIROC6: No units found

--> The keys in the returned dictionary of datasets are constructed as follows:
        'activity_id.institution_id.source_id.experiment_id.table_id.grid_label'
CESM2-WACCM-FV2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
MIROC-ES2L: No units found
MIROC6: No units found
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
```


100.00% [27/27 00:52<00:00]

```
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
MIROC6: No units foundMIROC-ES2L: No units found

CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
MIROC-ES2L: No units found
MIROC6: No units found
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
MIROC6: No units found
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
MIROC6: No units found
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
MIROC6: No units found
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
MIROC6: No units found
MIROC6: No units found
MIROC6: No units found
MIROC6: No units found

--> The keys in the returned dictionary of datasets are constructed as follows:
        'activity_id.institution_id.source_id.experiment_id.table_id.grid_label'
CESM2-WACCM-FV2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
MIROC-ES2L: No units found
MIROC6: No units found
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
```

100.00% [27/27 00:50<00:00]

```
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
MIROC-ES2L: No units found
MIROC6: No units found
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
MIROC-ES2L: No units found
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
MIROC6: No units found
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
MIROC6: No units found
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
MIROC6: No units found
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
MIROC6: No units found
CESM2: Unexpected unit (centimeters) for coordinate `lev` detected.
         Converted to `m`
MIROC6: No units found
MIROC6: No units found
MIROC6: No units found
MIROC6: No units found
```

Now we need to make sure to choose only the intersection between the keys of each sub-dictionary, because we need all 3 variables for each model.

```
ddict_multi_clean = {var:{} for var in variables}
for k in ddict_multi['thetao']:
    if all([k in ddict_multi[var] for var in variables]) and 'lev' in
ddict_multi['thetao'][k].dims:
        for var in variables:
            ddict_multi_clean[var][k] = ddict_multi[var][k]
```

And now similar as before we can create a complete grid dataset, but we can pass additional datasets to be combined using the `other_ds` argument. This can be a list of different variables, `cmip6_preprocessing` automatically detects the appropriate grid position.

```
import matplotlib.path as mpath
import numpy as np
from xgcm import Grid
import xarray as xr

fig, axarr = plt.subplots(
    ncols=4, nrows=6, figsize=[25, 20], subplot_kw={"projection": ccrs.PlateCarree()}
)
for ai ,(ax, k) in enumerate(zip(axarr.flat, ddict_multi_clean['thetao'].keys())):
    ds_t = ddict_multi_clean['thetao'][k] # One tracer should always be the reference
dataset!
    ds_u = ddict_multi_clean['uo'][k]
    ds_v = ddict_multi_clean['vo'][k]

    if 'lev' in ds_t: # show only models with depth coordinates in the vertical
        # aling only the intersection of the member_ids
        ds_t, ds_u, ds_v = xr.align(ds_t, ds_u, ds_v, join='inner', exclude=[di for di
in ds_t.dims if di != 'member_id'])

        # combine all datasets and create grid with metrics
        grid, ds = combine_staggered_grid(ds_t, other_ds=[ds_u, ds_v],
recalculate_metrics=True)


        # interpolate grid metrics at the corner points (these are not *yet*
constructed by cmip6_preprocessing)
        ds.coords['dx_temp'] = grid.interp(ds['dx_gx'], 'Y', boundary='extend')
        ds.coords['dy_temp'] = grid.interp(ds['dy_gy'], 'X')

        ds = ds.chunk({'x':360})

        # selecting the surface value
        ds = ds.isel(lev=0)

        # For demonstration purposes select the first member and a single monthly
timestep
        ds = ds.isel(time=10)

        if "member_id" in ds.dims:
            ds = ds.isel(member_id=0)

        grid = Grid(ds, periodic=['X'], metrics={'X':[co for co in ds.coords if 'dx'
in co],
                                                 'Y':[co for co in ds.coords if 'dy' in
co]})


        dv_dx = grid.derivative(ds.vo, 'X')
        du_dy = grid.derivative(ds.uo, 'Y',boundary='extend')

        # check the position of the derivatives and interpolate back to tracer point
for plotting
        if any(['x_' in di for di in dv_dx.dims]):
            dv_dx = grid.interp(dv_dx, 'X')
        if any(['y_' in di for di in dv_dx.dims]):
            dv_dx = grid.interp(dv_dx, 'Y', boundary='extend')

        if any(['x_' in di for di in du_dy.dims]):
            du_dy = grid.interp(du_dy, 'X')
        if any(['y_' in di for di in du_dy.dims]):
            du_dy = grid.interp(du_dy, 'Y', boundary='extend')

        curl = dv_dx - du_dy

        ds['curl'] = curl

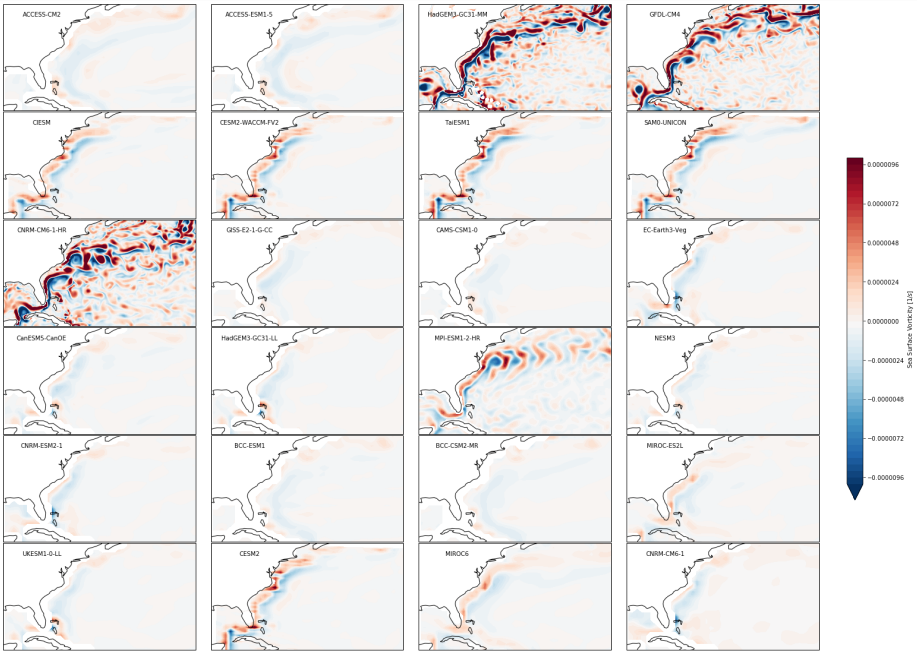
        # Select the North Atlantic
        ds = ds.sel(x=slice(270,335), y=slice(10,55))

        # and plot the resulting 2d array using xarray
        pp = ds.curl.plot.contourf(
            levels=51,
            ax=ax,
            x="lon",
            y="lat",
            vmax=1e-5,
            transform=ccrs.PlateCarree(),
            infer_intervals=False,
            add_colorbar=False,
            add_labels=False
        )

ax.text(0.2,0.9,ds.attrs['source_id'],horizontalalignment='center',verticalalignment='c
enter',
                transform=ax.transAxes, fontsize=10)
        ax.coastlines()
        ax.set_extent([-90, -45, 20, 45], ccrs.PlateCarree())
```

```
fig.subplots_adjust(wspace=0.01, hspace=0.01)
fig.subplots_adjust(right=0.9)
cbar_ax = fig.add_axes([0.92, 0.3, 0.015, 0.4])
fig.colorbar(pp, cax=cbar_ax, label="Sea Surface Vorticity [$1/s$]")
```

```
<matplotlib.colorbar.Colorbar at 0x7f8ef549ee10>
```



A caveat: Reconstructed metrics

For all the examples shown here, we have relied on a simplified reconstruction of the grid metrics (in this case, the distances between different points on the grid). We can check the quality of the reconstruction indirectly by comparing these to the only horizontal grid metric that is saved with the ocean model output: The horizontal grid cell area.

We reconstruct our area naively as the product of the x and y distances centered on the tracer cell, `dx_t`, and `dy_t`.

```
ds = ddict_multi_clean['thetao']['CMIP.NOAA-GFDL.GFDL-CM4.historical.Omon.gn']
_,ds = combine_staggered_grid(ds, recalculate_metrics=True)
area_reconstructed = ds.dx_t * ds.dy_t
```

Now lets load the actual area and compare the two.

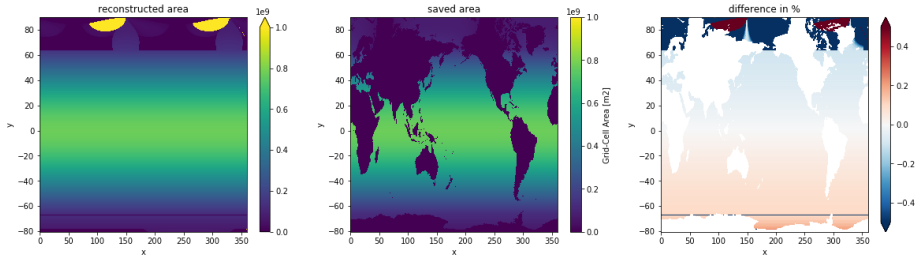
```
url = "https://raw.githubusercontent.com/NCAR/intake-esm-
datastore/master/catalogs/pangeo-cmip6.json"
col = intake.open_esm_datastore(url)
cat = col.search(table_id='Ofx', variable_id='areacello', source_id='GFDL-CM4' ,
grid_label='gn') # switch to `grid_label='gr'` for regridded file
ddict = cat.to_dataset_dict(zarr_kwargs={'consolidated':True},
preprocess=combined_preprocessing)
_,ds_area = ddict.popitem()
area = ds_area.areacello.isel(member_id=0).squeeze()
```

```
--> The keys in the returned dictionary of datasets are constructed as follows:
        'activity_id.institution_id.source_id.experiment_id.table_id.grid_label'
```

100.00% [2/2 00:00<00:00]

```
fig, (ax1, ax2, ax3) = plt.subplots(ncols=3, figsize=[20,5])
area_reconstructed.plot(ax=ax1, vmax=1e9)
area.plot(ax=ax2, vmax=1e9)
((area_reconstructed - area) / area * 100).plot(ax=ax3, vmax=0.5)
ax1.set_title('reconstructed area')
ax2.set_title('saved area')
ax3.set_title('difference in %')
```

```
Text(0.5, 1.0, 'difference in %')
```



You can see that for this particular model (`GFDL-CM4`), the reconstruction does reproduce the grid area with a less than 0.5% error between approximately 60S-60N. North of that, the grid geometry gets vastly more complicated, and the simple reconstruction fails. The area over which the reconstruction varies from model to model and as such caution should always be exercised when analyzing data using reconstructed metrics.

At this point, this is, however, the only way to run these kinds of analyses, since the original grid metrics are not routinely provided with the CMIP6 output. In order to truly reproduce analyses like e.g., budgets, the community requires the native model geometry.

Conclusions

We show that using `cmip6_preprocessing` and `xgcm` users can analyze complex native ocean model grids without the need to interpolate data or keep track of the intricacies of single model grids. Using these tools already enables users to calculate common operators like the gradient and curl, weighted averages and more, in a 'grid-agnostic' way, with decent precision outside of the polar regions.

Pending on the availability of more grid metric output and building on these tools, complex analyses like various budgets could become a matter of a few lines and be calculated across all models in the CMIP6 archive.

By combining generalizable and reproducible analysis with the publicly available CMIP6 data, more users will be able to analyze the data efficiently, leading to faster understanding and synthesis of the vast amount of data provided by modern climate modeling.

Acknowledgements

Julius Busecke's contributions to this work are motivated and developed as part of an ongoing project together with his postdoc advisor Prof. Laure Resplandy. This project investigates the influence of equatorial dynamics on possible expansions of Oxygen Minimum Zones in the Pacific and is funded under the NOAA Cooperative Institute for Climate Science agreement NA14OAR4320106.

Scikit-downscale: an open source Python package for scalable climate downscaling

Joseph Hamman (jhamman@ucar.edu) and Julia Kent (jkent@ucar.edu)

NCAR, Boulder, CO, USA

This notebook was developed for the 2020 EarthCube All Hands Meeting. The development of Scikit-downscale done in conjunction with the development of the Pangeo Project and was supported by the following awards:

- NSF-GEO-AGS 1928374: EarthCube Data Capabilities: Collaborative Proposal: Jupyter meets the Earth: Enabling discovery in geoscience through interactive computing at scale
- NSF-OIA 1937136: Convergence Accelerator Phase I (RAISE): Knowledge Open Network Queries for Research (KONQUER)

ECAHM 2020 ID: 143

---

1. Introduction

Climate data from Earth System Models (ESMs) are increasingly being used to study the impacts of climate change on a broad range of biogeophysical systems (forest fire, flood, fisheries, etc.) and human systems (water resources, power grids, etc.). Before this data can be used to study many of these systems, post-processing steps commonly referred to as

bias correction and statistical downscaling must be performed. “Bias correction” is used to correct persistent biases in climate model output and “statistical downscaling” is used to increase the spatiotemporal resolution of the model output (i.e. from 1 deg to 1/16th deg grid boxes). For our purposes, we’ll refer to both parts as “downscaling”.

In the past few decades, the applications community has developed a plethora of downscaling methods. Many of these methods are ad-hoc collections of processing routines while others target very specific applications. The proliferation of downscaling methods has left the climate applications community with an overwhelming body of research to sort through without much in the form of synthesis guilding method selection or applicability.

Motivated by the pressing socio-environmental challenges of climate change – and with the learnings from previous downscaling efforts (e.g. Gutmann et al. 2014, Lanzante et al. 2018) in mind – we have begun working on a community-centered open framework for climate downscaling: scikit-downscale. We believe that the community will benefit from the presence of a well-designed open source downscaling toolbox with standard interfaces alongside a repository of benchmark data to test and evaluate new and existing downscaling methods.

In this notebook, we provide an overview of the scikit-downscale project, detailing how it can be used to downscale a range of surface climate variables such as surface air temperature and precipitation. We also highlight how scikit-downscale framework is being used to compare exisiting methods and how it can be extended to support the development of new downscaling methods.

2. Scikit-downscale

Scikit-downscale is a new open source Python project. Within Scikit-downscale, we are been curating a collection of new and existing downscaling methods within a common framework. Key features of Scikit-downscale are:

- A high-level interface modeled after the popular `fit` / `predict` pattern found in many machine learning packages (Scikit-learn, Tensorflow, etc.),
- Uses Xarray and Pandas data structures and utilities for handling of labeled datasets,
- Utilities for automatic parallelization of pointwisde downscaling models,
- Common interface for pointwise and spatial (or global) downscaling models, and
- Extensible, allowing the creation of new downscaling methods through composition.

Scikit-downscale’s source code is available on GitHub.

2.1 Pointwise Models

We define pointwise methods as those that only use local information during the downscaling process. They can be often represented as a general linear model and fit independently across each point in the study domain. Examples of existing pointwise methods are:

- BCSD_[Temperature, Precipitation]: Wood et al. (2004)
- ARRM: Stoner et al. (2012)
- (Hybrid) Delta Method (e.g. Hamlet et al. (2010)
- GARD: Gutmann et al (in prep).

Because pointwise methods can be written as a stand-alone linear model, Scikit-downscale implements these models as a Scikit-learn LinearModel or Pipeline. By building directly on Scikit-learn, we inherit a well defined model API and the ability to interoperate with a robust ecosystem utilities for model evaluation and optimization (e.g. grid-search). Perhaps more importantly, this structure also allows us to compare methods at a high-level of granularity (single spatial point) before deploying them on large domain problems.

Begin interactive demo

From here forward in this notebook, we’ll jump back and forth between Python and text cells to describe how scikit-downscale works.

This first cell just imports some libraries and get’s things setup for our analysis to come.

```
%load_ext autoreload
%autoreload 2
%matplotlib inline

import warnings
warnings.filterwarnings("ignore")  # sklearn

import matplotlib.pyplot as plt
import seaborn as sns

import pandas as pd

from utils import get_sample_data

sns.set(style='darkgrid')
```

Now that we've imported a few libraries, let's open a sample dataset from a single point in North America. We'll use this data to explore Scikit-downscale and its existing functionality. You'll notice there are two groups of data, `training` and `targets`. The `training` data is meant to represent data from a typical climate model and the `targets` data is meant to represent our "observations". For the purpose of this demonstration, we've choosen training data sampled from a regional climate model (WRF) run at 50km resolution over North America. The observations are sampled from the nearest 1/16th grid cell in Livneh et al, 2013.

We have choosen to use the `tmax` variable (daily maximum surface air temperature) for demonstration purposes. With a small amount of effort, an interested reader could swap `tmax` for `pcp` and test these methods on precipitation.

```
# load sample data
training = get_sample_data('training')
targets = get_sample_data('targets')

# print a table of the training/targets data
display(pd.concat({'training': training, 'targets': targets}, axis=1))

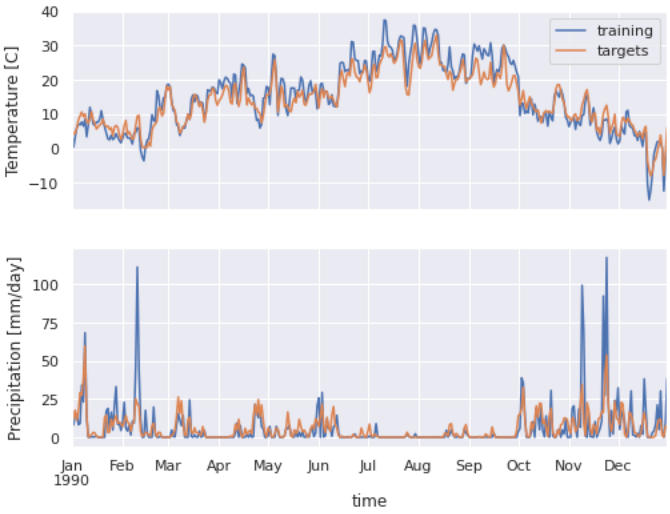
# make a plot of the temperature and precipitation data
fig, axes = plt.subplots(ncols=1, nrows=2, figsize=(8, 6), sharex=True)
time_slice = slice('1990-01-01', '1990-12-31')

# plot-temperature
training[time_slice]['tmax'].plot(ax=axes[0], label='training')
targets[time_slice]['tmax'].plot(ax=axes[0], label='targets')
axes[0].legend()
axes[0].set_ylabel('Temperature [C]')

# plot-precipitation
training[time_slice]['pcp'].plot(ax=axes[1])
targets[time_slice]['pcp'].plot(ax=axes[1])
_ = axes[1].set_ylabel('Precipitation [mm/day]')
```

	training		targets	
	tmax	pcp	tmax	pcp
time				
1950-01-01	NaN	NaN	-0.22	5.608394
1950-01-02	NaN	NaN	-4.54	2.919726
1950-01-03	NaN	NaN	-7.87	3.066762
1950-01-04	NaN	NaN	-5.08	4.684164
1950-01-05	NaN	NaN	-0.79	4.295568
...	...	...	...	...
2015-11-26	7.657013	0.000000e+00	NaN	NaN
2015-11-27	7.687256	0.000000e+00	NaN	NaN
2015-11-28	10.480835	0.000000e+00	NaN	NaN
2015-11-29	11.728516	0.000000e+00	NaN	NaN
2015-11-30	10.285431	3.152419e-13	NaN	NaN

24075 rows × 4 columns



2.2 Models as cattle, not pets

As we mentioned above, Scikit-downscale utilizes a similiar API to that of Scikit-learn for its pointwise models. This means we can build collections of models that may be quite different internally, but operate the same at the API level. Importantly, this means that all downscaling methods have two common API methods: `fit`, which trains the model given *training* and *targets* data, and `predict` which uses the fit model to perform the downscaling opperation. This is perhaps the most important feature of Scikit-downscale, the ability to test and compare arbitrary combinations of models under a common interface. This allows us to try many combinations of models and parameters, choosing only the best combinations. The following pseudo-code block describe the workflow common to all scikit-downscale models:

```
from skdownscale.pointwise_models import MyModel

...
# load and pre-process input data (X and y)
...

model = MyModel(**parameters)
model.fit(X_train, y)
predictions = model.predict(X_predict)

...
# evaluate and/or save predictions
...
```

In the cell below, we'll create nine different downscaling models, some from Scikit-downscale and some from Scikit-learn.

```
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor

from skdownscale.pointwise_models import PureAnalog, AnalogRegression
from skdownscale.pointwise_models import BcsdTemperature, BcsdPrecipitation


models = {
    'GARD: PureAnalog-best-1': PureAnalog(kind='best_analog', n_analogs=1),
    'GARD: PureAnalog-sample-10': PureAnalog(kind='sample_analogs', n_analogs=10),
    'GARD: PureAnalog-weight-10': PureAnalog(kind='weight_analogs', n_analogs=10),
    'GARD: PureAnalog-weight-100': PureAnalog(kind='weight_analogs', n_analogs=100),
    'GARD: PureAnalog-mean-10': PureAnalog(kind='mean_analogs', n_analogs=10),
    'GARD: AnalogRegression-100': AnalogRegression(n_analogs=100),
    'GARD: LinearRegression': LinearRegression(),
    'BCSD: BcsdTemperature': BcsdTemperature(return_anoms=False),
    'Sklearn: RandomForestRegressor': RandomForestRegressor(random_state=0)
}

train_slice = slice('1980-01-01', '1989-12-31')
predict_slice = slice('1990-01-01', '1999-12-31')
```

Now that we've created a collection of models, we want to train the models on the same input data. We do this by looping through our dictionary of models and calling the `fit` method:

```
# extract training / prediction data
X_train = training[['tmax']][train_slice]
y_train = targets[['tmax']][train_slice]
X_predict = training[['tmax']][predict_slice]

# Fit all models
for key, model in models.items():
    model.fit(X_train, y_train)
```

Just like that, we fit nine downscaling models. Now we want to use those models to downscale/bias-correct our data. For the sake of easy comparison, we'll use a different part of the training data:

```
# store predicted results in this dataframe
predict_df = pd.DataFrame(index = X_predict.index)

for key, model in models.items():
    predict_df[key] = model.predict(X_predict)

# show a table of the predicted data
display(predict_df.head())
```

	GARD: PureAnalog-best-1	GARD: PureAnalog-sample-10	GARD: PureAnalog-weight-10	GARD: PureAnalog-weight-100	GARD: PureAnalog-mean-10	GARD: AnalogRegress 100
time						
1990-01-01	4.50	5.67	5.375299	5.697786	5.895	5.93
1990-01-02	6.13	3.55	3.543398	3.264698	2.561	2.51
1990-01-03	5.46	3.04	4.963575	4.933534	4.692	4.86
1990-01-04	8.57	5.90	8.369125	8.239455	7.340	7.25
1990-01-05	5.67	7.03	7.424970	7.583703	7.705	7.71

Now, let's do some sample analysis on our predicted data. First, we'll look at a timeseries of all the downscaled timeseries for the first year of the prediction period. In the figure below, the `target` (truth) data is shown in black, the original (pre-correction) data is shown in grey, and each of the downscaled data timeseries is shown in a different color.

```
fig, ax = plt.subplots(figsize=(8, 3.5))
targets['tmax'][time_slice].plot(ax=ax, label='target', c='k', lw=1, alpha=0.75,
legend=True, zorder=10)
X_predict['tmax'][time_slice].plot(label='original', c='grey', ax=ax, alpha=0.75,
legend=True)
predict_df[time_slice].plot(ax=ax, lw=0.75)
ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))
_ = ax.set_ylabel('Temperature [C]')
```

Of course, its difficult to tell which of the nine downscaling methods performed best from our plot above. We may want to evaluate our predictions using a standard statistical score, such as \(r^2\). Those results are easily computed below:

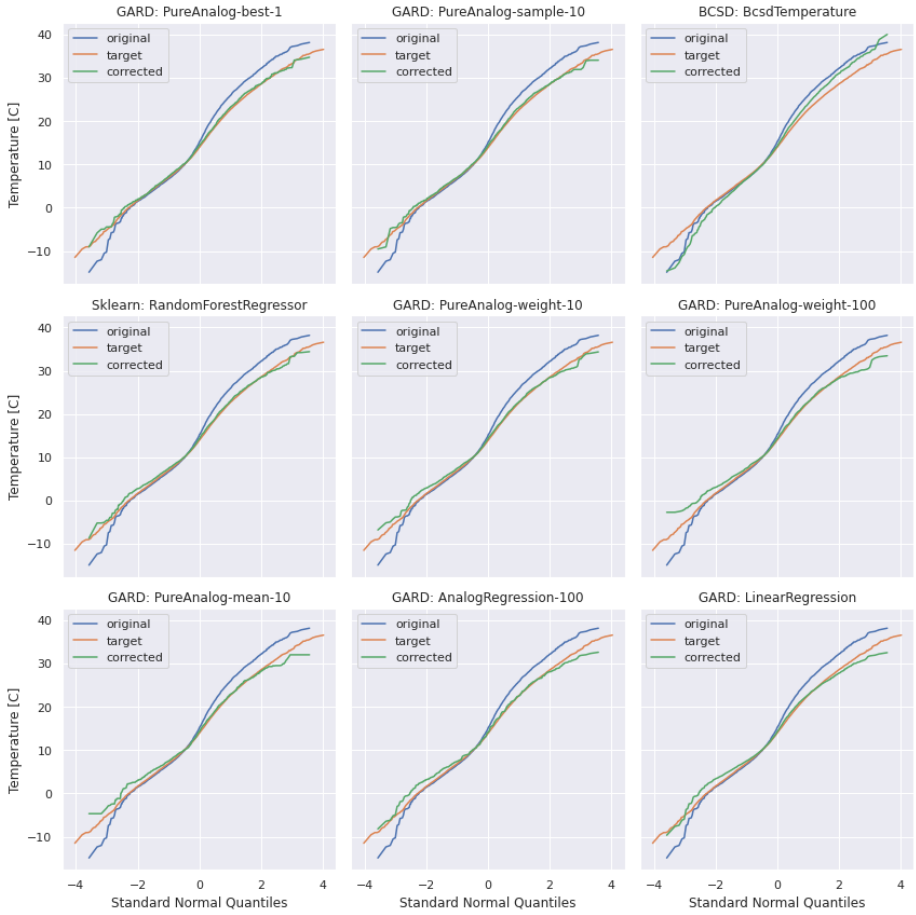
```
# calculate r2
score = (predict_df.corrwith(targets.tmax[predict_slice])
**2).sort_values().to_frame('r2_score')
display(score)
```

	r2_score
GARD: PureAnalog-best-1	0.820281
GARD: PureAnalog-sample-10	0.820977
BCSD: BcsdTemperature	0.858258
Sklearn: RandomForestRegressor	0.864160
GARD: PureAnalog-weight-10	0.881287
GARD: PureAnalog-weight-100	0.892049
GARD: PureAnalog-mean-10	0.899297
GARD: AnalogRegression-100	0.906217
GARD: LinearRegression	0.906316

All of our downscaling methods seem to be doing fairly well. The timeseries and statistics above shows that all our methods are producing generally resonable results. However, we are often interested in how our models do at predicting extreme events. We can quickly look into those aspects of our results using the `qq` plots below. There you'll see that the models diverge in some interesting ways. For example, while the `LinearRegression` method has the highest \(r^2\) score, it seems to have trouble capturing extreme heat events. Whereas many of the analog methods, as well as the `RandomForestRegressor`, perform much better on the tails of the distributions.

```
from utils import prob_plots

fig = prob_plots(X_predict, targets['tmax'], predict_df[score.index.values], shape=(3,
3), figsize=(12, 12))
```



In this section, we've shown how easy it is to fit, predict, and evaluate scikit-downscale models. The seamless interoperability of these models clearly facilitates a workflow that enables a deeper level of model evaluation that is otherwise possible in the downscaling world.

2.3 Tailor-made methods in a common framework

In the section above, we showed how it is possible to use scikit-downscale to bias-correct a timeseries of daily maximum air temperature using an arbitrary collection of linear models. Some of those models were general machine learning methods (e.g. `LinearRegression` or `RandomForestRegressor`) while others were tailor-made methods developed specifically for downscaling (e.g. `BCSDTemperature`). In this section, we walk through how new pointwise methods can be added to the scikit-downscale framework, highlighting the Z-Score method along the way.

2.3.1 Z-Score Method

Z-Score bias correction is a good technique for target variables with Gaussian probability distributions, such as zonal wind speed.

In essence the technique:

1. Finds the mean $\(\overline{x} = \sum_{i=0}^N \frac{x_i}{N}\)\(and standard deviation \)\(\sigma = \sqrt{\frac{\sum_{i=0}^N |x_i - \overline{x}|^2}{N-1}}\)\)$ of target (measured) data and training (historical modeled) data.
2. Compares the difference between the statistical values to produce a shift $\(shift = \overline{x_{target}} - \overline{x_{training}}\)\)\(and scale parameter \)\(scale = \sigma_{target} \div \sigma_{training}\)\)$
3. Applies these paramaters to the future model data to be corrected to get a new mean $\(\overline{x_{corrected}} = \overline{x_{future}} + shift\)\)\(and new standard deviation \)\(\sigma_{corrected} = \sigma_{future} \times scale\)\)$
4. Calculates the corrected values $\(x_{corrected_{i}} = z_i \times \sigma_{corrected} + \overline{x_{corrected}}\)\)\(from the future model's z-score values \)\(z_i = \frac{x_i-\overline{x}}{\sigma}\)\)$

In practice, if the wind was on average 3 m/s faster on the first of July in the models compared to the measurements, we would adjust the modeled data for all July 1sts in the future modeled dataset to be 3 m/s faster. And similarly for scaling the standard deviation

2.3.2 Building the ZScoreRegressor Class

Scikit-downscale's pointwise all implement Scikit-learn's `fit`/`predict` API. Each new downscaler must implement a minimum of three class methods: `__init__`, `fit`, `predict`.

```
class AbstractDownscaler(object):

    def __init__(self):
        ...

    def fit(self, X, y):
        ...
        return self

    def predict(X):
        ...
        return y_hat
```

Ommitting some of the complexity in the full implementation (which can be found in the full implementation on GitHub), we demonstrate how the `ZScoreRegressor` was built:

First, we define our `__init__` method, allowing users to specify specific options (in this case `window_width`):

```
class ZScoreRegressor(object):

    def __init__(self, window_width=31):
        self.window_width = window_width
```

Next, we define our `fit` method,

```
    def fit(self, X, y):
        X_mean, X_std = _calc_stats(X.squeeze(), self.window_width)
        y_mean, y_std = _calc_stats(y.squeeze(), self.window_width)

        self.stats_dict_ = {
            "X_mean": X_mean,
            "X_std": X_std,
            "y_mean": y_mean,
            "y_std": y_std,
        }

        shift, scale = _get_params(X_mean, X_std, y_mean, y_std)

        self.shift_ = shift
        self.scale_ = scale
        return self
```

Finally, we define our `predict` method,

```
    def predict(self, X):

        fut_mean, fut_std, fut_zscore = _get_fut_stats(X.squeeze(), self.window_width)
        shift_expanded, scale_expanded = _expand_params(X.squeeze(), self.shift_, self.scale_)

        fut_mean_corrected, fut_std_corrected = _correct_fut_stats(
            fut_mean, fut_std, shift_expanded, scale_expanded
        )

        self.fut_stats_dict_ = {
            "meani": fut_mean,
            "stdi": fut_std,
            "meanf": fut_mean_corrected,
            "stdf": fut_std_corrected,
        }

        fut_corrected = (fut_zscore * fut_std_corrected) + fut_mean_corrected

        return fut_corrected.to_frame(name)
```

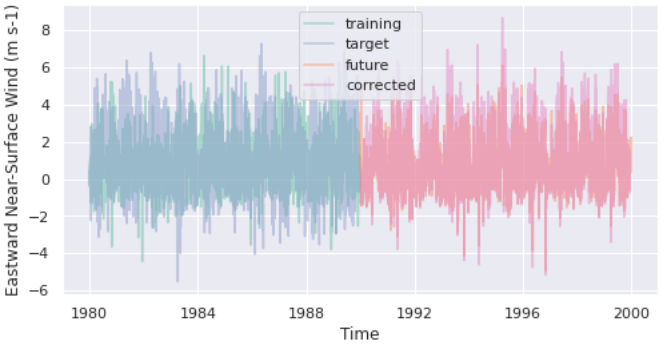
```
from skdownscale.pointwise_models import ZScoreRegressor
```

```
# open a small dataset
training = get_sample_data('wind-hist')
target = get_sample_data('wind-obs')
future = get_sample_data('wind-rcp')
```

```
# bias correction using ZScoreRegresssor
zscore = ZScoreRegressor()
zscore.fit(training, target)
fit_stats = zscore.fit_stats_dict_
out = zscore.predict(future)
predict_stats = zscore.predict_stats_dict_
```

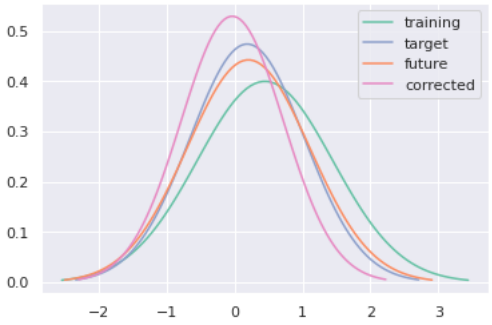
```
# visualize the datasets
from utils import zscore_ds_plot

zscore_ds_plot(training, target, future, out)
```




```
from utils import zscore_correction_plot

zscore_correction_plot(zscore)
```



2.4 Automatic Parallelization

In the examples above, we have performed downscaling on sample data sourced from individual points. In many downscaling workflows, however, users will want to apply pointwise methods at all points in their study domain. For this use case, scikit-downscale provides a high-level wrapper class: `PointWiseDownscaler`.

In the example below, we'll use the `BCSDTemperature` model, along with the `PointWiseDownscaler` wrapper, to downscale daily maximum surface air temperature from CMIP6 for all point in a subset of the Pacific Norwest. We'll use a local Dask Cluster to distribute the computation among our available processors. Though not the point of this example, we also use intake-esm to access CMIP6 data stored on Google Cloud Storage.

Data:

- Training / Prediction data: NASA-GISS-E2 historical data from CMIP6
- Targets: GridMet daily maximum surface air temperature

```
# parameters
train_slice = slice('1980', '1982')  # train time range
holdout_slice = slice('1990', '1991')  # prediction time range

# bounding box of downscaling region
lon_slice = slice(-124.8, -120.0)
lat_slice = slice(50, 45)

# chunk shape for dask execution (time must be contiguous, ie -1)
chunks = {'lat': 10, 'lon': 10, 'time': -1}
```

Step 1: Start a Dask Cluster. Xarray and the `PointWiseDownscaler` will make use of this cluster when it comes time to load input data and train/predict downscaling models.

```
from dask.distributed import Client

client = Client()
client
```

Client

Scheduler: tcp://127.0.0.1:41711
Dashboard: /user/jhamman/proxy/8787/status

Cluster

Workers: 4
Cores: 4
Memory: 25.77 GB

Step 2. Load our target data.

Here we use xarray directly to load a collection of OpenDAP endpoints.

```
import xarray as xr

fnames =
[f'http://thredds.northwestknowledge.net:8080/thredds/dodsC/MET/tmmx/tmmx_{year}.nc'
          for year in range(int(train_slice.start), int(train_slice.stop) + 1)]
# open the data and cleanup a bit of metadata
obs = xr.open_mfdataset(fnames, engine='pydap', concat_dim='day').rename({'day':
'time'}).drop('crs')

obs_subset = obs['air_temperature'].sel(time=train_slice, lon=lon_slice,
lat=lat_slice).resample(time='1d').mean().load(scheduler='threads').chunk(chunks)

# display
display(obs_subset)
obs_subset.isel(time=0).plot()
```

xarray.DataArray 'air_temperature' (**time**: 1096, **lat**: 106, **lon**: 115)

dask.array<chunksize=(1096, 10, 10), meta=np.ndarray>

▾ Coordinates:

time	(time)	datetime64[ns]	1980-01-01 ... 1982-12-31
lat	(lat)	float64	49.4 49.36 49.32 ... 45.07 45.03
lon	(lon)	float64	-124.8 -124.7 ... -120.1 -120.0

▸ Attributes: (0)

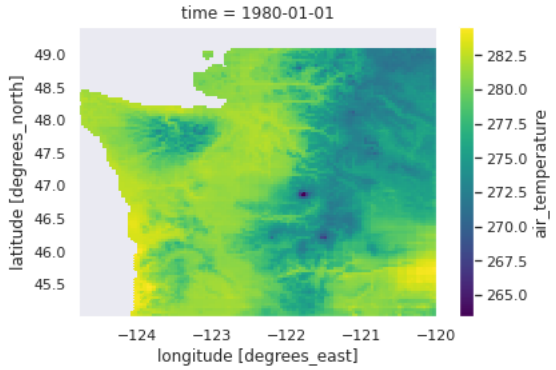
```
<matplotlib.collections.QuadMesh at 0x7fbffec07610>
```

Step 3: Load our training/prediction data.

Here we use `intake-esm` to access a single Xarray dataset from the Pangeo's Google Cloud CMIP6 data catalog.

```
import intake_esm
intake_esm.__version__
```

```
'2020.6.11'
```



```
import intake

# search the cmip6 catalog
col = intake.open_esm_datastore("https://storage.googleapis.com/cmip6/pangeo-
cmip6.json")
cat = col.search(experiment_id=['historical', 'ssp585'], table_id='day',
variable_id='tasmax',
                 grid_label='gn')

# access the data and do some cleanup
ds_model = cat['CMIP.NASA-GISS.GISS-E2-1-
G.historical.day.gn'].to_dask().squeeze(drop=True).drop(['height', 'lat_bnds',
'lon_bnds', 'time_bnds'])
ds_model.lon.values[ds_model.lon.values > 180] -= 360
ds_model = ds_model.roll(lon=72, roll_coords=True)

# regional subsets, ready for downscaling
train_subset =
ds_model['tasmax'].sel(time=train_slice).interp_like(obs_subset.isel(time=0,
drop=True), method='linear')
train_subset['time'] = train_subset.indexes['time'].to_datetimeindex()
train_subset =
train_subset.resample(time='1d').mean().load(scheduler='threads').chunk(chunks)

holdout_subset =
ds_model['tasmax'].sel(time=holdout_slice).interp_like(obs_subset.isel(time=0,
drop=True), method='linear')
holdout_subset['time'] = holdout_subset.indexes['time'].to_datetimeindex()
holdout_subset =
holdout_subset.resample(time='1d').mean().load(scheduler='threads').chunk(chunks)

# display
display(train_subset)
train_subset.isel(time=0).plot()
```

xarray.DataArray 'tasmax' (**time**: 1096, **lat**: 106, **lon**: 115)

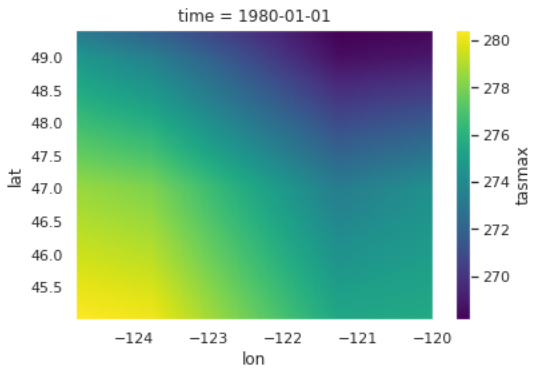
dask.array<chunksize=(1096, 10, 10), meta=np.ndarray>

▾ Coordinates:

time	(time)	datetime64[ns]	1980-01-01 ... 1982-12-31
lat	(lat)	float64	49.4 49.36 49.32 ... 45.07 45.03
lon	(lon)	float64	-124.8 -124.7 ... -120.1 -120.0

▸ Attributes: (0)

```
<matplotlib.collections.QuadMesh at 0x7fbffe6eff70>
```



Step 4. Now that we have loaded our training and target data, we can move on to fit our BcsdTemperature model at each x/y point in our domain. This is where the `PointWiseDownscaler` comes in:

```
from skdownscale.pointwise_models import PointWiseDownscaler
from dask.diagnostics import ProgressBar

model = PointWiseDownscaler(BcsdTemperature(return_anoms=False))
model
```

```
<skdownscale.PointWiseDownscaler>
  Fit Status: False
  Model:
    BcsdTemperature(return_anoms=False)
```

Step 5. We fit the `PointWiseDownscaler`, passing it data in Xarray data structures (our regional subsets from above). This opperation is lazy and return immediately. Under the hood, we can see that `PointWiseDownscaler._models` is an `Xarray.DataArray` of `BcsdTemperature` models.

Note: The following two cells may take a few minutes, or longer, to complete depending on how many cores your computer has, and your internet connection.

```
model.fit(train_subset, obs_subset)
display(model, model._models)
```

```
<skdownscale.PointWiseDownscaler>
  Fit Status: True
  Model:
    BcsdTemperature(return_anoms=False)
```

xarray.DataArray 'tasmax' (**lat**: 106, **lon**: 115)

dask.array<chunksize=(10, 10), meta=np.ndarray>

▾ Coordinates:

lat	(lat)	float64	49.4 49.36 49.32 ... 45.07 45.03
lon	(lon)	float64	-124.8 -124.7 ... -120.1 -120.0

▸ Attributes: (0)

Step 6. Finally, we can use our model to complete the downscaling workflow using the `predict` method along with our `holdout_subset` of CMIP6 data. We call the `.load()` method to eagerly compute the data. We end by plotting a map of downscaled data over our study area.

```
predicted = model.predict(holdout_subset).load()
display(predicted)
predicted.isel(time=0).plot()
```

xarray.DataArray (**time**: 730, **lat**: 106, **lon**: 115)

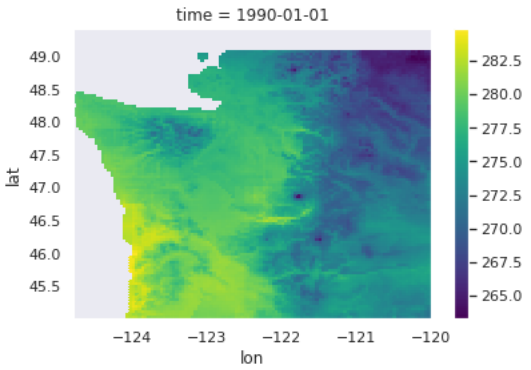
nan nan nan nan nan ... 280.30255 280.80515 280.2077 279.81027

▾ Coordinates:

lat	(lat)	float64	49.4 49.36 49.32 ... 45.07 45.03
time	(time)	datetime64[ns]	1990-01-01 ... 1991-12-31
lon	(lon)	float64	-124.8 -124.7 ... -120.1 -120.0

▸ Attributes: (0)

```
<matplotlib.collections.QuadMesh at 0x7fbffcad1be0>
```



2.2 Spatial Models

Spatial models is a second class of downscaling methods that use information from the full study domain to form relationships between observations and ESM data. Scikit-downscale implements these models as as SpatialDownscaler. Beyond providing fit and predict methods that accept Xarray objects, the internal layout of these methods is intentionally unspecified. We are currently working on wrapping a few popular spatial downscaling models such as:

- MACA: Multivariate Adaptive Constructed Analogs, Abatzoglou and Brown (2012)
- LOCA: Localized Constructed Analogs, Pierce, Cayan, and Thrasher (2014)

3. Discussion

3.1 Benchmark Applications

Its likely that one of the reasons we haven't seen strong consensus develop around particularl downscaling methodologies is the abscense of widely available benchamrk applications to test methods against eachother. While Scikit-downscale will not solve this problem on its own, we hope the ability to implemnt downscaling applications within a common framework will enable a more robust benchmarking inititive that previously possible.

3.2 Call for Participation

The Scikit-downscale effort is just getting started. With the recent release of CMIP6, we expect a surge of interest in downscaled climate data. There are clear opportunities for involvement from climate impacts practicioneers, computer scientists with an interest in machine learning for climate applications, and climate scientists alike. Please reach out if you are interested in participating in any way.

References

1. Abatzoglou, J. T. (2013), Development of gridded surface meteorological data for ecological applications and modelling. Int. J. Climatol., 33: 121–131.
2. Abatzoglou J.T. and Brown T.J. A comparison of statistical downscaling methods suited for wildfire applications, International Journal of Climatology (2012), 32, 772-780
3. Gutmann, E., Pruitt, T., Clark, M. P., Brekke, L., Arnold, J. R., Raff, D. A., and Rasmussen, R. M. (2014), An intercomparison of statistical downscaling methods used for water resource assessments in the United States, Water Resour. Res., 50, 7167– 7186, doi:10.1002/2014WR015559.
4. Gutmann, E., J. Hamman, M. Clark, T. Eidhammer, A. Wood, J. Arnold, K. Nowak (in prep), Evaluating the effect of statistical downscaling methodological choices in a common framework. To be submitted to JGR-Atomspheres.
5. Hamlet, A.F., Salathé, E.P., and Carrasco, P., 2010. Statistical downscaling techniques for global climate model simulations of temperature and precipitation with application to water resources planning studies. A report prepared by the Climate Impact Group for Columbia Basin Climate Change Scenario Project, University of Washington, Seattle, WA. http://www.hydro.washington.edu/2860/products/sites/r7climate/study_report/CBCCSP_chap4_gcm_draft_20100111.pdf
6. Lanzante JR, KW Dixon, MJ Nath, CE Whitlock, and D Adams-Smith (2018): Some Pitfalls in Statistical Downscaling of Future Climate. Bulletin of the American Meteorological Society. DOI: 0.1175/BAMS-D-17-0046.1.
7. Livneh B., E.A. Rosenberg, C. Lin, B. Nijssen, V. Mishra, K.M. Andreadis, E.P. Maurer, and D.P. Lettenmaier, 2013: A Long-Term Hydrologically Based Dataset of Land Surface Fluxes and States for the Conterminous United States: Update and Extensions, Journal of Climate, 26, 9384–9392.
8. Pierce, D. W., D. R. Cayan, and B. L. Thrasher, 2014: Statistical downscaling using Localized Constructed Analogs (LOCA). Journal of Hydrometeorology, volume 15, page 2558-2585
9. Stoner, A., K. Hayhoe, and X. Yang (2012), An asynchronous regional regression model for statistical downscaling of daily climate variables, Int. J. Climatol., 33, 2473–2494, doi:10.1002/joc.3603.
10. Wood, A., L. Leung, V. Sridhar, and D. Lettenmaier (2004), Hydrologic implications of dynamical and statistical approaches to downscaling climate model outputs, Clim. Change, 62, 189–216.

License

3D volume rendering of geophysical data using the yt platform

Authors: Christopher Havlin[1,2], Benjamin Holtzman[1], Kacper Kowalik[2], Madicken Munk[2], Sam Walkow[2], Matthew Turk[2]

1. Lamont-Doherty Earth Observatory, Columbia University
2. University of Illinois Urbana-Champagne

1. Abstract

We present novel applications of *yt*, a tool originally designed for analysis of astrophysics datasets, to the geosciences. *yt* (http://yt-project.org) is a python-based platform for volumetric data, which enables semantically meaningful analysis and visualization. As part of a wider effort to bring *yt* to the geosciences, we present an initial use-case of *yt* applied to 3D seismic tomography models of the upper mantle from the IRIS Earth Model Collaboration. While the rendering

capabilities of yt can in general be applied directly to 3D geoscience data, we add several graphical elements to the 3D rendering to aid in interpretation of volume renderings including latitude/longitude overlays and shapefile support for plotting political and tectonic boundaries along the Earth's surface. In this notebook, we focus on tomographic models of North America and the Western U.S., where high resolution models and rapidly varying seismic properties provide a rich dataset to explore systematically at a range of lengthscales. The notebook demonstrates loading and rendering of IRIS netcdf models, highlighting interesting 3D features of the Western U.S. upper mantle, and goes on to demonstrate how having direct control of the transfer functions used in creating the final volume rendering allows for a more systematic exploration of the role of the visualization method in our interpretation of 3D volumetric data. Finally, we conclude by demonstrating some of the semantically-aware capabilities of *yt* for analysis purposes, and demonstrate how these tools have cross-disciplinary functionality.

2. Background

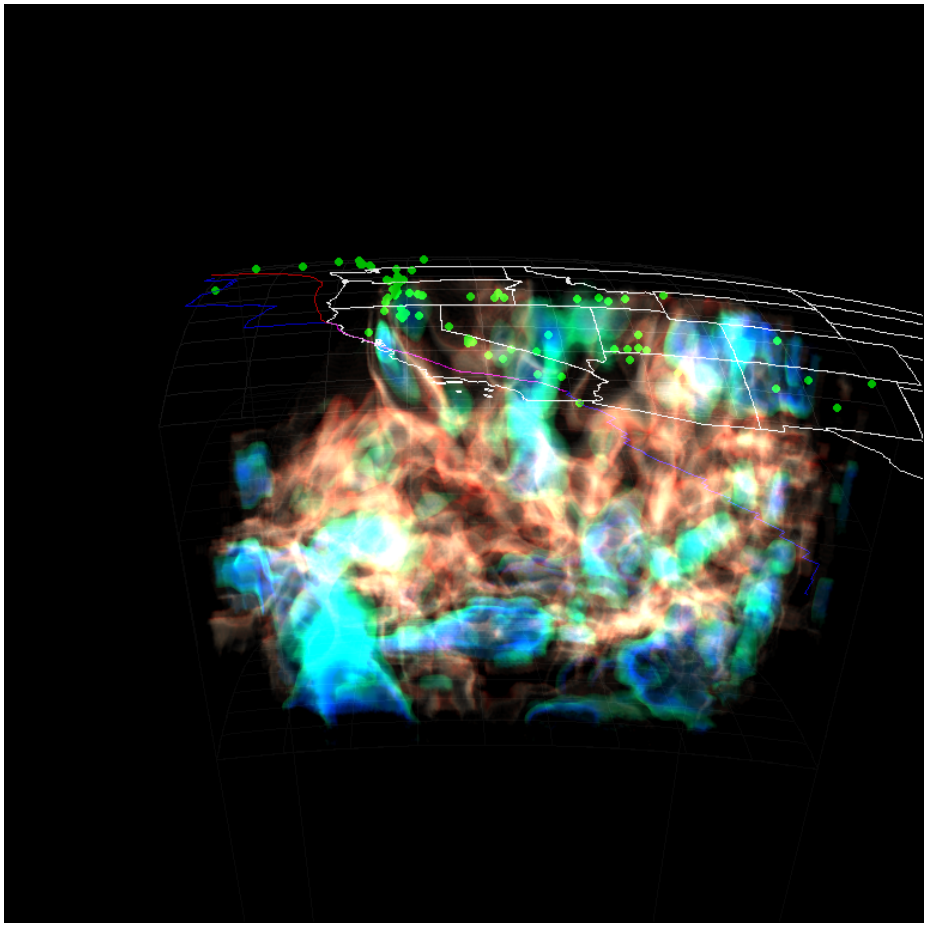
An increasing number of efforts have focused on expanding the geoscience domains in which *yt* is used, including weather simulations, observational astronomy, oceanography and hydrology. Within the realm of upper mantle geophysics, *yt* has been used to visualize seismic wavefronts from forward modeling (Holtzman et al. 2014) and in this work we demonstrate an initial use-case of *yt* in visualizing upper mantle geophysical data. Geophysical data contains rich 3D structure that maps the material properties of the upper mantle, including elastic properties (shear wave and compressional wave velocities, attenuation) and electrical properties (conductivity or resistivity). While 3D structure is inherent to all geophysical datasets, 3D visualization is not commonly used by seismologists and most studies interpret 2D slices through the 3D data. In the occasional studies where 3D visualization is used, authors rely on 3D isosurfaces (e.g., Obreski et al. 2010) and while isosurfaces can reveal 3D structure, the noisy nature of the data results in significantly different emergent structure depending on small deviations in the chosen value for an isosurface. In contrast to rendering isosurfaces, volume rendering allows spatially variable transparency, such that the noise is de-emphasized and the most robust anomalies dominate the image. The physical interpretation of 3D tomographic images should become less dependent on arbitrary choices in the visualization scheme. Volume rendering can be an important tool towards that aim.

In this notebook, we focus on a single tomographic model **NWUS11-S**, which maps shear wave velocity anomalies, \(dV_s\), in the northwest United States (James et al., 2011). Shear wave velocity in the Earth depends on composition and thermodynamic state (e.g., temperature, melt fraction) and to first order, maps of velocity anomalies highlight regions of the mantle that are hotter (slower) or cooler (faster) than expected relative to a 1D reference model. These thermal anomalies result in density and effective viscosity variations that drive mantle convection and plate tectonics, so maps of velocity anomalies provide important insight into many questions in geophysics. The upper mantle beneath the northwest U.S. is an ideal region for 3D visualization because it contains a wide range of regional tectonic processes including subduction of the Pacific plate beneath the Cascades and voluminous volcanic activity for the past ~50 Myrs including the Yellowstone Hot Spot (e.g. Humphreys 1995). And **NWUS11-S** is an ideal tomographic model to visualize in 3D with *yt* because of its relatively high spatial resolution thanks to leveraging of the Earthscope-USArray seismic network. Furthermore, because **NWUS11-S** is one of the models included in the Incorporated Research Institutions for Seismology (IRIS) Earth Model Collaboration (EMC) using the standardized netCDF format for 3-D models (citation), the code developed in the present repository is reasonably general and other models can be easily loaded.

3. Overview of Notebook

The following image shows an example of volume rendering of data from **NWUS11-S** in which slow and fast anomalies are mapped to warm and cool colors, respectively. Annotations along the Earth's surface include tectonic boundary lines (blue: divergent boundaries, red: convergent boundaries and pink: transform boundaries) and eruptions of the past 10,000 years (green circles).

```
import os
from IPython.display import Image
Image(filename = os.path.join('resources','InitVolume.png'))
```



The blue regions near the surface in the Pacific Northwest correspond to the cold, subducting Cascadian plate which drives volcanism within the Cascades. Farther east, James et al. (2011) interpret the fast anomalies beneath Idaho and Utah/Wyoming as remnant slabs from when the entirety of the tectonic border along the western U.S. was an active subduction zone. The “gap” between these two fast anomalies is the present day location of Yellowstone (the green dot on the Idaho/Wyoming border). James et al. (2011) hypothesize that the formation and migration of the Yellowstone hot spot is due to changes in subduction dynamics: the fragmentation and foundering of the slabs drove hot material through the slab gap. The anomalies deeper in the model domain are more enigmatic and the most salient feature is simply the prevalence of slow (hot) mantle beneath the western U.S., consistent with the ongoing volcanism of the past 50 Myrs (e.g., Humphreys 1995).

The above image contains detail that is difficult to achieve with 3D isosurfaces and in the remainder of the notebook, we describe the different components required for generating volume renderings from IRIS EMC files: in sections 4 and 5 demonstrate how to load uniformly gridded geophysical data sets into *yt* and how to set transfer functions for volume rendering to highlight ranges of data, respectively. Section 6 demonstrates how to adjust the orientation of the 3D view to aid interpretation. Together, these sections demonstrate the typical workflow for loading and rendering with *yt*:

1. initialize the *yt* scene: load data volume source and add annotation volume sources
2. set the viewing angle with a camera object
3. set the transfer function (how the data is rendered).
4. render the image

To modify this notebook, whether running a local jupyter server or running the binder-docker image, first run all cells to ensure the required packages are loaded and functions defined.

4. Loading IRIS Earth Model Collaboration (EMC) Files

The IRIS EMC (Trabant et al., 2012, http://ds.iris.edu/ds/products/emc-earthmodels/) is a rich database of geophysical models that serves as an excellent testing ground for using *yt* with geophysical data. As the 3D EMC models are standardized netCDF files, different models can be easily interchanged.

At present, the initialization workflow for loading 3D IRIS EMC models with *yt* consists of (4.1) loading required packages, (4.2) interpolation and (4.3) annotation.

4.1 Loading packages

We begin by importing the libraries and setting required environment variables. In addition to the standard `yt` package, the present notebook relies on a supplementary `yt_velmodel_vis` package (link) that facilitates data loading and transformations for seismic data as described below. Note that the present repository for the 2020 EarthCube Annual Meeting notebook (https://github.com/earthcube2020/ec20_havlin_etal) includes a copy of the package so that future changes to the package will not break the present notebook.

```
# imports and initialization
import os,yt, numpy as np
import matplotlib.pyplot as plt
from yt_velmodel_vis import seis_model as SM,shapeplotter as SP, transferfunctions as
TFs

os.environ['YTVELMODELDIR']='./data' # local repo path to data directory
for axtype in ['axes','xtick','ytick']:
    plt.rc(axtype, labelsize=14)
```

The `yt_velmodel_vis` package uses a simple filesystem database in which raw model files, interpolated model files and shapefiles for annotation are stored in the directory set by the `YTVELMODELDIR` environment variable. For normal usage outside this notebook, after installing `yt_velmodel_vis`, a user can set this environment variable to any directory and run the following from a Python interpreter to initialize the database:

```
from yt_velmodel_vis import datamanager as dm
dm.initializeDB()
```

This will setup the local filesystem database, building the expected file tree structure and fetching some initial data including IRIS models and useful shapefiles. Once files stored within the filesystem database, the user needs to only supply the filename, not the complete path (as long as file names are unique). In the case of the shapefiles, there are also short-names for easier access. If loading a file outside the database, the filenames must be full paths. In the above cell, we set the data directory relative to the location of the present notebook within the github repository, which already contains the data required for the notebook.

4.2 Interpolation

At present, 3D volume rendering in *yt* is restricted to cartesian coordinates, but IRIS EMC files are in geo-spherical coordinates (i.e., latitude, longitude and depth). A working implementation of spherical volume rendering exists (Kellison and Gyurgyik, in prep), but for the current demonstration we must first interpolate from spherical to cartesian coordinates. The `yt_velmodel_vis.seis_model` class contains a KDTree-Inverse Distance Weighting algorithm to populate values on the cartesian grid by finding the nearest N model data and weighting the values by distance from the point on the new cartesian grid.

While the resolution of the cartesian grid, number of nearest neighbors and max distance allowed in the nearest neighbor search are all adjustable parameters, the interpolation from spherical to cartesian coordinates is computationally demanding and so the current notebook loads a pre-built interpolation included in the repository's notebook data directory (`notebooks/data/`) for **NWUS11-S**, defined by the interpolation dictionary (`interp_dict`) in the following code cell.

```
# load interpolated data using the yt uniform grid loader (not editable)

# set the model file and the field to visualize
modelfile='NWUS11-S_percent.nc' # the model netCDF file
datafld='dvs' # the field to visualize, must be a variable in the netCDF file

# set the interpolation dictionary. If the interpolation for this model does
# not exist, SM.netcdf() will build it.
interp_dict={'field':datafld,'max_dist':50000,'res':[10000,10000,10000],
             'input_units':'m','interpChunk':int(1e7)}

# load the model
model=SM.netcdf(modelfile,interp_dict)

# set some objects required for loading in yt
bbox = model.cart['bbox'] # the bounding box of interpolated cartesian grid
data={datafld:model.interp['data'][datafld]} # data container for yt scene

# load the data as a uniform grid, create the 3d scene
ds = yt.load_uniform_grid(data,data[datafld].shape,1.0,bbox=bbox,nprocs=1,
                          periodicity=(True,True,True),unit_system="mks")

print("Data loaded.")
```

```
yt : [INFO     ] 2020-09-09 10:24:18,525 Parameters: current_time              = 0.0
yt : [INFO     ] 2020-09-09 10:24:18,526 Parameters: domain_dimensions         = [287
271 237]
yt : [INFO     ] 2020-09-09 10:24:18,526 Parameters: domain_left_edge          =
[-5201282.   -3445670.25  2740212.5 ]
yt : [INFO     ] 2020-09-09 10:24:18,527 Parameters: domain_right_edge         =
[-2328344.5     -735306.5625  5113796.5   ]
yt : [INFO     ] 2020-09-09 10:24:18,527 Parameters: cosmological_simulation   = 0.0
```

```
Data loaded.
```

It is worth noting that any netCDF file with uniformly spaced data could be loaded using the `yt.load_uniform_grid` loader. In the present application, the `SM.netcdf()` simply wraps the generic python netCDF4 in order to add methods that are useful for seismic tomography model. But, the `data` dictionary can contain any 3D numpy array.

4.3 Loading IRIS EMC Files: Building the scene

The `yt.scene` object contains the information on how and what to render. In addition to specifying which field of the data to render, we can add annotations. While *yt* has methods for adding annotations to a volume rendering, because we want to add annotations reflecting the original spherical domain, we use routines in the `yt_velmodel_vis` package to manually add annotations as line and point sources.

Additionally, as opposed to the astrophysical datasets that *yt* was originally designed for, interpretation of geophysical datasets typically requires correlation along one boundary of the 3D data (the Earth's surface). The surficial expression of plate tectonics is directly related to the material properties of the underlying mantle, and so any 3D visualization of geophysical data requires some level of mapping to the regional tectonics at the Earth's surface. And so the `yt_velmodel_vis.shapeplotter` module contains routines to facilitate plotting shapes at the Earth's surface in the 3D view including latitude and longitude grids and automated parsing and transformation of shapefiles. The `shapeplotter` module leverages the `geopandas` library for automating the reading of shapefiles and includes transformations from geo-spherical coordinates to geocentric cartesian coordinates.

In the present visualizations, we include tectonic boundaries (Coffin et al., 1998), sites of volcanism of the last 10,000 years (Simkin and Siebert, 1994) and US political boundaries (Natural Earth, https://www.naturalearthdata.com/).

```
def build_yt_scene():
    """ builds the yt scene:

    - Draws the spherical chunk bounding the dataset
    - Draws a latitude/longitude grid at surface
    - Draws shapefile data: US political boundaries, tectonic boundaries, volcanos

    """

    # create the scene (loads full dataset into the scene for rendering)
    sc = yt.create_scene(ds,datafld)

    # add useful annotations to the scene in two parts: 1. Domain Annotations and 2.
Shapefile Data

    # 1. Domain Annotations :
    # define the extent of the spherical chunk
    lat_rnge=
[np.min(model.data.variables['latitude']),np.max(model.data.variables['latitude'])]
    lon_rnge=
[np.min(model.data.variables['longitude']),np.max(model.data.variables['longitude'])]
    Depth_Range=[0,1200]
    R=6371.
    r_rnge=[(R-Depth_Range[1])*1000.,(R-Depth_Range[0])*1000.]

    # create a spherical chunk object
    Chunk=SP.sphericalChunk(lat_rnge,lon_rnge,r_rnge)

    # add on desired annotations to the chunk
    sc=Chunk.domainExtent(sc,RGBa=[1.,1.,1.,0.002],n_latlon=100,n_rad=50) # extent of
the domain
    sc=Chunk.latlonGrid(sc,RGBa=[1.,1.,1.,0.005]) # lat/lon grid at the surface
    sc=Chunk.latlonGrid(sc,RGBa=[1.,1.,1.,0.002],radius=(R-410.)*1000.) # lat/lon grid
at 410 km depth
    sc=Chunk.latlonGrid(sc,RGBa=[1.,1.,1.,0.002],radius=(R-Depth_Range[1])*1000.)
#lat/lon grid at lower extent
    sc=Chunk.wholeSphereReference(sc,RGBa=[1.,1.,1.,0.002]) # adds lines from Earth's
center to surface

    # 2. Shapefile Data
    # set the surficial bounding box, used for reading all shapefiles
    shp_bbox=[lon_rnge[0],lat_rnge[0],lon_rnge[1],lat_rnge[1]]

    # US political boundaries
    thisshp=SP.shapedata('us_states',bbox=shp_bbox,radius=R*1000.)
    sc=thisshp.addToScene(sc)

    # tectonic boundaries: buid a dictionary with unique RGBa values for each
    clrs={
        'transform':[0.8,0.,0.8,0.05],
        'ridge':[0.,0.,0.8,0.05],
        'trench':[0.8,0.,0.,0.05],
        'global_volcanos':[0.,0.8,0.,0.05]
    }
    for bound in ['transform','ridge','trench','global_volcanos']:
        tect=SP.shapedata(bound,radius=R*1000.,buildTraces=False)
        sc=tect.buildTraces(RGBa=clrs[bound],sc=sc,bbox=shp_bbox)

    return sc
```

In the preceding code, two types of annotations are added: domain and shapefile annotations.

While *yt* includes methods for annotating meshes and grids of volume renderings, those methods act on the interpolated cartesian grid supplied in the initial call to `yt.load_uniform_grid()`. For example, a call to `sc.annotate_domain()` will add the 3D boundaries of the volume being rendered. But it is more insightful to plot the boundaries of the original spherical dataset, and so the `SP.sphericalChunk()` class includes methods to facilitate annotating the original domain extent in spherical coordinates. Once spherical volume rendering is fully implemented in *yt*, domain annotation will be more straightforward.

For shapefile data, the data from each shapefile is added in two steps:

1. Loading and parsing the shapefile, `SP.shapedata()`
2. Adding the shapefile traces to the *yt* scene `SP.shapedata.addToScene()` or `SP.shapedata.buildTraces()`

When loading the shapefile, the `radius` argument specifies what radius to draw the shapes at (in the above cases, drawing tectonic boundaries at the Earth's surface). The traces are added as a `LineSource` or `PointSource` from the *yt* volume rendering package, `yt.visualization.volume_rendering.api`. If using default colors, the traces are built on initial data load and then added to the scene object (`sc`) with `SP.shapedata.addToScene(sc)`. The user can also specify

the RGBa to use in drawing each shapefile, in which case, `SP.shapedata()` is called with the `buildTraces` argument set to `False` and the traces are added to the scene object as soon as they are built by setting the `sc` argument in the call to `SP.shapedata.buildTraces()`.

Because the shapefiles being plotted are included in the `yt_velmodel_vis` package, they are referred to using their *shortnames* (`'us_states'`, `'transform'`, etc.) rather than full filename. To parse a shapefile that is not included, the user can supply the full path as the first argument rather than the *shortname*.

The final settings required before discussing the volume rendering are the properties controlling camera position and viewing angle. The *yt* documentation provides an overview of camera positioning and related settings (yt Volume Rendering Overview), and so for the present notebook, we simply create a function that adjusts the scene object for the present dataset to provide a reasonable initial viewing angle:

```
def getCenterVec():
    # center vector
    x_c=np.mean(bbox[0])
    y_c=np.mean(bbox[1])
    z_c=np.mean(bbox[2])
    center_vec=np.array([x_c,y_c,z_c])
    return center_vec / np.linalg.norm(center_vec)

def setCamera(sc):
    pos=sc.camera.position

    # center vector
    center_vec=getCenterVec()
    sc.camera.set_position(pos,north_vector=center_vec)

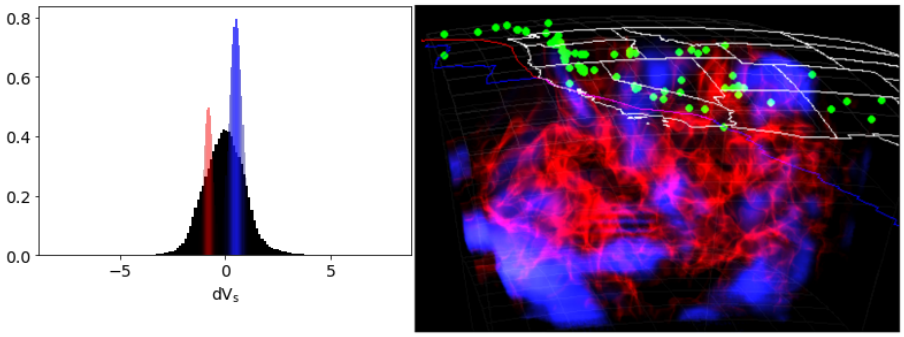
    # zoom in a bit
    zoom_factor=0.7 # < 1 zooms in
    init_width=sc.camera.width
    sc.camera.width = (init_width * zoom_factor)
```

5 Transfer Functions

3D volume rendering in `yt` uses ray-tracing through the 3D volume: rays are passed through the volume sources within a scene to the camera and the final RGB value of each pixel in the image is an integration of the RGBa values along each raypath. The **transfer function** maps the value of the data field at each voxel to an RGBa value, determining how much an individual voxel contributes to the final composite.

The following image contains an example of a transfer function (left) and volume rendering (right):

```
from IPython.display import Image
Image(filename = os.path.join('resources','TFexample.png'))
```



The transfer function plot (left) contains a normalized histogram of the velocity anomaly, \(dV_s\), over the whole domain, in black. The red and blue Guassian distributions represent the transfer function: the y-axis corresponds to the transmission coefficient (0 for transparent, 1 for opaque) and the \(dV_s\) values falling within each respective Gaussian are assigned the corresponding RGB value. The given transfer function results in the volume rendering to the right. The blues correspond to the range of positive velocity anomalies bounded by the blue Gaussian in the transfer function plot, while the reds correspond to the slow velocity anomalies bounded by the red Gaussian.

By modifying the functional form of the transform functions, different ranges of the data can be easily sampled. Furthermore, colormaps can be applied to data ranges to provide even more detail. In the following sections, we demonstrate how to create the above figure, and then how to modify the transfer function in more complex ways.

5.1 Transfer Functions: yt presets

yt provides a variety of preset transfer functions, but first, we define a simple function to pull out the values from a given transfer function to plot on top of a histogram of the data:

```
def plotTf_yt(tf,dvs_min,dvs_max):
    x = np.linspace(dvs_min,dvs_max,tf.nbins) # RGBa value defined for each dvs bin in
range
    y = tf.funcs[3].y # the alpha value of transfer function at each x
    w = np.append(x[1:]-x[:-1], x[-1]-x[-2])
    colors = np.array([tf.funcs[0].y, tf.funcs[1].y, tf.funcs[2].y,
                       tf.funcs[3].y]).T
    fig = plt.figure()
    ax = fig.add_axes([0.2, 0.2, 0.75, 0.75])
    d_hist=ax.hist(data['dvs']
[~np.isnan(data['dvs'])].ravel(),bins=100,density=True,log=False,color='k')
    ax.bar(x, tf.funcs[3].y, w, edgecolor=[0.0, 0.0, 0.0, 0.0],
           log=False, color=colors, bottom=[0])
    plt.xlabel('$\mathregular{dV_s}$')
    plt.show()
```

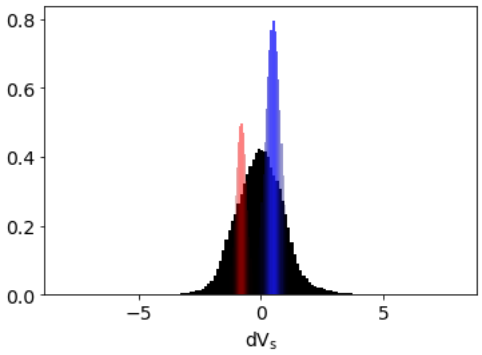
One of the simplest *yt* transfer function methods is to add a Gaussian to highlight a narrow range of the data. The user initializes a transfer function, and then adds a Gaussian described by the center of the peak, the peak width and the RGBa value of the center point:

```
# initialize the tf object by setting the data bounds to consider
dvs_min=-8
dvs_max=8
tf = yt.ColorTransferFunction((dvs_min,dvs_max))

# set gaussians to add
TF_gaussians=[
    {'center':-.8,'width':.1,'RGBa':(1.,0.,0.,.5)},
    {'center':.5,'width':.2,'RGBa':(0.1,0.1,1.,.8)}
]

for gau in TF_gaussians:
    tf.add_gaussian(gau['center'],gau['width'],gau['RGBa'])

# plot the transfer function
plotTf_yt(tf,dvs_min,dvs_max)
```



Once the transfer function is set, the volume rendering can be created. Because we will explore different transfer functions below, we first define a function to build the yt scene object, using functions defined in the previous section.

```
def configure_scene(the_transfer_function,res_factor=1.):
    # build scene, apply camera settings, set the transfer function
    sc = build_yt_scene()
    setCamera(sc)
    source = sc.sources['source_00']
    source.set_transfer_function(the_transfer_function)

    # adjust resolution of rendering
    res=sc.camera.get_resolution()
    new_res=(int(res[0]*res_factor),int(res[1]*res_factor))
    sc.camera.set_resolution(new_res)

    print("Scene ready to render")
    return sc
```

Given a transfer function, we can then run all the steps required before final rendering with:

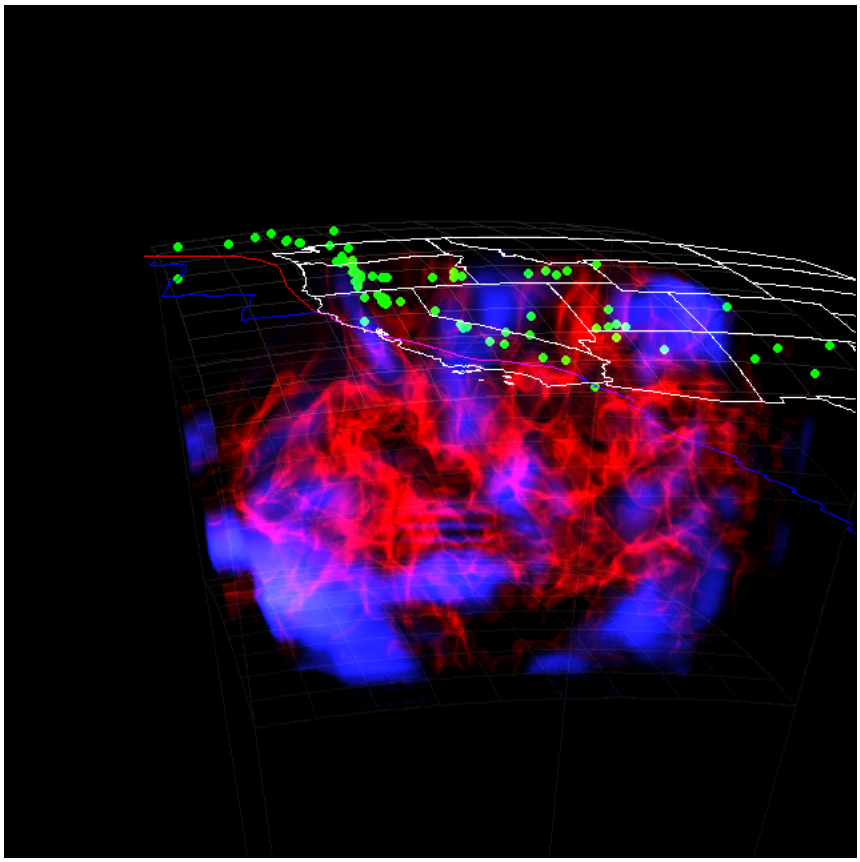
```
sc = configure_scene(tf,res_factor=1.25)
```

```
Scene ready to render
```

The scene is now ready to render, which is done by calling `sc.show()`. Note that the initial rendering may take a few minutes when running the notebook from the Binder image.

```
sc.show(sigma_clip=1.5)
```

```
yt : [INFO     ] 2020-09-09 10:24:47,471 Rendering scene (Can take a while).
yt : [INFO     ] 2020-09-09 10:24:47,606 Creating volume
/home/chavlin/miniconda3/envs/yt_vis/lib/python3.7/site-
packages/yt/units/yt_array.py:1373: RuntimeWarning: invalid value encountered in
log10
  out_arr = func(np.asarray(inp), out=out, **kwargs)
```



The argument `sigma_clip=1.5` controls the contrast of the final image, which is useful for fine tuning the final brightness of the image (e.g., Sigma Clip).

5.2 Transfer Functions: Customization

The transfer function object is a 4D numpy array of RGBa values, and so it is straightforward to create custom transfer functions for a particular application. Here, we demonstrate a more complex transfer function built to clearly differentiate slow and fast anomalies over a wider range of data than the Gaussian example. This example uses transfer function routines from the `yt_velmodel_vis.transfer_functions` module and so we define a new plotting function before building the transfer function:

```
def plotTf(tfOb):
    """ create a histogram-transfer function plot and display it"""
    f=plt.figure()
    ax=plt.axes()
    ax=tfOb.addHist(ax=ax,density=True,color=(0.,0.,0.,1.))
    ax=tfOb.addTFtoPlot(ax=ax)
    ax.set_xlabel('$\mathregular{dV_s}$')
    plt.show()
```

The `yt_velmodel_vis.transfer_functions` module allows different data ranges to use different colormaps to draw out contrast details. Any of the colormaps available in *yt* may be selected (see this link for how to view available colormaps).

In this example, we divide the transfer function into two separate segments for slow and fast anomalies and within each segment, the transmission coefficient varies linearly:

$\alpha(dV_s) = \alpha_o - \frac{\Delta{\alpha}}{\Delta{dV_s}}(dV_s-dV_s^o)$

where the α_o and change in α over the segment, $\Delta{\alpha}$ are constants that vary between segments, $\Delta{dV_s}$ is the range of each segment and dV_s^o is the min dV_s value within the segment.

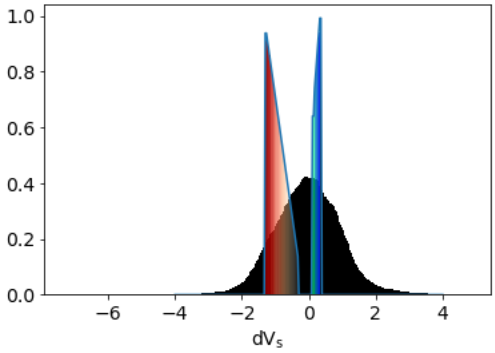
In our implementation below, the transmission coefficient varies inversely with the distribution of observations: having a lower transmission coefficient where there is more data prevents the more prevalent values from overwhelming the image and obscuring structure.

```
# setting up transfer functions
tfOb = TFs.dv(data[datafld].ravel(),bounds=[-4,4])

# segment 1, slow anomalies
bnds=[-1.3,-.3]
TFseg=TFs.TFsegment(tfOb,bounds=bnds,cmap='OrRd_r')
alpha_o=0.95
Dalpha=-0.85
alpha=alpha_o + Dalpha/(bnds[1]-bnds[0]) * (TFseg.dvbins_c-bnds[0])
tfOb.addTFsegment(alpha,TFseg)

# segment 2, fast anomalies
bnds=[.1,.35]
TFseg=TFs.TFsegment(tfOb,bounds=bnds,cmap='winter_r')
alpha_o=.6
Dalpha=.4
alpha=alpha_o+ Dalpha/(bnds[1]-bnds[0]) * (TFseg.dvbins_c-bnds[0])
tfOb.addTFsegment(alpha,TFseg)

plotTf(tfOb)
print("Ready to build scene")
```




```
Ready to build scene
```

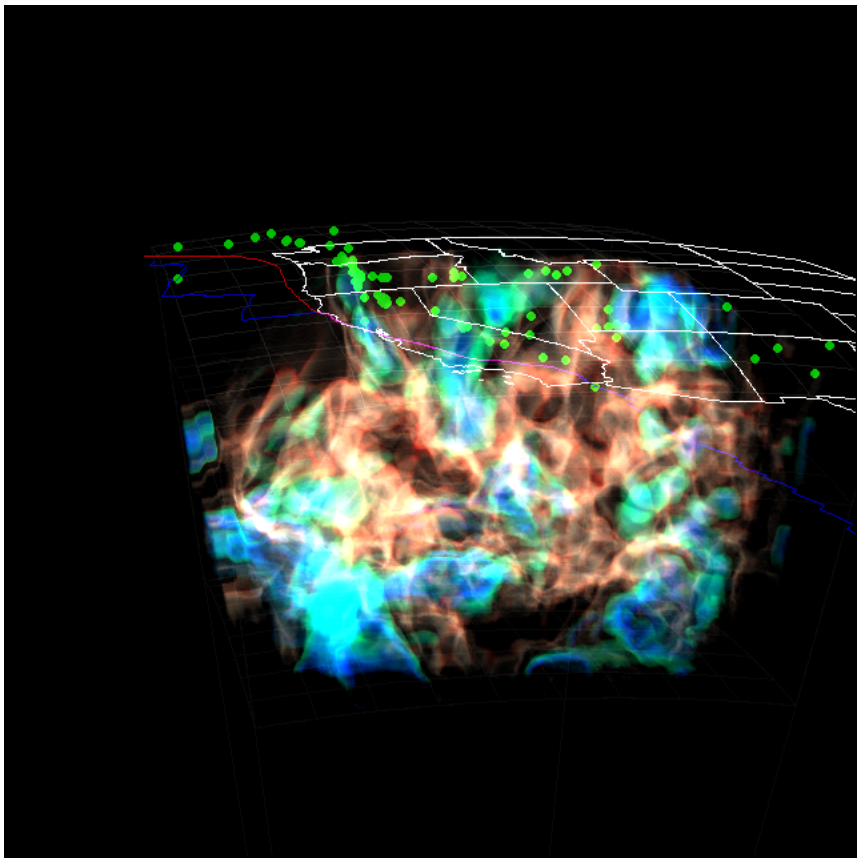
And once again, we build the scene and render:

```
# build scene, apply camera settings, set the transfer function
sc = configure_scene(tfOb.tf,res_factor=1.25)
```

```
Scene ready to render
```

```
sc.show(sigma_clip=1.5) # render the scene
```

```
yt : [INFO     ] 2020-09-09 10:25:10,399 Rendering scene (Can take a while).
yt : [INFO     ] 2020-09-09 10:25:10,627 Creating volume
```



While the same features are visible in both volume renderings above, the second one captures more variability, particularly within the fast anomalies.

6. Changing Perspective

To fully grasp the structure visible in 3D renderings, it is necessary to view the volumes from different angles. There are a number of ways to vary 3D orientation in *yt* by modifying the `camera` object in a scene (link to *yt* documentation) and we briefly demonstrate how to change the camera position and how to rotate the camera about a given axis.

Given that we are working in cartesian coordinates, the simplest way to set a new camera position is to modify the initial position. In the following cell, we first extract the position of the camera, a 3-element array, and then modify the first dimension by a factor (1.1 in this case), which moves the camera to a lower angle view relative to the center of the domain.

For the rotation, we rotate by -15 degrees around the vector that points through the center of the domain. The resulting perspective is roughly perpendicular to the Cascadian volcanic front and subducting slab.

```
sc = configure_scene(tfOb.tf,res_factor=1.25)

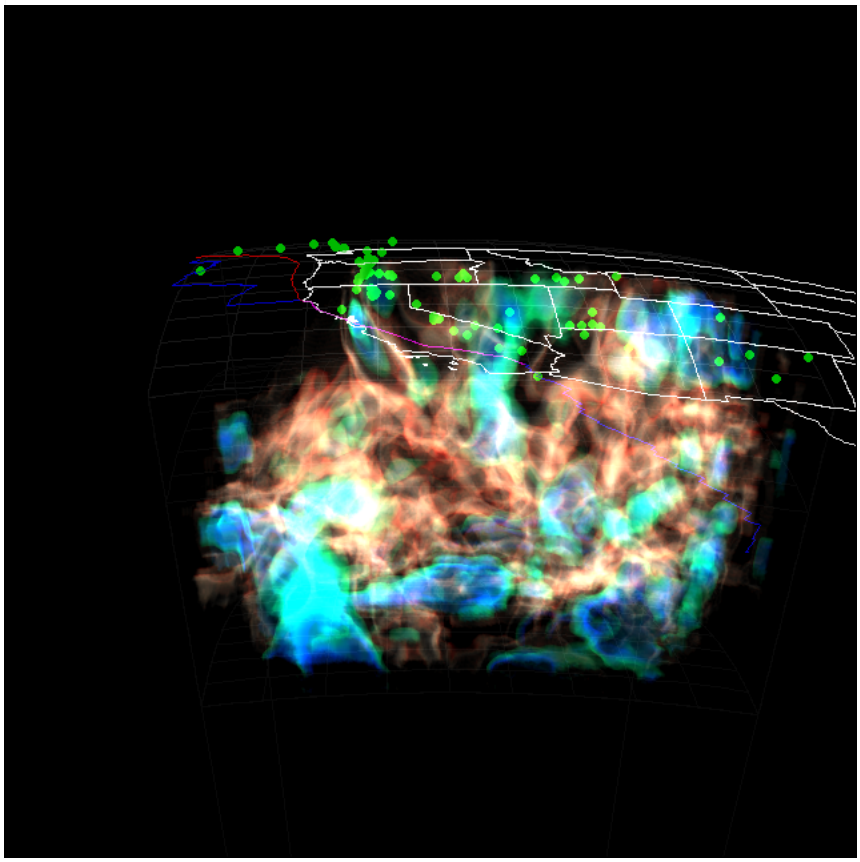
# adjust the camera position to a lower angle view
pos=sc.camera.get_position() # pos is a 3 element array
pos[0]=pos[0]*1.1
center_vec=getCenterVec()
sc.camera.set_position(pos,north_vector=center_vec)

# rotate by -15 degrees
sc.camera.rotate(-15*np.pi/180., rot_vector=center_vec)
sc.show(sigma_clip=1.5) # render the scene
```

```
yt : [INFO     ] 2020-09-09 10:25:28,232 Rendering scene (Can take a while).
```

```
Scene ready to render
```

```
yt : [INFO     ] 2020-09-09 10:25:28,552 Creating volume
```



By iteratively varying the position and rotation parameters, we can easily produce a serires of frames that can be stitched together into a video. The script NWUS11S_animation.py produces a video that lowers this viewing perspective, zooms and rotates the volume.

The following cell shows the resulting video. Note that the video may not display in static notebook viewers such as nbviewer.jupyter.org, in which case the video may be viewed externally at https://youtu.be/GEvu1PRNDjE or downloaded directly from here.

```
from IPython.core.display import Video
Video(os.path.join('resources','NWUS11-S-rotation.mp4'))
```

0:00

7. Future Work

The present notebook demonstrates an initial use-case of *yt* for volume rendering of seismic tomography. We have focused on laying out the technical steps required to produce volume renderings of IRIS Earth Model Collaboration netCDF files, which lays the groundwork for more interpretive work. In future work, we plan to more systematically investigate the influence of color map choices (e.g., Zeller and Rogers 2020) and transfer functions on our interpretation of geophysical data.

Additionally, a number of planned developments will simplify the process of using *yt* for tomography renderings and aid in interpreting seismic structure. These developments generally fall into categories of improving functionality of the `yt_velmodel_vis` package and improving the seismological context:

Improving functionality

- Further automation of loading IRIS netCDF files, including:
  - automatic import of all available field variables.
  - automatic calculation of velocity perturbations for a range of 1-D reference models when files include only absolute velocities.
- Support for loading tomographic models in other formats.
- Direct spherical volume rendering using recent implementation in *yt* (Kellison and Gyurgyik, in prep), removing the need for interpolation to cartesian coordinates.
- Expanding the *yt* units-awareness to include seismological fields.
- Support for composite model renderings to allow loading of regional studies covering different lengthscales.

Improving geophysical context

- overlaying earthquake epicenters.
- overlaying 3d reference ray paths calculated using open source seismological libraries (e.g., ObsPy http://obspy.org).

References

- Coffin, M.F., Gahagan, L.M., and Lawver, L.A., 1998. “Present-day Plate Boundary Digital Data Compilation.” University of Texas Institute for Geophysics Technical Report No. 174, pp. 5., http://www-udc.ig.utexas.edu/external/plates/data.htm
- Holtzman B., Candler J., Turk M., Peter D., 2014. “Seismic Sound Lab: Sights, Sounds and Perception of the Earth as an Acoustic Space.” In: Aramaki M., Derrien O., Kronland-Martinet R., Ystad S. (eds) Sound, Music, and Motion. CMMR 2013. Lecture Notes in Computer Science, vol 8905. Springer, Cham.
- IRIS EMC: http://ds.iris.edu/ds/products/emc-earthmodels/
- Humphreys, E. D. 1995. “Post-Laramide removal of the Farallon slab, western United States.” Geology, 23(11), 987-990.
- James D.E., M.J. Fouch, R.W. Carlson and J.B. Roth. 2011. “Slab fragmentation, edge flow and the origin of the Yellowstone hotspot track.” Earth Planet. Sci. Lett., https://doi.org/10.1016/j.epsl.2011.09.007. Data retrieved from http://ds.iris.edu/ds/products/emc-nwus11-s/
- Natural Earth, https://www.naturalearthdata.com/
- Obrebski, M., R.M. Allen, M. Xue, and Shu-Huei Hung. 2010. “Slab-Plume Interaction beneath the Pacific Northwest.” Geophys. Res. Lett. 37:L14305. https://doi.org/10.1029/2010GL043489.
- Simkin and Siebert, 1994. “Volcanoes of the world: An illustrated catalog of Holocene volcanoes and their eruptions.” https://earthworks.stanford.edu/catalog/harvard-glb-volc
- Trabant, C., A. R. Hutko, M. Bahavar, R. Karstens, T. Ahern, and R. Aster, 2012). “Data Products at the IRIS DMC: Stepping Stones for Research and Other Applications.” Seismological Research Letters, 83(5), 846–854, https://doi.org/10.1785/0220120032.
- *yt*: https://yt-project.org
- `yt_velmodel_vis`: https://github.com/chrishavlin/yt_velmodel_vis
- Zeller and Rogers, 2020, “Visualizing Science: How Color Determines What We See,” https://eos.org/features/visualizing-science-how-color-determines-what-we-see

Support
This work is funded in part by:

- PIs: M. Turk (Univ. of Illinois), B. K. Holtzman, and L. Orf (Univ. of Wisconsin), “Inquiry-Focused Volumetric Data Analysis Across Scientific Domains: Sustaining and Expanding the yt Community,” NSF Software Infrastructure for Sustained Innovation.
- PI: B. Holtzman, C. Havlin (LDEO), “Mapping variability in the thermo-mechanical structure of the North American Plate and upper mantle,” NSF Earthscope.

Intake / Pangeo Catalog: Making It Easier To Consume Earth’s Climate and Weather Data

Anderson Banihirwe (abanihi@ucar.edu), Charles Blackmon-Luca (blackmon@ldeo.columbia.edu), Ryan Abernathey (rpa@ldeo.columbia.edu), Joseph Hamman (jhamman@ucar.edu)

- NCAR, Boulder, CO, USA
- Columbia University, Palisades, NY, USA

2020 EarthCube Annual Meeting ID: 133

Introduction

Computer simulations of the Earth's climate and weather generate huge amounts of data. These data are often persisted on high-performance computing (HPC) systems or in the cloud across multiple data assets in a variety of formats (netCDF, Zarr, etc.). Finding, investigating, and loading these data assets into compute-ready data containers costs time and effort. The user should know what data are available and their associated metadata, preferably before loading a specific data asset and analyzing it.

In this notebook, we demonstrate intake-esm, a Python package and an intake plugin with aims of facilitating:

- the discovery of earth's climate and weather datasets.
- the ingestion of these datasets into xarray dataset containers.

The common/popular starting point for finding and investigating large datasets is with a data catalog. A *data catalog* is a collection of metadata, combined with search tools, that helps data analysts and other users to find the data they need. For a user to take full advantage of intake-esm, they must point it to an *Earth System Model (ESM) data catalog*. This is a JSON-formatted file that conforms to the ESM collection specification.

ESM Collection Specification

The ESM collection specification provides a machine-readable format for describing a wide range of climate and weather datasets, with a goal of making it easier to index and discover climate and weather data assets. An asset is any netCDF/HDF file or Zarr store that contains relevant data.

An ESM data catalog serves as an inventory of available data, and provides information to explore the existing data assets. Additionally, an ESM catalog can contain information on how to aggregate compatible groups of data assets into singular xarray datasets.

Use Case: CMIP6 hosted on Google Cloud

The Coupled Model Intercomparison Project (CMIP) is an international collaborative effort to improve the knowledge about climate change and its impacts on the Earth System and on our society. CMIP began in 1995, and today we are in its sixth phase (CMIP6). The CMIP6 data archive consists of data models created across approximately 30 working groups and 1,000 researchers investigating the urgent environmental problem of climate change, and will provide a wealth of information for the next Assessment Report (AR6) of the Intergovernmental Panel on Climate Change (IPCC).

Last year, Pangeo partnered with Google Cloud to bring CMIP6 climate data to Google Cloud's Public Datasets program. You can read more about this process here. For the remainder of this section, we will demonstrate intake-esm's features using the ESM data catalog for the CMIP6 data stored on Google Cloud Storage. This catalog resides in a dedicated CMIP6 bucket.

Loading an ESM data catalog

To load an ESM data catalog with intake-esm, the user must provide a valid ESM data catalog as input:

```
import warnings
warnings.filterwarnings("ignore")

import intake

col = intake.open_esm_datastore('https://storage.googleapis.com/cmip6/pangeo-cmip6.json')
col
```

pangeo-cmip6 catalog with 3936 dataset(s) from 269868 asset(s):

	unique
activity_id	15
institution_id	33
source_id	73
experiment_id	103
member_id	169
table_id	29
variable_id	370
grid_label	10
zstore	269868
dcpp_init_year	60

The summary above tells us that this catalog contains over 268,000 data assets. We can get more information on the individual data assets contained in the catalog by calling the underlying dataframe created when it is initialized:

```
col.df.head()
```

	activity_id	institution_id	source_id	experiment_id	member_id	table_id	variable_i
0	AerChemMIP	AS-RCEC	TaiESM1	histSST	r1i1p1f1	AERmon	od550a
1	AerChemMIP	BCC	BCC-ESM1	histSST	r1i1p1f1	AERmon	mmrb
2	AerChemMIP	BCC	BCC-ESM1	histSST	r1i1p1f1	AERmon	mmrdu
3	AerChemMIP	BCC	BCC-ESM1	histSST	r1i1p1f1	AERmon	mmrc
4	AerChemMIP	BCC	BCC-ESM1	histSST	r1i1p1f1	AERmon	mmrsc

The first data asset listed in the catalog contains:

- the ambient aerosol optical thickness at 550nm (`variable_id='od550aer'`), as a function of latitude, longitude, time,
- in an individual climate model experiment with the Taiwan Earth System Model 1.0 model (`source_id='TaiESM1'`),
- forced by the *Historical transient with SSTs prescribed from historical* experiment (`experiment_id='histSST'`),
- developed by the Taiwan Research Center for Environmental Changes (`instution_id='AS-RCEC'`),
- run as part of the Aerosols and Chemistry Model Intercomparison Project (`activity_id='AerChemMIP'`)

And is located in Google Cloud Storage at `gs://cmip6/AerChemMIP/AS-RCEC/TaiESM1/histSST/r1i1p1f1/AERmon/od550aer/gn/`.

Note: the amount of details provided in the catalog is determined by the data provider who builds the catalog.

Searching for datasets

After exploring the CMIP6 controlled vocabulary, it's straightforward to get the data assets you want using intake-esm's `search()` method. In the example below, we are are going to search for the following:

- variables: `tas` which stands for near-surface air temperature
- experiments: `['historical', 'ssp245', 'ssp585']`:
  - `historical`: all forcing of the recent past.
  - `ssp245`: update of RCP4.5 based on SSP2.
  - `ssp585`: emission-driven RCP8.5 based on SSP5.
- table_id: `Amon` which stands for Monthly atmospheric data.
- grid_label: `gr` which stands for regridded data reported on the data provider's preferred target grid.

For more details on the CMIP6 vocabulary, please check this website.

```
# form query dictionary
query = dict(experiment_id=['historical', 'ssp245', 'ssp585'],
             table_id='Amon',
             variable_id=['tas'],
             member_id = 'r1i1p1f1',
             grid_label='gr')

# subset catalog and get some metrics grouped by 'source_id'
col_subset = col.search(require_all_on=['source_id'], **query)
col_subset.df.groupby('source_id')[['experiment_id', 'variable_id',
'table_id']].nunique()
```

	experiment_id	variable_id	table_id
source_id			
CIESM	3	1	1
EC-Earth3	3	1	1
EC-Earth3-Veg	3	1	1
FGOALS-f3-L	3	1	1
IPSL-CM6A-LR	3	1	1
KACE-1-0-G	3	1	1

Loading datasets

Once you've identified data assets of interest, you can load them into xarray dataset containers using the `to_dataset_dict()` method. Invoking this method yields a Python dictionary of high-level aggregated xarray datasets. The logic for merging/concatenating the query results into higher level xarray datasets is provided in the input JSON file, under `aggregation_control`:

```
"aggregation_control": {
  "variable_column_name": "variable_id",
  "groupby_attrs": [
    "activity_id",
    "institution_id",
    "source_id",
    "experiment_id",
    "table_id",
    "grid_label"
  ],
  "aggregations": [{
      "type": "union",
      "attribute_name": "variable_id"
    },

    {
      "type": "join_new",
      "attribute_name": "member_id",
      "options": {
        "coords": "minimal",
        "compat": "override"
      }
    },
    {
      "type": "join_new",
      "attribute_name": "dcpp_init_year",
      "options": {
        "coords": "minimal",
        "compat": "override"
      }
    }
  ]
}
```

Though these aggregation specifications are sufficient to merge individual data assets into xarray datasets, sometimes additional arguments must be provided depending on the format of the data assets. For example, Zarr-based assets can be loaded with the option `consolidated=True`, which relies on a consolidated metadata file to describe the assets with minimal data egress.

```
dsets = col_subset.to_dataset_dict(zarr_kwargs={'consolidated': True}, storage_options=
{'token': 'anon'})

# list all merged datasets
[key for key in dsets.keys()]
```

```
--> The keys in the returned dictionary of datasets are constructed as follows:
        'activity_id.institution_id.source_id.experiment_id.table_id.grid_label'
```

100.00% [18/18 00:01<00:00]

```
['ScenarioMIP.CAS.FGOALS-f3-L.ssp245.Amon.gr',
 'ScenarioMIP.IPSL.IPSL-CM6A-LR.ssp245.Amon.gr',
 'ScenarioMIP.EC-Earth-Consortium.EC-Earth3.ssp585.Amon.gr',
 'ScenarioMIP.IPSL.IPSL-CM6A-LR.ssp585.Amon.gr',
 'CMIP.IPSL.IPSL-CM6A-LR.historical.Amon.gr',
 'CMIP.EC-Earth-Consortium.EC-Earth3.historical.Amon.gr',
 'ScenarioMIP.NIMS-KMA.KACE-1-0-G.ssp245.Amon.gr',
 'ScenarioMIP.EC-Earth-Consortium.EC-Earth3-Veg.ssp245.Amon.gr',
 'ScenarioMIP.THU.CIESM.ssp585.Amon.gr',
 'CMIP.EC-Earth-Consortium.EC-Earth3-Veg.historical.Amon.gr',
 'ScenarioMIP.THU.CIESM.ssp245.Amon.gr',
 'ScenarioMIP.EC-Earth-Consortium.EC-Earth3.ssp245.Amon.gr',
 'ScenarioMIP.EC-Earth-Consortium.EC-Earth3-Veg.ssp585.Amon.gr',
 'CMIP.NIMS-KMA.KACE-1-0-G.historical.Amon.gr',
 'ScenarioMIP.NIMS-KMA.KACE-1-0-G.ssp585.Amon.gr',
 'ScenarioMIP.CAS.FGOALS-f3-L.ssp585.Amon.gr',
 'CMIP.CAS.FGOALS-f3-L.historical.Amon.gr',
 'CMIP.THU.CIESM.historical.Amon.gr']
```

When the datasets have finished loading, we can extract any of them like we would a value in a Python dictionary:

```
ds = dsets['ScenarioMIP.THU.CIESM.ssp585.Amon.gr']
ds
```

xarray.Dataset

▸ Dimensions: (bnds: 2, **lat**: 192, **lon**: 288, **member_id**: 1, **time**: 1032)

▾ Coordinates:

lon_bnds	(lon, bnds)	float64	dask.array<chunksize=(288, 2), me…
lat_bnds	(lat, bnds)	float64	dask.array<chunksize=(192, 2), me…
time_bnds	(time, bnds)	object	dask.array<chunksize=(1032, 2), m…
height	()	float64	...
lon	(lon)	float64	0.0 1.25 2.5 ... 356.2 357.5 358.8
time	(time)	object	2015-01-16 12:00:00 ... 2100-12-1…
lat	(lat)	float64	-90.0 -89.06 -88.12 ... 89.06 90.0
member_id	(member_id)	<U8	'r1i1p1f1'

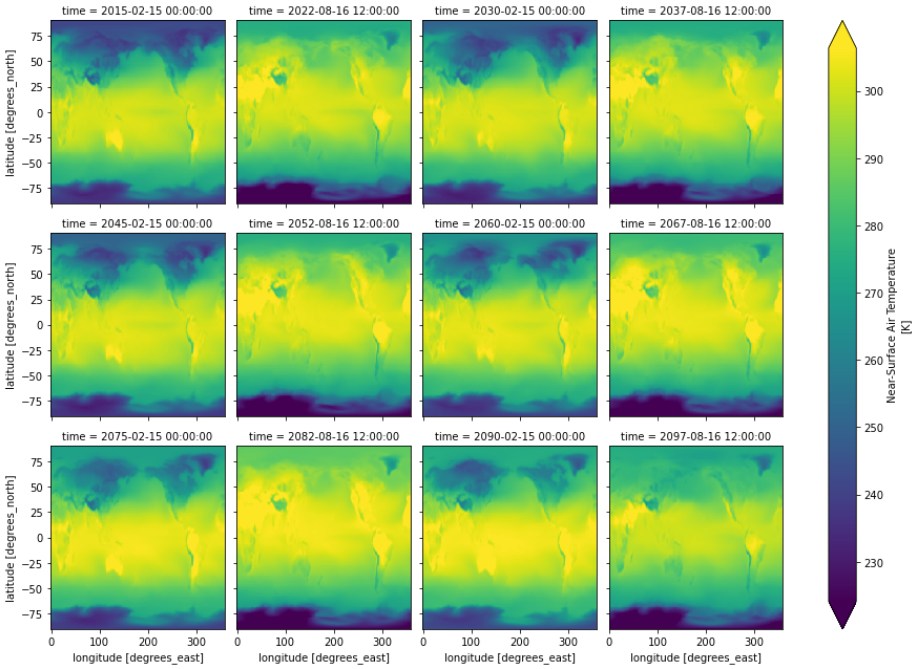
▾ Data variables:

tas	(member_id, time, lat, lon)	float32	dask.array<chunksize=(1, 303, 192…

▸ Attributes: (49)

```
# Let's create a quick plot for a slice of the data:
ds.tas.isel(time=range(1, 1000, 90))\
      .plot(col="time", col_wrap=4, robust=True)
```

```
<xarray.plot.facetgrid.FacetGrid at 0x7f3f315ca1f0>
```



Pangeo Catalog

Pangeo Catalog is an open-source project to enumerate and organize cloud-optimized climate data stored across a variety of providers. In addition to offering various useful climate datasets in a consolidated location, the project also serves as a means of accessing public ESM data catalogs.

Accessing catalogs using Python

At the core of the project is a GitHub repository containing several static intake catalogs in the form of YAML files. Thanks to plugins like intake-esm and intake-xarray, these catalogs can contain links to ESM data catalogs or data assets that can be loaded into xarray datasets, along with the arguments required to load them.

By editing these files using Git-based version control, anyone is free to contribute a dataset supported by the available intake plugins. Users can then browse these catalogs by providing their associated URL as input into intake's `open_catalog()`; their tree-like structure allows a user to explore their entirety by simply opening the root catalog and recursively walking through it:

```
cat = intake.open_catalog('https://raw.githubusercontent.com/pangeo-data/pangeo-datastore/master/intake-catalogs/master.yaml')
entries = cat.walk(depth=5)

[key for key in entries.keys()]
```

```
['ocean.sea_surface_height',
 'ocean.cesm_mom6_example',
 'ocean.ECCOv4r3',
 'ocean.SOSE',
 'ocean.GODAS',
 'ocean.ECCO_layers',
 'ocean.altimetry.al',
 'ocean.altimetry.alg',
 'ocean.altimetry.c2',
 'ocean.altimetry.e1',
 'ocean.altimetry.e1g',
 'ocean.altimetry.e2',
 'ocean.altimetry.en',
 'ocean.altimetry.enn',
 'ocean.altimetry.g2',
 'ocean.altimetry.h2',
 'ocean.altimetry.j1',
 'ocean.altimetry.j1g',
 'ocean.altimetry.j1n',
 'ocean.altimetry.j2',
 'ocean.altimetry.j2g',
 'ocean.altimetry.j2n',
```

```
 'ocean.altimetry.j3',
 'ocean.altimetry.s3a',
 'ocean.altimetry.s3b',
 'ocean.altimetry.tp',
 'ocean.altimetry.tpn',
 'ocean.altimetry',
 'ocean.LLC4320.LLC4320_grid',
 'ocean.LLC4320.LLC4320_SST',
 'ocean.LLC4320.LLC4320_SSS',
 'ocean.LLC4320.LLC4320_SSH',
 'ocean.LLC4320.LLC4320_SSU',
 'ocean.LLC4320.LLC4320_SSV',
 'ocean.LLC4320',
 'ocean.GFDL_CM2_6.GFDL_CM2_6_control_ocean',
 'ocean.GFDL_CM2_6.GFDL_CM2_6_control_ocean_surface',
 'ocean.GFDL_CM2_6.GFDL_CM2_6_control_ocean_3D',
 'ocean.GFDL_CM2_6.GFDL_CM2_6_control_ocean_transport',
 'ocean.GFDL_CM2_6.GFDL_CM2_6_control_ocean_boundary_flux',
 'ocean.GFDL_CM2_6.GFDL_CM2_6_control_ocean_budgets',
 'ocean.GFDL_CM2_6.GFDL_CM2_6_one_percent_ocean',
 'ocean.GFDL_CM2_6.GFDL_CM2_6_one_percent_ocean_surface',
 'ocean.GFDL_CM2_6.GFDL_CM2_6_one_percent_ocean_3D',
 'ocean.GFDL_CM2_6.GFDL_CM2_6_one_percent_ocean_transport',
 'ocean.GFDL_CM2_6.GFDL_CM2_6_one_percent_ocean_boundary_flux',
 'ocean.GFDL_CM2_6.GFDL_CM2_6_one_percent_ocean_budgets',
 'ocean.GFDL_CM2_6.GFDL_CM2_6_grid',
 'ocean.GFDL_CM2_6',
 'ocean.CESM_POP.CESM_POP_hires_control',
 'ocean.CESM_POP.CESM_POP_hires_RCP8_5',
 'ocean.CESM_POP',
 'ocean.channel.MITgcm_channel_flatbottom_02km_run01_phys-mon',
 'ocean.channel.MITgcm_channel_flatbottom_02km_run01_phys_snap15D',
 'ocean.channel.channel_ridge_resolutions_01km',
 'ocean.channel.channel_ridge_resolutions_05km',
 'ocean.channel.channel_ridge_resolutions_20km',
 'ocean.channel.run_tracers_restored_zarr',
 'ocean.channel',
 'ocean.MEOM_NEMO.NATL60_coord',
 'ocean.MEOM_NEMO.NATL60_horizontal_grid',
 'ocean.MEOM_NEMO.NATL60_vertical_grid',
 'ocean.MEOM_NEMO.NATL60_SSH',
 'ocean.MEOM_NEMO.NATL60_SSH_1',
 'ocean.MEOM_NEMO.NATL60_SSU',
 'ocean.MEOM_NEMO.NATL60_SSV',
 'ocean.MEOM_NEMO.eNATL60_grid',
 'ocean.MEOM_NEMO.eNATL60_BLBT02_SSH',
 'ocean.MEOM_NEMO.eNATL60_BLBT02_SSU',
 'ocean.MEOM_NEMO.eNATL60_BLBT02_SSV',
 'ocean.MEOM_NEMO.eNATL60_BLB002_SSU',
 'ocean.MEOM_NEMO.eNATL60_BLB002_SSV',
 'ocean.MEOM_NEMO',
 'ocean',
 'atmosphere.gmet_v1',
 'atmosphere.trmm_3b42rt',
 'atmosphere.sam_ngaqua_qobs_eqx_3d',
 'atmosphere.sam_ngaqua_qobs_eqx_2d',
 'atmosphere.gpcp_cdr_daily_v1_3',
 'atmosphere.wrf50_erai',
 'atmosphere.era5_hourly_reanalysis_single_levels_sa',
 'atmosphere',
 'climate.cmip6_gcs',
 'climate.GFDL_CM2_6',
 'climate.tracmip',
 'climate',
 'hydro.cgiar_pet',
 'hydro.hydrosheds_dir',
 'hydro.hydrosheds_acc',
 'hydro.hydrosheds_dem',
 'hydro.soil_grids_single_level',
 'hydro.soil_grids_multi_level',
 'hydro.camels.basin_mean_forcing_gcp',
 'hydro.camels.basin_mean_forcing_aws',
 'hydro.camels.usgs_streamflow_gcp',
 'hydro.camels.usgs_streamflow_aws',
 'hydro.camels.attributes_aws',
 'hydro.camels',
 'hydro']
```

The catalogs can also be explored using intake's own `search()` method:

```
cat_subset = cat.search('cmip6')

list(cat_subset)
```

```
['climate.cmip6_gcs', 'climate.GFDL_CM2_6', 'climate.tracmip', 'climate']
```

Once we have found a dataset or collection we want to explore, we can do so without the need of any user inputted argument:

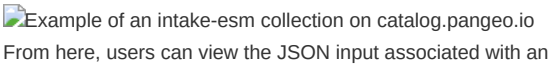
```
cat.climate.tracmip()
```

pangeo-tracmip catalog with 187 dataset(s) from 7067 asset(s):

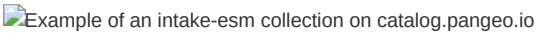
	unique
frequency	3
experiment	11
model	14
variable	47
version	10
source	7067

Accessing catalogs using catalog.pangeo.io

For those who don't want to initialize a Python environmemt to explore the catalogs, catalog.pangeo.io offers a means of viewing them from a standalone web application. The website directly mirrors the catalogs in the GitHub repository, with previews of each dataset or collection loaded on the fly:



From here, users can view the JSON input associated with an ESM collection and sort/subset its contents:



Conclusion

With intake-esm, much of the toil associated with discovering, loading, and consolidating data assets can be eliminated. In addition to making computations on huge datasets more accessible to the scientific community, the package also promotes reproducibility by providing simple methodology to create consistent datasets. Coupled with Pangeo Catalog (which in itself is powered by intake), intake-esm gives climate scientists the means to create and distribute large data collections with instructions on how to use them essentially written into their ESM specifications.

There is still much work to be done with respect to intake-esm and Pangeo Catalog; in particular, goals include:

- Merging ESM collection specifications into SpatioTemporal Asset Catalog (STAC) specification to offer a more universal specification standard
- Development of tools to verify and describe catalogued data on a regular basis
- Restructuring of catalogs to allow subsetting by cloud provider region

Please reach out if you are interested in participating in any way.

References

- intake-esm documentation
- intake documentation
- Pangeo Catalog on GitHub
- Pangeo documentation
- A list of existing, "known" catalogs

Python API to argovis.colorado.edu

The about page of the web app argovis.colorado.edu introduces the web app website and links to other useful information (e.g. FAQs, tutorials). The Argovis tutorials page provides more information about available APIs, along with sample code in different programming languages (e.g. Python, Matlab). Supplementary tutorial videos are found here.

Running this notebook requires some setup, installing several libraries and creating a separate python environment. A docker container has been created to run this noteobook in a separate environment. Further instructions are found at this Github repository.

A short summary of the background and of Argovis' goals:

For the first time in history, the [Argo] (http://www.argo.ucsd.edu/index.html) network of profiling floats provides real-time data of temperature T, salinity S, and pressure P for the global ocean to a depth of 2000 dbar, with Deep Argo floats going down to 6000-dbar depth. Argo floats have been deployed since the early 2000s and reached the expected spatial distribution in 2007 (Roemmich et al. 2009). Nearly 4000 floats are currently operating in the global ocean and provide a profile every 10 days, that is, measurements from a vertical column of the ocean as a single float ascends to the surface. The four-dimensional (4D) space-time Argo data have many scientific and technological advantages, two of which are 1) unprecedented spatial and temporal resolution over the global ocean and 2) no seasonal bias (Roemmich et al. 2009). More than two million T/S/P profiles have been collected through the Argo Program.

The web app argovis.colorado.edu aims to improve visualization and data retrieval of the Argo dataset. This web app is a maintainable, scalable, and portable tool written with representational state transfer (REST) architecture. The RESTful design offers us the opportunity to feature cloud computing applications; chiefly, map comparison from existing gridded Argo products, as well as parameter estimation such as basin mean T/S/P.

Currently, Argovis is expanding to co-locate Argo data with weather events, satellite, and other Earth science datasets.

Citation for the Argovis web app and the Argovis database:

Tucker, T., D. Giglio, M. Scanderbeg, and S.S.P. Shen, 0: Argovis: A Web Application for Fast Delivery, Visualization, and Analysis of Argo Data. J. Atmos. Oceanic Technol., 37, 401–416, [https://doi.org/10.1175/JTECH-D-19-0041.1] (https://doi.org/10.1175/JTECH-D-19-0041.1)

If using data from Argovis in publications, please cite both the above Argovis web application paper and the original Argo data source reference in your paper:

" These data were collected and made freely available by the International Argo Program and the national programs that contribute to it. (http://www.argo.ucsd.edu, http://argo.jcommops.org). The Argo Program is part of the Global Ocean Observing System. " Argo (2000). Argo float data and metadata from the Global Data Assembly Centre (Argo GDAC). SEANOE. http://doi.org/10.17882/42182

Acknowledgements:

Argovis is hosted on a server of the Department of Atmospheric and Oceanic Sciences (ATOC) at the University of Colorado Boulder. Currently, Argovis is funded by the NSF Earthcube program (Award #1928305).

In the past, Argovis has been funded by (starting with the most recent):

- Giglio's research funds provided by University of Colorado Boulder
- the SOCCOM Project through grant number NSF PLR-1425989
- the US Argo Program through NOAA Grant NA15OAR4320071 (CIMEC)
- the National Oceanic and Atmospheric Administration – Cooperative Science Center for Earth System Sciences and - Remote Sensing Technologies (NOAA-CREST) under the Cooperative Agreement Grant #: NA16SEC4810008
- the U.S. NOAA Cooperative Institute for Climate Science (Award No. 13342-Z7812001)
- The City College of New York, NOAA-CREST program and NOAA Office of Education, Educational Partnership Program which provided full fellowship support to Tyler Tucker at San Diego State University

The initial development of Argovis referenced the codes and ideas of the 4-Dimensional Visual Delivery (4DVD) technology developed at the Climate Informatics Lab, San Diego State University. The computer code for 4DVD is at https://github.com/dafrenchyman/4dvd and is available for download under the GNU General Public License open source license. All applicable restrictions, disclaimers of warranties, and limitations of liability in the GNU General Public License also apply to uses of 4DVD on this website.

Description of this notebook:

The following code is a way to get ocean data stored on Argovis. HTTP 'get' requests access the web app's database; however, without a browser. Essentially, this interface is used to query the same database that builds the website.

Citation:

Tucker, T., D. Giglio, M. Scanderbeg, and S.S.P. Shen, 0: Argovis: A Web Application for Fast Delivery, Visualization, and Analysis of Argo Data. J. Atmos. Oceanic Technol., 37, 401–416, [https://doi.org/10.1175/JTECH-D-19-0041.1] (https://doi.org/10.1175/JTECH-D-19-0041.1)

This notebook will guide a python user to:

1. Query a specific profile using its id, designated by its platform (WMO) number with its cycle number, connected by an underscore. For example '3900737_9'

2. Query a specified platform by number. Example '3900737'.

3.1 Query profiles within a given shape, date range, and pressure range.

3.2 Query profiles position, date, and cycle number within month and year (globally).

4. Plot query results

5. Create time series for a selected region and set of dates.

6. Query database using a gridded scheme

7. Overlay Atmospheric Rivers on the map

Firstly, the following libraries are called and styles are set.

```
import requests
import numpy as np
import pandas as pd

import cmocean
import matplotlib.pylab as plt
from scipy.interpolate import griddata
from scipy import interpolate
from datetime import datetime
import pdb
import os
import csv

from datetime import datetime, timedelta
import calendar

import matplotlib
matplotlib.font_manager._rebuild()

#used for map projections
from cartopy import config
import cartopy.crs as ccrs
import matplotlib.patches as mpatches

%matplotlib inline

#sets plot styles
import seaborn as sns
from matplotlib import rc
from matplotlib import rcParams
import matplotlib.ticker as mtick
rc('text', usetex=False)
rcStyle = {"font.size": 10,
           "axes.titlesize": 20,
           "axes.labelsize": 20,
           'xtick.labelsize': 16,
           'ytick.labelsize': 16}
sns.set_context("paper", rc=rcStyle)
sns.set_style("whitegrid", {'axes.grid' : False})
myColors = ["windows blue", "amber", "dusty rose", "prussian blue", "faded green",
"dusty purple", "gold", "dark pink", "green", "red", "brown"]
colorsBW = ["black", "grey"]
sns.set_palette(sns.xkcd_palette(myColors))

curDir = os.getcwd()
dataDir = os.path.join(curDir, 'data')

if not os.path.exists(dataDir):
    os.mkdir(dataDir)

import warnings
warnings.filterwarnings('ignore')
```

1. Get A Profile

The requests library handles the HTTP getting and receiving. If the message is received and the profile exists Argovis will return a JSON object. Python casts a JSON object as a native dictionary type.

in this example, we are going to access the profile from float 3900737 cycle 279. The function below builds the following URL, requests JSON data, and returns it in python.

https://argovis.colorado.edu/catalog/profiles/3900737_279

```
def get_profile(profile_number):
    url = 'https://argovis.colorado.edu/catalog/profiles/{}'.format(profile_number)
    resp = requests.get(url)
    # Consider any status other than 2xx an error
    if not resp.status_code // 100 == 2:
        return "Error: Unexpected response {}".format(resp)
    profile = resp.json()
    return profile
```

```
profileDict = get_profile('3900737_279')
```

`profileDict` is a set of key:value pairs enclosed by curly brackets. These are profile objects stored in the Argovis Database, imported in a Python environment. We expose the keys with the command `profileDict.keys()`

```
profileDict.keys()
```

```
dict_keys(['bgcMeasKeys', 'station_parameters', 'station_parameters_in_nc',
'PARAMETER_DATA_MODE', '_id', 'POSITIONING_SYSTEM', 'DATA_CENTRE', 'PI_NAME',
'WMO_INST_TYPE', 'VERTICAL_SAMPLING_SCHEME', 'DATA_MODE', 'PLATFORM_TYPE',
'measurements', 'pres_max_for_TEMP', 'pres_min_for_TEMP', 'pres_max_for_PSAL',
'pres_min_for_PSAL', 'max_pres', 'date', 'date_added', 'date_qc', 'lat', 'lon',
'geoLocation', 'position_qc', 'cycle_number', 'dac', 'platform_number', 'nc_url',
'DIRECTION', 'BASIN', 'bgcMeas', 'url', 'core_data_mode', 'jcommopsPlatform',
'euroargoPlatform', 'formatted_station_parameters', 'roundLat', 'roundLon', 'strLat',
'strLon', 'date_formatted', 'id'])
```

Core measuremenent data is stored in the field ‘measurements’. Its a list of dictionary objects, indexed by pressure. Each row is a pressure level and each key is a column. We can convert this tabular like data into a pandas dataframe. Essentially it is a spreadsheet table.

```
profileDict = get_profile('3900737_279')
profileDf = pd.DataFrame(profileDict['measurements'])
profileDf['cycle_number'] = profileDict['cycle_number']
profileDf['profile_id'] = profileDict['_id']
profileDf.head()
```

	temp	psal	pres	cycle_number	profile_id
0	27.165	35.421	4.4	279	3900737_279
1	27.063	35.421	10.0	279	3900737_279
2	27.055	35.422	16.9	279	3900737_279
3	27.048	35.422	23.7	279	3900737_279
4	27.046	35.421	30.9	279	3900737_279

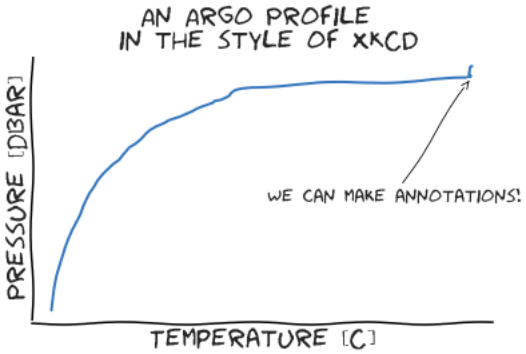
With the data in this form, we can plot it with our favorite library, matplotlib. Try different styles.

```
with plt.xkcd():
    fig = plt.figure()
    ax = fig.add_axes((0.1, 0.2, 0.8, 0.7))
    ax.spines['right'].set_color('none')
    ax.spines['top'].set_color('none')
    ax.set_xticks([])
    ax.set_yticks([])
    ax.invert_yaxis()

    dataX = profileDf.pres.values
    dataY = profileDf.temp.values
    ax.plot(dataY, dataX)

    ax.set_title('An Argo Profile \n in the style of XKCD')
    ax.set_xlabel('Temperature [C]')
    ax.set_ylabel('Pressure [dbar]')

    ax.annotate(
    'We can make annotations!',
    xy=(dataY[12], dataX[12]+10), \
        arrowprops=dict( color='k', arrowstyle='->'), xytext=(15, 1100))
```



2. Get A Platform

A platform consists of a list of profiles. An additional function 'parse_into_df' appends each profile to one data frame.

In this example we are constructing the url: https://argovis.colorado.edu/catalog/platforms/3900737

```
def get_platform_profiles(platform_number):
    url = 'https://argovis.colorado.edu/catalog/platforms/{}'.format(platform_number)
    resp = requests.get(url)
    # Consider any status other than 2xx an error
    if not resp.status_code // 100 == 2:
        return "Error: Unexpected response {}".format(resp)
    platformProfiles = resp.json()
    return platformProfiles

def parse_into_df(profiles):
    meas_keys = profiles[0]['measurements'][0].keys()
    df = pd.DataFrame(columns=meas_keys)
    for profile in profiles:
        profileDf = pd.DataFrame(profile['measurements'])
        profileDf['cycle_number'] = profile['cycle_number']
        profileDf['profile_id'] = profile['_id']
        profileDf['lat'] = profile['lat']
        profileDf['lon'] = profile['lon']
        profileDf['date'] = profile['date']
        df = pd.concat([df, profileDf], sort=False)
    return df
```

```
platformProfiles = get_platform_profiles('3900737')#('5904684')
platformDf = parse_into_df(platformProfiles)
print('number of measurements {}'.format(platformDf.shape[0]))
```

```
number of measurements 25609
```

```
platformDf.head()
```

	temp	psal	pres	cycle_number	profile_id	lat	lon	date
0	26.349	33.770	4.4	1.0	3900737_1	0.931	-84.083	2009-06-15T11:13:53.000Z
1	26.356	33.781	10.5	1.0	3900737_1	0.931	-84.083	2009-06-15T11:13:53.000Z
2	26.294	33.894	17.7	1.0	3900737_1	0.931	-84.083	2009-06-15T11:13:53.000Z
3	26.014	34.198	24.6	1.0	3900737_1	0.931	-84.083	2009-06-15T11:13:53.000Z
4	24.573	34.716	31.6	1.0	3900737_1	0.931	-84.083	2009-06-15T11:13:53.000Z

By the way, Pandas dataframes can handle large arrays efficiently, thanks to the underlying numpy library. Pandas allow for easy and quick computations, such as taking the mean of the measurements.

```
platformDf[['pres', 'psal', 'temp']].mean(0)
```

```
pres    539.346652
psal     34.905583
temp     12.602862
dtype: float64
```

Next, we shall plot these profiles' temperature at a level of interest. The function below calculates a linear interpolation of temperature and salinity. # Note that some of the profiles in Argovis may not have salinity (either because there is no salinity value in the original Argo file or the quality is bad)

A simple script then plots a scatter chart with the color set to temperature. Let's also use the Cartopy library for base layers and projections.

```
def parse_into_df_plev(profiles, plev):
    plevProfileList = []
    for profile in profiles:
        profileDf_bfr = pd.DataFrame(profile['measurements'])
        plevProfile = profile
        fT = interpolate.interp1d(profileDf_bfr['pres'], profileDf_bfr['temp'],
bounds_error=False)
        plevProfile['temp'] = fT(plev)
        # some of the profiles in Argovis may not have salinity
        # (either because there is no salinity value in the original Argo file or the
quality is bad)
        try:
            fS = interpolate.interp1d(profileDf_bfr['pres'], profileDf_bfr['psal'],
bounds_error=False)
            plevProfile['psal'] = fS(plev)
        except:
            plevProfile['psal'] = np.nan #  No salinity found in profile
        plevProfile['pres'] = plev
        plevProfileList.append(plevProfile)
    df = pd.DataFrame(plevProfileList)
    df = df.sort_values(by=['cycle_number'])
    df = df[['cycle_number','_id','date','lon','lat','pres','temp','psal']]
    return df
```

```
def plot_pmesh(df, measName, figsize=(16,24), shrinkcbar=.1, \
               delta_lon=10, delta_lat=10, map_proj=ccrs.PlateCarree(), \
               xlims=None):
    fig = plt.figure(figsize=figsize)
    x = df['lon'].values
    y = df['lat'].values
    points = map_proj.transform_points(ccrs.Geodetic(), x, y)
    x = points[:, 0]
    y = points[:, 1]

    z = df[measName].values
    map_proj._threshold /= 100.  # the default values is bad, users need to set them
manually
    ax = plt.axes(projection=map_proj, xlabel='long', ylabel='lats')
    plt.title(measName + ' on a map')

    sct = plt.scatter(x, y, c=z, s=15, cmap=cmocean.cm.dense,zorder=3)
    cbar = fig.colorbar(sct, cmap=cmocean.cm.dense, shrink=shrinkcbar)

    if not xlims:
        xlims = [df['lon'].max() + delta_lon, df['lon'].min()- delta_lon]

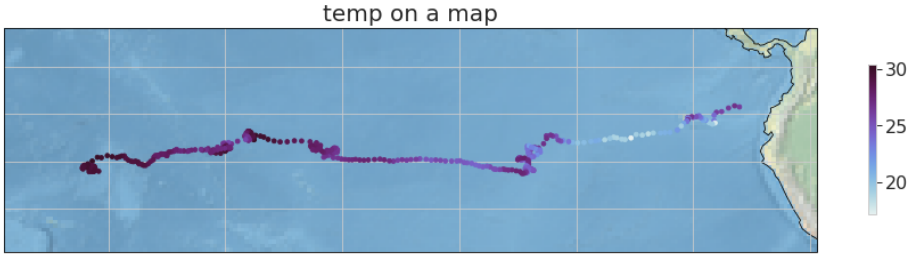
    ax.set_xlim(min(xlims), max(xlims))
    ax.set_ylim(min(df['lat']) - delta_lat, max(df['lat']) + delta_lat)

    ax.coastlines(zorder=1)
    ax.stock_img()
    ax.gridlines()
    return fig
```

```
plevIntp = 20
platformDf_plev = parse_into_df_plev(platformProfiles, plevIntp)
platformDf_plev.head()
```

	cycle_number	_id	date	lon	lat	pres	
0	1	3900737_1	2009-06-15T11:13:53.000Z	-84.083000	0.931	20	26.2006666666
111	2	3900737_2	2009-06-26T07:03:40.999Z	-84.698997	1.070	20	26.898189189
220	3	3900737_3	2009-07-06T23:34:37.001Z	-85.484001	0.789	20	26.356621621
288	4	3900737_4	2009-07-17T23:01:22.998Z	-86.365997	0.333	20	25
299	5	3900737_5	2009-07-28T15:05:59.000Z	-87.029999	0.245	20	26.1111791044

```
pmeshfig = plot_pmesh(platformDf_plev,'temp')
plt.show()
```



3.1 Get A Space-Time Selection

This query retrieves profiles within a given shape, date range, and optional depth range.

This region's data will be queried and plotted in Python.

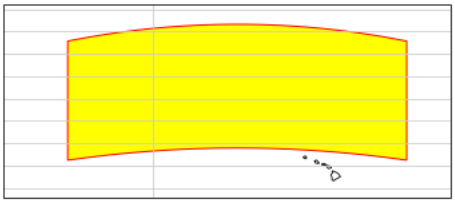
- start date: string formatted as 'YYYY-MM-DD'
- end date: string formatted as 'YYYY-MM-DD'
- pressure range (optional): string formatted as '[lowerPressure,upperPressure]'. No Spaces!
- shape: a list of lists containing [lng, lat] coordinates.

It is worth mentioning some information regarding how profiles are stored in Argovis. First, know that latitude lines are drawn on a map typically appear to be the shortest distance. Indeed this would be the case on a flat plane; however, for spherical-like objects like the Earth, Geodesic lines are shorter. Argovis indexes its coordinates on a sphere, thus takes spherical geometry when making spatial selections. Not only does this allow a more accurate representation of how a shape on a map behaves, but it also allows us to query profiles that fall in shapes that cross the antimeridian.

We can see an example of a shape in the yellow region below.

```
shape = [[[168.6,21.7],[168.6,37.7],[-145.9,37.7],[-145.9,21.7],[168.6,21.7]]]
lat_corners = np.array([lnglat[1] for lnglat in shape[0]])
lon_corners = np.array([lnglat[0] for lnglat in shape[0]])
poly_corners = np.zeros((len(lat_corners), 2), np.float64)
poly_corners[:,0] = lon_corners[::-1]
poly_corners[:,1] = lat_corners[::-1]
delta = 5
poly = mpatches.Polygon(poly_corners, closed=True, ec='r', fill=True, lw=1,
fc="yellow", transform=ccrs.Geodetic())
central_longitude = -180
map_proj = ccrs.PlateCarree(central_longitude=central_longitude)
map_proj._threshold /= 100.  # the default values is bad, users need to set them
manually

ax = plt.subplot(1, 1, 1, projection=map_proj)
#ax.set_global()
ax.add_patch(poly)
ax.gridlines()
ax.coastlines()
ax.set_ylim(lat_corners.min() - delta, lat_corners.max() + delta)
xrange = [lon_corners.min() - delta, lon_corners.max() + delta]
ax.set_xlim(-20, 40)
plt.show()
```



We should expect profiles made using the function `get_selection_profiles()` to fall within this region.

```
def get_selection_profiles(startDate, endDate, shape, presRange=None, printUrl=True):
    url = 'https://argovis.colorado.edu/selection/profiles'
    url += '?startDate={}'.format(startDate)
    url += '&endDate={}'.format(endDate)
    url += '&shape={}'.format(shape)
    if presRange:
        pressRangeQuery = '&presRange=' + presRange
        url += pressRangeQuery
    if printUrl:
        print(url)
    resp = requests.get(url)
    # Consider any status other than 2xx an error
    if not resp.status_code // 100 == 2:
        return "Error: Unexpected response {}".format(resp)
    selectionProfiles = resp.json()
    return selectionProfiles
```

```
startDate='2017-9-15'
endDate='2017-9-30'
strShape = str(shape).replace(' ', '')
presRange='[0,50]'
selectionProfiles = get_selection_profiles(startDate, endDate, strShape, presRange)
selectionProfiles_raw = selectionProfiles
if len(selectionProfiles) > 0:
    selectionDf = parse_into_df(selectionProfiles)
selectionDf.replace(-999, np.nan, inplace=True)
```

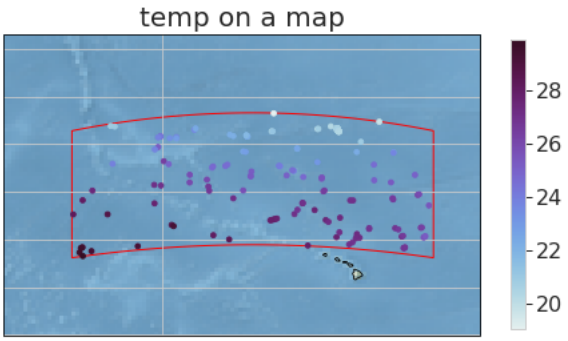
```
https://argovis.colorado.edu/selection/profiles?startDate=2017-9-15&endDate=2017-9-
30&shape=[[[168.6,21.7],[168.6,37.7],[-145.9,37.7],[-145.9,21.7],
[168.6,21.7]]]&presRange=[0,50]
```

As we did with the platform page, we interpolate the profiles to a pressure level of interest and plot the values.

```
selectionDf_plev = parse_into_df_plev(selectionProfiles_raw, plevIntp)
selectionDf_plev.head()
```

	cycle_number	_id	date	lon	lat	pres	
147	1	5905137_1	2017-09-20T14:17:14.001Z	-158.0020	38.0210	20	19
136	1	5905060_1	2017-09-21T12:04:44.999Z	169.9325	23.0209	20	
65	2	5905060_2	2017-09-25T17:00:17.999Z	169.8303	22.9688	20	29
146	2	5905137_2	2017-09-20T16:53:13.001Z	-158.0000	38.0170	20	20
2	3	5905060_3	2017-09-29T21:53:56.999Z	169.7212	22.9206	20	29.537879999

```
pmeshfig = plot_pmesh(selectionDf_plev,'temp', (8, 8), .5, 10, 10, map_proj, [-20,
40])
ax = pmeshfig.get_axes()[0]
poly = mpatches.Polygon(poly_corners, closed=True, ec='r', fill=False, lw=1,
transform=ccrs.Geodetic())
ax.add_patch(poly)
plt.show()
```



3.2 Metadata selection for month and year

For profile metadata, data can be accessed quickly with the following API.

```
def get_monthly_profile_pos(month, year):
    url = 'https://argovis.colorado.edu/selection/profiles'
    url += '/{0}/{1}'.format(month, year)
    resp = requests.get(url)
    if not resp.status_code // 100 == 2:
        return "Error: Unexpected response {}".format(resp)
    monthlyProfilePos = resp.json()
    return monthlyProfilePos

month = 1 # January
year = 2019 # Year 2019
monthlyProfilePos = get_monthly_profile_pos(month, year)
assert not isinstance(monthlyProfilePos, str)
monthlyDf = pd.DataFrame(monthlyProfilePos)
```

```
monthlyDf[['_id', 'date', 'POSITIONING_SYSTEM', 'DATA_MODE', \
           'dac', 'PLATFORM_TYPE', 'lat', 'lon']].head()
```

	_id	date	POSITIONING_SYSTEM	DATA_MODE	dac	PLATFOR
0	5904229_226	2019-01-31T23:58:35.000Z	GPS	D	csiro	
1	2902585_166	2019-01-31T23:57:33.999Z	ARGOS	A	csio	F
2	5905234_39D	2019-01-31T23:56:48.192Z	GPS	D	aoml	
3	5902387_140	2019-01-31T23:49:51.744Z	GPS	D	aoml	
4	5904899_157	2019-01-31T23:46:43.000Z	GPS	D	csiro	

Gathering metadata globally in a period of interest

We can loop over months, saving the results as CSV files.

```
def progress_ind(idx, maxIdx):
    print('{} percent complete'.format(100*round(idx/maxIdx, 3)), end='\r')

def make_grouped_meta_data(dataDir, dateRange):
    profTimeSeries = []
    typeTimeSeries = []
    psTimeSeries = []
    dacTimeSeries = []
    maxIdx = len(dateRange)
    for idx, date in enumerate(dateRange):
        progress_ind(idx, maxIdx)
        month = date.to_pydatetime().month
        year = date.to_pydatetime().year
        monthlyProfilePos = get_monthly_profile_pos(month, year)
        monthlyDf = pd.DataFrame(monthlyProfilePos)
        monthDict = {'date': date, 'nProf': len(monthlyProfilePos)}

        allDict = {}
        allDict.update(monthDict)

        platformType = monthlyDf.groupby('PLATFORM_TYPE')['_id'].count().to_dict()
        platformType.update(monthDict)
        typeTimeSeries.append(platformType)
        allDict.update(platformType)

        ps = monthlyDf.groupby('POSITIONING_SYSTEM')['_id'].count().to_dict()
        ps.update(monthDict)
        psTimeSeries.append(ps)
        allDict.update(ps)

        dac = monthlyDf.groupby('dac')['_id'].count().to_dict()
        dac.update(monthDict)
        dacTimeSeries.append(dac)
        allDict.update(dac)

        profTimeSeries.append(allDict)
    progress_ind(maxIdx, maxIdx)
    save_metadata_time_series(dataDir, 'groupedProfileTimeSeries.csv', profTimeSeries)
    save_metadata_time_series(dataDir, 'groupedProfilePositioningSystemTimeSeries.csv',
psTimeSeries)
    save_metadata_time_series(dataDir, 'groupedProfileTypeTimeSeries.csv',
typeTimeSeries)
    save_metadata_time_series(dataDir, 'groupedDacTimeSeries.csv', dacTimeSeries)

def save_metadata_time_series(dataDir, filename, metadataDict):
    filename = os.path.join(dataDir, filename)
    df = pd.DataFrame(metadataDict)
    df.to_csv(filename, index=False)

def get_grouped_metadata(dataDir, filename):
    df = pd.read_csv(os.path.join(dataDir,filename))
    df['date'] = pd.to_datetime(df['date'])
    return df
```

```
dateRange = pd.date_range('2015-01-01', '2016-06-01', periods=None, freq='M')
make_grouped_meta_data(dataDir, dateRange)
```

```
100.0 percent completeent completee
```

```
profTimeSeriesDf = get_grouped_metadata(dataDir, 'groupedProfileTimeSeries.csv')
psTimeSeriesDf = get_grouped_metadata(dataDir,
'groupedProfilePositioningSystemTimeSeries.csv')
typeTimeSeriesDf = get_grouped_metadata(dataDir, 'groupedProfileTypeTimeSeries.csv')
dacTimeSeriesDf = get_grouped_metadata(dataDir, 'groupedDacTimeSeries.csv')
```

```
dacTimeSeriesDf.head()
```

	aoml	bodc	coriolis	csio	csiro	incois	jma	kma	kordi	meds	nmdis	date	nP
0	7332	529	2055	833	1062	376	731	109	22	146	34.0	2015-01-31	132
1	6846	497	1944	762	993	575	684	71	23	127	31.0	2015-02-28	125
2	7389	556	2134	837	1070	440	787	86	19	139	34.0	2015-03-31	134
3	7227	535	2149	804	1033	396	850	86	21	141	29.0	2015-04-30	132
4	7752	541	2499	708	1058	408	945	76	22	154	28.0	2015-05-31	141

4. Plotting metadata results

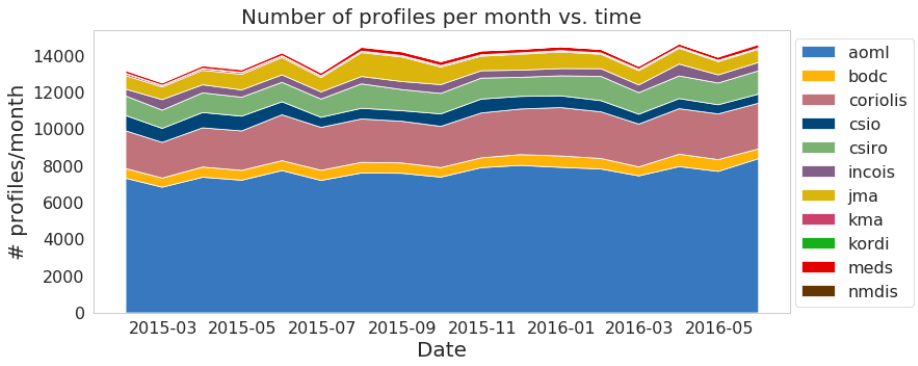
Argovis's plotting capabilities are limited to what is coded in JavaScript. There are many plotting libraries out there, but in this example we can create plots on our machine, thereby allowing customization.

We just made some dataframes containing profile metadata. We can plot these time series with the following function.

```
def make_stack_plot(df, figsize=(6,3)):
    dataDf = df.drop(['date', 'nProf'], axis=1)
    fig = plt.figure(figsize=figsize)
    axes = plt.axes()
    axes.set_title('Number of profiles per month vs. time')
    axes.set_ylabel('# profiles/month')
    axes.set_xlabel('Date')
    axes.stackplot(df['date'].values, dataDf.T, labels=dataDf.columns)
    axes.legend(loc=2, fontsize=16)
    return fig
```

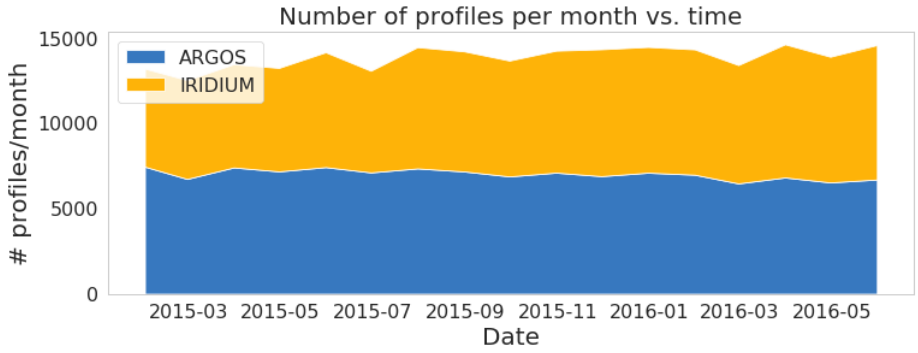
```
fig = make_stack_plot(dacTimeSeriesDf, figsize=(12,5))
axes = plt.axes()
axes.legend(bbox_to_anchor=(1.19, 1.00), fontsize=16)
```

```
<matplotlib.legend.Legend at 0x7f74c28038e0>
```



Here we can see which DACS release profile data over time. The next plot shows which transmission system the profiles are using. I lumped GPS into Iridium. They are essentially equivalent.

```
psdf=psTimeSeriesDf.copy()
if 'IRIDIUM' in psTimeSeriesDf:
    psdf['IRIDIUM'] = psTimeSeriesDf['GPS'] + psTimeSeriesDf['IRIDIUM']
    psdf.drop('GPS', axis = 1, inplace=True)
fig = make_stack_plot(psdf, figsize=(12,4))
```



5. Time series for the average of the data in a region (interpolated on pressure levels of interest)

We build a time series by stacking selection queries on top of each other. In this example, we average interpolated temperature data from profiles in a region of interest in monthly aggregates. We starting with January in `startYear` and ending in December of `endYear`.

In other words, we are using `get_selection_profiles` from [#section_three](section 3.1). and taking the mean of all the points of a given month. Note that we ought to take a *spatially* averaged mean if we were to put this into practice.

```
def get_month_day_range(date):
    '''gets first day and last day of the month a given month-year datetime object'''
    first_day = date.replace(day=1)
    last_day = date.replace(day=calendar.monthrange(date.year, date.month)[1])
    return first_day, last_day

# set region and pressure range of interest
shape = '[[[-65,20],[-65,25],[-60,25],[-60,20],[-65,20]]]'
presRange ='[0,50]'
startYear = 2019
endYear = 2020
```

```
def time_series(shape, presRange, startYear=2019, endYear=2020, plevIntp=(20)):
    tempMean = []
    times = []
    yearRange = range(startYear,endYear+1)
    maxYdx = len(yearRange)
    for ydx, yy in enumerate(yearRange):
        progress_ind(ydx, maxYdx)
        monthRange = range(1,13)
        for mm in monthRange:
            times.append(datetime(yy, mm, 15))
            [startDate, endDate] = get_month_day_range(datetime(yy, mm, 15))
            profiles = get_selection_profiles(startDate, endDate, shape, presRange,
printUrl=False)
            #pdb.set_trace()
            if len(profiles) > 0:
                df = parse_into_df_plev(profiles, plevIntp)
                df = df.dropna(subset=['temp'])
                mean = df['temp'].mean()
                tempMean.append(mean)
            else:
                tempMean.append(np.nan)
    progress_ind(maxYdx, maxYdx)
    return tempMean, times

tempMean, times = time_series(shape, presRange, startYear, endYear, plevIntp)
```

```
100.0 percent complete
```

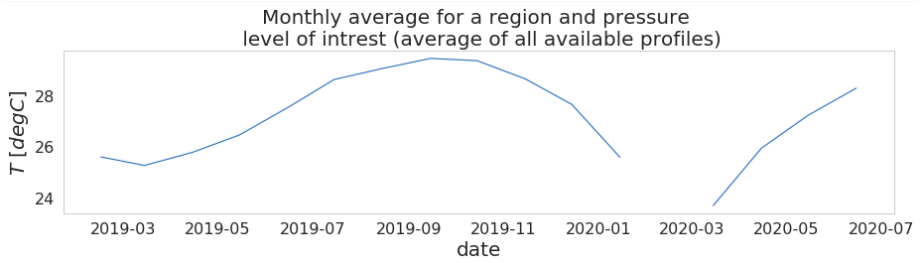
Once again, we save our output to a csv.

```
filename = os.path.join(dataDir, 'time_series.csv')
strTimes = [datetime.strftime(t, '%Y-%m-%d') for t in times]
with open(filename, 'w') as myfile:
    wr = csv.writer(myfile, quoting=csv.QUOTE_ALL)
    wr.writerow(('date', 'tempMean'))
    wr.writerows(zip(strTimes, tempMean))
```

```
tsdf = pd.read_csv(filename)
tsdf['date'] = tsdf['date'].apply(lambda t: datetime.strptime(t, '%Y-%m-%d'))
```

```
fig = plt.figure(999, figsize=(15,3))
ax = plt.axes()
ax.plot(tsdf['date'], tsdf['tempMean'])
ax.set_title('Monthly average for a region and pressure \n level of intrest (average of
all available profiles)')
ax.set_ylabel('$T\ [degC]$')
ax.set_xlabel('date')
```

```
Text(0.5, 0, 'date')
```



Large aggregations such as those seen in this section require multiple queries. The database that stores profiles can at most return 16MB for one query. Users attempting to query over too large an area or time range will receive an error message instead of the expected JSON. Hence, we segmented the routine in a loop, where each month/year is queried at one time.

6. Query globally within thin slice of pressure

In this section, we query all the data globally within a very short time and for a pressure range of interest. How small a query? Consider the following guidelines:

- Pressure range is no more than 50 dbar
- The date range is no more than 3 days This will ensure that JSON is returned instead of an unexpected error message.

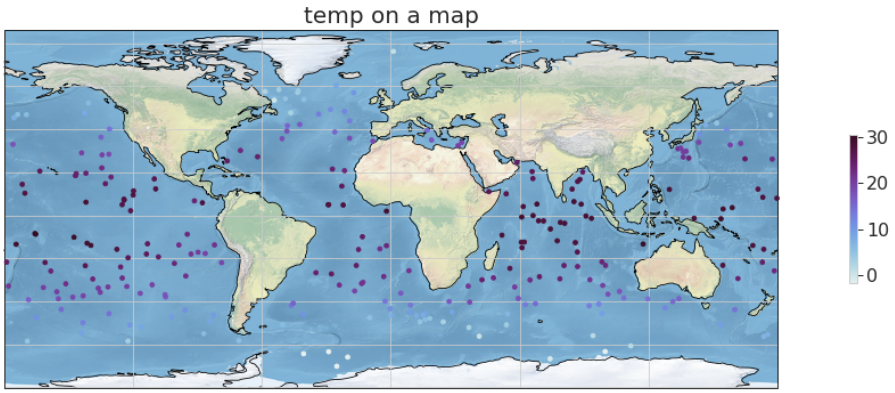
We then interpolate each profile to a pressure level of interest and plot the temperature using our `plot_pmesh()` function back from section 2.

```
def get_ocean_slice(startDate, endDate, presRange='[5,15]'):
    '''
    query horizontal slice of ocean for a specified time range
    startDate and endDate should be a string formated like so: 'YYYY-MM-DD'
    presRange should comprise of a string formatted to be: '[lowPres,highPres]'
    Try to make the query small enough so as to not pass the 16 MB limit set by the
database.
    '''
    baseURL = 'https://argovis.colorado.edu/gridding/presSlice/'
    startDateQuery = '?startDate=' + startDate
    endDateQuery = '&endDate=' + endDate
    presRangeQuery = '&presRange=' + presRange
    url = baseURL + startDateQuery + endDateQuery + presRangeQuery
    resp = requests.get(url)
    # Consider any status other than 2xx an error
    if not resp.status_code // 100 == 2:
        return "Error: Unexpected response {}".format(resp)
    profiles = resp.json()
    return profiles
```

```
presRange = '[0, 50]' #  used to query database
date = datetime(2010, 1, 1, 0, 0, 0) # this date will be used later for ARs
startDate='2010-1-1'
endDate='2010-1-2'
sliceProfiles = get_ocean_slice(startDate, endDate, presRange)
sliceDf = parse_into_df_plev(sliceProfiles, plevIntp)
sliceDf.head()
```

	cycle_number	_id	date	lon	lat	pres	te
67	0	1901426_0	2010-01-01T15:47:12.192Z	42.499	-29.988	20	24.323666666666
238	1	1900870_1	2010-01-01T03:08:44.000Z	64.854	-30.061	20	22.404916666666
255	1	1901400_1	2010-01-01T01:33:58.000Z	47.850	-29.065	20	22.962849462365
51	1	5903310_1	2010-01-01T18:14:36.095Z	80.669	-23.328	20	24
63	1	5901928_1	2010-01-01T16:39:14.999Z	99.993	-23.271	20	23.011072727272

```
pmeshfig = plot_pmesh(sliceDf,'temp')
plt.show()
```



Taking a step back, we can see several possibilities of using this API such as:

- Generate a global/regional daily state of the ocean, generated with a simple function call.
- Track a single platform throughout its history.
- Make historical time series for a region.

In addition to these items, Argovis is a tool for incorporating other data sets. A recent interest of Argovis is to include data other than Argo, I.E. hurricane track data, or satellite data. Even data generated from computer simulation or statistical mapping can be compared to argo profiles.

While still in its infancy, we have added some tools and features in the form of modules that explore the interaction between different data. One such module is mentioned in the next section.

7. Overlay Atmospheric Rivers on the map

Atmospheric rivers (AR) are bodies of moisture in the atmosphere. They are responsible for localized heavy rainfall such as the Pineapple express on the west coast of North America. Argovis has recently included an AR data set taken from by Guan and Walister. The dataset is comprised of shapes representing atmospheric rivers at a given time. Argovis charts these shapes and uses them to query for profiles. The module is located Here.

The following code shall query atmospheric rivers, and co-locate them with argo profiles.

```
def get_ar_by_date(date):
    url = "https://argovis.colorado.edu/arShapes/findByDate?date="
    url += date
    resp = requests.get(url)
    if not resp.status_code // 100 == 2:
        return "Error: Unexpected response {}".format(resp)
    ars = resp.json()
    return ars

def format_ars(ars):
    for ar in ars:
        coords = ar['geoLocation']['coordinates']
        del ar['geoLocation']
        longs, lats = list(zip(*coords))
        ar['coords'] = coords
        ar['longs'] = list(longs)
        ar['lats'] = list(lats)
    return ars

format_date_api = lambda date: datetime.strftime(date, "%Y-%m-%dT%H:%M:%SZ")
stringify_array = lambda arr: str(arr).replace(' ', '')
```

```
dateStr = format_date_api(date)
ars = get_ar_by_date(dateStr)
ars = format_ars(ars)
arDf = pd.DataFrame(ars)
```

```
presRange='[0,30]'
startDate = format_date_api(date - timedelta(days=3))
endDate = format_date_api(date + timedelta(days=3))
ar = arDf.iloc[2]
coords = list(zip(ar.longs,ar.lats))
coords = [list(elem) for elem in coords]
shape = stringify_array([coords])
```

The following chart takes one AR shape queried from Argovis and plots it with our slice query from section six

```
def plot_ar_over_profile_map(df,var,ar):
    fig = plt.figure(figsize=(8,12))
    x = df['lon'].values
    y = df['lat'].values
    z = df[var].values
    ax = plt.axes(projection=ccrs.PlateCarree())

    plt.title(var)

    ax.coastlines(zorder=1)

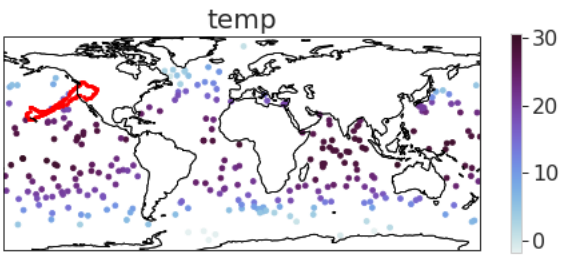
    ax.coastlines(zorder=1)
    sct = plt.scatter(x, y, c=z,s=15, cmap=cmocean.cm.dense,zorder=0)
    cbar = fig.colorbar(sct, cmap=cmocean.cm.dense, shrink=.25)

    ARy, ARx = ar.lats, ar.longs
    plt.scatter(ARx,ARy,marker='o',c='r',s=5)

    ax.set_ylabel('latitude')
    ax.set_xlabel('longitude')

    return fig

plot_ar_over_profile_map(sliceDf,'temp',ar)
plt.show()
```



Going even further, we can take AR shapes and make profile selections with them.

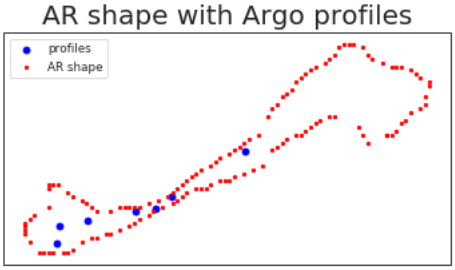
```
profiles = get_selection_profiles(startDate, endDate, shape, presRange)
profileDf = parse_into_df(profiles)
pdf = profileDf.drop_duplicates(subset='profile_id')
pdf.head()
```

```
https://argovis.colorado.edu/selection/profiles?startDate=2009-12-
29T00:00:00Z&endDate=2010-01-04T00:00:00Z&shape=[[[-115.625,37],[-118.125,36],
[-118.75,37],[-119.375,38],[-122.5,39.5],[-123.125,39.5],[-124.375,39],[-125,38.5],
[-125.625,38],[-126.25,37.5],[-126.875,37],[-127.5,37],[-128.75,36.5],[-129.375,36],
[-130,35.5],[-130.625,35],[-131.875,33],[-133.125,32.5],[-134.375,32],
[-135.625,31.5],[-136.25,31],[-136.875,31],[-137.5,31],[-138.75,30.5],[-139.375,30],
[-140,30],[-140.625,30],[-141.875,29.5],[-142.5,29],[-143.125,28.5],[-143.75,28],
[-145,27.5],[-146.25,27],[-146.875,26.5],[-148.125,26],[-148.75,25.5],[-149.375,25],
[-150.625,24.5],[-151.875,24],[-152.5,24],[-153.75,23.5],[-154.375,23.5],
[-155.625,23],[-156.875,22],[-157.5,21.5],[-158.125,21.5],[-159.375,21.5],
[-160,21.5],[-160.625,21.5],[-161.25,21.5],[-162.5,23],[-163.125,24],[-163.125,24.5],
[-163.125,25],[-163.125,25.5],[-162.5,26],[-161.875,26.5],[-160,27.5],[-160,30],
[-160,30.5],[-159.375,30.5],[-158.75,30.5],[-157.5,29.5],[-156.875,29],
[-155.625,28.5],[-155,28],[-154.375,27.5],[-153.75,27],[-151.875,27],[-150.625,27.5],
[-150,27.5],[-149.375,27.5],[-148.75,27.5],[-148.125,27.5],[-146.875,28],
[-145.625,28.5],[-144.375,29],[-143.75,29.5],[-143.125,30],[-142.5,30.5],
[-141.875,31],[-141.25,31.5],[-140.625,32],[-140,32.5],[-138.75,33],[-138.125,33.5],
[-137.5,34],[-136.25,34.5],[-135.625,35],[-135,35.5],[-134.375,36],[-133.75,36.5],
[-132.5,37],[-131.25,39.5],[-130.625,40.5],[-130,41],[-129.375,42],[-128.75,42.5],
[-128.125,43],[-127.5,44],[-126.875,44.5],[-126.25,45],[-125.625,45.5],[-125,46],
[-123.75,46.5],[-123.125,47],[-122.5,47.5],[-121.875,48.5],[-121.25,49],
[-120.625,49],[-120,49],[-118.75,48.5],[-118.125,47.5],[-117.5,47],[-116.25,46.5],
[-115,46],[-114.375,46],[-113.75,46],[-112.5,45.5],[-111.875,45],[-111.25,44.5],
[-110,44],[-110,43.5],[-110.625,42],[-110.625,41],[-111.25,40.5],[-112.5,40],
[-113.125,39],[-113.75,38.5],[-114.375,37.5],[-115,37],[-115.625,37]]]&presRange=
[0,30]
```

	pres	psal	temp	cycle_number	profile_id	lat	lon	date
0	5.5	35.392	21.512	97.0	5901759_97	27.448	-145.995	2010-01-02T19:31:48.000Z
0	5.7	35.511	22.681	122.0	5901336_122	25.165	-158.639	2010-01-02T12:35:27.000Z
0	5.5	35.488	22.040	97.0	5901757_97	25.868	-154.894	2010-01-02T02:22:05.088Z
0	5.5	35.448	21.640	97.0	5901756_97	26.966	-148.599	2010-01-02T00:39:58.464Z
0	5.5	34.832	19.976	46.0	5902210_46	28.997	-143.873	2010-01-01T21:54:46.656Z

```
def plot_profiles_and_ar(pdf, ar, central_longitude=-180):
    map_proj = ccrs.PlateCarree(central_longitude=central_longitude)
    map_proj._threshold /= 100.  # the default values is bad, users need to set them
manually
    #fig = plt.figure(projection=map_proj)
    ax = plt.axes(xlabel='longitude', ylabel='latitude', projection=map_proj)
    py, px = pdf.lat.values, pdf.lon.values
    ary, arx = ar.lats, ar.longs
    profax = ax.scatter(px,py,marker='o',c='b',s=25, label='profiles')
    arax = ax.scatter(arx,ary,marker='.',c='r',s=25, label='AR shape')
    ax.set_title('AR shape with Argo profiles')
    ax.legend(handles=[profax, arax])
    return fig

fig1 = plot_profiles_and_ar(pdf, ar)
plt.show()
```



As of the writing of this notebook, our only external data remains the Guan-Walister AR dataset. In time we intend to include tropical cyclone data and hurricane shapes which will be co-located in a similar fashion to the methods shown by this AR API.

Conclusion

We hope that you found these API functions usefull. Argovis itself uses them on its front-end interface. At its core, its API is a series of `get` requests that can be written in a number or languages, such as R, Matlab. Some Matlab scripts are provided here.

This project is still new, and will continue to evolve and improve. Feel free to email tyler.tucker@colorado.edu for questions/requests. Thanks!

```
import requests
import numpy as np
import pandas as pd

import cmocean
import matplotlib.pylab as plt
from scipy.interpolate import griddata
from scipy import interpolate
from datetime import datetime
import pdb
import os
import csv

from datetime import datetime, timedelta
import calendar

import matplotlib
matplotlib.font_manager._rebuild()

#used for map projections
from cartopy import config
import cartopy.crs as ccrs
import matplotlib.patches as mpatches

%matplotlib inline

#sets plot styles
import seaborn as sns
from matplotlib import rc
from matplotlib import rcParams
import matplotlib.ticker as mtick
rc('text', usetex=False)
rcStyle = {"font.size": 10,
           "axes.titlesize": 20,
           "axes.labelsize": 20,
           'xtick.labelsize': 16,
           'ytick.labelsize': 16}
sns.set_context("paper", rc=rcStyle)
sns.set_style("whitegrid", {'axes.grid' : False})
myColors = ["windows blue", "amber", "dusty rose", "prussian blue", "faded green",
"dusty purple", "gold", "dark pink", "green", "red", "brown"]
colorsBW = ["black", "grey"]
sns.set_palette(sns.xkcd_palette(myColors))

curDir = os.getcwd()
dataDir = os.path.join(curDir, 'data')

if not os.path.exists(dataDir):
    os.mkdir(dataDir)

import warnings
warnings.filterwarnings('ignore')
```

1. Get a BGC profile

If you know that a profile contains BGC parameters, the standard profile api contains the bgc measurements under the field bgcMeas.

```
def get_profile(profile_number):
    url = 'https://argovis.colorado.edu/catalog/profiles/{}'.format(profile_number)
    resp = requests.get(url)
    # Consider any status other than 2xx an error
    if not resp.status_code // 100 == 2:
        return "Error: Unexpected response {}".format(resp)
    profile = resp.json()
    return profile

def json2dataframe(profiles, measKey='measurements'):
    """ convert json data to Pandas DataFrame """
    # Make sure we deal with a list
    if isinstance(profiles, list):
        data = profiles
    else:
        data = [profiles]
    # Transform
    rows = []
    for profile in data:
        keys = [x for x in profile.keys() if x not in ['measurements', 'bgcMeas']]
        meta_row = dict((key, profile[key]) for key in keys)
        for row in profile[measKey]:
            row.update(meta_row)
            rows.append(row)
    df = pd.DataFrame(rows)
    return df
```

```
profileId = "5901069_270"
profile = get_profile(profileId)
df = json2dataframe([profile], 'bgcMeas')
```

```
> <ipython-input-44-da136a77d156>(20)json2dataframe()
-> for profile in data:
```

```
(Pdb)  c
```

```
df.head(5)
```

	pres	pres_qc	psal	psal_qc	temp	temp_qc	doxy_qc	bgcMeasKeys
0	5.400000	1	35.058998	1	19.774000	1	4	[pres, psal, temp, doxy]
1	9.300000	1	35.057999	1	19.778000	1	1	[pres, psal, temp, doxy]
2	19.500000	1	35.057999	1	19.777000	1	1	[pres, psal, temp, doxy]
3	29.700001	1	35.056000	1	19.768999	1	1	[pres, psal, temp, doxy]
4	39.400002	1	35.055000	1	19.768999	1	1	[pres, psal, temp, doxy]

5 rows × 49 columns

2. Get a BGC Platform, two variables at a time

Platform metadata is queried separatly from the BGC data. This is to keep the payload small enough for the server to operate efficiently. Platform BGC data is queried by two parameters at a time.

```
def get_platform_profile_metadata(platform_number):
    url =
f'https://argovis.colorado.edu/catalog/platform_profile_metadata/{platform_number}'
    print(url)
    resp = requests.get(url)
    # Consider any status other than 2xx an error
    if not resp.status_code // 100 == 2:
        return "Error: Unexpected response {}".format(resp)
    platformMetadata = resp.json()
    return platformMetadata

def get_platform_profile_data(platform_number, xaxis='doxy', yaxis='pres'):
    url = 'https://argovis.colorado.edu/catalog/bgc_platform_data/{0}/?xaxis={1}&yaxis=
{2}'.format(platform_number, xaxis, yaxis)
    print(url)
    resp = requests.get(url)
    # Consider any status other than 2xx an error
    if not resp.status_code // 100 == 2:
        return "Error: Unexpected response {}".format(resp)
    platformData = resp.json()
    return platformData

def join_platform_data(platformMetadata, platformData):
    platforms = []
    for idx, platform in enumerate(platformMetadata):
        metadata_id = platform['_id']
        data_id = platformData[idx]['_id']
        if (metadata_id == data_id) and ('bgcMeas' in platformData[idx].keys()) and
isinstance(platformData[idx]['bgcMeas'], list):
            platform['bgcMeas'] = platformData[idx]['bgcMeas']
            platforms.append(platform)
    return platforms
```

We merge the metadata and data and convert it into a dataframe

```
platformMetadata = get_platform_profile_metadata(5901464)
platformData = get_platform_profile_data(5901464, 'doxy', 'pres')
platforms = join_platform_data(platformMetadata, platformData)
df = json2dataframe(platforms, 'bgcMeas')
```

```
https://argovis.colorado.edu/catalog/platform_profile_metadata/5901464
https://argovis.colorado.edu/catalog/bgc_platform_data/5901464/?xaxis=doxy&yaxis=pres
```

```
df.head()
```

	doxy	doxy_qc	pres	pres_qc	_id	POSITIONING_SYSTEM	DAT
0	71.110497	1	1797.489990	1	5901464_115	GPS	
1	75.924095	1	1848.199951	1	5901464_115	GPS	
2	79.144302	1	1897.660034	1	5901464_115	GPS	
3	81.345016	1	1947.329956	1	5901464_115	GPS	
4	84.040329	1	1997.800049	1	5901464_115	GPS	

5 rows × 34 columns

3. Get a BGC selection

https://argovis.colorado.edu/selection/bgc_data_selection?xaxis=temp&yaxis=pres&startDate=2020-03-01&endDate=2020-03-11&presRange=[0,50]&shape=[[[-155.929898,27.683528],[-156.984448,13.752725],[-149.468316,8.819693],[-142.15318,3.741443],[-134.922845,-1.396838],[-127.660888,-6.512815],[-120.250934,-11.523088],[-110.056944,-2.811371],[-107.069051,12.039321],[-118.141833,20.303418],[-125.314828,22.509761],[-132.702476,24.389053],[-140.290513,25.90038],[-148.048372,27.007913],[-155.929898,27.683528]]]

```
def get_bgc_selection_profiles(startDate, endDate, shape, xaxis, yaxis, presRange=None,
printUrl=True):
    url = 'https://argovis.colorado.edu/selection/bgc_data_selection'
    url += '?startDate={}'.format(startDate)
    url += '&endDate={}'.format(endDate)
    url += '&shape={}'.format(shape)
    url += '&xaxis={}'.format(xaxis)
    url += '&yaxis={}'.format(yaxis)
    if presRange:
        pressRangeQuery = '&presRange='.format(presRange)
        url += pressRangeQuery
    url = url.replace(' ', '')
    if printUrl:
        print(url)
    resp = requests.get(url)
    # Consider any status other than 2xx an error
    if not resp.status_code // 100 == 2:
        return "Error: Unexpected response {}".format(resp)
    selectionProfiles = resp.json()
    return selectionProfiles
```

```
startDate = '2020-03-01'
endDate = '2020-03-11'
presRange = [0, 50]
shape = [[[-155.929898,27.683528],[-156.984448,13.752725],[-149.468316,8.819693],
[-142.15318,3.741443],[-134.922845,-1.396838],\
          [-127.660888,-6.512815],[-120.250934,-11.523088],[-110.056944,-2.811371],
[-107.069051,12.039321],[-118.141833,20.303418],\
          [-125.314828,22.509761],[-132.702476,24.389053],[-140.290513,25.90038],
[-148.048372,27.007913],[-155.929898,27.683528]]]
xaxis='doxy'
yaxis='pres'
profiles = get_bgc_selection_profiles(startDate, endDate, shape, xaxis, yaxis,
presRange, printUrl=True)
```

```
https://argovis.colorado.edu/selection/bgc_data_selection?startDate=2020-03-
01&endDate=2020-03-11&shape=[[[-155.929898,27.683528],[-156.984448,13.752725],
[-149.468316,8.819693],[-142.15318,3.741443],[-134.922845,-1.396838],
[-127.660888,-6.512815],[-120.250934,-11.523088],[-110.056944,-2.811371],
[-107.069051,12.039321],[-118.141833,20.303418],[-125.314828,22.509761],
[-132.702476,24.389053],[-140.290513,25.90038],[-148.048372,27.007913],
[-155.929898,27.683528]]]&xaxis=doxy&yaxis=pres&presRange=
```

```
df = json2dataframe(profiles, 'bgcMeas')
```

```
df.head()
```

	doxy	doxy_qc	pres	pres_qc	_id	POSITIONING_SYSTEM	DATA_
0	203.949081	1	7.460000	8	5906046_32	GPS	
1	203.940735	1	11.550000	8	5906046_32	GPS	
2	203.849197	1	16.520000	8	5906046_32	GPS	
3	203.733582	1	21.540001	8	5906046_32	GPS	
4	203.752197	1	26.670000	8	5906046_32	GPS	

Big Arrays, Fast: Profiling Cloud Storage Read Throughput

By Ryan Abernathey

Introduction

Many geoscience problems involve analyzing hundreds of Gigabytes (GB), many Terabytes (TB = 100 GB), or even a Petabyte (PB = 1000 TB) of data. For such data volumes, downloading data to personal computers is inefficient. Data-proximate computing, possibly using a parallel computing cluster, is one way to overcome this challenge. With data-proximate computing, a user access a remote computer which is connected tightly to shared, high-performance storage, enabling rapid processing without the need for a download step.

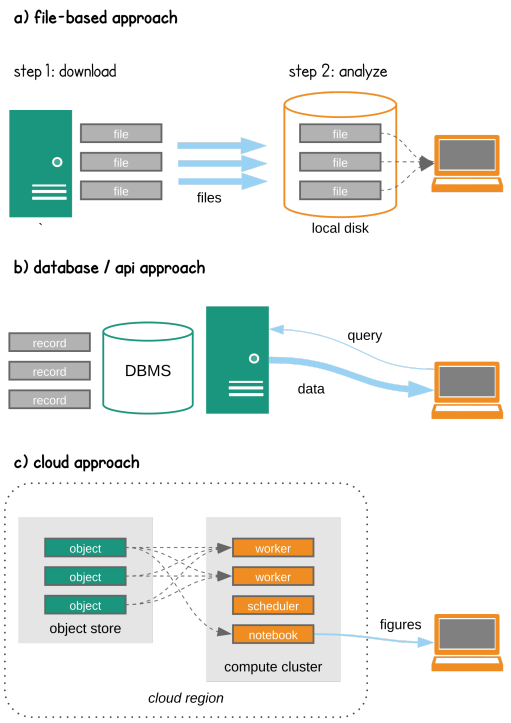


Figure from *Data Access Modes in Science*, by Ryan Abernathey. http://dx.doi.org/10.6084/M9.FIGSHARE.11987466.V1

The cloud offers many exciting new possibilities for scientific data analysis. It also brings new technical challenges. Many scientific users are accustomed to the environments found on High-Performance Computing (HPC) systems, such as NSF's XSEDE Computers or NCAR's Cheyenne system. These systems combine parallel computing with a high-performance shared filesystem, enabling massively parallel data analytics. When shifting workloads to the cloud, one of the biggest challenges is how to store data: *cloud computing does not easily support large shared filesystems, as commonly found on HPC systems*. The preferred way to store data in the cloud is using *cloud object storage*, such as Amazon S3 or Google Cloud Storage.

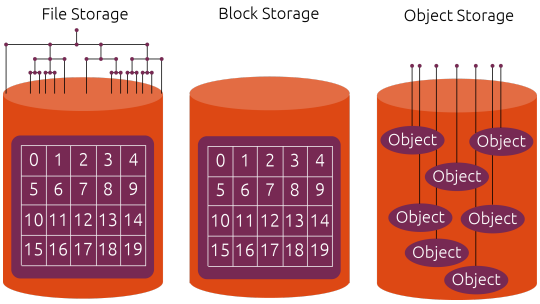


Image via Ubuntu Website, ©2020 Canonical Ltd. Ubuntu, Used with Permission

Cloud object storage is essentially a key/value storage system. They keys are strings, and the values are bytes of data. Data is read and written using HTTP calls. The performance of object storage is very different from file storage. On one hand, each individual read / write to object storage has a high overhead (10-100 ms), since it has to go over the network. On the other hand, object storage "scales out" nearly infinitely, meaning that we can make hundreds, thousands, or millions of concurrent reads / writes. This makes object storage well suited for distributed data analytics. However, the software architecture of a data analysis system must be adapted to take advantage of these properties.

The goal of this notebook is to demonstrate one software stack that can effectively exploit the scalable nature of cloud storage and computing for data processing. There are three key elements:

- Zarr - a library for the storage of multi-dimensional arrays using compressed chunks.

- Filesystem Spec - an abstraction layer for working with different storage technologies. Specific implementations of filesystem spec (e.g. gcsfs, Google Cloud Storage Filesystem) allow Zarr to read and write data from object storage.
- Dask - a distributed computing framework that allows us to read and write data in parallel asynchronously.

Accessing the Data

For this demonstration, we will use data stored in Google Cloud Storage. The data come from the GFDL CM2.6 high-resolution climate model. They are catalogged on the Pangeo Cloud Data Catalog here:

- https://catalog.pangeo.io/browse/master/ocean/GFDL_CM2_6/

The catalog returns an http://xarray.pydata.org/ object pointing to the data on google cloud:

```
import dask.array as dsa
import fsspec
import numpy as np
import pandas as pd
from contextlib import contextmanager
import pandas as pd
import intake
import time
import seaborn as sns
import dask
from matplotlib import pyplot as plt
```

```
from intake import open_catalog
cat = open_catalog("https://raw.githubusercontent.com/pangeo-data/pangeo-datastore/master/intake-catalogs/ocean/GFDL_CM2.6.yaml")
item = cat["GFDL_CM2_6_control_ocean"]
ds = item.to_dask()
ds
```

For this example, we are not interested in Xarray's advanced data analytics features, so we will work directly with the raw array data. Here we examine the data from the `temp` variable (ocean temperature):

```
data = ds.temp.data
data
```

Before dropping down to the dask level, we take a quick peek at the data using Xarray's built-in visualization capabilities.

```
ds.temp[0, 0].plot(figsize=(16, 8), center=False, robust=True)
```

Side note: We can open this array directly with dask, bypassing xarray, if we know the path on the object store. (These give identical results.)

```
item.urlpath
```

```
dsa.from_zarr(fsspec.get_mapper("gs://cmip6/GFDL_CM2_6/control/ocean/temp"))
```

Some important observations:

- The data is organized into 2400 distinct chunks–each one corresponds to an individual object in Google Cloud Storage
- The in-memory size of each chunk is 194.4 MB
- The chunks are contiguous across the latitude and longitude dimensions (axes 2 and 3) and span 5 vertical levels (axis 1). Each timestep (axis 0) is in a different chunk.
- Zarr + Dask know how to virtually combine these chunks into a single 4-dimensional array.

Set Up Benchmarking

Now we want to see how fast we can load data from the object store. *We are not interested in any computations.* Here the goal is simply to measure read performance and how it scales. In order to do this, we employ a trick: we store the data into a virtual storage target that simply discards the data, similar to piping data to `/dev/null` on a filesystem.

To do this, we define the following storage class:

```
class DevNullStore:

    def __init__(self):
        pass

    def __setitem__(*args, **kwargs):
        pass
```

This is basically a dictionary that forgets whatever you put in it:

```
null_store = DevNullStore()
# this line produces no error but actually does nothing
null_store['foo'] = 'bar'
```

Now let's see how long it takes to store one chunk of data into this null storage target. This essentially measures the read time.

```
%time dsa.store(data[0, :5], null_store, lock=False)
```

Now we want to scale this out to see how the read time scales as a function of number of parallel reads. For this, we need a Dask cluster. In our Pangeo environment, we can use Dask Gateway to create a Dask cluster. But there is an analogous way to do this on nearly any distributed computing environment. HPC-style environments (PBS, Slurm, etc.) are supported via Dask Jobqueue. Most cloud environments can work with Dask Kubernetes.

```
from dask_gateway import Gateway
from dask.distributed import Client

gateway = Gateway()
cluster = gateway.new_cluster()
cluster
```

```
client = Client(cluster)
client
```

Finally we create a context manager to help us keep track of the results of our benchmarking

```
class DiagnosticTimer:
    def __init__(self):
        self.diagnostics = []

    @contextmanager
    def time(self, **kwargs):
        tic = time.time()
        yield
        toc = time.time()
        kwargs["runtime"] = toc - tic
        self.diagnostics.append(kwargs)

    def dataframe(self):
        return pd.DataFrame(self.diagnostics)

diag_timer = DiagnosticTimer()
```

Do Benchmarking

We want to keep track of some information about our array. Here we figure out the size (in bytes) of the chunks.

```
chunksize = np.prod(data.chunksize) * data.dtype.itemsize
chunksize
```

And about how many workers / threads are in the cluster. The current cluster setup uses two threads and approximately 4 GB of memory per worker. These settings are the defaults for the current Pangeo environment and have not been explored extensively. We do not expect high sensitivity to these parameters here, as the task is I/O limited, not CPU limited.

```
def total_nthreads():
    return sum([v for v in client.nthreads().values()])

def total_ncores():
    return sum([v for v in client.ncores().values()])

def total_workers():
    return len(client.ncores())
```

This is the main loop where we time a distributed read for different numbers of worker counts. We also keep track of some other metadata about our testing conditions.

```
diag_kwargs = dict(nbytes=data.nbytes, chunksize=chunksize,
                   cloud='gcloud', format='zarr')


for nworkers in [30, 20, 10, 5]:
    cluster.scale(nworkers)
    time.sleep(10)
    client.wait_for_workers(nworkers)
    print(nworkers)
    with diag_timer.time(nthreads=total_nthreads(),
                         ncores=total_ncores(),
                         nworkers=total_workers(),
                         **diag_kwargs):

        future = dsa.store(data, null_store, lock=False, compute=False)
        dask.compute(future, retries=5)
```

```
client.close()
cluster.close()
```

```
df = diag_timer.dataframe()
df
```

```
df.plot(x='ncores', y='runtime', marker='o')
```

```
df['throughput_MBps'] = df.nbytes / 1e6 / df['runtime']
df.plot(x='ncores', y='throughput_MBps', marker='o')
```

```
df['throughput_MBps_per_thread'] = df.throughput_MBps / df.nthreads
df.plot(x='nthreads', y='throughput_MBps_per_thread', marker='o')
plt.ylim([0, 100])
```

Compare Many Results

Similar tests to the one above were performed with many different cloud storage technologies and data formats:

- Google Cloud Storage + Zarr Format (like the example above)
- Google Cloud Storage + NetCDF Format (NetCDF files were opened directly from object storage using `gcsfs` to provide file-like objects to `h5py`)
- Wasabi Cloud + Zarr Format (A discount cloud-storage vendor)
- Jetstream Cloud + Zarr Format (NSF's Cloud)
- Open Storage Network (OSN) NCSA Storage Pod + Zarr Format
- The NASA ECCO Data Portal (data read using `xmitgcm.llcreader`)
- Various OPeNDAP Servers, including:
  - The UCAR ESGF Node TDS Server
  - NOAA ESRL PSL TDS Server

The results have been archived on Zenodo: DOI 10.5281/zenodo.3829032

Despite the different sources and formats, all tests were conducted in the same basic manner:

1. A "lazy" dask array referencing the source data was created
2. A dask cluster was scaled up to different sizes
3. The array was "stored" into the `NullStore`, forcing all data to read from the source

Note: All computing was done in Google Cloud.

```
url_base = 'https://zenodo.org/record/3829032/files/'
files = {'GCS Zarr': ['gcloud-test-1.csv', 'gcloud-test-3.csv', 'gcloud-test-4.csv',
                      'gcloud-test-5.csv', 'gcloud-test-6.csv'],
         'GCS NetCDF': ['netcdf-test-1.csv'],
         'Wasabi Zarr': ['wasabi-test-1.csv', 'wasabi-test-2.csv', 'wasabi-test-
3.csv'],
         'Jetstream Zarr': ['jetstream-1.csv', 'jetstream-2.csv'],
         'ECCO Data Portal': ['ecco-data-portal-1.csv', 'ecco-data-portal-2.csv'],
         'ESGF UCAR OPeNDAP': ['esgf-ucar-1.csv', 'esgf-ucar-2.csv'],
         'NOAA ESRL OPeNDAP': ['noaa-esrl-1.csv', 'noaa-esrl-2.csv'],
         'OSN Zarr': ['OSN-test-1.csv', 'OSN-test-2.csv']
        }

data = {}
for name, fnames in files.items():
    data[name] = pd.concat([pd.read_csv(f'{url_base}/{fname}') for fname in fnames])
```

```
data['OSN Zarr']
```

```
fig0, ax0 = plt.subplots()
fig1, axs = plt.subplots(nrows=2, ncols=4, figsize=(20, 8))
palette = sns.color_palette('colorblind', len(files))

for (name, df), color, ax1 in zip(data.items(), palette, axs.flat):
    for ax in [ax0, ax1]:
        df.plot(kind='scatter', x='ncores', y='throughput_MBps',
                s=50, c=[color], edgecolor='k', ax=ax, label=name)
    ax1.grid()
    ax1.set_title(name)

ax0.grid()
ax0.legend(loc='upper left')

fig0.tight_layout()
fig1.tight_layout()
```

Discussion

The figure above shows a wide range of throughput rates and scaling behavior. There are a great many factors which might influence the results, including:

- The structure of the source data (e.g. data format, chunk size, compression, etc.)
- The rate at which the storage service can read and process data from its own disks
- The transport protocol and encoding (compression, etc.)
- The network bandwidth between the source and Google Cloud US-CENTRAL1 region
- The overhead of the dask cluster in computing many tasks simultaneously

The comparison is far from “fair”: it is clearly biased towards Google Cloud Storage, which has optimum network proximity to the compute location. Nevertheless, we make the following observations.

Zarr is about 10x faster than NetCDF in Cloud Object Storage

Using 40 cores (20 dask workers), we were able to pull netCDF data from Google Cloud Storage at a rate of about 500 MB/s. Using the Zarr format, we could get to 5000 MB/s (5 GB/s) for the same number of dask workers.

Google Cloud Storage Scales

Both formats, however, showed good scaling, with throughput increasingly nearly linearly with the size of the cluster. The largest experiment we attempted used nearly 400 cores: we were able to pull data from GCS at roughly 20 GB/s. That’s pretty fast! At this rate, we could process a Petebyte of data in about 14 hours. However, there is a knee in the GCS-Zarr curve around 150 cores, with the scaling becoming slightly poorer above that value. We hypothesize that this is due to Dask’s overhead from handling a very large task graph.

External Cloud Storage Providers Scale…then Saturate

We tested three cloud storage providers *outside* of Google Cloud: Wasabi (a commercial service), Jetstream (an NSF-operated cloud), and OSN (an NSF-sponsored storage provider). These curves all show a similar pattern: good scaling for low worker counts, then a plateau. We interpret this a saturation of the network bandwidth between the data and the compute location. Of these three, Jetstream saturated at the lowest value (2 GB/s), then Wasabi (3.5 GB/s, but possibly

not fully saturated yet), then OSN (5.5 GB/s). Also noteworthy is the fact that, for smaller core counts, OSN was actually *faster* than GCS *within Google Cloud*. These results are likely highly sensitive to network topology. However, they show unambiguously that OSN is an attractive choice for cloud-style storage and on-demand big-data processing.

OPeNDAP is Slow

The fastest we could get data out of an OPeNDAP server was 100 MB/s (UCAR ESGF Node). The NOAA ESRL server was more like 20 MB/s. These rates are many orders of magnitude than cloud storage. Perhaps these servers have throttling in place to limit their total bandwidth. Or perhaps the OPeNDAP protocol itself has some fundamental inefficiencies. OPeNDAP remains a very convenient protocol for remote data access; if we wish to use it for Big Data and distributed processing, data providers need to find some way to speed it up.

Conclusions

We demonstrated a protocol for testing the throughput and scalability Array data storage with the Dask computing framework. Using Google Cloud as our compute platform, we found, unsurprisingly, that a cloud-optimized storage format (Zarr), used with in-region cloud storage, provided the best raw throughput (up to 20 GB/s in our test) and scalability. We found that the same storage format also allowed us to transfer data on-demand from other clouds at reasonable rates (5 GB/s) for modest levels of parallelism. (Care must be taken in this case, however, because many cloud providers charge egress fees for data leaving their network.) Combined with Zarr, all of the cloud object storage platforms were able to deliver data orders of magnitude faster than traditional OPeNDAP servers, the standard remote-data-access protocol for the geosciences.

Future Work

This analysis could be improved in numerous ways. In the future we would like to

- Investigate other cloud-native storage formats, such as TileDB and Cloud-Optimized Geotiff
- Perform deeper analysis of parameter sensitivity (chunk size, compression, etc.)
- Explore other clouds (e.g. AWS, Microstoft Azure)

Finally, we did not attempt to assess the relative costs associated with cloud vs. on-premises data storage, as this involves difficult questions related to total cost of ownership. However, we hope these results are useful for data providers in deciding how to provision future infrastructure for data-intensive geoscience.

Notebook Reviews

Here we include the review of the original version of this notebook and our replies.

Review 1

Evaluation of the Contribution
Overall Recommendation (100%): 9 Total points (out of 10) : 9

Comments for the Authors
“Big Arrays, Fast: Profiling Cloud Storage Read Throughput” is a good quality and, most of all, interesting science meets computer science notebook. It addresses important issues relevant to this scientific arena. The problem statement, discussion and conclusions are all compelling and pertinent to this area of inquiry.

The most important criterion “does it run” is satisfied as I was able to run the notebook to completion on https://ocean.pangeo.io (though I could not run it on Binder for some reason, “KeyError: ‘GFDL_CM2_6_control_ocean’”).

> The error has been corrected in the current version.

I especially liked this notebook for the following reason:

- The “computational essay” aspect receives high marks for having expository prose and computation work together to build a logical conclusion.
- Figures are generated inline with matplotlib
- Figures are easy to understand and interesting (e.g., the set of throughput figures at the end of the notebook).
- The xarray.Dataset meta data allows for inspection via the attributes and data representation icons.
- Computational cells are small and intermixed with expository prose.

Problems with this notebook:

- Quite a few distracting spelling errors: Benachmarks, analagous, simultaneouslly, slighly, saturdation

> Unfortunately, JupyterLab has no spell checker, a feature I have obviously become to reliant upon.

A suggestion for improvement would be to include a reference section with HTML linked DOIs for further inquiry and background material, and perhaps also a future work section.

> A future work section has been included.

Review 2

Evaluation of the Contribution
Overall Recommendation (100%): 8 Total points (out of 10) : 8

Comments for the Authors

- the author should add to the discussion a reason for the choice of threads, cores and workers in the ‘do benchmarking’ section

> This has been done

- while the analysis is designed to provide a coarse grain view of the speed of zarr, it might be a valuable exercise to break the scales down to segments where they can be compared; e.g. of the four zarr tests, the n_cores scales vary from 0..400 (gcs), 0..60 (wasabi, jetstream) and 0..175 (osn) and thus these scales make it difficult to compare them when we do not have data past n_cores=60 and again n_cores=175

> This a reasonable suggestion but beyond what we could accomplish in a brief revision period.

- though the authors acknowledge the tests are not ‘fair’, indeed it would be more valuable to compare various configurations of netcdf to make such comparisons more easily interpretable

> It’s not clear what other “configurations” of netCDF are being suggested here.

- typo: change to “saturation” > We interpret this a saturdation of the network bandwidth between the data and the compute location.

> Thanks, done.

- the abstract should include “preliminary” to situate this work as in-progress and not a full quantitative exploration and assessment of accessing data

> Given the combinatoric explosion of possible configuration choices, is a comprehensive “full quantitative exploration” even possible?

- if at all possible the author should provide instructions for how to execute elsewhere outside the pangeo platform

> We have included some new links to how to deploy dask in other environments.

- licensing information as well as notebook requirements should also be provided with the notebook (or a link to the GH repo if one exists for the notebook)

> This was all included in the original submission.

Multi-Cloud Workflow with Pangeo

This example demonstrates a workflow using analysis-ready data provided in two public clouds.

- LENS (Hosted on AWS in the us-west-2 region)
- ERA5 (Hosted on Google Cloud Platform in multiple regions)

We'll perform a similar analysis on each of the datasets, a histogram of the total precipitation, compare the results. Notably, this computation reduces a large dataset to a small summary. The reduction can happen on a cluster in the cloud.

By placing a compute cluster in the cloud next to the data, we avoid moving large amounts of data over the public internet. The large analysis-ready data only needs to move within a cloud region: from the machines storing the data in an object-store like `S3` to the machines performing the analysis. The compute cluster reduces the large amount of data to a small histogram summary. At just a handful of KBs, the summary statistics can easily be moved back to the local client, which might be running on a laptop. This also avoids costly egress charges from moving large amounts of data out of cloud regions.

```
import getpass

import dask
from distributed import Client
from dask_gateway import Gateway, BasicAuth
import intake
import numpy as np
import s3fs
import xarray as xr
from xhistogram.xarray import histogram
```

Create Dask Clusters

We've deployed Dask Gateway on two Kubernetes clusters, one in AWS and one in GCP. We'll use these to create Dask clusters in the same cloud region as the data. We'll connect to both of them from the same interactive notebook session.

```
password = getpass.getpass()
auth = BasicAuth("pangeo", password)
```

```
 ····
```

```
# Create a Dask Cluster on AWS
aws_gateway = Gateway(
    "http://a00670d37945911eab47102a1da71b1b-524946043.us-west-2.elb.amazonaws.com",
    auth=auth,
)
aws = aws_gateway.new_cluster()
aws_client = Client(aws, set_as_default=False)
aws_client
```

Client

Scheduler: gateway://a00670d37945911eab47102a1da71b1b-524946043.us-west-2.elb.amazonaws.com:80/dask-gateway.ff367abfd96c4465a0782661a254e589
Dashboard: http://a00670d37945911eab47102a1da71b1b-524946043.us-west-2.elb.amazonaws.com/clusters/dask-gateway.ff367abfd96c4465a0782661a254e589/status

Cluster

Workers: 0
Cores: 0
Memory: 0 B

```
# Create a Dask Cluster on GCP
gcp_gateway = Gateway(
    "http://34.72.56.89",
    auth=auth,
)
gcp = gcp_gateway.new_cluster()
gcp_client = Client(gcp, set_as_default=False)
gcp_client
```

Client

Scheduler: gateway://34.72.56.89:80/dask-gateway.02ab011eaa054434916da9d3e3405c6c
Dashboard: http://34.72.56.89/clusters/dask-gateway.02ab011eaa054434916da9d3e3405c6c/status

Cluster

Workers: 0
Cores: 0
Memory: 0 B

We'll enable adaptive mode on each of the Dask clusters. Workers will be added and removed as needed by the current level of computation.

```
aws.adapt(minimum=1, maximum=200)
gcp.adapt(minimum=1, maximum=200)
```

ERA5 on Google Cloud Storage

We'll use `intake` and pangeo's data catalog to discover the dataset.

```
cat = intake.open_catalog(
    "https://raw.githubusercontent.com/pangeo-data/pangeo-datastore/master/intake-catalogs/master.yaml"
)
cat
```

```
<Intake catalog: master>
```

The next cell loads the *metadata* as an xarray dataset. No large amount of data is read or transfered here. It will be loaded on-demand when we ask for a concrete result later.

```
era5 = cat.atmosphere.era5_hourly_reanalysis_single_levels_sa(
    storage_options={"requester_pays": False, "token": "anon"}
).to_dask()
era5
```

xarray.Dataset

▸ Dimensions: (**latitude**: 721, **longitude**: 1440, **time**: 350640)

▾ Coordinates:

latitude	(latitude)	float32	90.0 89.75 89.5 ... -89.75 -90.0	
longitude	(longitude)	float32	0.0 0.25 0.5 ... 359.5 359.75	
time	(time)	datetime64[ns]	1979-01-01 ... 2018-12-31T23:00:00	

▸ Data variables:
(17)

▾ Attributes:

Conventions :	CF-1.6
history :	2019-09-20 05:15:01 GMT by grib_to_netcdf-2.10.0: /opt/ecmwf/eccodes/bin/grib_to_netcdf -o /cache/data7/adaptor.mars.internal-1568954670.105603-18230-3-5ca6e0df-a562-42ba-8b0b-948b2e1815bd.nc /cache/tmp/5ca6e0df-a562-42ba-8b0b-948b2e1815bd-adaptor.mars.internal-1568954670.1062171-18230-2-tmp.grib

We're computing the histogram on the total precipitation for a specific time period. xarray makes selecting this subset of data quite natural. Again, we still haven't loaded the data.

```
tp = era5.tp.sel(time=slice('1990-01-01', '2005-12-31'))
tp
```

xarray.DataArray 'tp' (**time**: 140256, **latitude**: 721, **longitude**: 1440)

dask.array<chunksize=(9, 721, 1440), meta=np.ndarray>

▾ Coordinates:

latitude	(latitude)	float32	90.0 89.75 89.5 ... -89.75 -90.0	
longitude	(longitude)	float32	0.0 0.25 0.5 ... 359.5 359.75	
time	(time)	datetime64[ns]	1990-01-01 ... 2005-12-31T23:00:00	

▾ Attributes:

long_name :	Total precipitation
units :	m

To compare to the 6-hourly LENS dataset, we'll aggregate to 6-hourly totals.

```
# convert to 6-hourly precip totals
tp_6hr = tp.coarsen(time=6).sum()
tp_6hr
```

xarray.DataArray (**time**: 23376, **latitude**: 721, **longitude**: 1440)

dask.array<chunksize=(1, 721, 1440), meta=np.ndarray>

▾ Coordinates:

latitude	(latitude)	float32	90.0 89.75 89.5 ... -89.75 -90.0
longitude	(longitude)	float32	0.0 0.25 0.5 ... 359.5 359.75
time	(time)	datetime64[ns]	1990-01-01T02:30:00 ... 2005-12-31T20:3…

▸ Attributes: (0)

We'll bin this data into the following bins.

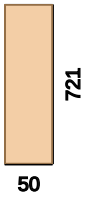
```
tp_6hr_bins = np.concatenate([[0], np.logspace(-5,  0, 50)])
tp_6hr_bins
```

```
array([0.00000000e+00, 1.00000000e-05, 1.26485522e-05, 1.59985872e-05,
       2.02358965e-05, 2.55954792e-05, 3.23745754e-05, 4.09491506e-05,
       5.17947468e-05, 6.55128557e-05, 8.28642773e-05, 1.04811313e-04,
       1.32571137e-04, 1.67683294e-04, 2.12095089e-04, 2.68269580e-04,
       3.39322177e-04, 4.29193426e-04, 5.42867544e-04, 6.86648845e-04,
       8.68511374e-04, 1.09854114e-03, 1.38949549e-03, 1.75751062e-03,
       2.22299648e-03, 2.81176870e-03, 3.55648031e-03, 4.49843267e-03,
       5.68986603e-03, 7.19685673e-03, 9.10298178e-03, 1.15139540e-02,
       1.45634848e-02, 1.84206997e-02, 2.32995181e-02, 2.94705170e-02,
       3.72759372e-02, 4.71486636e-02, 5.96362332e-02, 7.54312006e-02,
       9.54095476e-02, 1.20679264e-01, 1.52641797e-01, 1.93069773e-01,
       2.44205309e-01, 3.08884360e-01, 3.90693994e-01, 4.94171336e-01,
       6.25055193e-01, 7.90604321e-01, 1.00000000e+00])
```

The next cell applies the histogram to the `longitude` dimension and takes the mean over `time`. We're still just building up the computation here, we haven't actually loaded the data or executed it yet.

```
tp_hist = histogram(
    tp_6hr.rename('tp_6hr'), bins=[tp_6hr_bins], dim=['longitude']
).mean(dim='time')
tp_hist.data
```

	Array	Chunk
Bytes	288.40 kB	288.40 kB
Shape	(721, 50)	(721, 50)
Count	110889 Tasks	1 Chunks
Type	float64	numpy.ndarray



In total, we're going from the ~1.5TB raw dataset down to a small 288 kB result that is the histogram summarizing the total precipitation. We've built up a large sequence of operations to do that reduction (over 110,000 individual tasks), and now it's time to actually execute it. There will be some delay between running the next cell, the scheduler receiving the task graph, and the cluster starting to process it, but work is happening in the background. After a minute or so, tasks will start appearing on the Dask dashboard.

One thing to note: we request this result with the `gcp_client`, the client for the cluster in the same cloud region as the data.

```
era5_tp_hist_ = gcp_client.compute(tp_hist, retries=5)
```

`gcp_tp_hist_` is a `Future` pointing to the result on the cluster. The actual computation is happening in the background, and we'll call `.result()` to get the concrete result later on.

```
era5_tp_hist_
```

Future: finalize status: pending, key: finalize-827454f3f45ccd1c7c22f0b3907c098c

Because the Dask cluster is in adaptive mode, this computation has kicked off a chain of events: Dask has noticed that it suddenly has many tasks to compute, so it asks the cluster manager (Kubernetes in this case) for more workers. THe Kubernetes cluster then asks it's compute backend (Google Compute Engine in this case) for more virtual machines. As these machines come online, our workers will come to life and the cluster will start progressing on our computation.

LENS on AWS

This computation is very similar to the ERA5 computation. The primary difference is that the LENS dataset is an ensemble. We'll histogram a single member of that ensemble.

The Intake catalog created by NCAR includes many things, so we'll use `intake-esm` to search for the URL we want.

```
col = intake.open_esm_datastore(
    "https://raw.githubusercontent.com/NCAR/cesm-lens-aws/master/intake-catalogs/aws-cesm1-le.json"
)
res = col.search(frequency='hourly6-1990-2005', variable='PRECT')
res.df
```

	component	frequency	experiment	variable	path
0	atm	hourly6-1990-2005	20C	PRECT	s3://ncar-cesm-lens/atm/hourly6-1990-2005/cesm...

```
url = res.df.loc[0, "path"]
url
```

```
's3://ncar-cesm-lens/atm/hourly6-1990-2005/cesmLE-20C-PRECT.zarr'
```

We'll (lazily) load that data from S3 using s3fs, xarray, and zarr.

```
fs = s3fs.S3FileSystem(anon=True)
lens = xr.open_zarr(fs.get_mapper(url), consolidated=True)
lens
```

xarray.Dataset

▸ Dimensions: (**ilev**: 31, **lat**: 192, **lev**: 30, **lon**: 288, **member_id**: 36, nbnd: 2, **slat**: 191, **slon**: 288, **time**: 23360)

▾ Coordinates:

ilev	(ilev)	float64	2.255 5.032 10.16 ... 985.1 1e+03
lat	(lat)	float64	-90.0 -89.06 -88.12 ... 89.06 90.0
lev	(lev)	float64	3.643 7.595 14.36 ... 976.3 992.6
lon	(lon)	float64	0.0 1.25 2.5 ... 356.2 357.5 358.8
member_id	(member_id)	int64	1 2 3 4 5 6 ... 31 32 33 34 35 104
slat	(slat)	float64	-89.53 -88.59 ... 88.59 89.53
slon	(slon)	float64	-0.625 0.625 1.875 ... 356.9 358.1
time	(time)	object	1990-01-01 06:00:00 ... 2006-01-01 00:00:00

▸ Data variables:
(33)

▾ Attributes:

Conventions :	CF-1.0
NCO :	4.3.4
Version :	$Name$
history :	2019-08-01 00:15:18.487461 xarray.open_dataset('/glade/collections/cdg/data/cesmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC5CNBDRD.f09_g16.001.cam.h2.PRECT.1990010100Z-2005123118Z.nc') 2019-08-01 00:15:19.080785 xarray.open_dataset('/glade/collections/cdg/data/cesmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC5CNBDRD.f09_g16.002.cam.h2.PRECT.1990010100Z-2005123118Z.nc') 2019-08-01 00:15:20.252396 xarray.open_dataset('/glade/collections/cdg/data/cesmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC5CNBDRD.f09_g16.003.cam.h2.PRECT.1990010100Z-2005123118Z.nc') 2019-08-01 00:15:20.787281 xarray.open_dataset('/glade/collections/cdg/data/cesmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC5CNBDRD.f09_g16.004.cam.h2.PRECT.1990010100Z-2005123118Z.nc') 2019-08-01 00:15:21.279874 xarray.open_dataset('/glade/collections/cdg/data/cesmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC5CNBDRD.f09_g16.005.cam.h2.PRECT.1990010100Z-2005123118Z.nc') 2019-08-01 00:15:21.850205 xarray.open_dataset('/glade/collections/cdg/data/cesmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC5CNBDRD.f09_g16.006.cam.h2.PRECT.1990010100Z-2005123118Z.nc') 2019-08-01 00:15:22.423595 xarray.open_dataset('/glade/collections/cdg/data/cesmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC5CNBDRD.f09_g16.007.cam.h2.PRECT.1990010100Z-2005123118Z.nc') 2019-08-01 00:15:23.127816 xarray.open_dataset('/glade/collections/cdg/data/cesmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC5CNBDRD.f09_g16.008.cam.h2.PRECT.1990010100Z-2005123118Z.nc') 2019-08-01 00:15:23.695110 xarray.open_dataset('/glade/collections/cdg/data/cesmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC5CNBDRD.f09_g16.009.cam.h2.PRECT.1990010100Z-2005123118Z.nc') 2019-08-01 00:15:24.291352 xarray.open_dataset('/glade/collections/cdg/data/cesmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC5CNBDRD.f09_g16.010.cam.h2.PRECT.1990010100Z-2005123118Z.nc') 2019-08-01 00:15:24.873420 xarray.open_dataset('/glade/collections/cdg/data/cesmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC5CNBDRD.f09_g16.011.cam.h2.PRECT.1990010100Z-2005123118Z.nc') 2019-08-01 00:15:25.512516 xarray.open_dataset('/glade/collections/cdg/data/cesmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC5CNBDRD.f09_g16.012.cam.h2.PRECT.1990010100Z-2005123118Z.nc') 2019-08-01 00:15:26.061289 xarray.open_dataset('/glade/collections/cdg/data/cesmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC5CNBDRD.f09_g16.013.cam.h2.PRECT.1990010100Z-2005123118Z.nc') 2019-08-01 00:15:26.662665 xarray.open_dataset('/glade/collections/cdg/data/cesmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC

5CNBDRD.f09_g16.014.cam.h2.PRECT.1990010100Z-2005123118Z.nc')
2019-08-01 00:15:27.243923 xarray.open_dataset('/glade/collections/cdg/data/c
esmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC
5CNBDRD.f09_g16.015.cam.h2.PRECT.1990010100Z-2005123118Z.nc')
2019-08-01 00:15:27.799712 xarray.open_dataset('/glade/collections/cdg/data/c
esmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC
5CNBDRD.f09_g16.016.cam.h2.PRECT.1990010100Z-2005123118Z.nc')
2019-08-01 00:15:28.350833 xarray.open_dataset('/glade/collections/cdg/data/c
esmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC
5CNBDRD.f09_g16.017.cam.h2.PRECT.1990010100Z-2005123118Z.nc')
2019-08-01 00:15:28.882690 xarray.open_dataset('/glade/collections/cdg/data/c
esmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC
5CNBDRD.f09_g16.018.cam.h2.PRECT.1990010100Z-2005123118Z.nc')
2019-08-01 00:15:29.612376 xarray.open_dataset('/glade/collections/cdg/data/c
esmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC
5CNBDRD.f09_g16.019.cam.h2.PRECT.1990010100Z-2005123118Z.nc')
2019-08-01 00:15:30.142923 xarray.open_dataset('/glade/collections/cdg/data/c
esmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC
5CNBDRD.f09_g16.020.cam.h2.PRECT.1990010100Z-2005123118Z.nc')
2019-08-01 00:15:32.677487 xarray.open_dataset('/glade/collections/cdg/data/c
esmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC
5CNBDRD.f09_g16.021.cam.h2.PRECT.1990010100Z-2005123118Z.nc')
2019-08-01 00:15:33.314355 xarray.open_dataset('/glade/collections/cdg/data/c
esmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC
5CNBDRD.f09_g16.022.cam.h2.PRECT.1990010100Z-2005123118Z.nc')
2019-08-01 00:15:35.416995 xarray.open_dataset('/glade/collections/cdg/data/c
esmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC
5CNBDRD.f09_g16.023.cam.h2.PRECT.1990010100Z-2005123118Z.nc')
2019-08-01 00:15:37.400624 xarray.open_dataset('/glade/collections/cdg/data/c
esmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC
5CNBDRD.f09_g16.024.cam.h2.PRECT.1990010100Z-2005123118Z.nc')
2019-08-01 00:15:40.313590 xarray.open_dataset('/glade/collections/cdg/data/c
esmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC
5CNBDRD.f09_g16.025.cam.h2.PRECT.1990010100Z-2005123118Z.nc')
2019-08-01 00:15:42.594527 xarray.open_dataset('/glade/collections/cdg/data/c
esmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC
5CNBDRD.f09_g16.026.cam.h2.PRECT.1990010100Z-2005123118Z.nc')
2019-08-01 00:15:44.729537 xarray.open_dataset('/glade/collections/cdg/data/c
esmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC
5CNBDRD.f09_g16.027.cam.h2.PRECT.1990010100Z-2005123118Z.nc')
2019-08-01 00:15:46.637571 xarray.open_dataset('/glade/collections/cdg/data/c
esmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC
5CNBDRD.f09_g16.028.cam.h2.PRECT.1990010100Z-2005123118Z.nc')
2019-08-01 00:15:48.589381 xarray.open_dataset('/glade/collections/cdg/data/c
esmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC
5CNBDRD.f09_g16.029.cam.h2.PRECT.1990010100Z-2005123118Z.nc')
2019-08-01 00:15:50.705311 xarray.open_dataset('/glade/collections/cdg/data/c
esmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC
5CNBDRD.f09_g16.030.cam.h2.PRECT.1990010100Z-2005123118Z.nc')
2019-08-01 00:15:51.206031 xarray.open_dataset('/glade/collections/cdg/data/c
esmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC
5CNBDRD.f09_g16.031.cam.h2.PRECT.1990010100Z-2005123118Z.nc')
2019-08-01 00:15:51.683246 xarray.open_dataset('/glade/collections/cdg/data/c
esmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC
5CNBDRD.f09_g16.032.cam.h2.PRECT.1990010100Z-2005123118Z.nc')
2019-08-01 00:15:52.156426 xarray.open_dataset('/glade/collections/cdg/data/c
esmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC
5CNBDRD.f09_g16.033.cam.h2.PRECT.1990010100Z-2005123118Z.nc')
2019-08-01 00:15:54.220732 xarray.open_dataset('/glade/collections/cdg/data/c
esmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC
5CNBDRD.f09_g16.034.cam.h2.PRECT.1990010100Z-2005123118Z.nc')
2019-08-01 00:15:56.793005 xarray.open_dataset('/glade/collections/cdg/data/c

esmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC5CNBDRD.f09_g16.035.cam.h2.PRECT.1990010100Z-2005123118Z.nc')
2019-08-01 00:15:58.913802 xarray.open_dataset('/glade/collections/cdg/data/cesmLE/CESM-CAM5-BGC-LE/atm/proc/tseries/hourly6/PRECT/b.e11.B20TRC5CNBDRD.f09_g16.104.cam.h2.PRECT.1990010100Z-2005123118Z.nc')
2019-08-01 00:16:06.104904 xarray.concat(<ALL_MEMBERS>, dim='member_id', coords='minimal')

important_note :	This data is part of the project 'Blind Evaluation of Lossy Data-Compression in LENS'. Please exercise caution before using this data for other purposes.
nco_openmp_t…	1
revision_Id :	Id
source :	CAM
title :	UNSET

```
hour = 60*60

precip_in_m = lens.PRECT * (6 * hour)
precip_in_m
```

xarray.DataArray 'PRECT' (**member_id**: 36, **time**: 23360, **lat**: 192, **lon**: 288)

dask.array<chunksize=(2, 504, 192, 288), meta=np.ndarray>

▾ Coordinates:

lat	(lat)	float64	-90.0 -89.06 -88.12 ... 89.06 90.0
lon	(lon)	float64	0.0 1.25 2.5 ... 356.2 357.5 358.8
member_id	(member_id)	int64	1 2 3 4 5 6 ... 31 32 33 34 35 104
time	(time)	object	1990-01-01 06:00:00 ... 2006-01-01 00:00:00

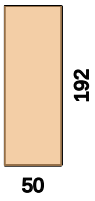
▸ Attributes: (0)

We'll select the first member for comparison with the ERA5 histogram.

```
lens_hist = histogram(
    precip_in_m.isel(member_id=0).rename("tp_6hr"),
    bins=[tp_6hr_bins], dim=["lon"]
).mean(dim=('time'))
```

```
lens_hist.data
```

	Array	Chunk
Bytes	76.80 kB	76.80 kB
Shape	(192, 50)	(192, 50)
Count	2369 Tasks	1 Chunks
Type	float64	numpy.ndarray



Note that we're using the `aws_client`, because LENS is stored in an AWS region.

```
lens_hist_ = aws_client.compute(lens_hist)
```

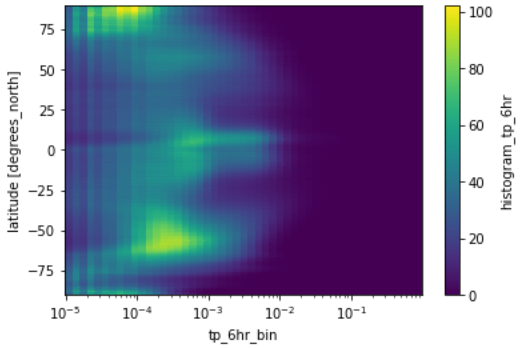
Compare results

Let's plot the histograms for both the ERA5 and LENS data. These are small results so it's safe to transfer the result from the cluster to the client machine for plotting.

```
lens_tp_hist_ = lens_hist_.result()
era5_tp_hist_ = era5_tp_hist_.result()
```

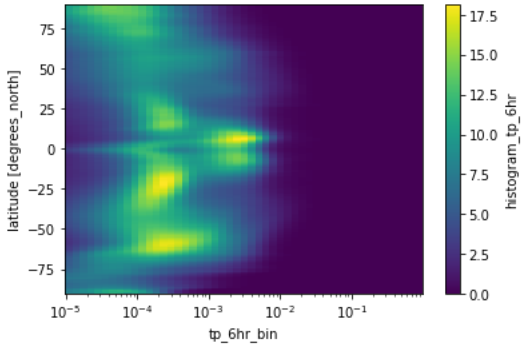
For ERA5:

```
era5_tp_hist_[: ,1:].plot(xscale='log');
```



And for LENS:

```
lens_tp_hist_[: ,1:].plot(xscale='log');
```



Cleanup

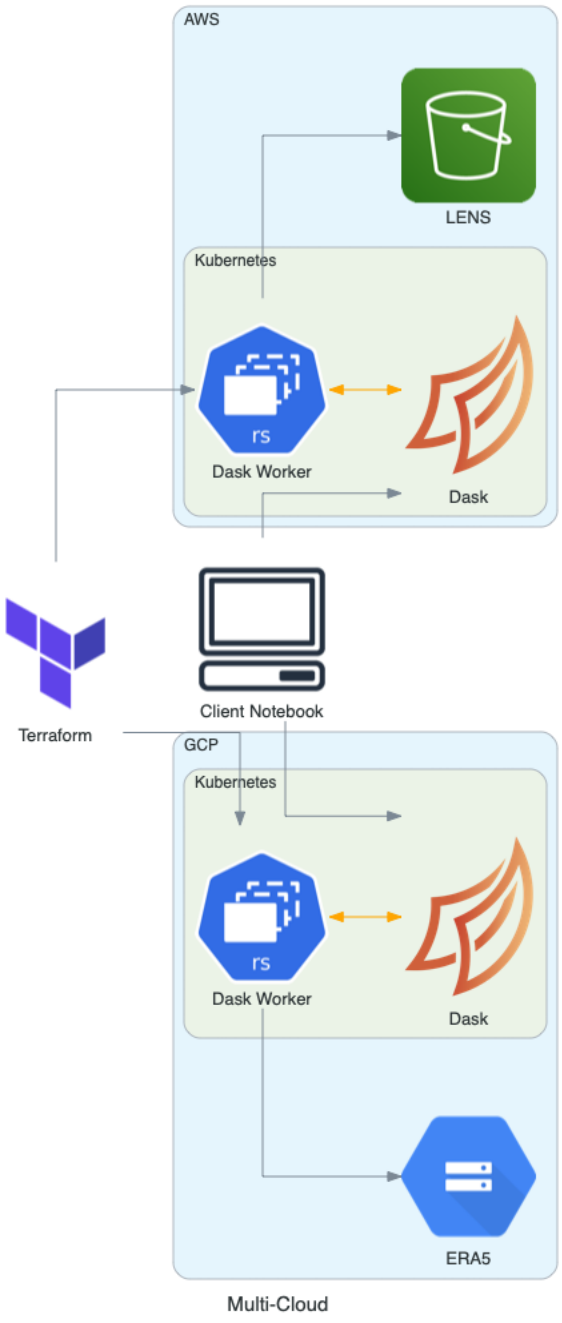
Closing the clients will free all our resources.

```
aws_client.close()
aws.close()

gcp_client.close()
gcp.close()
```

Behind the Scenes

We deployed some infrastructure to make this notebook runnable. In line with one of Pangeo's guiding principles, each of these technologies has an [open architechture](#).



From low-level to high-level

- **Terraform** provides the tools for provisioning the cloud resources needed for the clusters.
- **Kubernetes** provides the container orchestration for deploying the Dask Clusters. We created kubernetes clusters in AWS's `us-west-2` and GCP's `us-central1` regsions.
- **Dask-Gatway** provides centralized, secure access to Dask Clusters. These clusters were deployed using helm on two Kubernetes clusters.
- **Dask** provides scalable, distributed computation for analyzing these large datasets

- **xarray** provides high-level APIs and high-performance data structures for working with this data
- **Intake, gcsfs, s3fs** provide catalogs for data discover and libraries for loading that data
- **Jupyterlab** provides a user interface for interactive computing. The client laptop interacts with the clusters through Jupyterlab.

All of the resources for this demo are available at https://github.com/pangeo-data/multicloud-demo.

PmagPy Online: Jupyter Notebooks, the PmagPy Software Package and the Magnetics Information Consortium (MagIC) Database

Lisa Tauxe\(^1\), Rupert Minnett\(^2\), Nick Jarboe\(^1\), Catherine Constable\(^1\), Anthony Koppers\(^2\), Lori Jonestrask\(^1\), Nick Swanson-Hysell\(^3\)

\(^1\)Scripps Institution of Oceanography, United States of America; \(^2\) Oregon State University; \(^3\) University of California, Berkely; ltauxe@ucsd.edu

The Magnetics Information Consortium (MagIC), hosted at http://earthref.org/MagIC is a database that serves as a Findable, Accessible, Interoperable, Reusable (FAIR) archive for paleomagnetic and rock magnetic data. It has a flexible, comprehensive data model that can accomodate most kinds of paleomagnetic data. The **PmagPy** software package is a cross-platform and open-source set of tools written in Python for the analysis of paleomagnetic data that serves as one interface to MagIC, accommodating various levels of user expertise. It is available through github.com/PmagPy. Because PmagPy requires installation of Python, several non-standard Python modules, and the PmagPy software package, there is a speed bump for many practitioners on beginning to use the software. In order to make the software and MagIC more accessible to the broad spectrum of scientists interested in paleo and rock magnetism, we have prepared a set of Jupyter notebooks, hosted on jupyterhub.earthref.org which serve a set of purposes. 1) There is a complete course in Python for Earth Scientists, 2) a set of notebooks that introduce PmagPy (pulling the software package from the github repository) and illustrate how it can be used to create data products and figures for typical papers, and 3) show how to prepare data from the laboratory to upload into the MagIC database. The latter will satisfy expectations from NSF for data archiving and for example the AGU publication data archiving requirements.

Getting started

- To use the PmagPy notebooks online, go to website at https://jupyterhub.earthref.org/. Create an Earthref account using your ORCID and log on. [This allows you to keep files in a private work space.]
- Open the PmagPy Online - Setup notebook and execute the two cells. Then click on File = > Open and click on the PmagPy_Online folder. Open the PmagPy_online notebook and work through the examples. There are other notebooks that are useful for the working paleomagnetist.
- Alternatively, you can install Python and the PmagPy software package on your computer (see https://earthref.org/PmagPy/cookbook for instructions). Follow the instructions for “Full PmagPy install and update” through section 1.4 (Quickstart with PmagPy notebooks). This notebook is in the collection of PmagPy notebooks.

Overview of MagIC

The Magnetics Information Consortium (MagIC), hosted at http://earthref.org/MagIC is a database that serves as a Findable, Accessible, Interoperable, Reusable (FAIR) archive for paleomagnetic and rock magnetic data. Its datamodel is fully described here: https://www2.earthref.org/MagIC/data-models/3.0. Each contribution is associated with a publication via the DOI. There are nine data tables:

- contribution: metadata of the associated publication.
- locations: metadata for locations, which are groups of sites (e.g., stratigraphic section, region, etc.)
- sites: metadata and derived data at the site level (units with a common expectation)
- samples: metadata and derived data at the sample level.
- specimens: metadata and derived data at the specimen level.
- criteria: criteria by which data are deemed acceptable
- ages: ages and metadata for sites/samples/specimens
- images: associated images and plots.

Overview of PmagPy

The functionality of **PmagPy** is demonstrated within notebooks in the **PmagPy** repository:

- PmagPy_online.ipynb: serves as an introdution to PmagPy and MagIC (this conference). It highlights the link between **PmagPy** and the Findable Accessible Interoperable Reusabe (FAIR) database maintained by the Magnetics Information Consortium (MagIC) at https://earthref.org/MagIC.

Other notebooks of interest are:

- PmagPy_calculations.ipynb: demonstrates many of the PmagPy calculation functions such as those that rotate directions, return statistical parameters, and simulate data from specified distributions.
- PmagPy_plots_analysis.ipynb: demonstrates PmagPy functions that can be used to visual data as well as those that conduct statistical tests that have associated visualizations.
- PmagPy_MagIC.ipynb: demonstrates how PmagPy can be used to read and write data to and from the MagIC database format including conversion from many individual lab measurement file formats.

Please see also our YouTube channel with more presentations from the 2020 MagIC workshop here:
https://www.youtube.com/playlist?list=PLirL2unikKCgUkHQ3m8nT29tMCJNBj4kj

Semantic Annotation of Data using JSON Linked Data

Luigi Marini (lmarini@illinois.edu), Diego Calderon (diegoac2@illinois.edu), Praveen Kumar (kumar1@illinois.edu)

University of Illinois at Urbana-Champaign

Abstract

The Earthcube Geosemantics Framework (https://ecgs.ncsa.illinois.edu/) developed a prototype of a decentralized framework that combines the Linked Data and RESTful web services to annotate, connect, integrate, and reason about integration of geoscience resources. The framework allows the semantic enrichment of web resources and semantic mediation among heterogeneous geoscience resources, such as models and data.

This notebook provides examples on how the Semantic Annotation Service can be used to manage linked controlled vocabularies using JSON Linked Data (JSON-LD), including how to query the built-in RDF graphs for existing linked standard vocabularies based on the Community Surface Dynamics Modeling System (CSDMS), Observations Data Model (ODM2) and Unidata udunits2 vocabularies, how to query build-in crosswalks between CSDMS and ODM2 vocabularies using SKOS, and how to add new linked vocabularies to the service. JSON-LD based definitions provided by these endpoints will be used to annotate sample data available within the IML Critical Zone Observatory data repository using the Clowder Web Service API (https://data.imlczo.org/). By supporting JSON-LD, the Semantic Annotation Service and the Clowder framework provide examples on how portable and semantically defined metadata can be used to better annotate data across repositories and services.

Table of contents

Introduction

We face many challenges in the process of extracting meaningful information from data. Frequently, these obstacle compel scientists to perform the integration of models with data manually. Manual integration becomes exponentially difficult when a user aims to integrate long-tail data (data collected by individual researchers or small research groups) and long-tail models (models developed by individuals or small modeling communities). We focus on these long-tail resources because despite their often-narrow scope, they have significant impacts in scientific studies and present an opportunity for addressing critical gaps through automated integration. The goal of the Goesemantics Framework is to provide a framework rooted in semantic techniques and approaches to support “long-tail” models and data integration.

The Linked Data paradigm emerged in the context of Semantic Web technologies for publishing and sharing data over the Web. It connects related individual Web resources in a Graph database, where resources represent the graph nodes, and an edge connects a pair of nodes. Publishing and linking scientific resources using Semantic Web technologies require that the user community follows the three principles of Linked Data:

1. Each resource needs to be represented using a unique Uniform Resource Identifier (URI), which consists of: (i) A Uniform Resource Locator (URL) to define the server path over the Web, and (ii) A Uniform Resource Name (URN) to describe the exact name of the resource.
2. The relationships between resources are described using the triple format, where a subject S has a predicate P with an object O. A predicate is either an undirected relationship (bi-directional), where it connects two entities in both ways or a directed relationship (uni-directional), where the presence of a relationship between two entities in one direction does not imply the presence of a reverse relationship. The triple format is the structure unit for the Linked Data system.
3. The HyperText Transfer Protocol (HTTP) is used as a universal access mechanism for resources on the Web.

For more information about linked data, please visit https://www.w3.org/standards/semanticweb/data.

The Geosemantics Integration Service (GSIS) is a playground to show a lot of these principles in practice with respect to Earth science. Below we show many of the endpoints available in the GSIS and how they can enable new ways to manage metadata about data and models. By virtue of leveraging Semntic Web Technologies, the approaches below are compatible with other efforts such as the ESIP science-on-schema effort. Some of the endpoints shown below are currently used in production, others are proof of concept developement. The goal of this notebook is to show what is possible when using Linked Data approaches.

Basic Requirements

We will be interacting with two web services. The first is the Geosemantics Integration Service (GSIS) available at http://hcgs.ncsa.illinois.edu. This service provides support for standandard vocabularies and methods for transforming typical strings used for tracking time, space and physical variables into well formed Linked Data documents. The second service is the NSF Intensively Managed Landscape Critical Zone Observatory data management system (https://data.imlczo.org/). We will be retrieving data from it and uploading data and metadata back to it using the Clowder web service API. Clowder is a customizable and scalable data management system to support any data format (Marini et al. 2018). It is under active development and deployed for a variety of research projects. For more information about Clowder please visit https://clowderframework.org/.

We first setup some basic requirements used throughout the notebook. We use the ubiqutous Requests Python library to intereact with both APIs.

```
import requests
import json

gsis_host = 'https://ecgs.ncsa.illinois.edu'

clowder_host = 'https://data.imlczo.org/clowder'
```

Geosemantics Integration Service (GSIS)

Since geospatial data comes in many formats, from shapefiles to geotiffs to comma-delimited text files, it is often helpfull to annotate the files with portable metadata that can be used to identify what each files contains. For geospatial data the temporal, spatial, and physical properties dimensions are important and often used to search over a large collection of data. The Geosemantics Integration Service (GSIS) provides a series of endpoints to simplify annotating geospatial data. It includes temporal endpoints so that generic formats for date and times can be translated to well formted JSON-LD

formats (Elag et al. 2015). It includes the ability to store standard vocabularies as generic RDF graphs and retrieve those as simple JSON documents for easy integration in external services (for example Clowder). It also includes the ability to make links between terms from two different standard vocabularies using SKOS and OWN same as predicates.

RDF Graphs

All information stored in the GSIS is stored in the form of RDF graphs using Apache Jena. The following endpoints list all know RDF graphs and let the client retrieve each graph as JSON-LD. The content of these graphs can vary greatly, from standardad vocabularies to definitions of computational models. For a full list of methods please see the documentation available at http://hcgs.ncsa.illinois.edu/. New RDF graphs can be added to the system using private HTTP POST endpoints which accept RDF graphs serialized as JSON-LD or Turtle format.

```
# Get a lists of the names of all graphs in the Knowledge base
r = requests.get(f"{gsis_host}/gsis/listGraphNames")
r.json()
```


```
{'graph_names': ['csdms',
  'odm2-vars',
  'udunits2-base',
  'udunits2-derived',
  'udunits2-accepted',
  'udunits2-prefix',
  'google-unit',
  'model-2',
  'model-3',
  'data-1',
  'data-2',
  'data-3',
  'variable_name_crosswalk',
  'variable_name_crosswalk-owl',
  'variable_name_crosswalk-skos',
  'model_test',
  'model_test11',
  'config_vars.ttl',
  'model-x',
  'Info',
  'Inf',
  'demo-model',
  'csv-mappings',
  'models_graph9d7d400f53864989a05d3ae539f30a78',
  'models_graph37baec3114d74ca6abd72cce75f966db',
  'models_graphe604316f14334985aaf4ebd6fe220e77',
  'models_graph26e29f5026664f11b244072bf6956f74']}
```


```
# List the content of the  Graph (for example, CSDMS Standard Names)
graph = 'csdms'
r = requests.get(f"{gsis_host}/gsis/read?graph={graph}")
r.json().get('@graph')[0:5] # We just show the top 10 results, to see all results
remove the slice operator [0:5]
```

```
[{'@id': 'csn:air__dielectric_constant',
  '@type': 'csn:name',
  'csn:base_object': 'air',
  'csn:base_quantity': 'constant',
  'csn:object_fullname': 'air',
  'csn:object_part': 'air',
  'csn:quantity_fullname': 'dielectric_constant',
  'csn:quantity_part': ['dielectric', 'constant']},
 {'@id': 'csn:air__dynamic_shear_viscosity',
  '@type': 'csn:name',
  'csn:base_object': 'air',
  'csn:base_quantity': 'viscosity',
  'csn:object_fullname': 'air',
  'csn:object_part': 'air',
  'csn:quantity_fullname': 'dynamic_shear_viscosity',
  'csn:quantity_part': ['dynamic', 'shear', 'viscosity']},
 {'@id': 'csn:air__dynamic_volume_viscosity',
  '@type': 'csn:name',
  'csn:base_object': 'air',
  'csn:base_quantity': 'viscosity',
  'csn:object_fullname': 'air',
  'csn:object_part': 'air',
  'csn:quantity_fullname': 'dynamic_volume_viscosity',
  'csn:quantity_part': ['dynamic', 'viscosity', 'volume']},
 {'@id': 'csn:air__kinematic_shear_viscosity',
  '@type': 'csn:name',
  'csn:base_object': 'air',
  'csn:base_quantity': 'viscosity',
  'csn:object_fullname': 'air',
  'csn:object_part': 'air',
  'csn:quantity_fullname': 'kinematic_shear_viscosity',
  'csn:quantity_part': ['shear', 'viscosity', 'kinematic']},
 {'@id': 'csn:air__kinematic_volume_viscosity',
  '@type': 'csn:name',
  'csn:base_object': 'air',
  'csn:base_quantity': 'viscosity',
  'csn:object_fullname': 'air',
  'csn:object_part': 'air',
  'csn:quantity_fullname': 'kinematic_volume_viscosity',
  'csn:quantity_part': ['viscosity', 'volume', 'kinematic']}]
```

Standard Vocabularies

Two RDF graphs in the GSIS store two external standard vocabularies in RDF. The first one is the CSDMS Standard Terms (CSN) (Peckham et al. 2014). The second is the ODM2 Variable Name Vocabulary (Horsburgh et al. 2016). To make it easier to query these RDF by graphs, the GSIS provide simplified methods to search across standard vocabularies by label and look attributes of a specific term in a vocabulary.

```
# Search standard vocabularies by search query
query = 'wind speed'
r = requests.get(f'{gsis_host}/gsis/sas/vars/list?term={query}')
r.json()
```

```
['csn:earth_surface_wind__range_of_speed',
 'csn:land_surface_wind__reference_height_speed',
 'csn:land_surface_wind__speed_reference_height',
 'csn:projectile_origin_wind__speed',
 'odm2:windGustSpeed',
 'odm2:windSpeed']
```

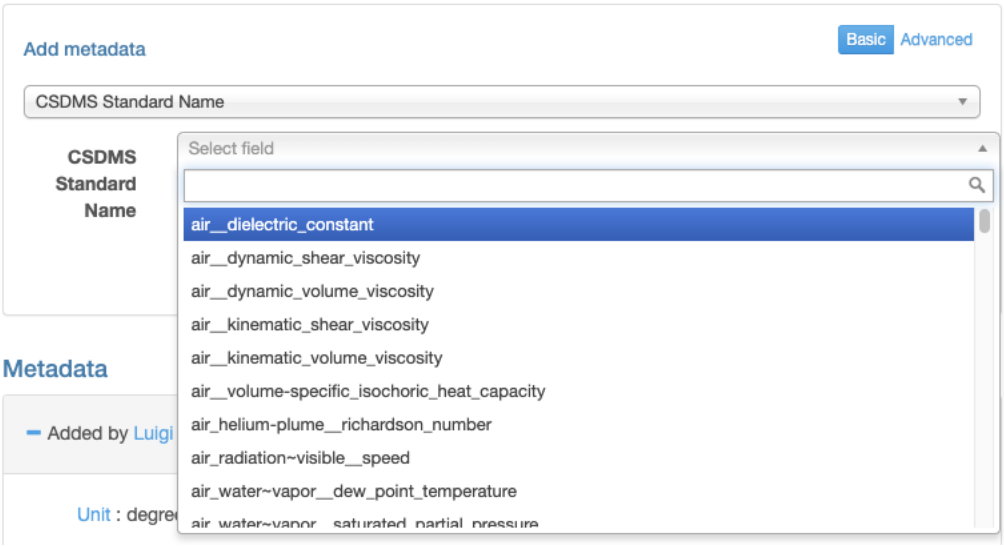
```
# Get all properties of a given CSDMS Standard Name from a specific graph
graph = 'csdms'
name = 'air__dynamic_shear_viscosity'
r = requests.get(f"{gsis_host}/gsis/CSNqueryName?graph={graph}&name={name}")
r.json()
```

```
{'name': 'air__dynamic_shear_viscosity',
 'type': 'http://ecgs.ncsa.illinois.edu/2015/csn/name',
 'object_fullname': 'air',
 'quantity_fullname': 'dynamic_shear_viscosity',
 'base_object': 'air',
 'base_quantity': 'viscosity',
 'object_part': ['air'],
 'quantity_part': ['dynamic', 'shear', 'viscosity']}
```

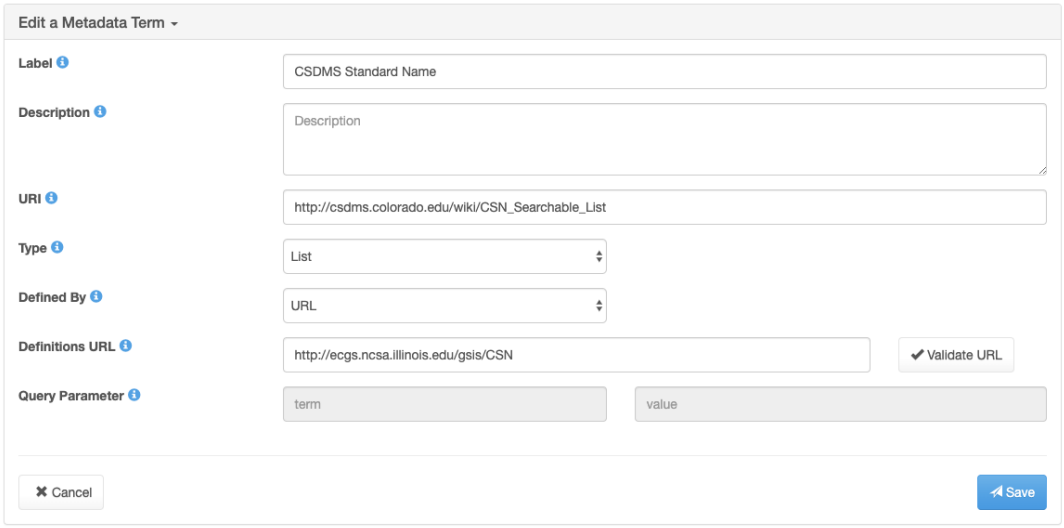
List of CSDMS Standard Names and ODM2 as JSON Arrays

To simplify clients' ability to parse these standard vocabularies, the GSIS provides ways to list all unique identifiers from both vocabularies as play JSON arrays. This for example is used by Clowder to show a list of standard term for each vocabulary in its interface. It is worth noting that Clowder lets users define these lists through its GUI both as local list, but more importantly as remote JSON endpoints so that, as lists are updated, the latest version is always shown to the user. Here is an example from the IMLCZO instance.

This is what a user sees when manually adding metadata to a file or dataset. The list is dynamically loaded at run time.



This is what an adminstrator of the system or a data sharing space see when adding more options to what users can defined from the GUI. This is only for metadata added by users from the GUI. Later we will show how to programmatically add any type of metadata to a file or dataset using the web service API.



The widget listing options above is populated by calling the endpoints below. The `Definitions URL` is how the system is aware of which external endpoint to call. Any service providing the same interface can be utilized.

```
# Get the CSDMS Standard Names as a flat list.
r = requests.get(f"{gsis_host}/gsis/sas/sn/csn")
csn_terms = r.json()
print(f'Found {len(csn_terms)} terms. Showing top 20 results:')
csn_terms[0:20] # We just show the top 20 results, to see all results remove the slice
operator [0:20]
```

```
Found 2573 terms. Showing top 20 results:
```

```
['air__dielectric_constant',
 'air__dynamic_shear_viscosity',
 'air__dynamic_volume_viscosity',
 'air__kinematic_shear_viscosity',
 'air__kinematic_volume_viscosity',
 'air__volume-specific_isochoric_heat_capacity',
 'air_helium-plume__richardson_number',
 'air_radiation~visible__speed',
 'air_water~vapor__dew_point_temperature',
 'air_water~vapor__saturated_partial_pressure',
 'aircraft__flight_duration',
 'airfoil__drag_coefficient',
 'airfoil__lift_coefficient',
 'airfoil_curve~enclosing__circulation',
 'airplane__altitude',
 'airplane__mach_number',
 'airplane_wing__span',
 'air~dry__mass-specific_gas_constant',
 'air~dry_water~vapor__gas_constant_ratio',
 'aluminum__mass-specific_isobaric_heat_capacity']
```

```
# Get the ODM2 Variable Names as a flat list.
r = requests.get(f"{gsis_host}/gsis/sas/sn/odm2")
odm2_terms = r.json()
print(f'Found {len(odm2_terms)} terms. Showing top 20 results:')
r.json()[0:20] # We just show the top 20 results, to see all results remove the slice operator [0:20]
```

```
Found 792 terms. Showing top 20 results:
```

```
['19_Hexanoyloxyfucoxanthin',
 '1_1_1_Trichloroethane',
 '1_1_2_2_Tetrachloroethane',
 '1_1_2_Trichloroethane',
 '1_1_Dichloroethane',
 '1_1_Dichloroethene',
 '1_2_3_Trimethylbenzene',
 '1_2_4_5_Tetrachlorobenzene',
 '1_2_4_Trichlorobenzene',
 '1_2_4_Trimethylbenzene',
 '1_2_Dibromo_3_Chloropropane',
 '1_2_Dichlorobenzene',
 '1_2_Dichloroethane',
 '1_2_Dichloropropane',
 '1_2_Dimethylnaphthalene',
 '1_2_Dinitrobenzene',
 '1_2_Diphenylhydrazine',
 '1_3_5_Trimethylbenzene',
 '1_3_Dichlorobenzene',
 '1_3_Dimethyladamantane']
```

With some simple Python we can search specific substrings from these lists.

```
[i for i in csn_terms if 'temperature' in i]
```

```
['air_water~vapor__dew_point_temperature',
 'atmosphere_air__anomaly_of_temperature',
 'atmosphere_air__azimuth_angle_of_gradient_of_temperature',
 'atmosphere_air__east_derivative_of_temperature',
 'atmosphere_air__elevation_angle_of_gradient_of_temperature',
 'atmosphere_air__equivalent_potential_temperature',
 'atmosphere_air__equivalent_temperature',
 'atmosphere_air__increment_of_temperature',
 'atmosphere_air__magnitude_of_gradient_of_temperature',
 'atmosphere_air__north_derivative_of_temperature',
 'atmosphere_air__potential_temperature',
 'atmosphere_air__temperature',
 'atmosphere_air__temperature_dry_adiabatic_lapse_rate',
 'atmosphere_air__temperature_environmental_lapse_rate',
 'atmosphere_air__temperature_lapse_rate',
 'atmosphere_air__temperature_saturated_adiabatic_lapse_rate',
 'atmosphere_air__x_derivative_of_temperature',
 'atmosphere_air__y_derivative_of_temperature',
 'atmosphere_air__z_derivative_of_temperature',
 'atmosphere_air_water~vapor__bubble_point_temperature',
 'atmosphere_air_water~vapor__dew_point_temperature',
 'atmosphere_air_water~vapor__frost_point_temperature',
```

```
'atmosphere_air_water~vapor__virtual_potential_temperature',
'atmosphere_air_water~vapor__virtual_temperature',
'atmosphere_bottom_air__temperature',
'atmosphere_bottom_air_water~vapor__dew_point_temperature',
'atmosphere_bottom_air_water~vapor__frost_point_temperature',
'atmosphere_top_air__temperature',
'atmosphere_water~vapor__dew_point_temperature',
'atmosphere_water~vapor__frost_point_temperature',
'channel_bottom_water__temperature',
'channel_water__temperature',
'channel_water_surface_air__temperature',
'channel_water_surface_water__temperature',
'chocolate__melting_point_temperature',
'earth_black-body__temperature',
'earth_equator__average_temperature',
'earth_interior__down_z_derivative_of_temperature',
'earth_surface__average_temperature',
'earth_surface__range_of_diurnal_temperature',
'glacier_bed__down_z_derivative_of_temperature',
'glacier_bottom_ice__temperature',
'glacier_ice__azimuth_angle_of_gradient_of_temperature',
'glacier_ice__depression_of_melting_point_temperature',
'glacier_ice__down_derivative_of_temperature',
'glacier_ice__east_derivative_of_temperature',
'glacier_ice__elevation_angle_of_gradient_of_temperature',
'glacier_ice__magnitude_of_gradient_of_temperature',
'glacier_ice__melting_point_temperature',
'glacier_ice__north_derivative_of_temperature',
'glacier_ice__pressure_melting_point_temperature',
'glacier_ice__temperature',
'glacier_ice__x_derivative_of_temperature',
'glacier_ice__y_derivative_of_temperature',
'glacier_ice__z_derivative_of_temperature',
'glacier_top__temperature',
'glacier_top_ice__temperature',
'glacier_top_ice__time_derivative_of_temperature',
'glacier_top_surface__temperature',
'hydrometeor__temperature',
'ice__melting_point_temperature',
'iron__melting_point_temperature',
'land_surface__anomaly_of_temperature',
'land_surface__temperature',
'land_surface_air__temperature',
'land_surface~10m-above_air__temperature',
'sea_bottom_water__temperature',
'sea_ice__depression_of_melting_point_temperature',
'sea_ice__melting_point_temperature',
'sea_ice_bottom_water__temperature',
'sea_ice_surface_air__temperature',
'sea_surface_air-vs-water__difference_of_temperature',
'sea_surface_air__reference_temperature',
'sea_surface_air__temperature',
'sea_surface_water__anomaly_of_temperature',
'sea_surface_water__temperature',
'sea_water__azimuth_angle_of_gradient_of_temperature',
'sea_water__east_derivative_of_temperature',
'sea_water__elevation_angle_of_gradient_of_temperature',
'sea_water__magnitude_of_gradient_of_temperature',
'sea_water__north_derivative_of_temperature',
'sea_water__potential_temperature',
'sea_water__temperature',
'sea_water__time_average_of_square_of_potential_temperature',
'sea_water__time_derivative_of_temperature',
'sea_water__x_derivative_of_temperature',
'sea_water__y_derivative_of_temperature',
'sea_water__z_derivative_of_temperature',
'snow__temperature',
'snow__threshold_of_degree-day-temperature',
'snowpack__degree-day_threshold_temperature',
'snowpack__diurnal_max_of_temperature',
'snowpack__diurnal_min_of_temperature',
'snowpack__diurnal_range_of_temperature',
'snowpack__mean_of_temperature',
'snowpack__time_derivative_of_temperature',
'snowpack_bottom__temperature',
'snowpack_top__temperature',
'snowpack_top_air__temperature',
'soil__reference_depth_temperature',
'soil__temperature',
'soil__temperature_reference_depth',
'water__boiling_point_temperature',
'water__freezing_point_temperature']
```

Crosswalks Between Standard Vocabularies

With so many standard vocabularies available, it is helpful to defined equivalency between terms from separate vocabularies. To this end, the GSIS provides the ability to establish mappings between terms in different graphs using the `skos:sameAs` predicate. The user can the query this graph of relationships using the following endpoint.

```
# Given a term from one vocabulary, find equivalent ones in other vocabularies.
var = 'http://vocabulary.odm2.org/variablename/windSpeed'
r = requests.get(f'{gsis_host}/gsis/var/sameAs/skos?varName={var}')
r.json()
```

```
['http://csdms.colorado.edu/wiki/CSN_Searchable_List/land_surface_air_flow__speed',
 'http://vocabulary.odm2.org/variablename/windSpeed',
 'http://vocabulary.odm2.org/variablename/windGustSpeed']
```

Units

Physical variables are not the only type of standard vocabularies the GSIS stores. Following are examples of two different lists of standard units imported in the GSIS, Unidata udunits2 and Google Units.

```
# Get the list of udunits2 units in JSON format.
r = requests.get(f"{gsis_host}/gsis/sas/unit/udunits2")
r.json()[0:20] # We just show the top 20 results, to see all results remove the slice
operator [0:20]
```

```
['ampere',
 'arc_degree',
 'arc_minute',
 'arc_second',
 'candela',
 'coulomb',
 'day',
 'degree_Celsius',
 'electronvolt',
 'farad',
 'gram',
 'gray',
 'henry',
 'hertz',
 'hour',
 'joule',
 'katal',
 'kelvin',
 'kilogram',
 'liter']
```

```
## Get the list of Google units in JSON format.
r = requests.get(gsis_host + "/gsis/sas/unit/google")
r.json()[0:20] # We just show the top 20 results, to see all results remove the slice
operator [0:20]
```

```
['acre',
 'acre-foot',
 'Algerian dinar',
 'ampere',
 'ampere hour',
 'amu',
 'arc minute',
 'arc second',
 'are',
 'Argentine peso',
 'Astronomical Unit',
 'ATA pica',
 'ATA point',
 'atmosphere',
 'atomic mass unit',
 'Australian cent',
 'Australian dollar',
 'Bahrain dinar',
 "baker's dozen",
 'bar']
```

Temporal Annotation

To convert from strings representing time to a more formal definition, the GSIS provides three endpoints to represent instant, interval, and time series. Time values are represented in UTC (Coordinated Universal Time) format. Times are expressed in local time, together with a time zone offset in hours and minutes. For more information about date and time formats, please visit https://www.w3.org/TR/NOTE-datetime.

Time Instant Annotation
Query parameters:

- **time** (string): time value in UTC format

```
# Get a temporal annotation for a time instant in a JSON-LD format.
time = '2014-01-01T08:01:01-09:00'
r = requests.get(f"{gsis_host}/gsis/sas/temporal?time={time}")
r.json()
```


```
{'@context': {'dc': 'http://purl.org/dc/elements/1.1/',
  'dcterms': 'http://purl.org/dc/terms/',
  'xsd': 'http://www.w3.org/2001/XMLSchema#',
  'time': 'http://www.w3.org/2006/time#',
  'tzont': 'http://www.w3.org/2006/timezone-us'},
 '@id': 'http://ecgs.ncsa.illinois.edu/time_instant',
 '@type': 'time:Instant',
 'dc:date': "yyyy-MM-dd'T'HH:mm:ssZ",
 'time:DateTimeDescription': {'time:year': '2014',
  'time:month': '1',
  'time:day': '1',
  'tzont': '-09:00',
  'time:hours': '8',
  'time:minutes': '1',
  'time:seconds': '1'}}
```


Time Interval Annotation
Query parameters:

- **beginning** (string): time value in UTC format.
- **end** (string): time value in UTC format.

```
# Get a temporal annotation for a time interval in a JSON-LD format.
beginning = '2014-01-01T08:01:01-10:00'
end = '2014-12-31T08:01:01-10:00'
r = requests.get(f"{gsis_host}/gsis/sas/temporal?beginning={beginning}&end={end}")
r.json()
```


```
{'@context': {'dc': 'http://purl.org/dc/elements/1.1/',
  'dcterms': 'http://purl.org/dc/terms/',
  'xsd': 'http://www.w3.org/2001/XMLSchema#',
  'time': 'http://www.w3.org/2006/time#',
  'tzont': 'http://www.w3.org/2006/timezone-us'},
 '@id': 'http://ecgs.ncsa.illinois.edu/time_interval',
 '@type': 'time:Interval',
 'time:Duration': {'time:hasBeginning': {'dc:date': "yyyy-MM-dd'T'HH:mm:ssZ",
   'time:DateTimeDescription': {'time:year': 2014,
    'time:month': 1,
    'time:day': 1,
    'tzont': '-10:00',
    'time:hours': 8,
    'time:minutes': 1,
    'time:seconds': 1}},
  'time:hasEnd': {'dc:date': "yyyy-MM-dd'T'HH:mm:ssZ",
   'time:DateTimeDescription': {'time:year': 2014,
    'time:month': 12,
    'time:day': 31,
    'tzont': '-10:00',
    'time:hours': 8,
    'time:minutes': 1,
    'time:seconds': 1}}}}
```


Time Series Annotation
Query parameters:

- **beginning** (string): time value in UTC format.
- **end** (string): time value in UTC format.
- **interval** (float): time step.

```
# Get a temporal annotation for a time series in a JSON-LD format.
beginning = '2014-01-01T08:01:01-10:00'
end = '2014-03-01T08:01:01-10:00'
interval = '4'
r = requests.get(f"{gsis_host}/gsis/sas/temporal?beginning={beginning}&end=
{end}&interval={interval}")
r.json()
```

```
{'@context': {'dc': 'http://purl.org/dc/elements/1.1/',
  'dcterms': 'http://purl.org/dc/terms/',
  'xsd': 'http://www.w3.org/2001/XMLSchema#',
  'time': 'http://www.w3.org/2006/time#',
  'tzont': 'http://www.w3.org/2006/timezone-us'},
 '@id': 'http://ecgs.ncsa.illinois.edu/time_series',
 'time:Duration': {'time:hasBeginning': {'dc:date': "yyyy-MM-dd'T'HH:mm:ssZ",
   'time:DateTimeDescription': {'time:year': 2014,
    'time:month': 1,
    'time:day': 1,
    'tzont': '-10:00',
    'time:hours': 8,
    'time:minutes': 1,
    'time:seconds': 1}},
  'time:hasEnd': {'dc:date': "yyyy-MM-dd'T'HH:mm:ssZ",
   'time:DateTimeDescription': {'time:year': 2014,
    'time:month': 3,
    'time:day': 1,
    'tzont': '-10:00',
    'time:hours': 8,
    'time:minutes': 1,
    'time:seconds': 1}},
  'time:temporalUnit': {'@type': 'time:unitSecond', '@value': 4}}}
```

Annotating Data in the IMLCZO Data Management System

The IMLCZO data management system is comprised of two different services. A Clowder instance stores raw data streaming in from sensors, manually uploaded by users, collected in the lab or in the field. A Geodashboard including a subset of all the data collected and presented using interactive maps and graphs. We will focus on the Clowder service and how users can store arbitrary metadata as JSON-LD on datasets and files stored within. We will leverage the GSIS to make sure that our metadata is based on existing standards.

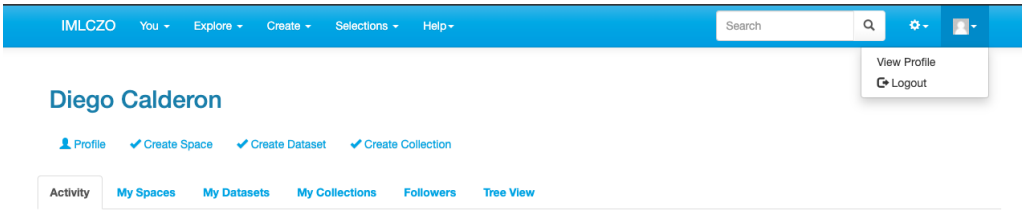
Because we will be adding information to the Clowder instance, we will be required to register an account on the IMLCZO Clowder instance and create an API key and added to the cell below.

Register an account on IMLCZO

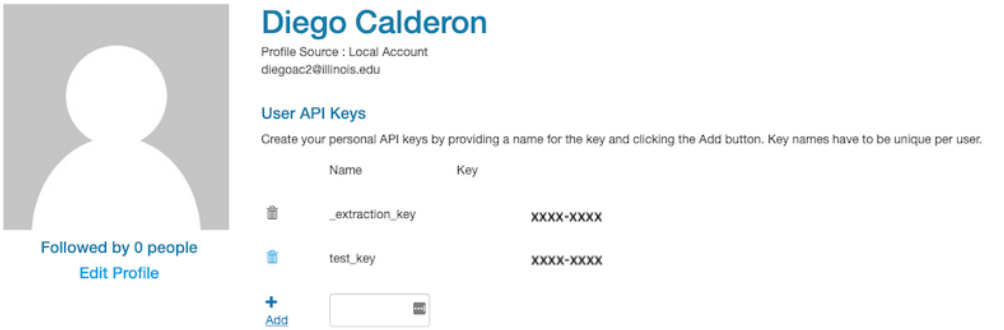
To register an account on the IMLCZO Clowder instance, please go to https://data.imlczo.org/clowder/signup and enter your email. You will receive an email from us. To activate your account, please reply back to the email saying you are using the ecgs jupyter notebook. Once your account is activated, you can generate an API key.

Generate and use an API Key

1. Login to https://data.imlczo.org/clowder/login and navigate to your profile (click on the icon in the top right and select “View Profile”).



2. Add a name to your key and hit the “Add” button. In the example below, we created a test_key.



3. Copy your key from step 2.
4. If you are running this notebook in your machine, create a .env file in the same directory as this notebook and, using your favorite text editor, add the following line: `CLOWDER-KEY=paste-your-key-here`.
5. If you are running the notebook in Binder, uncomment and paste the key in line 5, `%env CLOWDER-KEY=` in the block below.
6. Run the following block.

Setting API key and request headers

We use python-dotenv to set the Clowder API key for this session (you can also just manually set it in the notebook if you prefer). We also set the headers for most requests here and set the default content type to JSON and the Clowder API key to the one we just created. All calls will only provide information based on the user making the request. This means that the quality of the results could vary greatly. We will be creating a dataset and adding a file to it to make sure that the user can make the appropriate against this specific resource.

```
# Please create an API key as described above
%load_ext dotenv
%dotenv
# If using Binder, uncomment and paste your key below
# %env CLOWDER-KEY=paste your key here
import os
clowder_key = os.getenv("CLOWDER-KEY")

headers = {'Content-type': 'application/json', 'X-API-Key': clowder_key}
```

Search by generic search query

We will start by searching the system for a generic string `precipitation`. Depending on your permissions you might be able to see around 11 results. The results of this query are based on any information available on the resource (dataset, file, or collection).

```
query = 'precipitation'
r = requests.get("{}/api/search?query={}".format(clowder_host, query), headers=headers)
r.raise_for_status()
r.json()
```

```
{'count': 10,
 'size': 10,
 'scanned_size': 240,
 'results': [{'id': '596faa3b4f0c0b1c81fa42de',
   'name': 'Trimpe East Site (Precip tipping bucket site)',
   'description': 'Precipitation, temperature and solar radiation sensor (E-P) at
Trimpe Farm on the east end of the farm. Installed on 7/11/17. Precip array was moved
from the west end of the farm and relocated on the east end. Precip records before
7/11/17 can be found in the Trimpe West dataset.Trimpe East Deep Well has a decagon
well transducer measuring hydraulic head and specific conductivity; the transducer
has a depth of (6.2m - .73m well casing length = 5.47m). Trimpe East Shallow Well has
a decagon well transducer measuring hydraulic head and specific conductivty; the
transducer has a depth of (4.57m -1.3m well casing = 3.27m). These well files are
denoted as Trimpe_E-W . ',
   'created': 'Wed Jul 19 13:51:39 CDT 2017',
   'thumbnail': '596fade24f0c0b1c81fa4414',
   'authorId': '58b05a8c5792756cf490e664',
   'spaces': ['5967f7004f0c0b1c81f8654a'],
   'resource_type': 'dataset'},
  {'id': '59444c82e4b001915f9624ee',
```

```
   'collectionname': 'Precipitation',
   'description': '',
   'created': 'Fri Jun 16 16:24:18 CDT 2017',
   'thumbnail': '58ff9534e4b0ed131b4cb554',
   'authorId': '587e90a7a3ad2109e72c8651',
   'resource_type': 'collection'},
  {'id': '59444cebe4b001915f96254f',
   'name': 'C-1-1',
   'description': 'Climate - Precipitation',
   'created': 'Fri Jun 16 16:26:03 CDT 2017',
   'thumbnail': None,
   'authorId': '587e90a7a3ad2109e72c8651',
   'spaces': ['594441eae4b001915f96205c'],
   'resource_type': 'dataset'},
  {'id': '59444cf6e4b001915f962553',
   'name': 'C-1-2',
   'description': 'Climate - Precipitation',
   'created': 'Fri Jun 16 16:26:14 CDT 2017',
   'thumbnail': None,
   'authorId': '587e90a7a3ad2109e72c8651',
   'spaces': ['594441eae4b001915f96205c'],
   'resource_type': 'dataset'},
  {'id': '5500f526e4b0feb5ae15458d',
   'name': 'ClearCreekIA_HistoricalClimateData.xlsx',
   'description': 'Historical climate data for Clear Creek Temperature,
Precipitation, Snowfall, Snow Depth, Wind Speed, Pan Evaporation',
   'created': 'Thu Oct 24 23:21:16 CDT 2013',
   'thumbnail': None,
   'authorId': '5500f508e4b0feb5ae154555',
   'spaces': ['594441eae4b001915f96205c'],
   'resource_type': 'dataset'},
  {'id': '564a0843e4b0b6c112b46cac',
   'name': 'Clear Creek IA - WA_RAW_Maas_6',
   'description': 'Clear Creek, IA\nZone I Site 1-1\nWeathering Array
EM30136\nAnemometer, Precipitation',
   'created': 'Mon Nov 16 10:45:55 CST 2015',
   'thumbnail': '57bc65cfe4b0b805e47a37a8',
   'authorId': '577bfd99e4b054f1fa78f685',
   'spaces': ['594441f6e4b001915f96206f'],
   'resource_type': 'dataset'},
  {'id': '596fadd04f0c0b1c81fa4405',
   'name': 'WA_RAW_Trimpe_E-P_2017-7-17.xls',
   'status': 'PROCESSED',
   'thumbnail': '596fade24f0c0b1c81fa4414',
   'created': 'Wed Jul 19 14:06:56 CDT 2017',
   'resource_type': 'file'},
  {'id': '5a32c7674f0c6b32ffa397e7',
   'name': 'S-Fowler Farm Weather Station (CZO_FF1)',
   'description': 'Site #151\nLocation: 40°09\'02.6" N -88°19\'52.1" W\nSite/Logger
Name: CZO_FF1\nDate deployed: 03/29/2017\nSensors: PYR pyranometer, ECRN-100 rain
gauge, VP-4 RH/Temp/AtmP, LWS leaf wetness, and Davis Cup anemometer. DS-2 sonic
anemometer used from 03/29/2017-01/31/2018.\nLogger(s): METER Group EM60G, SN: 06-
01890 (06/28/2018-current)\nDecagon EM50G, SN: 5G106751 (used 11/14/2017-
06/28/2018)\nDecagon EM50G; SN: 5G106590 (used 03/29/2017-10/14/2017)\n\nData
collection frequency: 1-min\nData collected: solar radiation (W/m^2), precipitation
(mm), relative humidity (%), temperature (*C), atmospheric pressure (kPa), LWS
minutes wet (min) and count, wind speed (m/s), wind direction (degree), and maximum
wind speed (m/s)',
   'created': 'Thu Dec 14 12:48:07 CST 2017',
   'thumbnail': None,
   'authorId': '57222685a30d71e0572adf99',
   'spaces': ['594441eae4b001915f96205c'],
   'resource_type': 'dataset'},
  {'id': '5a32e5094f0c6b32ffa39e86',
   'name': 'S-River Bend Weather Station (CZO_RB1)',
   'description': 'Site #149\nLocation: 40°10\'49.1" N -88°26\'00.0" W\nSite/Logger
Name: CZO_RB1\nDate deployed: 09/15/2016\nSensors: PYR pyranometer, ECRN-100 rain
gauge, VP-4 RH/Temp/AtmP, LWS leaf wetness, and Davis Cup anemometer. DS-2 sonic
anemometer was used from 09/15/2016-2/22/2018.\nLogger(s): METER Group EM60G, SN: 06-
01883 (06/14/2018-current)\nDecagon EM50, SN: EM20692 (used 04/30/2018-
06/14/2018)\nDecagon EM50G, SN: 5G106588 (used 09/15/2016-04/30/2018)\n\nMeasurement
Interval: 1-min\nData collected: solar radiation (W/m^2), precipitation (mm),
relative humidity (%), temperature (*C), atmospheric pressure (kPa), LWS minutes wet
(min) and count, wind speed (m/s), wind direction (degree), and maximum wind speed
(m/s)',
   'created': 'Thu Dec 14 14:54:33 CST 2017',
   'thumbnail': None,
   'authorId': '57222685a30d71e0572adf99',
   'spaces': ['594441eae4b001915f96205c'],
   'resource_type': 'dataset'},
  {'id': '57c0b485e4b0b805e47b3df8',
   'name': 'Previous measurement station',
   'description': 'These sensor locations are previously installed inside USRB,
including precipitation stations, sediment station, USGS stream station, Ameriflux
tower, ISWS Stream and nutrient station, etc.',
   'created': 'Fri Aug 26 16:28:37 CDT 2016',
   'thumbnail': None,
```

```
    'authorId': '5500f508e4b0feb5ae15455e',
    'spaces': ['5953ba884f0c5558e1684bf4'],
    'resource_type': 'dataset'}],
  'from': 0,
  'total_size': 12}
```

Search by metadata field

To be more specific, we will search for any resource which contains metadata for `ODM2 Variable Name` that is equal to `precipitation`.

```
query = '"ODM2 Variable Name":"precipitation"'
r = requests.get("{}/api/search?query={}".format(clowder_host, query), headers=headers)
datasets = r.json().get('results')

dataset = [d for d in datasets if d.get('name') == 'Trimpe East Site (Precip tipping
bucket site)']

datasetId = dataset[0].get('id')

dataset[0]
```

```
{'id': '596faa3b4f0c0b1c81fa42de',
 'name': 'Trimpe East Site (Precip tipping bucket site)',
 'description': 'Precipitation, temperature and solar radiation sensor (E-P) at
Trimpe Farm on the east end of the farm. Installed on 7/11/17. Precip array was moved
from the west end of the farm and relocated on the east end. Precip records before
7/11/17 can be found in the Trimpe West dataset.Trimpe East Deep Well has a decagon
well transducer measuring hydraulic head and specific conductivity; the transducer
has a depth of (6.2m - .73m well casing length = 5.47m). Trimpe East Shallow Well has
a decagon well transducer measuring hydraulic head and specific conductivty; the
transducer has a depth of (4.57m -1.3m well casing = 3.27m). These well files are
denoted as Trimpe_E-W . ',
 'created': 'Wed Jul 19 13:51:39 CDT 2017',
 'thumbnail': '596fade24f0c0b1c81fa4414',
 'authorId': '58b05a8c5792756cf490e664',
 'spaces': ['5967f7004f0c0b1c81f8654a'],
 'resource_type': 'dataset'}
```

We list all files in the dataset and download the first in the list. This is just to provide us with a relevant file locally but if you prefer you can ignore this step and later on upload your own file to the system.

```
# List files in dataset
url = "{}/api/datasets/{}/files".format(clowder_host, datasetId)
r = requests.get(url)
files = r.json()
# Download the first file
file_id = files[0].get('id')
file_name = files[0].get('filename')
url = "{}/api/files/{}/blob".format(clowder_host, file_id)
r = requests.get(url)
with open(file_name, 'wb') as f:
    f.write(r.content)
print(f'Downloaded file {file_name} to local disk')
```

```
Downloaded file WA_RAW_Trimpe_E-P_2017-7-17.xls to local disk
```

Create new dataset

Create a new dataset to contain the file we just downloaded (or a new one) and metadata.

```
url = "{}/api/datasets/createempty".format(clowder_host)
payload = json.dumps({'name': 'Geosemantics Demo',
                      'description': 'A dataset used for demoing basic metadata
annotation functionality',
                      'access': 'PRIVATE',
                      'space': [],
                      'collection': []})

r = requests.post(url, data=payload, headers=headers)
r.raise_for_status()
new_dataset = r.json()
new_dataset_id = new_dataset.get('id')
print(f'Created new dataset {clowder_host}/datasets/{new_dataset_id}')
```

```
Created new dataset https://data.imlczo.org/clowder/datasets/5ee7d6a64f0ccc5274e45431
```

Upload file to new dataset

We now upload a file to the dataset that we just created. If you prefer uploading a different file, change the file name below.

```
url = "{}/api/uploadToDataset/{}".format(clowder_host, new_dataset_id)
# change file_name if you prefer uploading a different file from your local directory
files = {'file': open(file_name, 'rb')}
r = requests.post(url, files=files, headers={'X-API-Key': clowder_key})
r.raise_for_status()
uploaded_file_id = r.json().get('id')
print(f'Uploaded file {clowder_host}/files/{uploaded_file_id}')
```

```
Uploaded file https://data.imlczo.org/clowder/files/5ee7d6a94f0ccc5274e45437
```

Add metadata to new file

We now upload metadata to the file we have just uploaded. Note that this operation can be executed multiple times with different payloads. Every time a new entry is added to the list of metadata documents associated with a file or dataset. The same user can update multiple values of a specific entry or different users can specify alternative values of the same metadata types. It is is up to the client to decide which version is the most accurate. Users can delete entries that are not valid anymore. This type of generic metadata is compatible to the advanced publishing techniquies described in the ESIP science-on-schema.org and could be added to a DCAT Dataset as described there.

```
url = "{}/api/files/{}/metadata.jsonld".format(clowder_host, uploaded_file_id)
payload = {
    "@context":[
        "https://clowder.ncsa.illinois.edu/contexts/metadata.jsonld",
        {
            "CSDMS Standard Name":
"http://csdms.colorado.edu/wiki/CSN_Searchable_List#atmosphere_air__temperature",
            "Unit":
"http://ecgs.ncsa.illinois.edu/gsis/sas/unit/udunits2#degree_Celsius"
        }
    ],
    "agent": {
        "@type":"cat:extractor",
        "name":"ECGS Notebook",
        "extractor_id":"https://clowder.ncsa.illinois.edu/api/extractors/ecgs"
    },
    "content": {
        "CSDMS Standard Name": "atmosphere_air__temperature",
        "Unit": "degree_Celsius"
    }
}
r = requests.post(url, headers = headers, data=json.dumps(payload))
r.raise_for_status()
print('Response ' + r.json())
print(f'View metadata you have just uploaded on the file page
{clowder_host}/files/{uploaded_file_id}')
```

```
Response Metadata successfully added to db
View metadata you have just uploaded on the file page
https://data.imlczo.org/clowder/files/5ee7d6a94f0ccc5274e45437
```

We can also retrieve all the metadata available on the file, including metadata automatically created by the system.

```
url = "{}/api/files/{}/metadata.jsonld".format(clowder_host, uploaded_file_id)
r = requests.get(url, headers = headers)
r.json()
```

```
[{'@context': ['https://clowder.ncsa.illinois.edu/contexts/metadata.jsonld',
   {'CSDMS Standard Name':
'http://csdms.colorado.edu/wiki/CSN_Searchable_List#atmosphere_air__temperature',
    'Unit': 'http://ecgs.ncsa.illinois.edu/gsis/sas/unit/udunits2#degree_Celsius'}],
  'attached_to': {'resource_type': 'cat:file',
   'url': 'https://data.imlczo.org/clowder/files/5ee7d6a94f0ccc5274e45437'},
  'created_at': 'Mon Jun 15 15:14:33 CDT 2020',
  'agent': {'@type': 'cat:extractor',
   'name': 'ECGS Notebook',
   'extractor_id': 'https://clowder.ncsa.illinois.edu/api/extractors/ecgs'},
  'content': {'CSDMS Standard Name': 'atmosphere_air__temperature',
   'Unit': 'degree_Celsius'}}]
```

We have defined the context by including an external one that contains the basic elements of any Clowder metadata document such as `agent` as well as specific ones for the two entries in `content`. We can view the rest of the context here:

```
# The context file describes the basic elements of a Clowder metadata document
r = requests.get('https://clowder.ncsa.illinois.edu/contexts/metadata.jsonld')
r.json()
```

```
{'@context': {'cat': 'https://clowder.ncsa.illinois.edu/#',
  'extractor_id': {'@id': 'cat:extractor/id', '@type': '@id'},
  'user_id': {'@id': 'cat:user/id', '@type': '@id'},
  'created_at': {'@id': 'http://purl.org/dc/terms/created',
   '@type': 'http://www.w3.org/2001/XMLSchema#dateTime'},
  'agent': {'@id': 'http://www.w3.org/ns/prov#Agent'},
  'user': 'cat:user',
  'extractor': 'cat:extractor',
  'content': {'@id': 'https://clowder.ncsa.illinois.edu/metadata#content'}}}
```

Matching Models with Data

The GSIS also prototyped a method to match simulations models developed using the Basic Model Interface (BMI) (Jiang et al. 2017) to input datasets. Given definitions for both, the system checks if a specific variable is compatible between data and model. The user provides the id of an RDF graph representing a model, one of an RDF graph representing a dataset, and the RDF predicates used to identify the variables in each graph. It then tries to match the two lists leveraging the `skos:sameAs` crosswalks defined above. The results includes whether a match was found, what variables are missing if so, and what variables will require a crosswalk and potentially a conversion.

For example, in the case below a match was found but the crosswalk between `http://vocabulary.odm2.org/variablename/windSpeed` and `http://csdms.colorado.edu/wiki/CSN_Searchable_List/land_surface_air_flow__speed` was required.

```
model_id = 'model-3'
model_var_property_name =
'http://ecgs.ncsa.illinois.edu/bmi_models/temp/hasStandardName'
data_id = 'data-3'
data_var_property_name = 'http://ecgs.ncsa.illinois.edu/gsis/sas/vars'

r = requests.get(f'{gsis_host}/gsis/modelDataMatches?model=
{model_id}&modelVarPropertyName={model_var_property_name}&data=
{data_id}&dataVarPropertyName={data_var_property_name}')
r.json()
```

```
{'Matching status': 'true',
 'missing variables': '',
 'model variables':
'http://vocabulary.odm2.org/variablename/windSpeed,http://csdms.colorado.edu/wiki/CSN
_Searchable_List/atmosphere_bottom_air__brutsaert_emissivity_canopy_factor',
 'data variables, original':
'http://csdms.colorado.edu/wiki/CSN_Searchable_List/atmosphere_bottom_air__brutsaert_
emissivity_canopy_factor,http://csdms.colorado.edu/wiki/CSN_Searchable_List/land_surf
ace_air_flow__speed',
 'data variables, with inference':
'http://csdms.colorado.edu/wiki/CSN_Searchable_List/atmosphere_bottom_air__brutsaert_
emissivity_canopy_factor,http://csdms.colorado.edu/wiki/CSN_Searchable_List/land_surf
ace_air_flow__speed,http://vocabulary.odm2.org/variablename/windGustSpeed,http://voca
bulary.odm2.org/variablename/windSpeed'}
```

For reference here are the two RDF graphs being compared:

```
r = requests.get(f"{gsis_host}/gsis/read?graph={model_id}")
r.json()
```

```
{'@graph': [{'@id': '_:b0',
   '@type': 'ns1:variable',
   'ns1:hasScriptName': 'beta',
   'hasSpatialProperties': '_:b2',
   'hasStandardName': 'odm2:windSpeed',
   'hasTemporalProperties': '_:b3',
   'ns1:hasUnits': 'radians',
   'hasValueType': 'xsd:double'},
  {'@id': '_:b1',
   '@type': 'ns1:variable',
   'ns1:hasScriptName': 'canopy_factor',
   'hasSpatialProperties': '_:b6',
   'hasStandardName':
'csn:atmosphere_bottom_air__brutsaert_emissivity_canopy_factor',
   'hasTemporalProperties': '_:b7',
   'ns1:hasUnits': '1',
   'hasValueType': 'xsd:double'},
  {'@id': '_:b2',
   'hasGridNumberInX': 'ns1:unknown',
   'hasGridNumberInY': 'ns1:unknown',
   'hasGridSpacingX': '_:b4',
   'hasGridSpacingY': '_:b5',
   'ns1:hasGridType': 'uniform',
   'hasSouthBounding': 'ns1:unknown',
   'hasWestBounding': 'ns1:unknown'},
  {'@id': '_:b3',
   'hasTimeStep': 'ns1:unknown',
   'hasTimeStep:Type': 'fixed',
   'ns1:hasTimeUnits': 'seconds'},
  {'@id': '_:b4', 'ns1:hasUnits': 'meters', 'hasValue': 'ns1:unknown'},
  {'@id': '_:b5', 'ns1:hasUnits': 'meters', 'hasValue': 'ns1:unknown'},
  {'@id': '_:b6',
   'hasGridNumberInX': 'ns1:unknown',
   'hasGridNumberInY': 'ns1:unknown',
   'hasGridSpacingX': '_:b8',
   'hasGridSpacingY': '_:b9',
   'ns1:hasGridType': 'uniform',
   'hasSouthBounding': 'ns1:unknown',
   'hasWestBounding': 'ns1:unknown'},
  {'@id': '_:b7',
   'hasTimeStep': 'ns1:unknown',
   'hasTimeStep:Type': 'fixed',
   'ns1:hasTimeUnits': 'seconds'},
  {'@id': '_:b8', 'ns1:hasUnits': 'meters', 'hasValue': 'ns1:unknown'},
  {'@id': '_:b9', 'ns1:hasUnits': 'meters', 'hasValue': 'ns1:unknown'},
  {'@id': 'ns1:TopoFlow_Meteorology',
   '@type': 'ns1:model',
   'hasInput': ['_:b0', '_:b1']}],
 '@context': {'hasInput': {'@id':
'http://ecgs.ncsa.illinois.edu/bmi_models/temp/hasInput',
   '@type': '@id'},
  'hasScriptName': {'@id':
'http://ecgs.ncsa.illinois.edu/bmi_models/temp/hasScriptName',
   '@type': 'http://www.w3.org/2001/XMLSchema#string'},
  'hasUnits': {'@id': 'http://ecgs.ncsa.illinois.edu/bmi_models/temp/hasUnits',
   '@type': 'http://www.w3.org/2001/XMLSchema#string'},
  'hasSpatialProperties': {'@id':
'http://ecgs.ncsa.illinois.edu/bmi_models/temp/hasSpatialProperties',
   '@type': '@id'},
  'hasStandardName': {'@id':
'http://ecgs.ncsa.illinois.edu/bmi_models/temp/hasStandardName',
   '@type': '@id'},
  'hasTemporalProperties': {'@id':
'http://ecgs.ncsa.illinois.edu/bmi_models/temp/hasTemporalProperties',
   '@type': '@id'},
  'hasValueType': {'@id':
'http://ecgs.ncsa.illinois.edu/bmi_models/temp/hasValueType',
   '@type': '@id'},
  'hasGridNumberInX': {'@id':
'http://ecgs.ncsa.illinois.edu/bmi_models/temp/hasGridNumberInX',
   '@type': '@id'},
  'hasGridNumberInY': {'@id':
'http://ecgs.ncsa.illinois.edu/bmi_models/temp/hasGridNumberInY',
   '@type': '@id'},
  'hasGridSpacingX': {'@id':
'http://ecgs.ncsa.illinois.edu/bmi_models/temp/hasGridSpacingX',
   '@type': '@id'},
  'hasGridSpacingY': {'@id':
'http://ecgs.ncsa.illinois.edu/bmi_models/temp/hasGridSpacingY',
   '@type': '@id'},
  'hasGridType': {'@id': 'http://ecgs.ncsa.illinois.edu/bmi_models/temp/hasGridType',
   '@type': 'http://www.w3.org/2001/XMLSchema#string'},
  'hasSouthBounding': {'@id':
'http://ecgs.ncsa.illinois.edu/bmi_models/temp/hasSouthBounding',
   '@type': '@id'},
  'hasWestBounding': {'@id':
'http://ecgs.ncsa.illinois.edu/bmi_models/temp/hasWestBounding',
```

```
    '@type': '@id'},
   'hasValue': {'@id': 'http://ecgs.ncsa.illinois.edu/bmi_models/temp/hasValue',
    '@type': '@id'},
   'hasTimeStep': {'@id': 'http://ecgs.ncsa.illinois.edu/bmi_models/temp/hasTimeStep',
    '@type': '@id'},
   'hasTimeStepType': {'@id':
'http://ecgs.ncsa.illinois.edu/bmi_models/temp/hasTimeStepType',
    '@type': 'http://www.w3.org/2001/XMLSchema#string'},
   'hasTimeUnits': {'@id':
'http://ecgs.ncsa.illinois.edu/bmi_models/temp/hasTimeUnits',
    '@type': 'http://www.w3.org/2001/XMLSchema#string'},
   'rdf': 'http://www.w3.org/1999/02/22-rdf-syntax-ns#',
   'xml': 'http://www.w3.org/XML/1998/namespace',
   'xsd': 'http://www.w3.org/2001/XMLSchema#',
   'odm2': 'http://vocabulary.odm2.org/variablename/',
   'rdfs': 'http://www.w3.org/2000/01/rdf-schema#',
   'ns1': 'http://ecgs.ncsa.illinois.edu/bmi_models/temp/',
   'csn': 'http://csdms.colorado.edu/wiki/CSN_Searchable_List/'}}
```

```
r = requests.get(f"{gsis_host}/gsis/read?graph={data_id}")
r.json()
```

```
{'@id': '_:b0',
 'http://ecgs.ncsa.illinois.edu/gsis/sas/unit': 'degree',
 'vars': ['csn:atmosphere_bottom_air__brutsaert_emissivity_canopy_factor',
  'csn:land_surface_air_flow__speed'],
 '@context': {'vars': {'@id': 'http://ecgs.ncsa.illinois.edu/gsis/sas/vars',
   '@type': '@id'},
  'unit': {'@id': 'http://ecgs.ncsa.illinois.edu/gsis/sas/unit',
   '@type': 'http://www.w3.org/2001/XMLSchema#string'},
  'csn': 'http://csdms.colorado.edu/wiki/CSN_Searchable_List/'}}
```

Conclusions

This short notebook provides a few simple examples of developing web applications around the principles of Linked Data. By developing services built around interoperability, we hope it will be easier in the future to build services that can easily interoperate. Earth sciences provide unique challenges in that the way researchers store their data can vary greatly. The Linked Data approach can be useful in overcoming some of these challenges even thought it provides its own technical challenges in terms of adoption. Over time efforts like Schema.org are showing that the principles of Linked Data are important but that simplifying some of their approaches might be the key to widespread adoption. Even though the GSIS stores information as RDF graphs, it provides simple HTTP web services to make it easier to be used in the existing ecosystem of tools. Furthermore, the Clowder data framework provides simple GUI and APIs to store rich metadata documents next to the raw bytes, but it tries to find a good compromise between expressivenes of the metadata and simplicity of use.

References

1. Marini, L., I. Gutierrez-Polo, R. Kooper, S. Puthanveetil Satheesan, M. Burnette, J. Lee, T. Nicholson, Y. Zhao, and K. McHenry. 2018. Clowder: Open Source Data Management for Long Tail Data. In Proceedings of the Practice and Experience on Advanced Research Computing (PEARC '18). ACM, New York, NY, USA, Article 40, 8 pages. DOI: https://doi.org/10.1145/3219104.3219159
2. Jiang, Peishi and Elag, Mostaf and Kumar, Praveen and Peckham, Scott and Marini, Luigi and Liu, Rui, "A service-oriented architecture for coupling web service models using the Basic Model Interface (BMI)", Environmental Modelling & Software, 2017
3. Elag, M.M., P. Kumar, L. Marini, S.D. Peckham (2015) Semantic interoperability of long-tail geoscience resources over the Web, In: Large-Scale Machine Learning in the Earth Sciences, Eds. A.N. Srivastava, R. Nemani and K. Steinhaeuser, Taylor and Francis
4. Peckham, S.D. (2014a) The CSDMS Standard Names: Cross-domain naming conventions for describing process models, data sets and their associated variables, Proceedings of the 7th Intl. Congress on Env. Modelling and Software, International Environmental Modelling and Software Society (iEMSs), San Diego, CA. (Eds. D.P. Ames, N.W.T. Quinn, A.E. Rizzoli)
5. Horsburgh, J. S., Aufdenkampe, A. K., Mayorga, E., Lehnert, K. A., Hsu, L., Song, L., Spackman Jones, A., Damiano, S. G., Tarboton, D. G., Valentine, D., Zaslavsky, I., Whitenack, T. (2016). Observations Data Model 2: A community information model for spatially discrete Earth observations, Environmental Modelling & Software, 79, 55-74, http://dx.doi.org/10.1016/j.envsoft.2016.01.010

License

Vertical regridding and remapping of CMIP6 ocean data in the cloud

C Spencer Jones, Julius Busecke, Takaya Uchida and Ryan Abernathey

1. Introduction

Many ocean and climate models output ocean variables (like velocity, temperature, oxygen concentration etc.) in depth space. Property transport in the ocean generally follows isopycnals, but isopycnals often move up and down in depth space. A small difference in the vertical location of isopycnals between experiments can cause a large apparent difference in ocean properties when the experiments are compared in depth space. As a result, it is often useful to compare ocean variables in density space.

This work compares two algorithms for plotting ocean properties in density coordinates, one written in FORTRAN with a python wrapper (xlayers), and one written in xarray (xarrayutils). Both algorithms conserve total salt content in the vertical, and both algorithms are easily parallelizable to enable plotting large datasets in density coordinates. As shown here, xlayers is a faster algorithm, but on some machines it requires more setup due to its reliance on a FORTRAN compiler.

Here, we apply these algorithms to plot salinity in density space in one of the Coupled Model Intercomparison Project Phase 6 (CMIP-6) models. We compare the salinity as a function of density in two future greenhouse-gas scenarios. In general, areas with net precipitation today are getting fresher and areas with net evaporation today are getting saltier (Durack and Wijffels, 2010). In climate models with a 1% per year CO\(_2\) increase, areas with net precipitation today experience increasing precipitation, and areas with net evaporation today experience a further reduction in precipitation in higher greenhouse-gas scenarios (Vallis et al. 2015). Here we compare two different greenhouse gas scenarios in the Shared Socioeconomic Pathways experiments. By plotting salinity in density space, we visualize how changes in salinity at the surface propagate along isopycnals to influence salinity concentrations in the ocean interior.

2. Loading CMIP-6 data

We choose to load temperature and salinity data from the ACCESS-ESM1-5 model, but this calculation can be performed on almost any CMIP-6 model currently available through Google's public datasets program.

```
#Load the packages needed
import numpy as np
import pandas as pd
import xarray as xr
import xesmf as xe
import cartopy
import cartopy.crs as ccrs
import zarr
import gcsfs
import matplotlib.pyplot as plt
import time
import warnings

# suppress warnings to improve readability
warnings.filterwarnings('ignore')
```

```
# Access the catalog of available CMIP-6 data
df = pd.read_csv('https://storage.googleapis.com/cmip6/cmip6-zarr-consolidated-
stores.csv')
# Connect to google cloud storage (this only needs to be created once)
gcs = gcsfs.GCSFileSystem(token='anon')

# Write a query to find temperature (thetao) and salinity (so) in the ACCESS model
df_sub = df.query("source_id == 'ACCESS-ESM1-5' and member_id=='r1i1p1f1' and
table_id=='Omon' \
                           and (experiment_id=='ssp126' or experiment_id=='ssp585')
\
                            and (variable_id=='thetao' or variable_id=='so')")
google_cloud_stores = df_sub.zstore.values
google_cloud_stores
```

```
array(['gs://cmip6/ScenarioMIP/CSIRO/ACCESS-ESM1-5/ssp126/r1i1p1f1/Omon/so/gn/',
       'gs://cmip6/ScenarioMIP/CSIRO/ACCESS-ESM1-5/ssp126/r1i1p1f1/Omon/thetao/gn/',
       'gs://cmip6/ScenarioMIP/CSIRO/ACCESS-ESM1-5/ssp585/r1i1p1f1/Omon/so/gn/',
       'gs://cmip6/ScenarioMIP/CSIRO/ACCESS-ESM1-5/ssp585/r1i1p1f1/Omon/thetao/gn/'],
      dtype=object)
```

We choose to compare properties for two future greenhouse gas scenarios, sometimes called Shared Soceioeconomic Pathways. The first is a lower-emissions scenario, SSP1-2.6, and the second is a higher-emissions scenario, SSP5-8.5. The following code accesses the specific zarr stores that contain temperature and salinity data for these two pathways and sets up pointers to this data.

```
# read the metadata from the cloud
datasets = [xr.open_zarr(gcs.get_mapper(zstore), consolidated=True)
            for zstore in google_cloud_stores]

# fix vertices coordinate
dsets_fixed = [ds.set_coords(['vertices_latitude', 'vertices_longitude'])
                for ds in datasets]

# separate data so we can merge it more sensibly
ds_so_ssp126, ds_thetao_ssp126, ds_so_ssp585, ds_thetao_ssp585 = datasets

# merge temperature and salinity
ds_ssp126 = xr.merge([dsets_fixed[0], dsets_fixed[1]], compat='identical')
ds_ssp126.coords['experiment_id'] = 'experiment_id', ['ssp126']
ds_ssp585 = xr.merge([dsets_fixed[2], dsets_fixed[3]], compat='identical')
ds_ssp585.coords['experiment_id'] = 'experiment_id', ['ssp585']

# merge SSP1-2.6 with SSP5-8.5
ds = xr.concat([ds_ssp126, ds_ssp585], dim='experiment_id', compat='identical')
ds
```

```
<xarray.Dataset>
Dimensions:             (bnds: 2, experiment_id: 2, i: 360, j: 300, lev: 50, time:
1032, vertices: 4)
Coordinates:
    vertices_longitude  (j, i, vertices) float64 80.0 81.0 81.0 ... 80.0 80.0
    time_bnds           (time, bnds) datetime64[ns] 2015-01-01 ... 2101-01-01
    lev_bnds            (lev, bnds) float64 0.0 10.0 10.0 ... 5.665e+03 6e+03
    longitude           (j, i) float64 80.5 81.5 82.5 83.5 ... 79.96 79.97 79.99
    vertices_latitude   (j, i, vertices) float64 -78.0 -78.0 ... 65.0 65.42
    latitude            (j, i) float64 -77.88 -77.88 -77.88 ... 65.63 65.21
  * j                   (j) int32 0 1 2 3 4 5 6 ... 293 294 295 296 297 298 299
  * i                   (i) int32 0 1 2 3 4 5 6 ... 353 354 355 356 357 358 359
  * time                (time) datetime64[ns] 2015-01-16T12:00:00 ... 2100-12-
16T12:00:00
  * lev                 (lev) float64 5.0 15.0 25.0 ... 5.499e+03 5.831e+03
  * experiment_id       (experiment_id) object 'ssp126' 'ssp585'
Dimensions without coordinates: bnds, vertices
Data variables:
    so                  (experiment_id, time, lev, j, i) float32
dask.array<chunksize=(1, 6, 50, 300, 360), meta=np.ndarray>
    thetao              (experiment_id, time, lev, j, i) float32
dask.array<chunksize=(1, 5, 50, 300, 360), meta=np.ndarray>
```

3. Preparing data for plotting in density coordinates

The data in these stores is given in depth space. In order to plot the salinity in potential density coordinates, we need the potential density field in depth space. In the cell below, the gibbs seawater toolbox is applied to find the surface-referenced potential density from the temperature and salinity. The ACCESS model actually uses the Jackett et al 2006 equation of state, but the density calculated here is a good approximation of the density in the ACCESS model.

```
import gsw

# calculate potential density from temperature and salinity
ds['dens'] = xr.apply_ufunc(gsw.density.sigma0, ds.so, ds.thetao,
                            dask='parallelized', output_dtypes=[float]).rename('dens')
ds
```

```
<xarray.Dataset>
Dimensions:             (bnds: 2, experiment_id: 2, i: 360, j: 300, lev: 50, time:
1032, vertices: 4)
Coordinates:
    vertices_longitude  (j, i, vertices) float64 80.0 81.0 81.0 ... 80.0 80.0
    time_bnds           (time, bnds) datetime64[ns] 2015-01-01 ... 2101-01-01
    lev_bnds            (lev, bnds) float64 0.0 10.0 10.0 ... 5.665e+03 6e+03
    longitude           (j, i) float64 80.5 81.5 82.5 83.5 ... 79.96 79.97 79.99
    vertices_latitude   (j, i, vertices) float64 -78.0 -78.0 ... 65.0 65.42
    latitude            (j, i) float64 -77.88 -77.88 -77.88 ... 65.63 65.21
  * j                   (j) int32 0 1 2 3 4 5 6 ... 293 294 295 296 297 298 299
  * i                   (i) int32 0 1 2 3 4 5 6 ... 353 354 355 356 357 358 359
  * time                (time) datetime64[ns] 2015-01-16T12:00:00 ... 2100-12-
16T12:00:00
  * lev                 (lev) float64 5.0 15.0 25.0 ... 5.499e+03 5.831e+03
  * experiment_id       (experiment_id) object 'ssp126' 'ssp585'
Dimensions without coordinates: bnds, vertices
Data variables:
    so                  (experiment_id, time, lev, j, i) float32
dask.array<chunksize=(1, 6, 50, 300, 360), meta=np.ndarray>
    thetao              (experiment_id, time, lev, j, i) float32
dask.array<chunksize=(1, 5, 50, 300, 360), meta=np.ndarray>
    dens                (experiment_id, time, lev, j, i) float64
dask.array<chunksize=(1, 5, 50, 300, 360), meta=np.ndarray>
```

Here, we plot sea surface salinity in 2015 (white contours in the figure below). Surface salinity tends to be lower in areas of high net precipitation and higher in areas of low net precipitation. Hence, if wet areas get wetter and dry areas get drier, we expect to see salty areas get saltier and fresh areas get fresher.

```
# find sea surface properties for SSP1-2.6 in 2015
ds_to_plot = ds.sel(experiment_id='ssp126').isel(lev=0).sel(time='2015').mean('time')

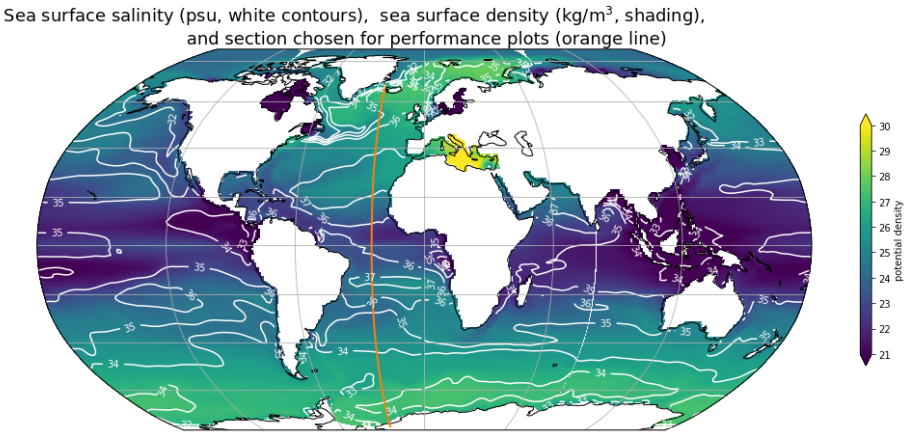
# Define sea-surface salinity and sea-surface density
ssd = ds_to_plot.dens
sss = ds_to_plot.so

# regrid sss for plotting
ds_out = xr.Dataset({'lat': (['lat'], np.arange(-80, 80, 1.0)),
                     'lon': (['lon'], np.arange(0, 360, 1.5)),
                    })
regridder = xe.Regridder(sss.rename({'longitude':'lon','latitude':'lat'}), ds_out,
'bilinear');
sss_regrid = regridder(sss);


# plot sea-surface salinity, sea-surface density and chosen section for performance
plots
fig = plt.figure(figsize=(18, 12))
ax = plt.axes(projection=ccrs.Robinson())
ax.coastlines()
ax.gridlines()
ssd.plot(x='longitude', y='latitude', ax=ax, transform=ccrs.PlateCarree(),
         vmin=21, vmax=30, cbar_kwargs={'shrink': 0.4, 'label':'potential density'})
CS = sss_regrid.plot.contour(levels=np.arange(32, 38, 1), transform=ccrs.PlateCarree(),
 colors='w')
CL = plt.clabel(CS, fmt='%d')

# note that the section location is i=255, j=0,250
lat_sec=ssd.sel(i=255, j=slice(0, 250)).latitude
lat_sec.plot(x='longitude', ax=ax, transform=ccrs.PlateCarree(), color='tab:orange',
linewidth=2)
ax.set_title('\n Sea surface salinity (psu, white contours),  sea surface density
(kg/m$^3$, shading),\
                          \n and section chosen for performance plots (orange line)',
fontsize=18);
```

```
Overwrite existing file: bilinear_300x360_160x240.nc
 You can set reuse_weights=True to save computing time.
```



These simulations begin in 2015. We choose to compare salinities in 2100, the final year of the simulation. In the parts 4 and 5 of this notebook, we assess the performance of the two packages by plotting salinity in density space on a single section, which is shown by the orange line in the plot above.

In order to do this, we load a small amount of data in the cell below.

```
# Define the densities used in the new coordinate system
t_vals = np.arange(20.4, 28.1, 0.1)


# Define lev_bounds to be the cell edges in the vertical
ds = ds.assign_coords(lev_bounds = np.concatenate([ds.lev_bnds[:, 0].values, [6000]]),
dims='lev_bounds');

# fill nans in the salinity with zeros
ds['so'] = ds['so'].fillna(0)

# load density and salinity on the chosen section in January 2100,
# so that both methods are on a level playing field for benchmarking
ds_slice = ds.sel(i=255, j=slice(0, 250), time='2100-
01').sel(experiment_id='ssp126').load()
ds_slice.load()
```

```
<xarray.Dataset>
Dimensions:             (bnds: 2, j: 251, lev: 50, lev_bounds: 51, time: 1, vertices:
4)
Coordinates:
    vertices_longitude  (j, vertices) float64 335.0 336.0 336.0 ... 335.9 334.9
    time_bnds           (time, bnds) datetime64[ns] 2100-01-01 2100-02-01
    lev_bnds            (lev, bnds) float64 0.0 10.0 10.0 ... 5.665e+03 6e+03
    longitude           (j) float64 335.5 335.5 335.5 ... 335.5 335.4 335.4
    vertices_latitude   (j, vertices) float64 -78.0 -78.0 -77.75 ... 67.67 67.66
    latitude            (j) float64 -77.88 -77.63 -77.38 ... 66.55 67.0 67.44
  * j                   (j) int32 0 1 2 3 4 5 6 ... 244 245 246 247 248 249 250
    i                   int32 255
  * time                (time) datetime64[ns] 2100-01-16T12:00:00
  * lev                 (lev) float64 5.0 15.0 25.0 ... 5.499e+03 5.831e+03
    experiment_id       <U6 'ssp126'
  * lev_bounds          (lev_bounds) float64 0.0 10.0 20.0 ... 5.665e+03 6e+03
    dims                <U10 'lev_bounds'
Dimensions without coordinates: bnds, vertices
Data variables:
    so                  (time, lev, j) float32 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0
    thetao              (time, lev, j) float32 nan nan nan nan ... nan nan nan
    dens                (time, lev, j) float64 nan nan nan nan ... nan nan nan
```

4. Evaluating the performance of xarrayutils

In this section, we use xarrayutils to transform salinity into density space.

```
# import xarrayutils
from xarrayutils.vertical_coordinates import conservative_remap
from xarrayutils.vertical_coordinates import linear_interpolation_regrid
```

xarrayutils transforms salinity into density space in two steps. First, there is a regridding step, in which the depths of the density surfaces are found. Then there is a remapping step, in which the salinity is remapped onto these new depth levels. Below, we perform both steps on the loaded section in order to assess the performance of the package.

```
# here, we define  a function that applies the xarrayutils package to the data
def apply_xarrayutils(ds_slice, t_vals):
    # define the new density grid
    density_values = xr.DataArray(t_vals, coords=[('t', t_vals)])

    # Find the depth of these surfaces
    z_dens_bounds = linear_interpolation_regrid(ds_slice.lev, ds_slice.dens,
density_values, z_bounds=ds_slice.lev_bounds ,target_value_dim='t',
z_bounds_dim='lev_bounds')

    # define the cell boundaries in the original salinity field
    bounds_original = ds_slice.lev_bounds

    # Remap salinity into density space
    ds_dens_cons = conservative_remap(ds_slice.so.squeeze(), bounds_original,
z_dens_bounds.squeeze(),
                                      z_dim='lev', z_bnd_dim='lev_bounds',
z_bnd_dim_target='regridded', mask=True
                                       ) # the associated depth dimensions for each array

    # replace the new depth dimension values with the appropriate depth (here the
middle of the density cell bounds)
    t_vals = z_dens_bounds.coords['regridded'].data
    t_vals = 0.5 * (t_vals[1:] + t_vals[0:-1])

    ds_dens_cons.coords['remapped'] = xr.DataArray(t_vals, coords=[('remapped',
t_vals)])
    return ds_dens_cons, z_dens_bounds


# here we apply the function to the loaded data
# start the timer
start = time.time()

# apply xarrayutils to a single slice of salinity data
ds_dens_cons, z_dens_bounds=apply_xarrayutils(ds_slice, t_vals)

# stop the timer
end = time.time()

#print how long this took
print('It takes {:.2f} seconds to perform this transformation using
xarrayutils'.format(end-start))
```

```
It takes 0.15 seconds to perform this transformation using xarrayutils
```

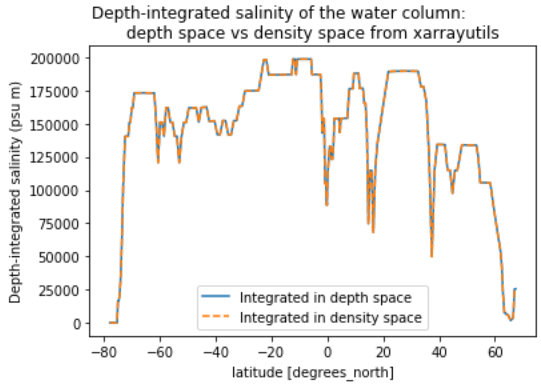
Next, we check that the depth-integrated salinity of the water column before the transformation is equal to the depth-integrated salinity of the column after the transformation.

```
# find the height of the grid cells in density space
dz_remapped = z_dens_bounds.diff('regridded').rename({'regridded':'remapped'})
dz_remapped.coords['remapped'] = ds_dens_cons.coords['remapped'].values

# find the depth integrated salinity in depth space (blue line) for a single experiment
for easy comparison
sal_depth = ((ds_slice.so*ds.lev_bnds.diff('bnds')).sum('lev'))

# find the depth integrated salinity in density space (orange dashed line)
sal_dens = (ds_dens_cons*dz_remapped).sum('remapped')

# plot the depth-integrated salinity before and after transformation
sal_depth.swap_dims({'j':'latitude'}).plot()
sal_dens.swap_dims({'j':'latitude'}).plot(linestyle='dashed')
plt.ylabel('Depth-integrated salinity (psu m)')
plt.legend(('Integrated in depth space', 'Integrated in density space'))
plt.title('Depth-integrated salinity of the water column:\
            \ndepth space vs density space from xarrayutils');
```

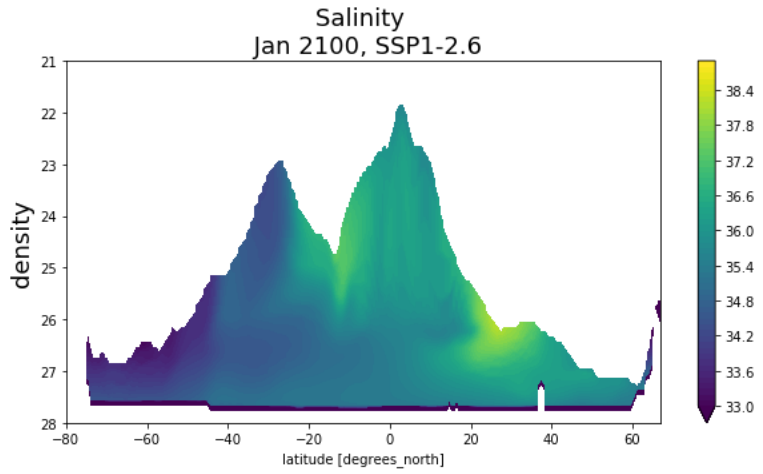


Note that the two lines in this figure are on top of each other: the total salt content of the water column before transformation equals the total salt content of the water column after transformation.

Next, we plot the difference in the salinity in January of 2100.

```
# find the salinity in density space
sal_dens = ds_dens_cons.where(dz_remapped>0)

# plot salinity in density space
fig, ax = plt.subplots(1, 1, figsize=(10, 5))
(sal_dens.squeeze().swap_dims({'j':'latitude'})
                       ).plot.contourf(x='latitude', y='remapped', ax=ax,
levels=np.arange(33, 39, 0.1))
ax.set_title('Salinity \n Jan 2100, SSP1-2.6', fontsize=18)
ax.set_ylabel('latitude', fontsize=18)
ax.set_ylabel('density', fontsize=18)
ax.set_ylim(28, 21)
ax.set_xlim(-80, 67);
```



5. Evaluating the performance of xlayers

xlayers uses a different algorithm from xarrayutils to transform data in one vertical coordinate system into another coordinate system. Here, we apply xlayers to the same dataset in order to compare its performance and results with those of xarrayutils.

```
# Load xlayers
from xlayers import finegrid, layers
from xlayers.core import layers_apply
```

```
# xlayers takes the locations of cell edges in a particular format, as calculated here
def finegrid_metrics(levs, lev_bnds):
    drF = np.diff(lev_bnds, axis=1)
    drC = np.concatenate((np.array([levs[0]]), np.diff(levs, axis=0),
np.array([lev_bnds[-1, -1]-levs[-1]])))
    return(drF, drC)

# fine_drf is the depth of cells in the vertical and fine_drc is the height difference
# between cell centers in the vertical
fine_drf, fine_drc = finegrid_metrics(ds_slice.lev.values, ds.lev_bnds.values)
```

xlayers does both regridding and remapping all in one step, but requires an initial calculation of various indices needed for binning. The output of `layers_apply` is thickness weighted i.e. \begin{equation} O(\rho_1 \mbox{ to } \rho_2)=\int_{z(\rho_1)}^{z(\rho_2)} I(z) dz , , \end{equation} where \(I(z)\) is the input and \(O(\rho_1 \mbox{ to } \rho_2)\) is the output in the layer between \(\rho_1\) and \(\rho_2\).

Now, we time the calculation of the thickness-weighted salinity, `sal_lay` and the thickness of each layer, `th_lay`, in density space using xlayers.

```
# start the timer
start = time.time()

# Calculated the thickness-weighted salinity using xlayers
sal_lay = layers_apply(ds_slice.so.squeeze(), ds_slice.dens.squeeze(), t_vals,
                           fine_drf, fine_drc, 10, 'lev', 'Tlev')


# Calculated the thickness of each new layer (needed to back out the real salinity)
# using xlayers
th_lay = layers_apply(xr.ones_like(ds_slice.so.squeeze()), ds_slice.dens.squeeze(),
t_vals,
                           fine_drf, fine_drc, 10, 'lev', 'Tlev')

# stop the timer
end = time.time()

#print how long this took
print('It takes {:.5f} seconds to perform this transformation using
xlayers'.format(end-start))
```

```
Warning: theta_in may not be monotonically ascending/descending
Warning: theta_in may not be monotonically ascending/descending
It takes 0.06132 seconds to perform this transformation using xlayers
```

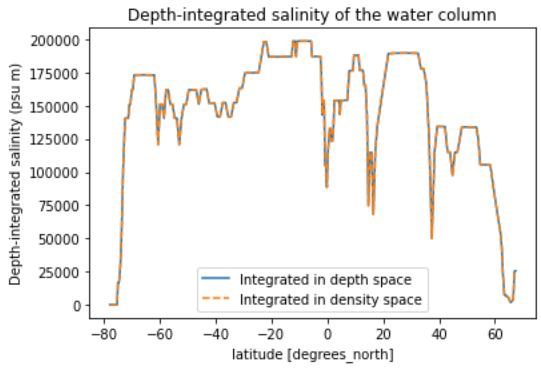
xlayers generally takes about 0.05 seconds to transform this salinity field into density coordinates. In other words xlayers is at least 2x faster than xarrayutils.

Again, we confirm that this algorithm conserves ocean properties by checking that the depth-integrated salinity before transformation is equal to the depth-integrated salinity after the transformation.

```
# find the depth integrated salinity in depth space (blue line)
sal_depth = ((ds_slice.so*ds.lev_bnds.diff('bnds')).sum('lev'))

# find the depth integrated salinity in density space (orange dashed line)
sal_dens = sal_lay.sum('Tlev')


# plot as a function of latitude
sal_depth.swap_dims({'j':'latitude'}).plot()
sal_dens.swap_dims({'j':'latitude'}).plot(linestyle='--')
plt.title('Salt content of the water column')
plt.ylabel('Depth-integrated salinity (psu m)')
plt.legend(('Integrated in depth space','Integrated in density space'))
plt.title('Depth-integrated salinity of the water column');
```

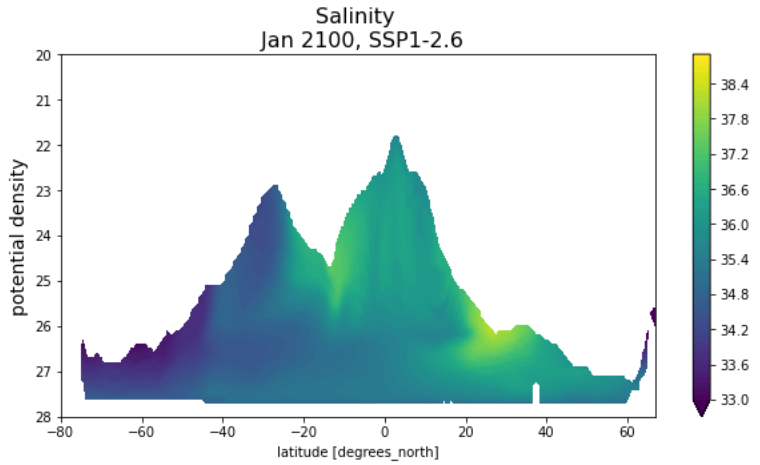


The orange dashed line is on top of the blue line, indicating that the total salt content of the water column before the transformation into density space equals the total salt content of the water column after transformation into density space.

We plot the difference in salinity in January 2100.

```
# Find the salinity in density space
sal_dens = sal_lay/th_lay

# Plot the salinity in density space
fig, ax = plt.subplots(1, 1, figsize=(10, 5))
sal_dens.plot.contourf(x='latitude',y='Tlev', levels=np.arange(33, 39, 0.1), ax=ax)
ax.set_title('Salinity \n Jan 2100, SSP1-2.6', fontsize=16)
ax.set_ylabel('potential density', fontsize=14)
ax.set_ylabel('latitude', fontsize=14)
ax.set_ylim(28, 20)
ax.set_xlim(-80 ,67);
```



Note that xarrayutils and xlayers produce very similar results, but xlayers generally produces more sensible values for salinity at the deepest density levels.

6. Using a cluster to examine “wet gets wetter, dry gets drier” pattern in density coordinates

The two scenarios begin from identical initial conditions, with greenhouse gas emissions growing faster in SSP5-8.5 than in SSP1-2.6. Hence we expect that wet regions will be wetter dry regions will be drier to a greater extent in SSP5-8.5 than in SSP1-2.6.

We want to compare our salinity in density space to sea surface salinity at the beginning of the simulation, in order to find out whether salty regions are getting saltier and fresh regions are getting fresher. Below, we calculate surface salinity as a function of density in the year 2015.

```
# This calculation is performed using the xhistogram package
from xhistogram.xarray import histogram

# Define density bins
bins1 = np.hstack([np.arange(20.4, 28.1, 0.2)])
bins2 = np.arange(20, 40, 0.1)

# define salinity at the surface in 2015
sss15 = ds.so.sel(experiment_id='ssp126').isel(lev=0).sel(time='2015')

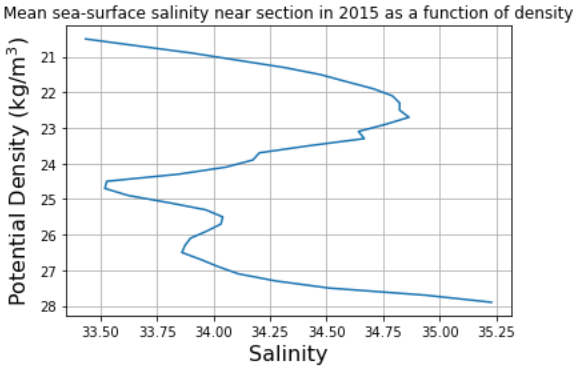
# define density at the surface in 2015
ssd15 =
ds.dens.sel(experiment_id='ssp126').isel(lev=0).sel(time='2015').chunk({'time':6})

#histogram of density at surface in 2015, weighted by salinity
hist_sal = histogram(ssd15, bins=[bins1], weights=sss15)

#histogram of density at surface in 2015
hist_dens = histogram(ssd15, bins=[bins1])

#mean sea-surface salinity as a function of density
mean_sal = hist_sal/hist_dens

# plot
mean_sal.plot(y='dens_bin')
plt.gca().invert_yaxis()
plt.grid()
plt.xlabel('Salinity', fontsize=16)
plt.ylabel('Potential Density (kg/m$^3$)', fontsize=16)
plt.title('Mean sea-surface salinity near section in 2015 as a function of density');
```



Surface salinities on isopycnals lighter than 24kg/m\(^3\) and heavier than 27kg/m\(^3\) are generally salty and surface salinities on isopycnals between 24kg/m\(^3\) and 27kg/m\(^3\) are generally fresh. Based on this plot, we expect isopycnals with densities less than 24kg/m\(^3\) to get saltier and isopycnals with densities between 24kg/m\(^3\) and 27kg/m\(^3\) to get fresher.

We would like to compare the salinities for the globe the whole year. However, in order to find the mean salinity in density space over the whole year, we must take the thickness-weighted average. Described in Young 2012, thickness weighting is important for preventing relatively thin isopycnal layers from disproportionately affecting the mean salinity. In discretized form it follows \begin{equation} \tilde{S}(\rho_1 \mbox{ to } \rho_2)=\frac{\int_{t_1}^{t_2}\int_{z(\rho_1)}^{z(\rho_2)} S(z) dz , dt}{\int_{t_1}^{t_2}\int_{z(\rho_1)}^{z(\rho_2)} dz, dt} , , \end{equation} where \(\rho_1\) and \(\rho_2\) are the bounds of the isopycnal layer, and \(\tilde{S}\) is the thickness weighted average salinity in the density layer between \(\rho_1\) and \(\rho_2\). The thickness-weighted time and zonal average salinity for the year 2100 is plotted below.

```
# Set up a cluster using dask
from dask_gateway import Gateway
from dask.distributed import Client

gateway = Gateway()
cluster = gateway.new_cluster()

cluster.scale(5)
cluster
```

```
#Make a client so you can see the progress of tasks (click the link that appears below)
client = Client(cluster)
client
```

Client

Scheduler: gateway://traefik-prod-dask-gateway.prod:80/prod.783b554396454767af7d5d3e8e0f0e16
Dashboard: https://hub.binder.pangeo.io/services/dask-gateway/clusters/prod.783b554396454767af7d5d3e8e0f0e16/status

Cluster

Workers: 0
Cores: 0
Memory: 0 B

First, we plot the mean for the year 2100 using xarrayutils.

```
# define section for the year 2100
ds_glob = ds.sel(i=255, time='2100', j=slice(0, 250))

# calculate the salinity in density space
ds_dens_cons, z_dens_bounds = apply_xarrayutils(ds_glob, t_vals)

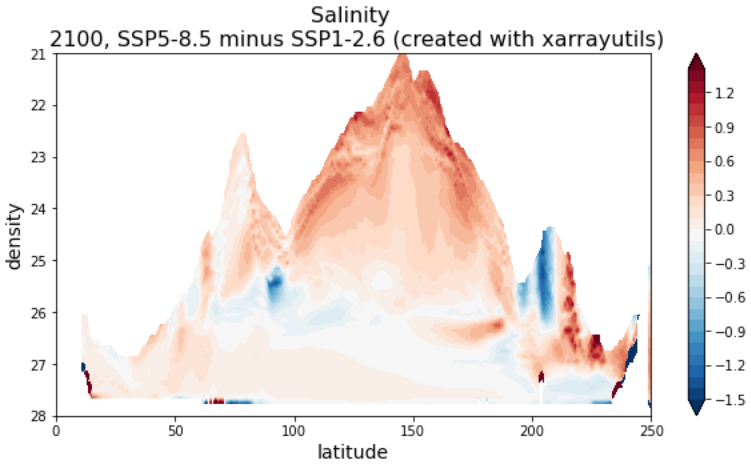
# find thickness weighting
dz_remapped = z_dens_bounds.diff('regridded').rename({'regridded':'remapped'})
dz_remapped.coords['remapped'] = ds_dens_cons.coords['remapped'].values

# Find thickness weighted salinity, averaged in time
tw_xau2100 = (ds_dens_cons*dz_remapped).sum('time')/dz_remapped.sum('time')

# Find difference between experiments SSP5-8.5 and SSP1-2.6
difference = tw_xau2100.sel(experiment_id='ssp585')-
tw_xau2100.sel(experiment_id='ssp126')

# plot difference
fig, ax = plt.subplots(1, 1, figsize=(10, 5))

difference.plot.contourf(x='j', y='remapped', ax=ax, levels=np.arange(-1.5, 1.5, 0.1),
cmap='RdBu_r')
ax.set_title('Salinity \n 2100, SSP5-8.5 minus SSP1-2.6 (created with xarrayutils)',
fontsize=16)
ax.set_ylabel('density', fontsize=14)
ax.set_xlabel('latitude', fontsize=14)
ax.set_ylim(28, 21);
```



We can plot the global and year average with xlayers though:

```
# define gloabel dataset for the year 2100
ds_glob = ds.sel(j=slice(0, 250), time='2100')

# prepare inputs for xarrayutils
salinity_in = ds_glob.so.chunk({'time': 1})
density_in = ds_glob.dens.chunk({'time': 1})

# Redefine density coords (the global ocean has some water that is lighter than the
lightest water in our section)
denslayers = np.hstack([np.arange(15, 28.1, 0.1)])

# find thicknes-weighted salinity
sal_lay = layers_apply(salinity_in, density_in, denslayers,
                           fine_drf, fine_drc, 10, 'lev', 'Tlev')

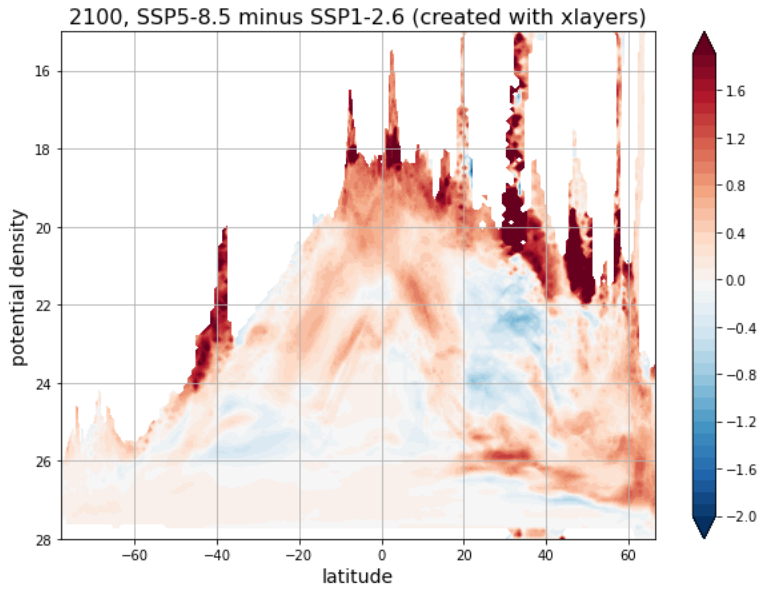

# Calculated the thickness of each new layer (needed to back out the real salinity)
th_lay = layers_apply(xr.ones_like(salinity_in), density_in, denslayers,
                           fine_drf, fine_drc, 10, 'lev', 'Tlev')


# Find thickness-weighted average salinity, over the whole globe for the year 2100
tw_xlay = (sal_lay.sum('time').sum('i')/th_lay.sum('time').sum('i'))

# Find the difference between experiments SSP5-8.5 and SSP1-2.6
difference = tw_xlay.sel(experiment_id='ssp585')-tw_xlay.sel(experiment_id='ssp126')

#define latitude for plotting (at the latitudes chosen, latitude is constant in i)
difference['mlatitude']=salinity_in['latitude'].mean('i')

# plot difference
fig, ax = plt.subplots(1, 1, figsize=(10, 7))
difference.plot.contourf(x='mlatitude',y='Tlev', levels=np.arange(-2, 2, 0.1), ax=ax)
ax.set_title('2100, SSP5-8.5 minus SSP1-2.6 (created with xlayers)', fontsize=16)
ax.set_ylabel('potential density', fontsize=14)
ax.set_xlabel('latitude', fontsize=14)
ax.grid()
ax.set_ylim(28, 15);
```



The above plot is the zonal average for the whole globe. In general the salinity appears to be increasing for densities greater than 24kg/m\(^3\) and decreasing for densities between 24kg/m\(^3\) and 27kg/m\(^3\), as predicted by “wet gets wetter, dry gets drier”. However, between 20N and 40N, there is a lot of freshening at lighter densities: further investigation of this is needed.

7. Conclusions

We transform salinity from the ACCESS model into density coordinates using two different property-conserving algorithms, xarrayutils and xlayers. Xlayers is about 2x faster than xarrayutils, but xarrayutils is easier to set up on some systems because it does not require a FORTRAN compiler. The salinity in isopycnal space is similar between the two algorithms. We plan to continue to refine these algorithms and perhaps combine them into a single package in future.

In general the subtropics has net evaporation, i.e. it is a dry region, and the subpolar region has net precipitation, i.e. it is a wetter region. As greenhouse gas levels increase, wet regions are generally thought to get wetter and dry regions are thought to get drier (see e.g. Vallis et al. 2015). Using output from two scenarios in the ACCESS model, we show here that the salinity in density space is consistent with this “wet gets wetter and dry gets drier” hypothesis: for scenarios with more greenhouse gases in the atmosphere, higher salinities are seen in the tropics and lower salinities are seen in the subtropics.

```
cluster.close()
```

8. Acknowledgements

Parts of this notebook were originally developed as part of the CMIP6 Hackathon, which was supported by funding from US-CLIVAR and OCB. We acknowledge support from Pangeo (NSF award 1740648).

9. References

Durack, P. J. and Wijffels, S. E., 2010: Fifty-Year Trends in Global Ocean Salinities and Their Relationship to Broad-Scale Warming. *J. Clim.*, **23**, 4342–4362, doi: 10.1175/2010JCLI3377.1

Vallis, G. K., Zurita-Gotor, P., Cairns, C., and Kidston, J., 2014: Response of the large-scale structure of the atmosphere to global warming. *Q. J. R. Meteorol. Soc*, **141**, 1479-1501, doi:10.1002/qj.2456

Bi, D., Dix, M., Marsland, S. J., O’Farrell, S., Rashid, H., Uotila, P., … and Yan, H. 2013., The ACCESS coupled model: description, control climate and evaluation. *Aust. Meteorol. Oceanogr. J*, **63**(1), 41-64.

Riahi, K., Van Vuuren, D. P., Kriegler, E., Edmonds, J., O’neill, B. C., Fujimori, S., … and Lutz, W., 2017. The shared socioeconomic pathways and their energy, land use, and greenhouse gas emissions implications: an overview. *Global Environmental Change*, **42**, 153-168.

By EarthCube Office