Creating a GEDI Geo-Locator

A Novel Method for Geographically Searching GEDI Version 1 Lidar Data

Ryan Waters

July, 2020

Element 84

Summary

This paper presents a novel method for geographically searching GEDI lidar data version 1. An example execution time over a simulated 2 years worth of GEDI coordinate data (10+ billion points and 149 GB in size) leveraging a parallel search on a single compute instance ran in less than 1/100th of a second and used 4.64 GB of memory. The search accuracy included an area averaging 4.73 meters above and below GEDI's swath path +/- error introduced by using a spherical earth model (< ~ 20 meters of error). The included code is from a working prototype in python using mostly SciPy, GeoPy and Numba libraries.

Introduction

GEDI[1] (Global Ecosystem Dynamics Investigation) released data to the public on January 21, 2020[2] and a geo-locator was released shortly thereafter on February 10th, 2020[3]. When satellites and other remote sensing instruments introduce new data to research communities a geo-locator may be written to enable that data to be searchable and more usable.

This paper presents an accurate and computationally efficient method of geo-searching GEDI data by taking the reader through the investigative process and some of the trial-and-error along the way to a solution. If you are a programmer, a scientist or somewhere on the path to becoming either (or both!) then you're the intended audience. Let's begin!

What is GEDI?

GEDI is a light ranging and detection (lidar) / laser altimeter mounted to the International Space Station (ISS). According to GEDI's website, *"GEDI will provide answers to how deforestation has contributed to atmospheric CO2 concentrations, how much carbon forests will absorb in the future, and how habitat degradation will affect global biodiversity." "GEDI has the highest resolution and densest sampling of any lidar ever put in orbit (and) is a full-waveform lidar instrument that makes detailed measurements of the 3D structure of the Earth's surface".*

[1] GEDI website - https://gedi.umd.edu/
[2] GEDI initial public release - https://earthdata.nasa.gov/learn/articles/first-gedi-data-available
[3] GEDI Finder announcement - https://gedi.umd.edu/lp-daac-release-of-gedi-finder/

What is the Problem?

GEDI, over the lifetime of its 2+ year mission, is projected to have 200+ TB of data files (known as granules) containing 149+ GB of geo-located point data. Working with that much data can be unwieldy and maybe we're only interested in a land area within a single country or large forest and, therefore, only need to download or otherwise access a subset of the available files. Our geo-locator service would take the geometry of our area of interest (AOI) and only return the URLs for data files that may be relevant. Specifically, we'll create a bounding box or rectangle of longitude and latitude around our AOI and then we'll determine which data intersect the bounding box.

Our goal, then, is to write software which can quickly search point data within a reasonable margin of error based on a user-supplied bounding box.

Understanding the Data

The path of a GEDI orbit drawn on a map has the shape of an imperfect sinusoidal line–a string laid across the globe. When those orbits build up over time they criss-cross forming a net or latticework.

An illustration of orbit 2352, courtesy of LP DAAC[4]
(from: https://lpdaac.usgs.gov/media/images/GEDI_L1B_Orbit02352_Orbit.original.png

[4] https://lpdaac.usgs.gov/products/gedi01_bv001/

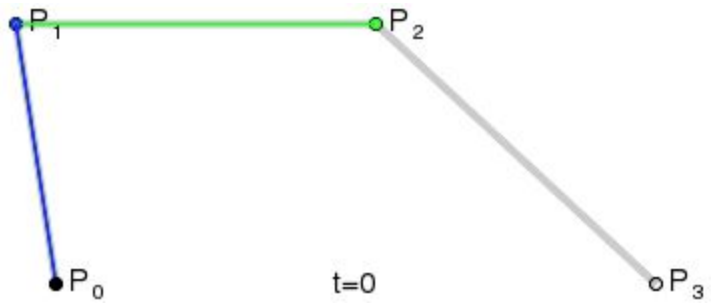
The range of a single orbit is about 51.6 degrees to -51.6 degrees latitude and will cover most longitudes. The data usually crosses the anti-meridian and the start and end of each orbit will be at different longitudes. For instance, one of the orbits (not shown) begins at 56.51247116483483 degrees longitude, crosses the antimeridian between longitudes 179.99995720557044 and -179.99976982688509, and then finishes its orbit at longitude 32.930112087033024.

If you zoom in on an orbit (and one orbit has multiple 'products' with one granule or file per orbit per product) you would see eight parallel ground tracks made by regularly spaced points of laser observations. If you're familiar with how farmers commonly grow corn, think of each point as a corn stalk with a string of points forming a corn row.

The geo-located waveforms or data points are about 25 meters wide and, on a given row or track, are spaced about 60 meters apart. There is about 600 meters of space between each track for a total swath width of 4.2 kilometers. The eight GEDI beams are: 0000, 0001, 0010, 0011, 0101, 0110, 1000, and 1011. Beam 0110 or beam 'six' is also the sixth beam ordinally and is the only one with associated coordinate data. The first beam has the highest latitude and the last beam has the lowest. The location data are degrees of longitude and latitude and are stored as double floats (8 bytes) in a Geographic Coordinate Reference System (CRS) ("WGS84"). Granules are persisted in a complicated filesystem-in-a-file format called HDF5[5].

What will be our Approach?

The nature of GEDI data makes for interesting work when implementing a locator. GEDI has no hard boundaries apart from each orbit, just individual points. The granules themselves aren't associated with pre-defined tiles or bounding boxes like some other remote sensing products. If you've ever worked with splines in a vector graphics program like Inkscape, you might know they're based on math. With the path of the data being mostly sinusoidal, what if we were able to fit a mathematical function to each orbit's curve?



A spline animation showing how a curve can be constructed from control points.

[5] HDF file format - https://en.wikipedia.org/wiki/Hierarchical_Data_Format

Animation by Phil Tregoning, Public Domain.

Earlier I had mentioned we need to be accurate within a margin of error. Geocoding, for example, is considered “highly accurate” if it has a margin of error less than 50 meters[6]. We’ll see how close we can get to that.

I’ll include some code examples written in Python but many programming languages would be appropriate for a project like this one.

Investigations: Fitting a Trig Function

First, we’ll want to download a sample of Level 1B GEDI granules[7] and extract the longitude and latitude for each point of the 0110 beam. They’re about 40MB in size when stored as JSON and will be the basis data for fitting functions.

```
import h5py
import json
import re
import os

def extract_coords():
    from_path = './path/to/GEDI01_B'
    to_path = './path/to/gedi_l1b_coords_2019_05'

    for root, dirs, files in os.walk(from_path):
        if len(files) > 0:
            for f in files:
                with open(os.path.join(root, f), 'rb') as g:
                    granule = h5py.File(g, 'r')
                    output_name = re.sub(r'\.h5$', '', f)
                    print(output_name)
                    with open(os.path.join(to_path, output_name), 'w') as output_file:
                        output_data = {'name': output_name,
                                       'lons':
granule['BEAM0110/geolocation/longitude_bin0'][()].tolist(),
                                       'lats':
granule['BEAM0110/geolocation/latitude_bin0'][()].tolist()}
                        json.dump(output_data, output_file)
```

6 https://www.esri.com/arcgis-blog/products/analytics/analytics/geocoding-delivering-high-location-accuracy/

7 https://lpdaac.usgs.gov/tools/data-pool/

Our first test will be to see how a sinusoidal function fits the data. The following code uses SciPy's `curve_fit`[8] to take a sine function and adjust its arguments according to a least squares linear regression[9]. If you're unfamiliar with least squares regression this short video gives a great introduction.

> *Regarding the anti-meridian: it's the global boundary opposite the (prime) meridian where longitude is 0. The anti-meridian is 180 degrees (or -180 degrees). From a coordinate reference system standpoint, we consider the anti-meridian to be the beginning and the end and geographic features which cross it should be vertically cut in two. In our case, any orbit of GEDI data we deal with will be cut at this point as needed.*

Here's an attempt at fitting a full orbit of data:

```
from geopy.distance import geodesic
import json
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import curve_fit
from statistics import mean
import time


def split_on_anti_meridian(lons, lats):
    """ Take two lists of lists, lons and lats, and return two lists of lists
        with lons and lats split on the antimeridian """
    # Working across the anti meridian boundary is numerically challenging so
    #  it's best to break up our data along that boundary.

    def _fn(lons, lats):
        am = 0
        for i in range(1, len(lons)):
            if (lons[i-1] > 0) and (lons[i] < 0):
                am = i
                break
        if am > 0:
            return [lons[0:am], lons[am:]], [lats[0:am], lats[am:]]
        else:
            return [lons], [lats]
```

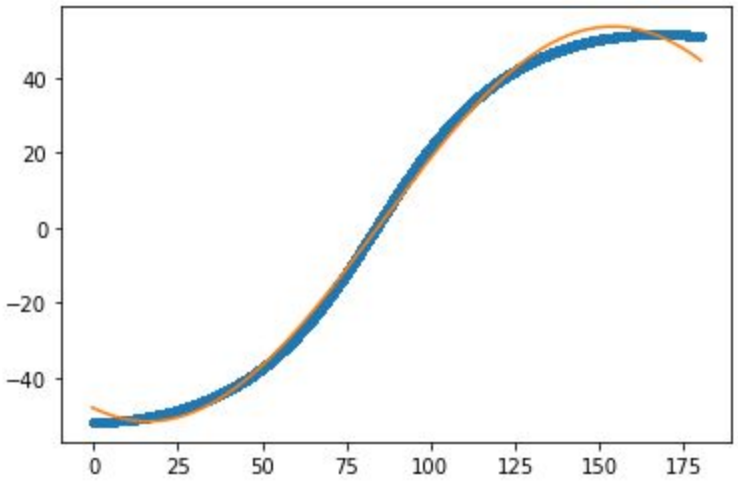
[8] Scipy curve_fit - https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html
[9] Least squares regression - https://en.wikipedia.org/wiki/Least_squares

```
    acc_lons = []
    acc_lats = []
    for lons_part, lats_part in zip(lons, lats):
        lons_parts, lats_parts = _fn(lons_part, lats_part)
        for lons_ in lons_parts:
            acc_lons.append(lons_)
        for lats_ in lats_parts:
            acc_lats.append(lats_)

    return acc_lons, acc_lats


def get_fn_error_rates(fn, fit, lons, lats):
    """ Take a function fn, array of longitude values lons, and array of
        latitude values lats and return the mean average error and maximum
        error between function-calculated latitude and reference latitude
        values. """
    # Because we're computing distance between latitudes on the same longitude
    #   the longitudinal value may be any value; 0, in this case
    # One degree latitude is 110.567 km at the equator and 111.699 at the poles.
    #   We could have used the Haversine formula which assumes a Great-circle but
    #   the geodetic distance uses an ellipsoid representation and is more
    #   accurate (WGS-84 ellipsoid by default)
    deltas_lat = [((lats[i], 0), (fn(lons[i], *fit[0]), 0)) for i in range(len(lons))]
    deltas_km = [geodesic(*points).kilometers for points in deltas_lat]
    return mean(deltas_km), max(deltas_km)


# sinusoidal function
def my_sin(x, freq, amplitude, phase, offset):
    return np.sin(x * freq + phase) * amplitude + offset

##########################################################################

start_time = time.time()
filename = "./gedi_coords/GEDI01_B_2019108002011_O01959_T03909_02_003_01"
lons = []
lats = []
with open(filename) as coords:
    lonlat = json.load(coords)
    lons_, lats_ = split_on_anti_meridian([lonlat['lons']], [lonlat['lats']])
    for i in range(len(lons_)):
        lons.append(np.array([float(x) for x in lons_[i]]))
```

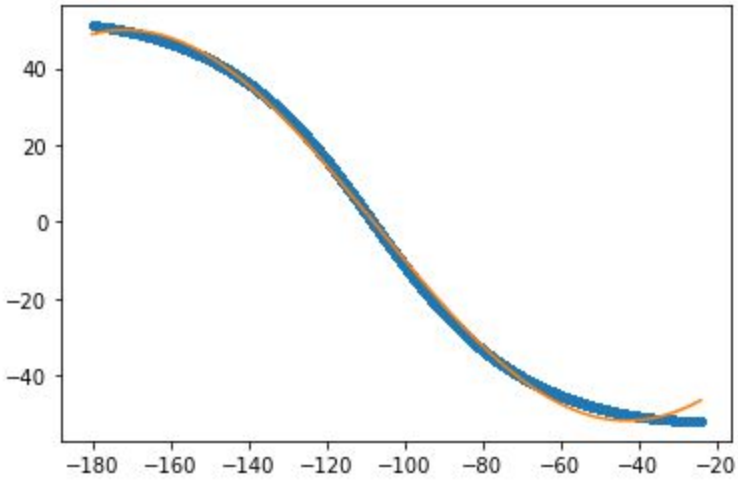
```
        lats.append(np.array([float(y) for y in lats_[i]]))

guess_freq = [0.1, 0.04]
for i in range(0, 2):
    guess_amplitude = 1
    guess_phase = 1
    guess_offset = 0

    p0=[guess_freq[i], guess_amplitude, guess_phase, guess_offset]

    # find the parameters that fit the function to all of the data
    fit = curve_fit(my_sin, lons[i], lats[i], p0=p0)

    # create lat data based on lon data and fit parameters
    data_fit = my_sin(lons[i], *fit[0])

    # error rates
    avg_error, max_error = get_fn_error_rates(my_sin, fit, lons[i], lats[i])
    print(f'Avg error: {avg_error}, Max error: {max_error}')

    print(*fit[0])

    # lon, lat format
    plt.plot(lons[i], lats[i], '.')  # blue
    plt.plot(lons[i], data_fit)      # orange
    plt.show()

total_time = time.time() - start_time
print('Total time taken: {0:.3f} seconds'.format(total_time))
```

(output)

```
Avg error: 174.42631416942262, Max error: 713.7965575081116
0.022819569935377544 -52.727145520176194 7.484391428110654 1.0427322123266742
```




```
Avg error: 143.76553519578917, Max error: 595.444003946346
0.024425584080654244 50.83506880084666 -0.5238782974421087 -0.9183431695966431
```




```
Total time taken: 250.415 seconds
```

(end output)

What we see above are two outputs for granule `GEDI01_B_2019108002011_O01959_T03909_02_003_01`, one output each for data before and after the anti-meridian. Data points are the blue line and our fitted function is in orange. We derive the error by taking each of the 1,097,865 data points and finding an ellipsoid model geodetic distance[10] between actual latitude data and the fitted function

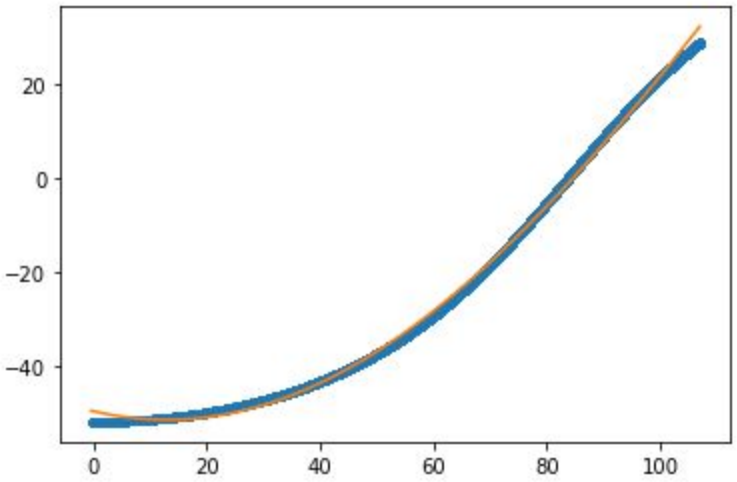
[10] https://en.wikipedia.org/wiki/Geodesic

computed latitude at a data point's longitude. The maximum error is the largest of these distances and the average is the mean across the population.

Results will vary by granule/orbit but, in this case, data before the anti-meridian has an average error of ~ 174 km and max error of ~ 714 km. Unfortunately these results are abysmal. There are 511,535 data points before the anti-meridian; would we get better results by fitting the same function across fewer points?

(output)

```
Avg error: 89.71874863551194, Max error: 383.5056917972365
0.013393636818693024 120.91332836643203 4.538724550082946 69.54807178206747
```




```
Total time taken: 90.574 seconds
```

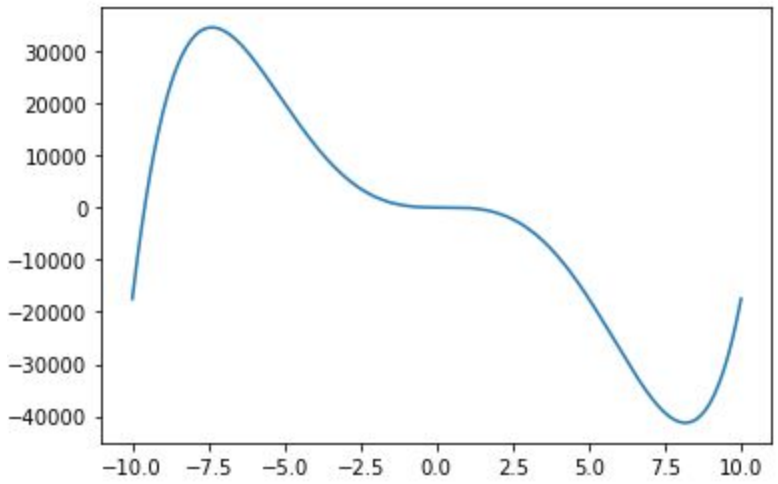
(end of output)

Taking 400,000 points instead of 511,535 is still far from our maximum error goal but notice how the average error is ~ 90 km and maxes out at ~ 383 km. In this case, decreasing the amount of data we're trying to fit by 22% improved our error rates by about 37%. We'll use this as a hint for future work.

In addition to fitting functions to smaller data sets, what if we also tried fitting a different kind of function like polynomials?

Investigations: Fitting a Polynomial, Part 1

A polynomial is "an expression consisting of variables … and coefficients"[11]. The degree of a polynomial is based on the highest exponent.



An example polynomial: $2x^5 - 3x^4 - 200x^3 + 125x^2 + 1.5x + 9$

To establish a baseline of accuracy over our sample granule, here are the results for each side of the anti-meridian split (the same spatial extent used in each series) using 5 to 50 degree polynomials in 5 degree increments:

```
Before anti-meridian:
polynomial degree, avg error, max error
5, 76.55509637535208, 428.51018690324673
10, 10.61696980812977, 52.66230616071602
15, 1.3742996807505743, 8.416259452795414
20, 0.9463589112804677, 5.84428483614433
25, 0.36587500365615094, 3.7612231913568124
30, 0.6121521654083976, 3.884122556954003
35, 0.3946447194103193, 3.752943781548051
40, 0.20621685112677862, 2.297003965498448
45, 0.34909398944147363, 3.2346368757710366
50, 0.17560606903268472, 2.0494112740157284

After anti-meridian:
polynomial degree, avg error, max error
5, 56.536427851501415, 299.8989631295974
10, 4.858796049696968, 39.10767524010038
15, 0.8223061044425902, 5.550106052769198
20, 0.6138984906060114, 3.9214406917995404
25, 0.44339075908992576, 3.437786979647548
30, 0.24080619500467015, 2.7728586944941496
35, 0.11282835661474254, 1.2100705819471451
```

[11] https://en.wikipedia.org/wiki/Polynomial

```
40, 0.17938472689470317, 2.041906852656545
45, 0.08118271662070943, 0.9414158801294034
50, 0.12419120539731891, 1.5798505249781858
```

It's interesting how, in this case, beyond a ~ 35 degree polynomial the fit stops reliably improving.

> *SciPy documentation warns us against using high degree polynomials due to loss of precision. The loss of digits beyond the floating point representation's mantissa[12] become significant when a number is taken to such a high exponent as the loss of precision compounds itself. We could change our representation to something more than 64-bits but that trades speed for accuracy and would limit our 3rd-party software library options. That said, it would be interesting to experiment with higher-bit representations.*

To achieve greater accuracy we can repeat what we did earlier when we fit a trig function: apply function fitting across slices or partitions of data of varying sizes. The following experiment divides the data into an increasing number of partitions, 1 through 100, and applies polynomial fitting functions of degree 2 through 40 (quadratic, cubic, quartic, and so on). It's worth pointing out that NumPy is doing all the heavy lifting here - thank you NumPy:

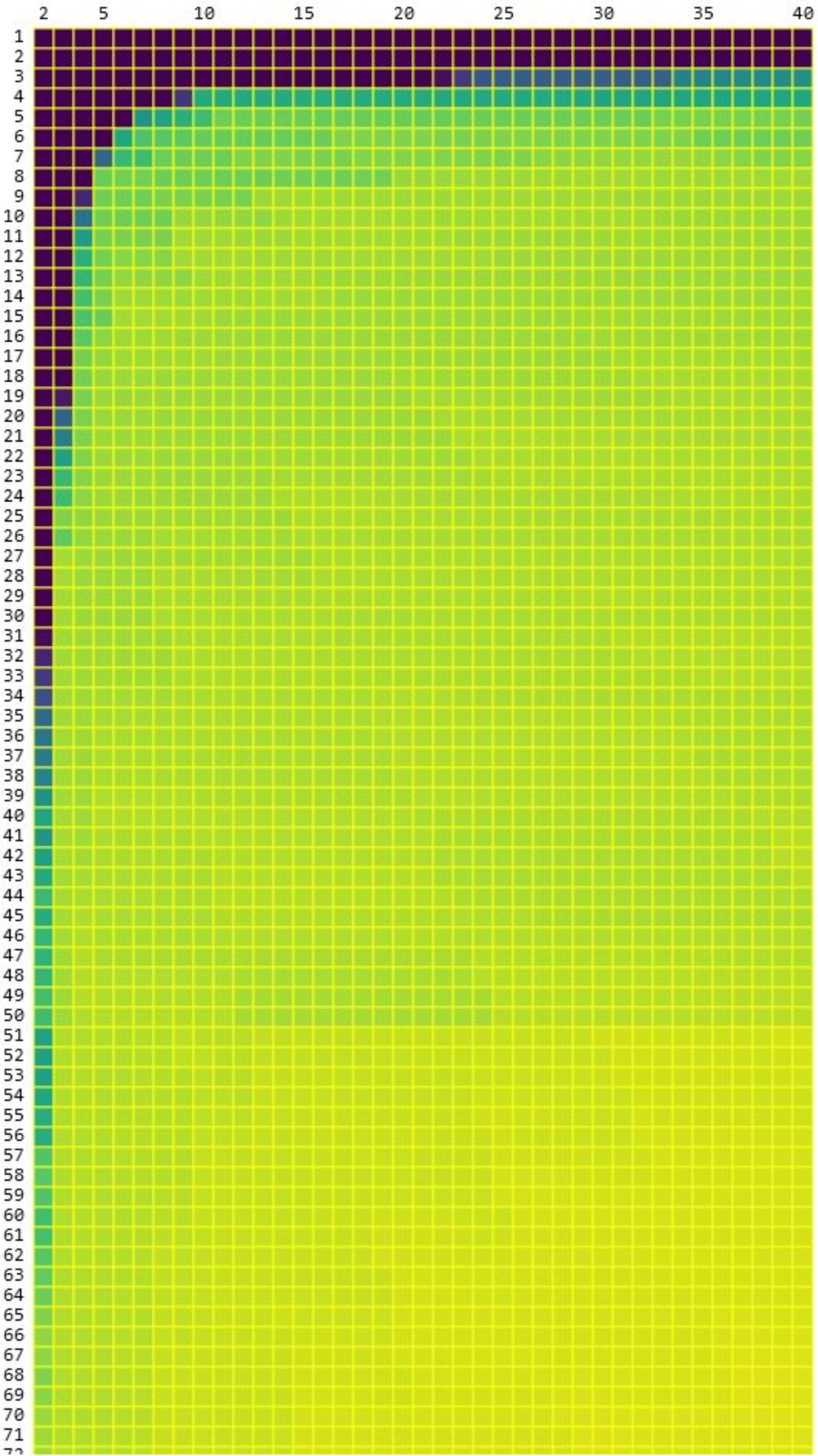
```
from geopy.distance import geodesic
import json
import math
import numpy as np
from statistics import mean
import time
import warnings


def split_on_anti_meridian(lons, lats):
    """ Take two lists of lists, lons and lats, and return two lists of lists
        with lons and lats split on the antimeridian """
    # Working across the anti meridian boundary is numerically challenging so
    #  it's best to break up our data along that boundary.

    def _fn(lons, lats):
        am = 0
        for i in range(1, len(lons)):
            if (lons[i-1] > 0) and (lons[i] < 0):
                am = i
                break
```

[12] https://fabiensanglard.net/floating_point_visually_explained/

```
        if am > 0:
            return [lons[0:am], lons[am:]], [lats[0:am], lats[am:]]
        else:
            return [lons], [lats]

    acc_lons = []
    acc_lats = []
    for lons_part, lats_part in zip(lons, lats):
        lons_parts, lats_parts = _fn(lons_part, lats_part)
        for lons_ in lons_parts:
            acc_lons.append(lons_)
        for lats_ in lats_parts:
            acc_lats.append(lats_)

    return acc_lons, acc_lats

def generate_polynomial(deg, lons, lats):
    """ Takes degree of polynomial deg, array of longitude values lons, and
        array of latitude values lats and attempts to fit a polynomial to the
        data via non-linear regression (least squares).  Returns the
        polynomial."""
    with warnings.catch_warnings():
        warnings.simplefilter('ignore', np.RankWarning)
        pf = np.polyfit(lons, lats, deg)
        return np.poly1d(pf)


def get_fn_error_rates(fn, lons, lats):
    """ Take a function fn, array of longitude values lons, and array of
        latitude values lats and return the mean average error and maximum
        error between function-calculated latitude and reference latitude
        values. """
    # Because we're computing distance between latitudes on the same longitude
    #   the longitudinal value may be any value; 0, in this case
    # One degree latitude is 110.574 km at the equator and 111.699 at the poles.
    #   We could have used the Haversine formula which assumes a Great-circle but
    #   the geodetic distance uses an ellipsoid representation and is more
    #   accurate (WGS-84 ellipsoid by default, same as GEDI)
    deltas_lat = [((lats[i], 0), (fn(lons[i]), 0)) for i in range(len(lons))]
    deltas_km = [geodesic(*points).kilometers for points in deltas_lat]
    return mean(deltas_km), max(deltas_km)

# # #
```

```
input_filename = "./gedi_coords/GEDI01_B_2019108002011_O01959_T03909_02_003_01"
output_filename = "./error_rates.csv"
partition_range = [1, 101]
pn_degree_range = [2, 41]

start_time = time.time()

with open(input_filename) as coords:
    with open(output_filename, 'w') as output:
        lonlat = json.load(coords)
        lons_, lats_ = split_on_anti_meridian([lonlat['lons']], [lonlat['lats']])

        output.write('total partitions, polynomial degree, avg error, max error\n')
        for parts in range(*partition_range):
            # We're only going to deal with points before the anti-meridian for
            #  this test.
            sample_size = math.ceil(len(lons_[0]) / parts)
            lons = np.array(lons_[0][:sample_size])
            lats = np.array(lats_[0][:sample_size])

            for deg in range(*pn_degree_range):
                pn = generate_polynomial(deg, lons, lats)
                avg_error, max_error = get_fn_error_rates(pn, lons, lats)
                output.write(f'{parts}, {deg}, {avg_error}, {max_error}\n')

total_time = time.time() - start_time
print('Total time taken: {0:.3f} seconds'.format(total_time))
```

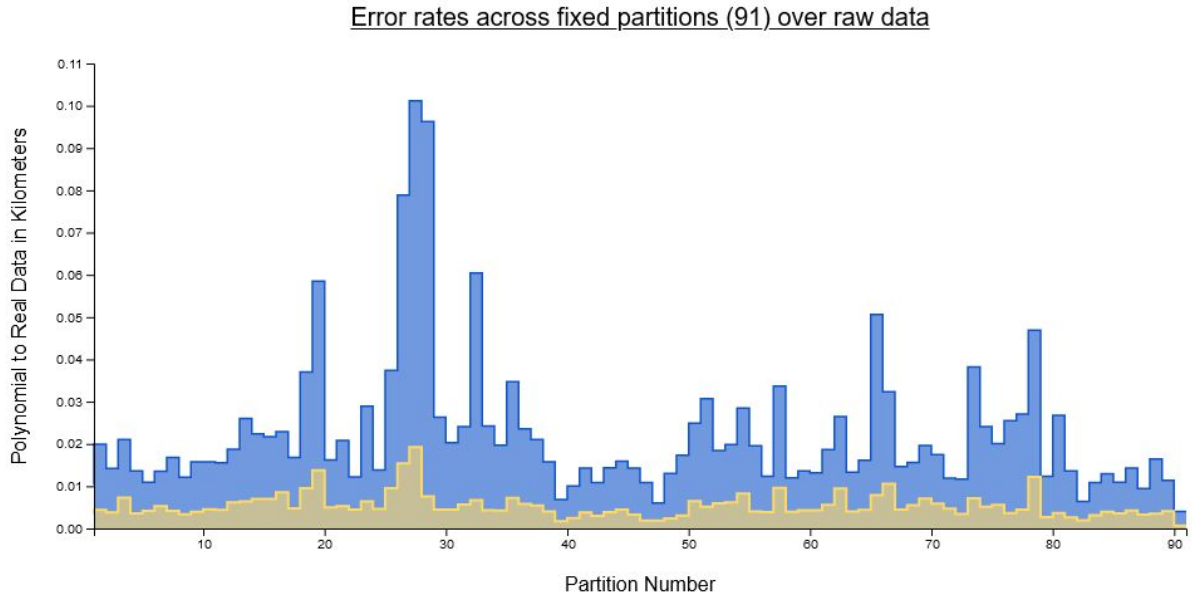
In order to better visualize the results, here is a heatmap with hue based on maximum error. The y-axis represents the number of partitions and the x-axis is the degree of polynomial. The first partition is shown and meant to be a representative sample of the rest (graph code not shown):


2 5 10 15 20 25 30 35 40
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71

It took 7.75 hours to generate the data for this graph on an AMD FX-8350

There are a couple patterns worth noting: first, within a given number of partitions and as we increase the degree of the polynomials we at some point early on see diminishing and then negligible returns on error rate reduction. Second, we seem to achieve decent accuracy with low degree polynomials if the number of partitions is relatively high; increasing the total number of partitions reduces the amount of relative change and flattens the curve within the dataset. According to this test 90 partitions and a 7 degree polynomial achieves an accuracy of better than 20 meters.

However, if we look at the accuracy of the partitions beyond the first we see a very different picture. Shown in the chart below are all ~90 partitions and the error rate varies widely. Here's the accuracy over our sample granule `GEDI01_B_2019108002011_O01959_T03909_02_003_01`:



Maximum error in Blue, Average error in yellow

The polynomials fit to each partition tend to achieve a maximum error of 20 meters or less but they can also fit much worse at over 100 meters (0.1 km). This variance is concerning and suggests we should probably seek another approach.

On the Uses of Error

The significance of generating a maximum error rate for each partition is twofold–we've seen one use so far and we'll see a second use later. The error rate indicates how accurate a fitted function is to the actual GEDI coordinates. If the curve is of acceptable quality then we've fulfilled an important requirement and come closer to a usable mathematical proxy.

The utility of maximum error continues when we use fitted functions as the 'engine' in our search engine. We'll take the lon/lat values of the bounding box around a user's AOI and see if they intersect with data generated by the polynomial, +/- the maximum error. Knowing the longitudinal extent of a given partition, a polynomial and the maximum (latitudinal) error of the polynomial over that partition means we'll know whether a granule *might* intersect a bounding box, with our degree of uncertainty equal to the error. In the end we may end up returning a granule that doesn't actually intersect an AOI but we won't miss any granules that should match. In other words, given the tools we're building up we know there may be false positives but there shouldn't be any false negatives.

Investigations: Fitting a Polynomial, Part 2

Our strategy for fitting polynomials to the data so far has revolved around defining the number of fixed width partitions into which we subdivide the data as well as the degree of polynomial applied to each partition. What if instead we turn things around and take our error rate as our goal, our invariant, and let the code choose both the number of partitions and the partition width needed to achieve that goal? We could also let the code choose the degree of polynomial but because degree has diminishing to negligible returns as we increase its value, we could instead make that, too, an invariant and choose the polynomial degree ourselves. A drawback to using arbitrarily large polynomials is that their storage cost is degree-plus-one times the number of partitions times the number of orbits in our dataset and it's beginning to seem possible that the final searchable dataset could reside entirely in memory; this would make for faster searches. Having consistent memory sizes for each polynomial also means we may be able to traverse our data structures more efficiently and use libraries such as numpy and numba which work best (in the case of NumPy) or expect (in the case of Numba) uniformity of type (homogeneous arrays).

Here is the code to dynamically size partitions over the sample granules:

```
# (other functions same as before)

def apply_partition_latitudes(p):
    """ Takes a dict p representing partition data and mutates that partition
        data to include its minimum and maximum latitude values. """
    bbox = poly_bbox(p['pn'], p['lon_min'], p['lon_max'], p['max_error'])
    lat_range = {'lat_min': bbox[0][1], 'lat_max': bbox[1][1]}
    return {**p, **lat_range}


def create_dynamic_partitions(lons__, lats__):
    max_inc_idx = 0
    max_increments = [100, 1000, 20000]
```

```
    pn_degree = 7
    error_threshold = ERROR_THRESHOLD
    results = []
    #partition_num = 1

    lons_, lats_ = split_on_anti_meridian([lons__], [lats__], pn_degree)

    for s_lons, s_lats in zip(lons_, lats_):
        lower_bound = 0
        upper_bound = pn_degree + 1
        known_good = pn_degree + 1
        known_bad = len(s_lons)

        while (upper_bound + 1) < len(s_lons):
            while True:
                lons = s_lons[lower_bound:upper_bound]
                lats = s_lats[lower_bound:upper_bound]
                pn = generate_polynomial(pn_degree, lons, lats)
                avg_error, max_error = get_fn_error_rates(pn, lons, lats)

                #print(f'p:{partition_num}, lower_bound:{lower_bound},
known_good:{known_good}, upper_bound:{upper_bound}, known_bad:{known_bad},
max_error:{max_error}')

                # Partition width must be at least one more than the
                #   degree of polynomial.
                if ((known_bad - known_good) == 1) or (known_good > known_bad):
                    upper_bound = known_good
                    lons = s_lons[lower_bound:upper_bound]
                    lats = s_lats[lower_bound:upper_bound]
                    pn = generate_polynomial(pn_degree, lons, lats)
                    avg_error, max_error = get_fn_error_rates(pn, lons, lats)
                    break

                # the partition may grow
                if max_error <= error_threshold:
                    known_good = upper_bound
                    maybe_increment = int((known_bad - upper_bound) / 2)
                    if maybe_increment > max_increments[max_inc_idx]:
                        increment = max_increments[max_inc_idx]
                        max_inc_idx += 1
                        if max_inc_idx >= len(max_increments):
                            max_inc_idx = len(max_increments) - 1
```

```
                else:
                    increment = maybe_increment
                upper_bound = upper_bound + increment
                if upper_bound == known_good:
                    upper_bound += 1
            # the partition must shrink
            else:
                known_bad = upper_bound
                upper_bound = upper_bound - int((known_bad - known_good) / 2)
                if upper_bound == known_bad:
                    upper_bound -= 1
                max_inc_idx = 0

        lon_min = s_lons[lower_bound]
        lon_max = s_lons[upper_bound]

        partition = {'left_extent': lower_bound,
                     'right_extent': upper_bound,
                     'pn': tuple(pn.c),
                     'lon_min': lon_min,
                     'lon_max': lon_max,
                     'max_error': max_error,
                     'avg_error': avg_error}

        # compute bounding box for current partition
        partition = apply_partition_latitudes(partition)

        results.append(partition)

        lower_bound = upper_bound + 1
        upper_bound = lower_bound + pn_degree + 1
        known_good = lower_bound + pn_degree + 1
        known_bad = len(s_lons)

        #partition_num += 1

# Sort partitions by longitude. This will introduce a gap in the data if
#   the orbit passes through the anti-meridian (which happens most of the
#   time). It's not a problem for the search but it's worth knowing it's
#   there. Makes the data binary search compatible on a per granule or
#   orbit basis.
return sorted(results, key=lambda x: x['lon_min'])
```

```
def filter_invalid_coords(lons_, lats_):
    """ Takes an array of longitude values lons_ and an array of latitude
        values lats_ and returns a copy of each with invalid with invalid
        point data removed. """
    lons = deepcopy(lons_)
    lats = deepcopy(lats_)
    for i in range(len(lons)-1, 0, -1):
        if (lons[i] < -180) or (lons[i] > 180) or math.isnan(lons[i]) \
          or (lats[i] < -90) or (lats[i] > 90) or math.isnan(lats[i]):
            print(f'Bad data removed at index {i}: {lons[i]}, {lats[i]}')
            del lons[i]
            del lats[i]
    return lons, lats


def do(input_filenames, input_path, output_path):
    for input_filename in input_filenames:
        print('\n' + input_filename)
        with open(input_path + input_filename) as coords:

            coords_json = json.load(coords)

            # Check for valid data; without this check I was receiving the
            #   following error on some files: "ValueError: On entry to DLASCL
            #   parameter number 4 had an illegal value". Turns out there are
            #   some NaN's in GEDI coordinate data.
            lons, lats = filter_invalid_coords(coords_json['lons'],
                                               coords_json['lats'])

            partitions = create_dynamic_partitions(lons, lats)

            with open(output_path + input_filename + '.json', 'w') as output_file:
                json.dump(partitions, output_file)


if __name__ == '__main__':
    start_time = time.time()

    with Pool(multiprocessing.cpu_count()) as p:
       p.starmap(do, [[[x], input_path, OUTPUT_PATH] for x in
(get_filenames(INPUT_PATH))])
```

```
    print('Total time taken: {0:.3f} seconds'.format(time.time() - start_time))
```

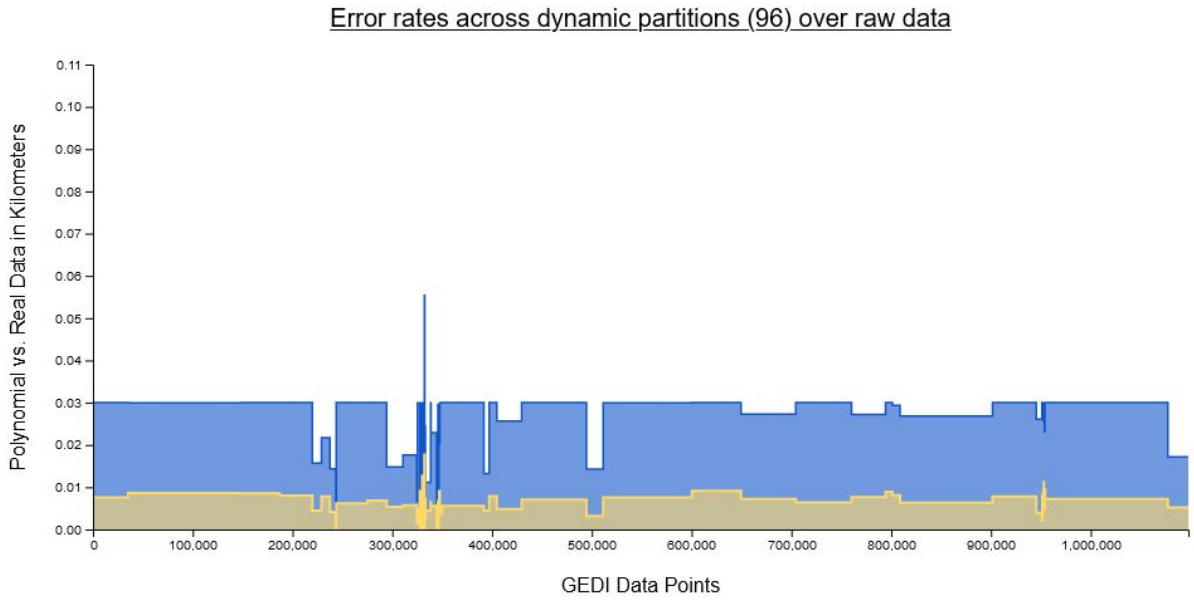
And here is some sample output which may help in understanding some of what it does:

```
GEDI01_B_2019108002011_O01959_T03909_02_003_01
p:1, lower_bound:0, known_good:8, upper_bound:8, known_bad:496607,
max_error:7.589719382119511e-06
p:1, lower_bound:0, known_good:8, upper_bound:108, known_bad:496607,
max_error:0.00013878764480470632
p:1, lower_bound:0, known_good:108, upper_bound:1108, known_bad:496607,
max_error:0.0011892317332065987
p:1, lower_bound:0, known_good:1108, upper_bound:21108, known_bad:496607,
max_error:0.027517041144147266
p:1, lower_bound:0, known_good:21108, upper_bound:41108, known_bad:496607,
max_error:0.03017484874312236
p:1, lower_bound:0, known_good:21108, upper_bound:31108, known_bad:41108,
max_error:0.027939061828534333
p:1, lower_bound:0, known_good:31108, upper_bound:31208, known_bad:41108,
max_error:0.02796734990838053
p:1, lower_bound:0, known_good:31208, upper_bound:32208, known_bad:41108,
max_error:0.028312941969671275
p:1, lower_bound:0, known_good:32208, upper_bound:36658, known_bad:41108,
max_error:0.030076797456290955
p:1, lower_bound:0, known_good:32208, upper_bound:34433, known_bad:36658,
max_error:0.02938518122491594
p:1, lower_bound:0, known_good:34433, upper_bound:34533, known_bad:36658,
max_error:0.029478135143909876
p:1, lower_bound:0, known_good:34533, upper_bound:35533, known_bad:36658,
max_error:0.030125655634307472
p:1, lower_bound:0, known_good:34533, upper_bound:35033, known_bad:35533,
max_error:0.02980586985488824
p:1, lower_bound:0, known_good:35033, upper_bound:35133, known_bad:35533,
max_error:0.02988107715667559
p:1, lower_bound:0, known_good:35133, upper_bound:35333, known_bad:35533,
max_error:0.030029824399015387
p:1, lower_bound:0, known_good:35133, upper_bound:35233, known_bad:35333,
max_error:0.029957015041170522
p:1, lower_bound:0, known_good:35233, upper_bound:35283, known_bad:35333,
max_error:0.029994421261588314
p:1, lower_bound:0, known_good:35283, upper_bound:35308, known_bad:35333,
max_error:0.030012615469099238
p:1, lower_bound:0, known_good:35283, upper_bound:35296, known_bad:35308,
max_error:0.030003960888411656
p:1, lower_bound:0, known_good:35283, upper_bound:35290, known_bad:35296,
max_error:0.029999573562684914
p:1, lower_bound:0, known_good:35290, upper_bound:35293, known_bad:35296,
max_error:0.030001771139518976
```

```
p:1, lower_bound:0, known_good:35290, upper_bound:35292, known_bad:35293,
max_error:0.03000103943413725
p:1, lower_bound:0, known_good:35290, upper_bound:35291, known_bad:35292,
max_error:0.03000030689072727
p:1, lower_bound:0, known_good:35290, upper_bound:35290, known_bad:35291,
max_error:0.029999573562684914
p:2, lower_bound:35291, known_good:35299, upper_bound:35299, known_bad:496607,
max_error:6.402811027698382e-08
p:2, lower_bound:35291, known_good:35299, upper_bound:35399, known_bad:496607,
max_error:3.712876595607178e-05
p:2, lower_bound:35291, known_good:35399, upper_bound:36399, known_bad:496607,
max_error:0.0009124606499388271
p:2, lower_bound:35291, known_good:36399, upper_bound:56399, known_bad:496607,
max_error:0.026815840995010092
p:2, lower_bound:35291, known_good:56399, upper_bound:76399, known_bad:496607,
max_error:0.02821008014045565
p:2, lower_bound:35291, known_good:76399, upper_bound:96399, known_bad:496607,
max_error:0.026437130476392765
p:2, lower_bound:35291, known_good:96399, upper_bound:116399, known_bad:496607,
max_error:0.02717588934470578
...
```

The first column starts with a repeating ‘p:1’ which indicates which partition the dynamic partitioner is currently making. For a given set of coordinates, if the error is too high it will shrink the coordinate range to a number between a known good range and a known bad range. If the error is too low it will do the opposite and increase the range. It proceeds back and forth until it reaches the longest range to fit the specified error threshold at which point it stores the partition information for later writing and continues to find the next partition, and so on, until it goes through all points of an orbit. The error calculation here is relatively slow but accurate to within millimeters.

Here’s a graph from data generated by the code above for one granule (graph code not shown but it’s using D3js v5):



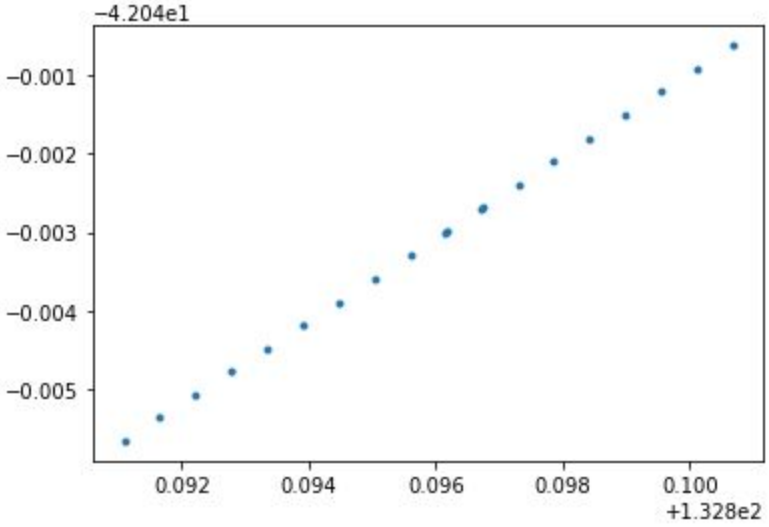
Maximum error in Blue, Average error in yellow
(It took 1754 seconds to generate the data for this graph - AMD Ryzen 9 3900X)

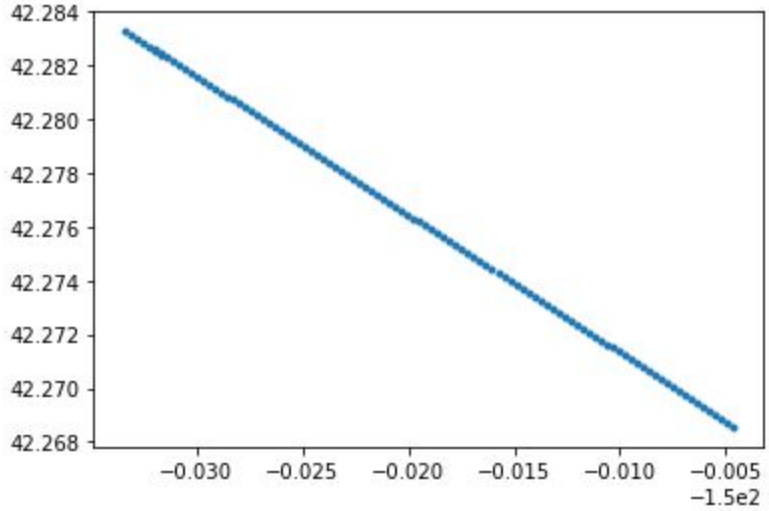
Note that this graph shows maximum and average error rates over the same data as the previous graph and the axes here use the same linear scale and range. What has changed is the x-axis is now “number of data points” instead of “partition number”. The actual partition boundaries are not shown.

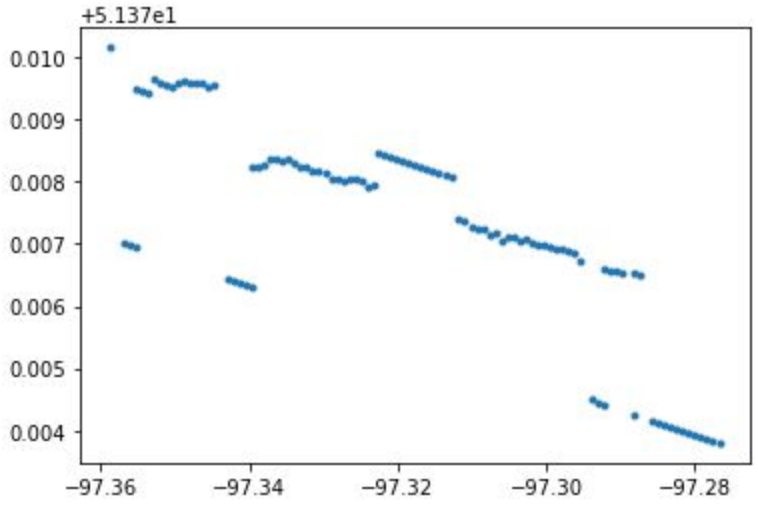
We specified an error rate of 30 meters (0.030 km) fitting 7 degree polynomials and the code created partitions of varying widths in an attempt to keep each at or below 30 meters. Seeing the results on this chart is both heartening and puzzling - why is there still a deviation in accuracy greater than what we specified? This behavior can be seen across many of over 100 sample granules to which I fit polynomials. In fact, some granules were exceedingly problematic. The worst in the sample set had to have 9679 partitions to reach 30 meter max error across most of its data. If the data we are fitting functions to were uniform then it should be reasonable to expect more uniform results.

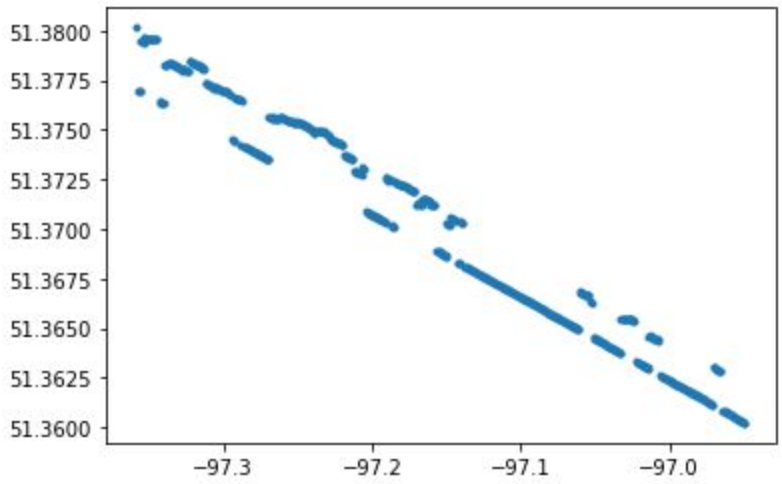
The Shape of GEDI Data, Part 1

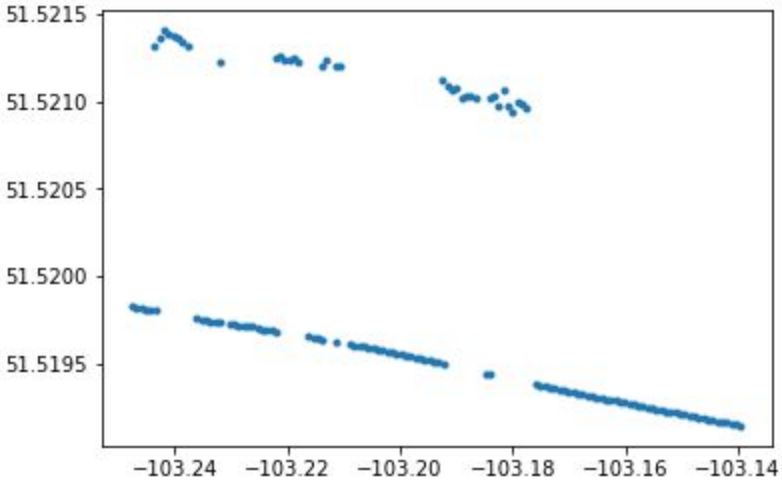
Let’s zoom in on some problem areas of different granules and see what’s going on.


−4.204e1
−0.001
−0.002
−0.003
−0.004
−0.005
0.092
0.094
0.096
0.098
0.100
+1.328e2


42.284
42.282
42.280
42.278
42.276
42.274
42.272
42.270
42.268
−0.030
−0.025
−0.020
−0.015
−0.010
−0.005
−1.5e2


+5.137e1
0.010
0.009
0.008
0.007
0.006
0.005
0.004
−97.36
−97.34
−97.32
−97.30
−97.28


51.3800
51.3775
51.3750
51.3725
51.3700
51.3675
51.3650
51.3625
51.3600
−97.3
−97.2
−97.1
−97.0


51.5215
51.5210
51.5205
51.5200
51.5195
−103.24
−103.22
−103.20
−103.18
−103.16
−103.14

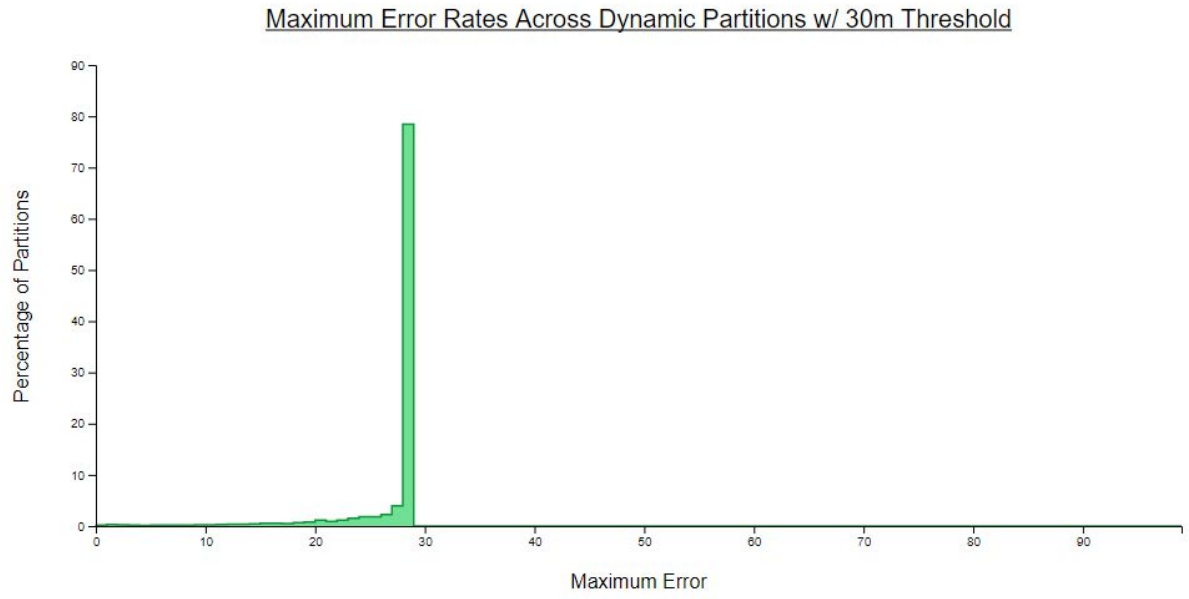
Some data have minor deviations, like small stair steps or an occasional point that's almost on top of another one while other data deviate strongly from the main line. Why do they do this? Is the data valid? And the question most relevant to our work: Do we need to consider these data? Would an individual who uses our search want to include these outliers in their results?

We know GEDI is a taskable satellite, meaning the instrument can be pointed as needed. Specifically, it has the ability to move up to 6 degrees latitude off nadir[13] so there will be times when the instrument will be directed 'off-path'. It's best to assume data released by the GEDI team is vetted good data so we will account for it.

Accuracy of Our Approach

Over a sample month of data, May of 2019, I ran the dynamic partitioner with 7-degree polynomials and specified error thresholds of 30m, 20m, 10m, and 5m. Shown below is a histogram of achieved accuracy for each as well as other information that will help us evaluate the results.

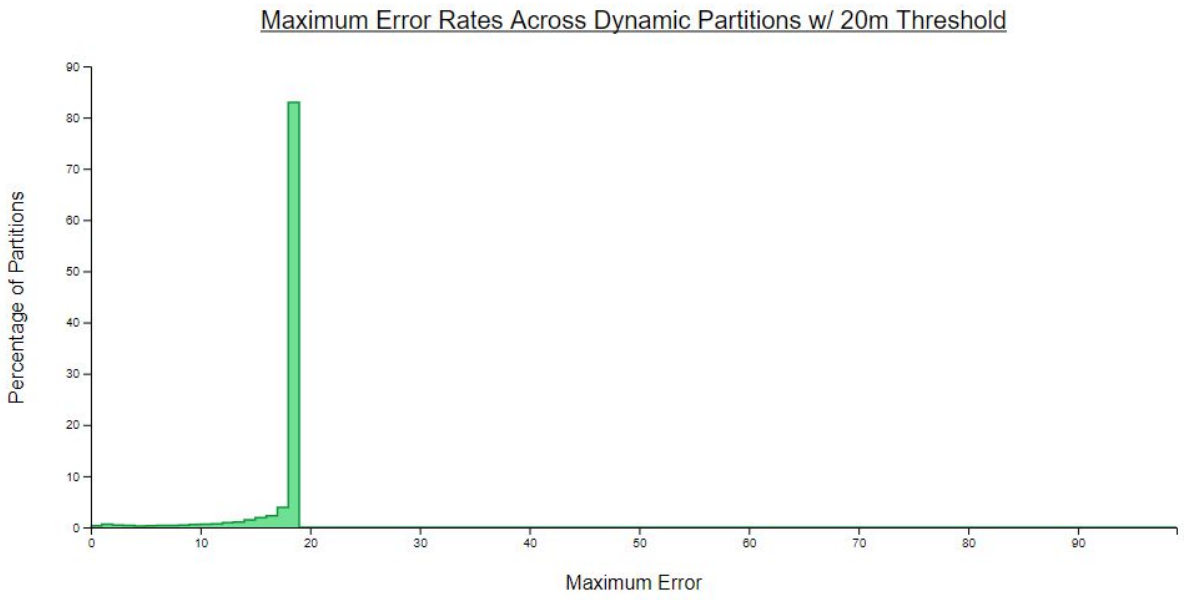
Polynomial Fit for Partitions of 30m (0.030 km) Specified Maximum Error



[13] https://gedi.umd.edu/instrument/instrument-overview/

Average, mean:	0.028216439662905695 km
Average, median:	0.029954263344824216 km
Max error for worst fitting polynomial:	0.6869073938382132 km
Standard deviation or σ:	0.005039136508381175 km
Percentage of data points at or less than specified error threshold:	99.937 %
Percentage of data points above specified error threshold:	0.063 %
Total number of partitions for sample month:	402,865
Average partitions per granule:	943
Most partitions for a granule:	9679
Least partitions for a granule:	11
Total size on disk for sample month (as JSON):	164 MB

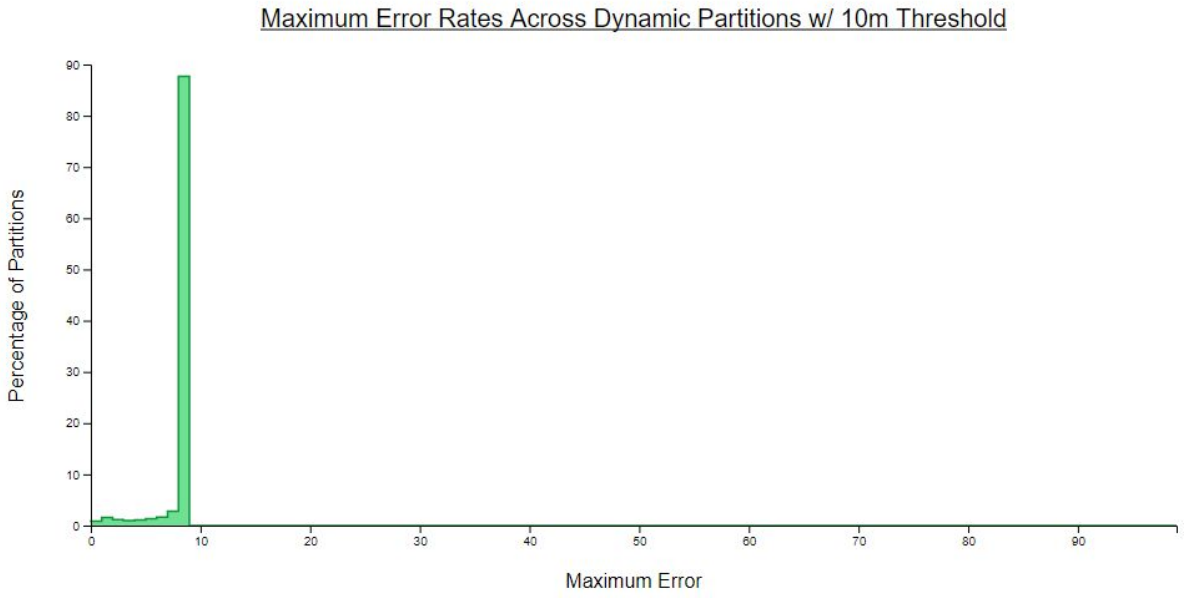
Polynomial Fit for Partitions of 20m (0.020 km) Specified Maximum Error



Average, mean:	0.01888378813809257 km
Average, median:	0.01996206643293634 km
Max error for worst fitting polynomial:	0.6869073938382132 km

Standard deviation or σ:	0.003850757467096522 km
Percentage of data points at or less than specified error threshold:	99.908 %
Percentage of data points above specified error threshold:	0.092 %
Total number of partitions for sample month:	606,931
Average partitions per granule:	1421
Most partitions for a granule:	11,333
Least partitions for a granule:	15
Total size on disk for sample month (as JSON):	246 MB

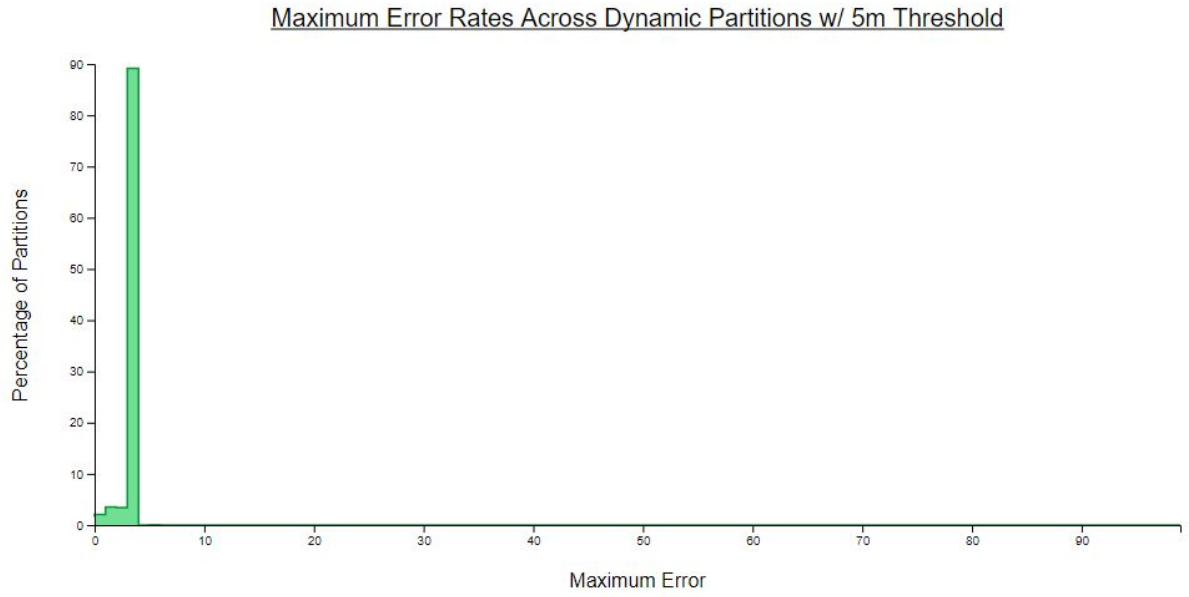
Polynomial Fit for Partitions of 10m (0.010 km) Specified Maximum Error



Average, mean:	0.009420246194762704 km
Average, median:	0.009966840586964055 km
Max error for worst fitting polynomial:	0.6869073938382132 km
Standard deviation or σ:	0.0030379619266287868 km
Percentage of data points at or less than specified error threshold:	99.828 %
Percentage of data points above specified error threshold:	0.172 %

Total number of partitions for sample month:	1,077,172
Average partitions per granule:	2523
Most partitions for a granule:	15,982
Least partitions for a granule:	27
Total size on disk for sample month (as JSON):	436 MB

Polynomial Fit for Partitions of 5m (0.005 km) Specified Maximum Error



Average, mean:	0.00473401961305397 km
Average, median:	0.004969986895258308 km
Max error for worst fitting polynomial:	0.6869073938382132 km
Standard deviation or σ:	0.0029390885193709146 km
Percentage of data points at or less than specified error threshold:	99.784 %
Percentage of data points above specified error threshold:	0.216 %
Total number of partitions for sample month:	1,696,332
Average partitions per granule:	3973
Most partitions for a granule:	19,957

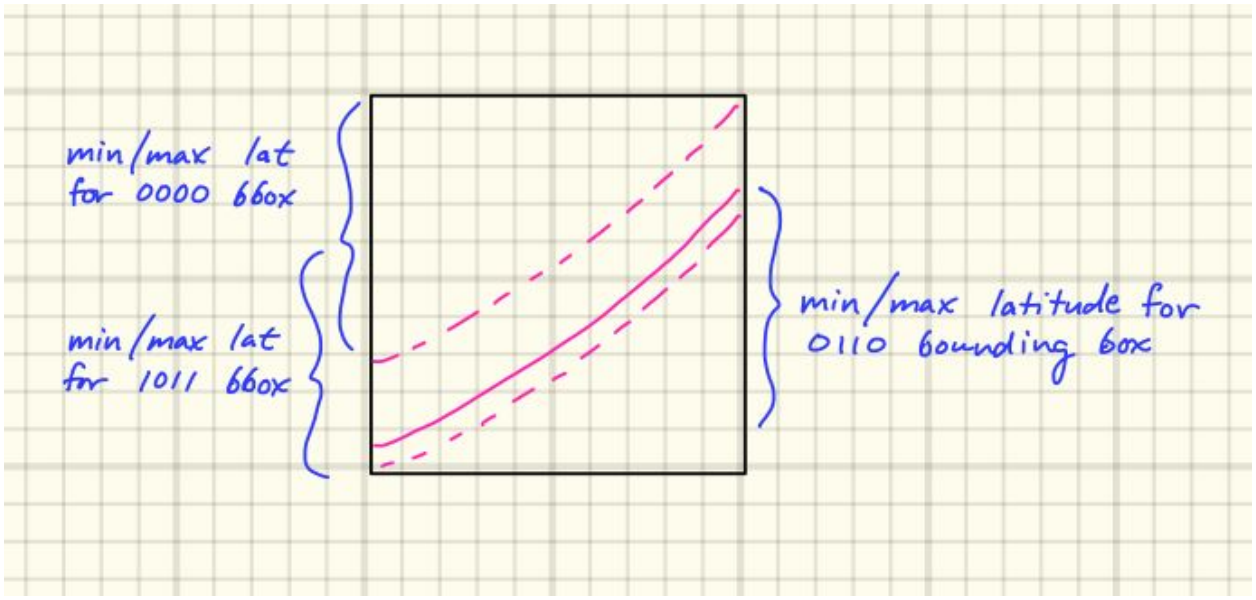
Least partitions for a granule:	54
Total size on disk for sample month (as JSON):	687 MB

These stats demonstrate strong and consistent results from our dynamic partitioning code. For example, to put 5 m of specified error (above and below) into perspective, it increases our calculated swath footprint by only 0.237% or less over GEDI's actual swath and this is the case for over 99.828% of our sample.

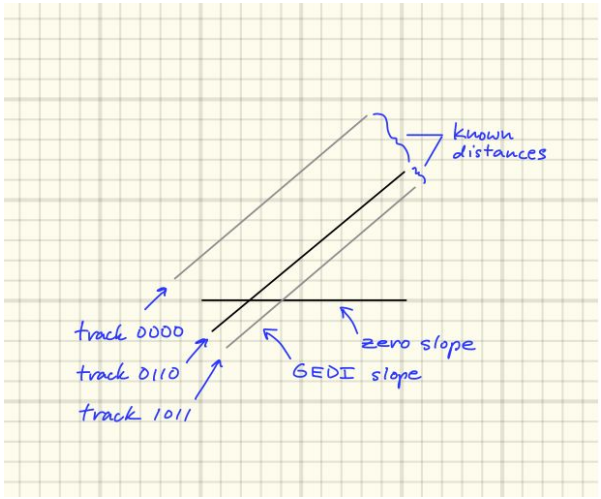
Notice that we didn't test a specified error threshold below 5 meters. While it would be possible to do so, the accuracy probably isn't necessary for this application and we'll soon see the error introduced by using spherical trigonometry (versus ellipsoid) can be a fair amount larger (2x to 4x). Conversely, having some 'extra slack' in the fit may actually end up being a desirable feature. A scientist may want to include all granules which have data that intersect their AOI as well as any granules which are in very close proximity. As we've now demonstrated, a wide range of specified error thresholds may be used with corresponding memory tradeoffs. The search run-time differences will be covered soon.

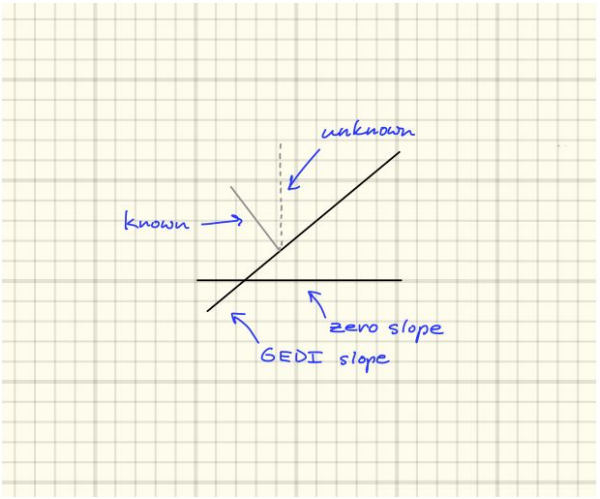
The Shape of GEDI Data, Part 2

Before detailing the search process we should dig deeper into the differences between our search model vs. the actual shape of GEDI data. As stated previously, we're using lon/lat coordinates from beam 0110 (6) to fit our polynomials. One of the first steps in the search process will be to see if the bounding box of a partition intersects the bounding box containing the user-supplied AOI. That means we'll need to develop a way to include the entire swath path, from the first track to the last, in our bounding box.



The swath width is 4200m + (half the width of the first beam) + (half the width of the last beam) = 4225m. The distance from beam 6 to beam 0 will always be 3012.5m and the distance from beam 6 to beam “8” will always be 1212.5m. The pitfall in accounting for all of the beams based on beam 6 is that our polynomials can only tell us a distance in latitude perpendicular to the equator but our known distances are perpendicular to the path of a GEDI orbit. We need to find a way to take track 6, add the due north distance to track 0 as well as the due south distance to track “8”. If we were to simply add the constant perpendicular-to-slope distance then our estimated bounding box would be inaccurate proportional to the slope of the orbit at that longitude.





E.g. to find the latitudinal distance between track 6 and track 0 we must find the distance labelled “unknown” above.

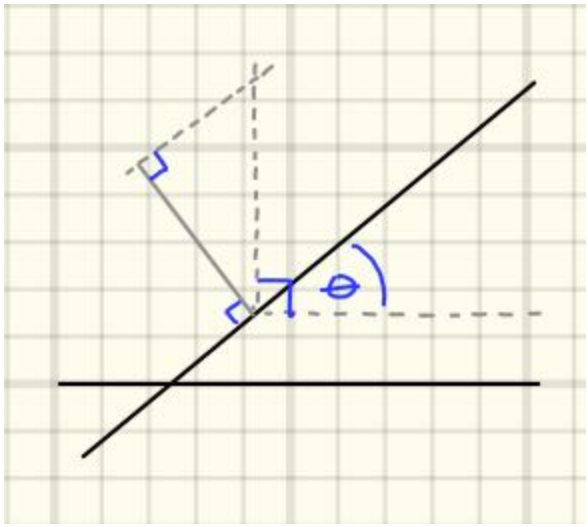
Thankfully there's math that can take our 'known' and the slope of the data path to derive the 'unknown'.

> *The earth is slightly larger around the equator than it is pole-to-pole. Because of this the earth, like other planets, is called an 'oblate spheroid'[14]. The difference in the earth's radius at the equator vs. the north or south pole is about 21.385km. We'll be using spherical trigonometry which treats the earth as a perfect sphere; it's more accurate than euclidean equations but slightly less accurate than ellipsoid[15] geodesics[16].*

The first step is to translate our constant distance into degrees of arc length: The distance from the northernmost swath edge to track 6 perpendicular to the orbit is 3.0125 km and the constant distance we'll use for 1 degree latitude is 110.5743 km[17].

The perpendicular distance between tracks 0000 and 0110 is 3.0125 km. Therefore 0110 to 0000 arc length is 3.6125 km / 110.5743 km or 0.027244... degrees.

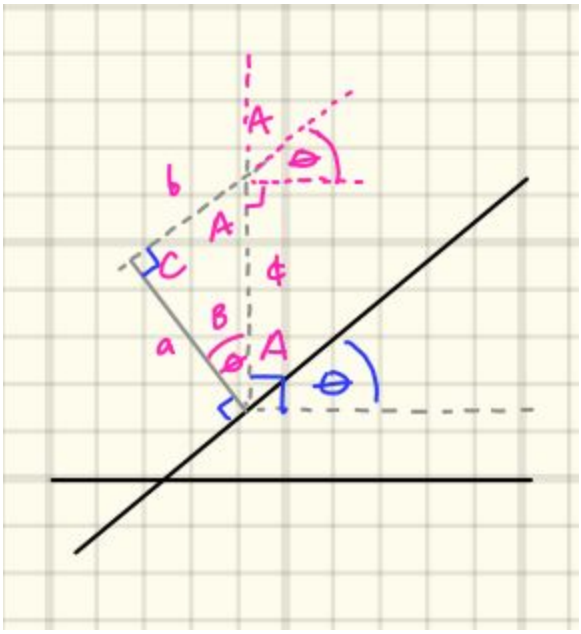
Next, we need to know the orbital slope. This can be accomplished by taking a fitted polynomial's first derivative[18] and supplying a longitude to get a slope value at that location. We then take the arctangent or inverse tangent of the slope to find the slope angle. This value is commonly called 'theta':



To find the latitudinal distance (the hypotenuse in the above right spherical triangle) we'll use trigonometry based on Napier's Rules for right spherical triangles[19]. The diagram below

[14] https://en.wikipedia.org/wiki/Spheroid#Oblate_spheroids
[15] https://en.wikipedia.org/wiki/Earth_ellipsoid
[16] https://en.wikipedia.org/wiki/Geodesics_on_an_ellipsoid
[17] https://calgary.rasc.ca/latlong.htm
[18] https://docs.scipy.org/doc/scipy/reference/generated/scipy.misc.derivative.html
[19] https://en.wikipedia.org/wiki/Spherical_trigonometry

has been marked up with traditional designations for each angle and side as well as which angles are equivalent. The capital letters are angles and the lower-case letters sides.



Based on theta, which is also ‘B’, and the known distance ‘a’, we can solve for ‘c’. For the sake of demonstrating an example solution let’s set the value of theta or B to 1 degree which would make ‘a’ very nearly equal to ‘c’.

Rules for right spherical triangles:

$\cos \phi = \cos a \cos b$,
$\sin a = \sin A \sin \phi$,
$\sin b = \sin B \sin \phi$,
$\tan a = \tan A \sin b$,
$\tan b = \tan B \sin a$,
$\tan b = \cos A \tan \phi$,
$\tan a = \cos B \tan \phi$,
$\cos A = \sin B \cos a$,
$\cos B = \sin A \cos b$,
$\cos \phi = \cot A \cot B$

1° at equator is 110.5743 km

$a = 3.0125$ km or $\frac{3.0125 \text{ km}}{110.5743} = 0.027244...^\circ$

$B = \theta = 1^\circ$ (this makes ϕ almost a)

$\tan a = \cos B \cdot \tan \phi$

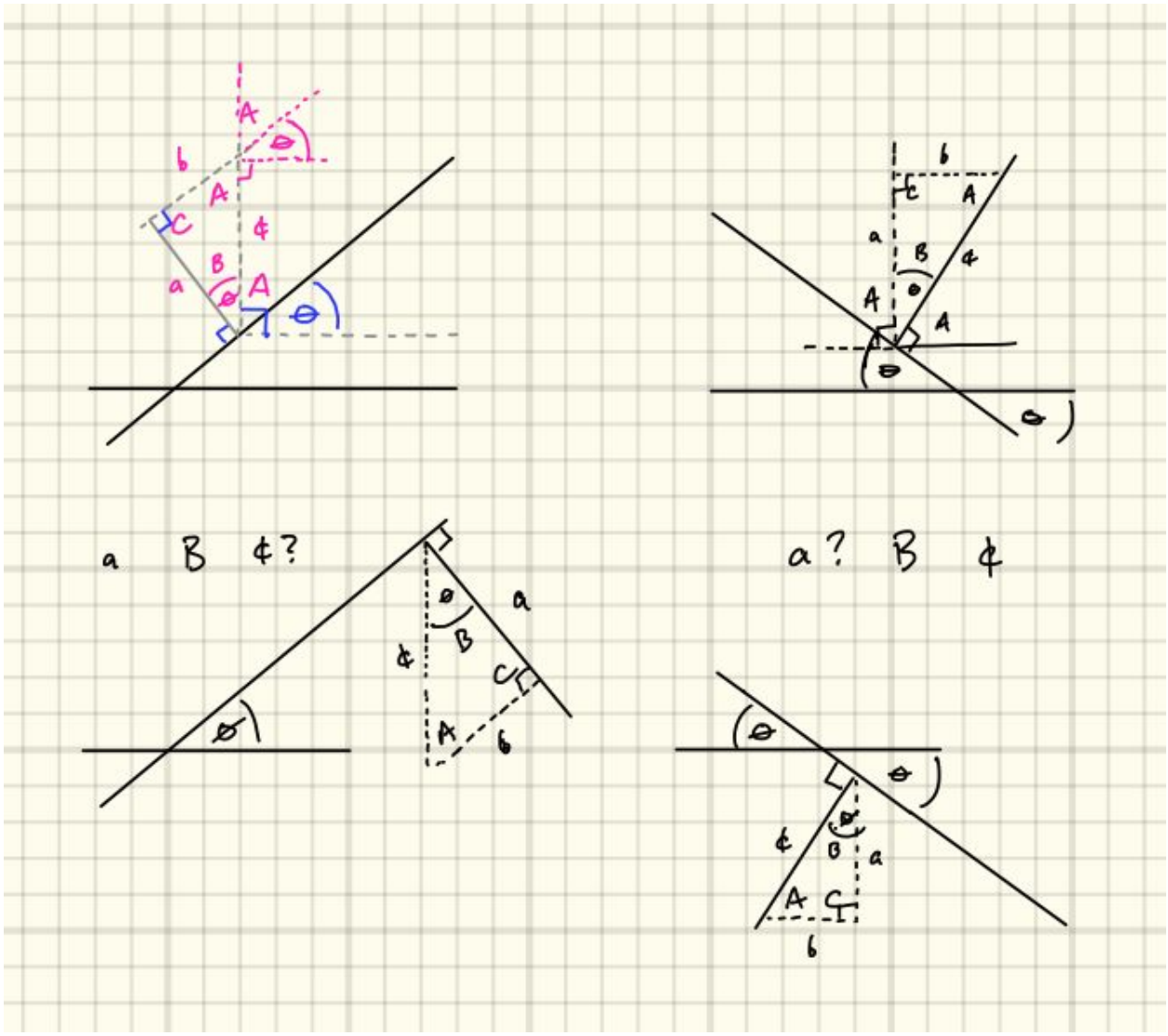
$\therefore \quad \frac{\tan a}{\cos B} = \tan \phi$

$\therefore \quad \tan^{-1}\left(\frac{\tan a}{\cos B}\right) = \phi$

$\tan^{-1}\left(\frac{0.00047...}{0.99984...}\right) = \phi$

$\phi = 0.027248...^\circ$

Also note that we can use the same formula for distances both above and below track 6 as the angles and distances are mirror opposites. And these are just the details for a positive slope - if the slope is negative then we'll apply a different formula. In the drawings below, positive slope is demonstrated on the left and negative slope is demonstrated on the right.



Accuracy Notes

By using spherical geometry to solve for our distances we gain simplicity and speed at the cost of accuracy. "Big Circle" calculations are about 0.3% less accurate than the millimeters-level accuracy of ellipsoid formulae from e.g. Vincenty[20]. For our purposes, that

20 https://community.esri.com/groups/coordinate-reference-systems/blog/2017/10/11/vincenty-formula

loss of accuracy is, again, proportional to the slope of GEDI's orbit at a given longitude. Specifically the vertical distance above track 6 will range from 3.0125 km at a 0 degree slope to 4.6866 km at a 50 degree slope. That means across an orbit the additional error will range from 9 m to 15 m above track 6 and 3.6 m to 6 m below. When accounting for the total accuracy of our solution we need to add a static factor of 1.003 (0.3 %) to our calculated swath path which will then naturally vary the amount of total error as needed.

What We've Discovered So Far

We've learned about GEDI's tracks, which one is geo-located and how the others are a known distance away from each other. The first and last track define the extent of the swath width or path. We've determined polynomials can very accurately fit GEDI's orbital path within a specified and configurable distance and that the distance guides the number of times a GEDI orbit must be partitioned.

We're almost done with our pre-search code - we'll cover the last piece we need under the Search section, which is next, because it helps to first understand the Search algorithm. This is the order in which I discovered it as well - realizing how the search works makes it clear what we need to write next.

Search

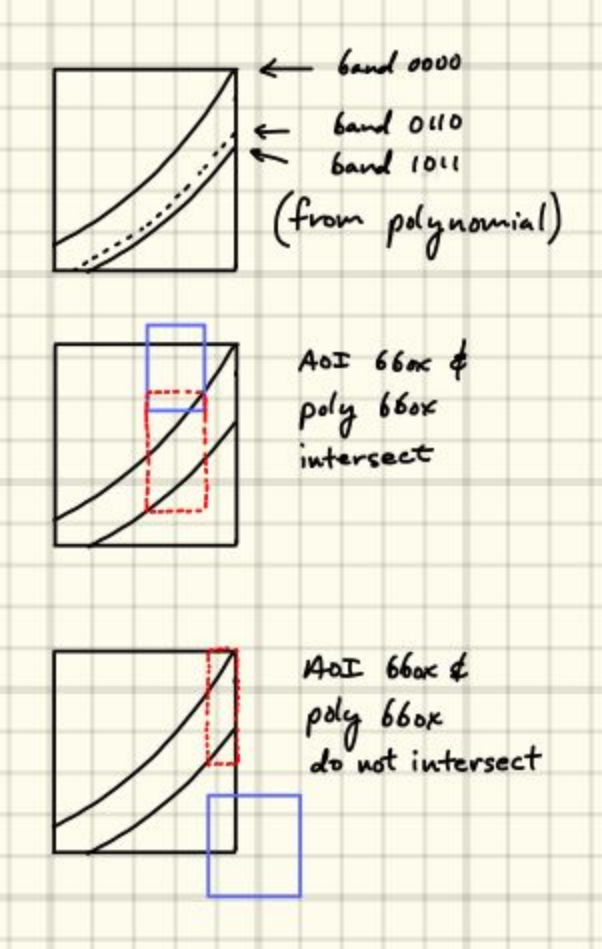
The heart of the search process is the ability to take an AOI bounding box and know if it intersects the swath extent. Our polynomial values won't be used in the search directly but rather used to calculate this extent. While we haven't been overly concerned with performance so far, code for these steps should now be as fast as possible.

We've touched on bounding boxes a number of times already and they're a commonly used approximation for more complex shapes. Calculating bounding box intersection, shown later, is also very fast code to run so it's probably best to use bounding box intersection detection as far as we can in order to minimize more computationally expensive steps.

How then do we make a final determination for swath/AOI intersection? There's complex math that can figure it out for us, for example here, or if we zoom in far enough on our polynomial or generalize it maybe we could treat it as a straight line and use Rotational Directions or other methods demonstrated here.

As I was drawing bounding boxes around polynomials I realized we can achieve true swath intersection detection with bounding boxes alone so long as we take the set intersection of

longitude values between the AOI bounding box and a polynomial bounding box and then follow-up with latitude overlap detection:



(The AOI bounding box is in blue and the bounding box based on the set intersection of longitude values of AOI and polynomial is in red)

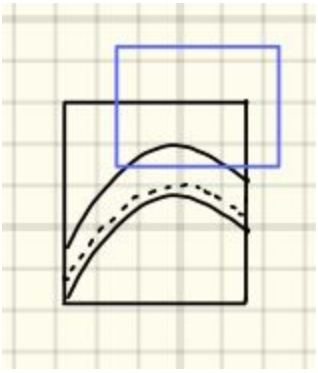
In other words, to find our longitude range we would:

1. take the greater of: the minimum longitude of the AOI bounding box vs. the minimum longitude of the current partition
2. take the the lesser of: the maximum longitude of the AOI bounding box vs. the maximum longitude of the current partition

Then, if the latitude ranges for each bounding box (AOI and current partition's swath width) overlap we know there's a search hit between the AOI and the granule the partition belongs to; no difficult or long-running math required. That makes our search turtles-and-bounding-boxes all the way down! (minus the turtles).

Inflections

And this is where we revisit the last piece of preparation required. Alongside the dynamic partitioning code, similar to what we did with anti-meridian splitting, we must include something that will cause a partition split along an orbital path inflection[21], i.e. the tops and bottoms for each orbit. We can have true swath intersection detection by checking for overlapping latitudes at the extremes of common longitudes so long as the polynomial's slope doesn't change sign over the common longitudinal range. If we didn't split at inflections, notice how the intersection detection described above would fail in this example:



(The AOI bounding box is in blue. In order to find the upper extent of this polynomial we would e.g. need to find the inflection point.)

We know GEDI's orbit will only have inflections at its extremes of latitude. Our search is going to happen one partition at a time so we can avoid inflections in our search process by splitting partitions at an inflection point, if they have one.

So - how do we detect for inflections and find inflections points? Because the raw geo-located GEDI data is available to us my initial approach was to go back to that - the source. Some example ideas included:

- Take the latitude absolute values, filter for those above 51.4 degrees, add up sample ranges and find the one which has the highest value. Then, take the midpoint as the inflection point.
- Total the inter-point slope across ranges and find the midpoint of the least
- Create arrays over a predetermined width which collect whether point i < i + 1 and assign -1 or 1. Then take the array that has a value closest to 0 and use the midpoint as the inflection point.

[21] https://en.wikipedia.org/wiki/Inflection_point

I think I had other ideas but none were very good. All of these ideas were a programmatic hassle because GEDI can have irregular data. Jittery data might cause us to label local maxima or minima as whole orbit inflections so I decided to go back to regression which, of course, is regularly used to generalize data such as this.

My initial approach was to solve for longitude where the first derivative of each polynomial equaled zero but once again there is a python library to help us out: scipy.optimize[22]. It has the 'minimize' function which can find the lowest value in a polynomial using a variety of algorithms. For our purposes we need to find the minimum and maximum over a given range but scipy.optimize doesn't offer a 'maximize' function. To get around that we can either invert the sign of our inputs (longitude) or multiply the polynomial coefficients by -1, either of which would invert the polynomial and allow us to use 'minimize' to find a maximum.

```
def _split_on_inflections(lons, lats, pn_degree):
    # A low degree polynomial will be less prone to fitting local maxima.
    # We're optimizing over a path which is convex and smooth.
    inf_pn_degree = 3
    lats_peak_threshold = 51.4
    skip_amount = 250000
    inflection_threshold = 1e-6
    abs_lats = np.abs(lats)
    equatorial_markers = []
    inflection_indices = []

    # assemble a list of equatorial markers
    search = True
    left_boundary = 0
    while search:
        search = False
        for i in range(left_boundary, len(abs_lats)):
            if 0 < abs_lats[i] < 1:
                equatorial_markers.append(i)
                left_boundary = i + skip_amount
                if left_boundary < len(abs_lats):
                    search = True
                break

    # add beginning and end indices to equatorial markers
    markers = [0, *equatorial_markers, len(lons) - 1]
```

[22] https://docs.scipy.org/doc/scipy/reference/optimize.html

```
    # Determine if there are any inflection points
    for marker_index in range(1, len(markers)):
        left_boundary = markers[marker_index-1]
        right_boundary = markers[marker_index]

        # Determine possible inflection range
        peak_indices = None
        for i in range(left_boundary, right_boundary):
            if abs_lats[i] > lats_peak_threshold:
                for ii in range(i, right_boundary):
                    if abs_lats[ii] < lats_peak_threshold:
                        peak_indices = [i, ii]
                        break
                if not peak_indices:
                    peak_indices = [i, right_boundary - 1]
                break

        if peak_indices:
            x1, x2 = peak_indices
            # Fit a function and test for inflection point
            pn = generate_polynomial(inf_pn_degree, lons[x1:x2], abs_lats[x1:x2])
            # Need to take the inverse of the fitted function because
            #   scipy.optimize doesn't have 'maximize'
            pn_inverse = np.poly1d([x * -1 for x in pn])
            opt = minimize(pn_inverse, x0=lons[x1], bounds=[[lons[x1], lons[x2]]])
            if opt.success:
                lon_min = opt.x[0]
                z = inflection_threshold
                # If polynomial max value isn't located at either end of our range
then
                #   an inflection point has been found
                if (abs(lon_min - lons[x1]) > z) and (abs(lon_min - lons[x2]) > z):
                    # find the offset value based on inflection point
                    inflection_indices.append(bisect(lons, lon_min))

    # split up lons and lats according to inflection_indices
    if inflection_indices:
        final_indices_ = [0, *inflection_indices, len(lons) - 1]
        final_indices = []
        # Discard split at inflection if it is too small for us to fit a
        #   polynomial function
        for i in range(1, len(final_indices_)):
```

```
            if (final_indices_[i] - final_indices_[i-1]) > pn_degree:
                final_indices.append(final_indices_[i-1])
        final_indices.append(final_indices_[-1])

        result_lons = []
        result_lats = []
        for i in range(1, len(final_indices)):
            left_boundary = final_indices[i-1]
            right_boundary = final_indices[i]
            result_lons.append(lons[left_boundary:right_boundary])
            result_lats.append(lats[left_boundary:right_boundary])
        return result_lons, result_lats
    else:
        return [lons], [lats]


def split_on_inflections(lons, lats, pn_degree):
    """ Find the inflections through the anti-meridian halves of a GEDI
        orbit. Takes an array of arrays of longitude values lons, and array
        of arrays of latitude values lats, and the degree of polynomial to
        use to find inflections and returns an array of arrays of longitude
        and latitude values. """
    acc_lons = []
    acc_lats = []
    for lons_, lats_ in zip(lons, lats):
        lons_parts, lats_parts = _split_on_inflections(lons_, lats_, pn_degree)
        acc_lons.extend(lons_parts)
        acc_lats.extend(lats_parts)
    return acc_lons, acc_lats
```

Search Algorithm

Here's an outline for the search:

- Take a user-supplied AOI bounding box
- For each granule:
    - Check that the AOI bbox intersects the granule bbox
    - Perform a binary search (bisect function) over the granule's partitions to see where we need to start looking.
    - Loop over partitions from the index indicated in the binary search until a partition is entirely outside of the range of the AOI.

        - Check if the AOI intersects the partition bbox
            - Check for swath overlap based on longitudinal set intersection and latitude overlap between partition latitude extremes and AOI latitude extremes.

As soon as we receive a 'hit' on any of a granule's partitions we may record that granule as having relevant data to the AOI and may stop further searching within that granule.

Because we've stored each granule's partitions in sorted order we can perform a binary search[23] to discover the starting point for partition bounding box detection. This will be faster than starting at the beginning and testing each one; such a strategy would on average find a starting point after N/2 comparisons where N is the number of partitions. A binary search, however, will average log2(N). For example let's say an orbit contains 1000 partitions: searching from the beginning means averaging 500 tests for a left-most match whereas a binary search will average 10 tests.

Search Code

The search code comes in two major sections - loading the data into memory and then performing the actual search. In a production environment, e.g. for a web service, this prototype would be loaded once and then left to run searches for a period of time because loading the data can be slow as detailed in the table below.

What is NumPy?

If you're not familiar with NumPy[24], it's a Python library that primarily deals in arrays and mathematical functions. Part of it is written in C and compiled to machine code which allows it to perform operations outside of the python interpreter[25].

What is Numba?

Numba "is an open source JIT compiler that translates a subset of Python and NumPy code into fast machine code."[26] Many performance critical libraries like NumPy are compiled for speed but writing one's own code in the same way usually requires at least two different programming ecosystems. An advantage Numba offers is its ability to take code which can run under the Python interpreter and compile some or all of it into native machine code via the LLVM compiler, all without a change in the programming ecosystem. The very large caveat is you'll be restricted from using any third party libraries, directly or transitively, that aren't pure Python. Also, not all Python or NumPy is supported by Numba. The compiler

[23] https://en.wikipedia.org/wiki/Binary_search_algorithm
[24] https://numpy.org/doc/stable/user/whatisnumpy.html
[25] https://stackoverflow.com/questions/8385602/why-are-numpy-arrays-so-fast
[26] http://numba.pydata.org/

features are leveraged with a @jit python decorator and in our code we're supplying keyword args *nopython, nogil,* and *parallel*.

There are three files that make up the search: search.py, shared.py, and polyder.py and they are listed in that order below:

search.py

```
import json
import numpy as np
from numba import jit, prange, int8, float64, int64, void
import random
from time import sleep
from timeit import default_timer as timer

import settings
from shared import bbox_intersect, get_filenames, is_overlap_sorted_values, poly_bbox


INPUT_PATH = settings.PARTITIONS_05M_PATH

# partitions: [('granules_idx'), ('polynomial', (8)), ('bbox', (4)), ('max_error')] ==
14 wide
PARTITIONS_WIDTH = 14
GRANULES_IDX = 0
POLYNOMIAL_BEGIN = 1
POLYNOMIAL_END = 9
P_BBOX_BEGIN = 9
P_BBOX_END = 13
MAX_ERROR = 13
# granules:   [('bbox', (4)), ('partitions_offset_left'), ('partitions_offset_right')]
== 6 wide
GRANULES_WIDTH = 6
G_BBOX_BEGIN = 0
G_BBOX_END = 4
PARTITIONS_OFFSET_LEFT = 4
PARTITIONS_OFFSET_RIGHT = 5


def _orbit_bbox(partitions):
    """ Takes a granule's partitions 'partitions' and returns the bounding box
        containing all of them. Bounding box is ll, ur format
        [[lon, lat], [lon, lat]]. """
```

```
    lon_min = partitions[0]['lon_min']
    lat_min = partitions[0]['lat_min']
    lon_max = partitions[0]['lon_max']
    lat_max = partitions[0]['lat_max']
    for p in partitions[1:]:
        if p['lon_min'] < lon_min:
            lon_min = p['lon_min']
        if p['lat_min'] < lat_min:
            lat_min = p['lat_min']
        if p['lon_max'] > lon_max:
            lon_max = p['lon_max']
        if p['lat_max'] > lat_max:
            lat_max = p['lat_max']
    return [[lon_min, lat_min], [lon_max, lat_max]]


def load_data(input_path):
    """ Takes a filesystem path input_path and returns a tuple of
        (urls, granules, partitions, partitions_lons_max, min_lat,
        max_lat). """
    # NOTE: simulating two years of data with 1 month repeated
    dataset_multiplier = 24

    # Determine the dimensions of ndarrays
    filenames = get_filenames(input_path)
    granules_count = 0
    partitions_count = 0
    for f in filenames:
        granules_count += 1
        with open(input_path + f) as g:
            partitions_count += len(json.load(g))

    granules_count = granules_count * dataset_multiplier
    partitions_count = partitions_count * dataset_multiplier

    # Create ndarrays
    partitions = np.zeros((partitions_count, PARTITIONS_WIDTH))
    granules = np.zeros((granules_count, GRANULES_WIDTH))

    # Populate data
    urls = []
    partitions_lons_max = np.zeros((partitions_count))
    granules_idx = 0
```

```
    partitions_idx = 0
    for _ in range(dataset_multiplier):
        for f in filenames:
            # append to list of urls
            urls.append('https://placeholder.url/' + f)
            # assign values to granule and partitions
            with open(input_path + f) as g:
                partitions_json = json.load(g)
            granules[granules_idx][PARTITIONS_OFFSET_LEFT] = partitions_idx
            granules[granules_idx][PARTITIONS_OFFSET_RIGHT] = partitions_idx +
len(partitions_json) - 1
            for p in partitions_json:
                # set all the values in this partition
                partitions_view = partitions[partitions_idx]
                partitions_view[GRANULES_IDX] = granules_idx
                np.put(partitions_view, np.arange(POLYNOMIAL_BEGIN, POLYNOMIAL_END),
tuple(p['pn']))
                np.put(partitions_view, np.arange(P_BBOX_BEGIN, P_BBOX_END),
[p['lon_min'], p['lat_min'], p['lon_max'], p['lat_max']])
                partitions_view[MAX_ERROR] = p['max_error']
                # set a separate value for binary search
                partitions_lons_max[partitions_idx] = p['lon_max']
                # increment for the next partition
                partitions_idx += 1
            np.put(granules[granules_idx], np.arange(G_BBOX_BEGIN, G_BBOX_END),
np.array(_orbit_bbox(partitions_json)).flatten())
            granules_idx += 1

    # Find the minimum latitude and maximum latitude across all partitions
    min_lat = 0
    max_lat = 0
    for i in range(0, len(granules)):
        minl = (granules[i][G_BBOX_BEGIN:G_BBOX_END])[1]
        maxl = (granules[i][G_BBOX_BEGIN:G_BBOX_END])[3]
        if min_lat > minl:
            min_lat = minl
        if max_lat < maxl:
            max_lat = maxl

    return urls, granules, partitions, partitions_lons_max, min_lat, max_lat


@jit
```

```
def _search(granules_idx, aoi_bbox, granules, partitions, partitions_lons_max,
matched_granules):
    aoi_lon_min = aoi_bbox[0][0]
    aoi_lon_max = aoi_bbox[1][0]
    aoi_lat_min = aoi_bbox[0][1]
    aoi_lat_max = aoi_bbox[1][1]

    g_bbox = (granules[granules_idx][G_BBOX_BEGIN:G_BBOX_END]).copy().reshape((2, 2))

    # Does aoi_bbox intersect this granule's bbox
    if bbox_intersect(aoi_bbox, g_bbox):
        left_idx = int(round(granules[granules_idx][PARTITIONS_OFFSET_LEFT]))
        right_idx = int(round(granules[granules_idx][PARTITIONS_OFFSET_RIGHT] + 1))
        partitions_view = partitions[left_idx:right_idx]
        partitions_lons_max_view = partitions_lons_max[left_idx:right_idx]
        # Binary search on sorted values
        start_idx = np.searchsorted(partitions_lons_max_view, aoi_lon_min)
        # Iterate through relevant partitions
        for i in range(start_idx, right_idx+1):
            # Check for longitude overlap. NOTE: is_overlap_sorted_values is
            #   not used here because NaN raw data values can cause partition
            #   splits resulting in a premature end to this loop if we were
            #   to use that function.
            p_lon_min = (partitions_view[i][P_BBOX_BEGIN:P_BBOX_END])[0]
            if aoi_lon_max > p_lon_min:
                # Check for whole partition bbox overlap with aoi_bbox
                p_bbox =
(partitions_view[i][P_BBOX_BEGIN:P_BBOX_END]).copy().reshape((2, 2))
                if bbox_intersect(p_bbox, aoi_bbox):
                    # Check for specific calculated swath overlap. Generate a
                    # bounding box for the polynomial that is:
                    #  - the greater of aoi_lon_min, p_lon_min
                    #  - the lesser of aoi_lon_max, p_lon_max
                    # p_g_ is short for polynomial, generated
                    p_lon_max = (partitions_view[i][P_BBOX_BEGIN:P_BBOX_END])[2]
                    p_g_lon_min = aoi_lon_min if aoi_lon_min > p_lon_min else
p_lon_min
                    p_g_lon_max = aoi_lon_max if aoi_lon_max < p_lon_max else
p_lon_max
                    p_g_bbox =
poly_bbox(partitions_view[i][POLYNOMIAL_BEGIN:POLYNOMIAL_END], p_g_lon_min,
p_g_lon_max, partitions_view[i][MAX_ERROR])
```

```
                    # We know specific longitudes of each bbox overlap; now
                    #   detect specific latitude overlap.
                    p_g_lat_min = p_g_bbox[0][1]
                    p_g_lat_max = p_g_bbox[1][1]
                    if is_overlap_sorted_values(p_g_lat_min, p_g_lat_max, aoi_lat_min,
aoi_lat_max):
                        # aoi_bbox and partition overlap; update matched_granules and
end search for this granule
                        matched_granules[granules_idx] = True
                        break
            else:
                break


@jit(nopython=True, nogil=True, parallel=True)
def search(aoi_bbox, granules, partitions, partitions_lons_max, granules_bbox):
    # An array that will have each element set to 1 according to each granule
    #   that matches a given aoi_bbox.
    matched_granules = np.zeros(len(granules), dtype=np.bool_)

    # Verify the aoi_bbox is within at least one of the granules_bbox
    if bbox_intersect(aoi_bbox, granules_bbox):
        # parallel processing across granules
        for granules_idx in prange(len(granules)):
            _search(granules_idx, aoi_bbox, granules, partitions, partitions_lons_max,
matched_granules)

    return np.nonzero(matched_granules)[0]


def granules_to_urls(granules_idxs, urls):
    """ Takes an array of granules indexes granules_idx and urls array 'urls'
        and returns a list of urls that correspond to the indices. """
    results = []
    for idx in granules_idxs:
        results.append(urls[idx])
    return results


def random_bbox():
    """ Returns random bbox around part of the amazon (roughly) in ll, ur
        format [[lon, lat], [lon, lat]]. """
    lons_fn = lambda: random.uniform(-74.0, -70)
```

```
    lats_fn = lambda: random.uniform(1.1, 3.9)
    lons = sorted((lons_fn(), lons_fn()))
    lats = sorted((lats_fn(), lats_fn()))
    return np.array([[lons[0], lats[0]], [lons[1], lats[1]]])


def benchmark():
    test_bboxes = [np.array([[-74.1, 3.1], [-73.7, 3.3]]),
                   np.array([[-74.1, 2.9], [-73.3, 3.3]]),
                   np.array([[-74.1, 2.7], [-72.9, 3.3]]),
                   np.array([[-74.1, 2.5], [-72.5, 3.3]]),
                   np.array([[-74.1, 2.3], [-72.1, 3.3]]),
                   np.array([[-74.1, 2.1], [-71.7, 3.3]]),
                   np.array([[-74.1, 1.9], [-71.3, 3.3]]),
                   np.array([[-74.1, 1.7], [-70.9, 3.3]]),
                   np.array([[-74.1, 1.5], [-70.5, 3.3]]),
                   np.array([[-74.1, 1.3], [-70.1, 3.3]])]

    rand_bboxes = [random_bbox() for _ in range(10)]

    # Load data
    start_time = timer()
    urls, granules, partitions, partitions_lons_max, min_lat, max_lat =
load_data(INPUT_PATH)
    end_time = timer()
    print('Load time: {0:.3f} seconds'.format(end_time - start_time))

    sleep(3)

    granules_bbox = np.array([[-180.0, min_lat], [180.0, max_lat]])

    # warm up for Numba and JIT
    for _ in range(10):
        for t in test_bboxes:
            granules_idxs = search(t, granules, partitions, partitions_lons_max,
granules_bbox)

    # establish search time
    start_time = timer()
    for b in rand_bboxes:
        granules_idxs = search(b, granules, partitions, partitions_lons_max,
granules_bbox)
        granules_to_urls(granules_idxs, urls)
```

```
    end_time = timer()
    print('Search time: {0:.9f} seconds'.format((end_time - start_time) /
len(test_bboxes)))

    # return URLs for a search
    granules_idxs = search(test_bboxes[0], granules, partitions, partitions_lons_max,
granules_bbox)
    print(sorted(granules_to_urls(granules_idxs, urls)))


if __name__ == "__main__":
    benchmark()
```

shared.py

```
import math
import os
import numpy as np
from numba import jit

from polyder import polyder

# one degree latitude distance in kilometers
ODL_DISTANCE = 110.5743
# Factor in error introduced by spherical trigonometry
STATIC_MULT = 1.003
# Distance from track 6 to 0 perpendicular to orbit
PERPENDICULAR_ABOVE_DISTANCE = 3.0125  # in km
# Distance from track 6 to 7 perpendicular to orbit
PERPENDICULAR_BELOW_DISTANCE = 1.2125  # in km


def get_filenames(path):
    """ Takes a filesystem path and returns a sorted list of filenames under
        that path. """
    xs = []
    for (dirpath, dirnames, filenames) in os.walk(path):
        xs.extend(filenames)
        break
    xs.sort()
    return xs
```

```
@jit
def polyval(p, x):
    """ Takes a sequence p representing a polynomial and a number x and
        returns the value of p at x. This version is Numba-compatible; NumPy's
        version is not. """
    val = 0
    ii = len(p) - 1
    for i in range(len(p) - 1):
        val += p[i] * (x ** ii)
        ii -= 1
    return val + p[-1]


@jit
def is_overlap_sorted_values(v1, v2, w1, w2):
    """ Takes two pairs of values, v1, v2 and w1, w2 and returns a boolean
        result indicating whether the range v1, v2 (inclusive) contains any
        values in the range w1, w2 (inclusive). """
    if (v2 < w1) or (v1 > w2):
        return False
    else:
        return True


@jit
def bbox_intersect(a_ary, b_ary):
    """ Takes two nested two dimensional arrays, a_ary and b_ary,
        representing a bounding box in ll, ur format
        [[lon, lat], [lon, lat]]. Returns a boolean result as to whether the
        bounding boxes overlap. """
    # Do any of the 4 corners of one bbox lie inside the other bbox?
    # bbox format of [ll, ur]
    # bbx[0] is lower left
    # bbx[1] is upper right
    # bbx[0][0] is lower left longitude
    # bbx[0][1] is lower left latitude
    # bbx[1][0] is upper right longitude
    # bbx[1][1] is upper right latitude

    # Detect longitude and latitude overlap
    if is_overlap_sorted_values(a_ary[0][0], a_ary[1][0], b_ary[0][0], b_ary[1][0]) \
```

```
            and is_overlap_sorted_values(a_ary[0][1], a_ary[1][1], b_ary[0][1],
b_ary[1][1]):
        return True
    else:
        return False


@jit
def _deg_to_rad(deg):
    """ Takes a degree value deg and returns the equivalent value in
        radians. """
    return deg * math.pi / 180


@jit
def _rad_to_deg(rad):
    """ Takes a radian value rad and returns the equivalent value in
        degrees. """
    return rad * 180 / math.pi


@jit
def _angle_from_slope(poly_ary, lon):
    """ Takes a polynomial array poly_ary and a longitude value lon and
        returns the angle in radians at that longitude. """
    # find the first derivative
    poly = polyder(poly_ary, 1)
    # get the slope at our point of interest
    slope = polyval(poly, lon)
    # get the angle from the slope
    angle = math.atan(slope)
    return angle  # in radians


@jit
def _slope_pos_vert_distance(B, perp):
    """ Takes an acute angle B and distance in kilometers perp and returns the
        distance perpendicular to the equator. Assumes an orbital section
        positive in slope. """
    # get the arc length of 'a' at around this latitude
    a = _deg_to_rad(perp / ODL_DISTANCE)
    # arclength from beam 0110 to first or last beam vertically
    arclength = math.atan(math.tan(a) / math.cos(B))
```

```
    # distance in km from beam 0110 to first or last vertically
    beam_distance = _rad_to_deg(arclength) * ODL_DISTANCE
    return beam_distance  # in km


@jit
def _slope_neg_vert_distance(B, perp):
    """ Takes an acute angle B and distance in kilometers perp and returns the
        distance perpendicular to the equator. Assumes an orbital section
        negative in slope. """
    # get the arc length of 'a' at around this latitude
    c = _deg_to_rad(perp / ODL_DISTANCE)
    # arclength from beam 0110 to first or last beam vertically
    arclength = math.atan(math.cos(B) * math.tan(c))
    # distance in km from beam 0110 to first or last vertically.
    beam_distance = _rad_to_deg(arclength) * ODL_DISTANCE
    return beam_distance  # in km


@jit
def poly_bbox(poly_ary, lon_min, lon_max, max_error):
    """ Takes a polynomial array poly_ary, starting longitude lon_min, ending
        longitude lon_max, and error distance max_error and returns a nested
        array representing a bounding box for a GEDI swath extent. Return
        value is in ll, ur format [[lon, lat], [lon, lat]]. """
    # Angle 'B' is also theta
    B_lon_min = _angle_from_slope(poly_ary, lon_min)
    B_lon_max = _angle_from_slope(poly_ary, lon_max)

    # Slope assumptions:
    # 1) will never be 0 at orbit major maximum or minimum
    # 2) will never change sign because of #1
    # Therefore, slopes are assumed continuously increasing or decreasing
    slope_is_positive = True if B_lon_min > 0 else False

    if slope_is_positive:
        # lat min will be at min lon
        lat_min_ = polyval(poly_ary, lon_min)
        # lat max will be at max lon
        lat_max_ = polyval(poly_ary, lon_max)

        # find the latitudinal distance using rules for right spherical triangles
```

```
        distance_below = _slope_pos_vert_distance(B_lon_min,
PERPENDICULAR_BELOW_DISTANCE)
        distance_above = _slope_pos_vert_distance(B_lon_max,
PERPENDICULAR_ABOVE_DISTANCE)

    else:  # slope is negative
        # lat min will be at max lon
        lat_min_ = polyval(poly_ary, lon_max)
        # lat max will be at min lon
        lat_max_ = polyval(poly_ary, lon_min)

        # find the latitudinal distance using rules for right spherical triangles
        distance_below = _slope_neg_vert_distance(B_lon_min,
PERPENDICULAR_BELOW_DISTANCE)
        distance_above = _slope_neg_vert_distance(B_lon_max,
PERPENDICULAR_ABOVE_DISTANCE)

    # subtract distance from 0110 beam to 0111 beam
    lat_min = lat_min_ - ((distance_below * STATIC_MULT) - max_error) / ODL_DISTANCE
    # add distance from 0110 beam to 0000 beam
    lat_max = lat_max_ + ((distance_above * STATIC_MULT) + max_error) / ODL_DISTANCE

    # bbox format of [ll, ur]
    bbox = np.array([[lon_min, lat_min], [lon_max, lat_max]])

    return bbox
```

polyder.py

```
import numpy.core.numeric as NX
from numba import jit


"""
This code has been lifted and modified from NumPy and is licensed under
a BSD 3-Clause "New" or "Revised" License.

- - - - -

Copyright (c) 2005-2020, NumPy Developers.
All rights reserved.
```

```
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

    * Redistributions of source code must retain the above copyright
       notice, this list of conditions and the following disclaimer.

    * Redistributions in binary form must reproduce the above
       copyright notice, this list of conditions and the following
       disclaimer in the documentation and/or other materials provided
       with the distribution.

    * Neither the name of the NumPy Developers nor the names of any
       contributors may be used to endorse or promote products derived
       from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
© 2020 GitHub, Inc.
"""


@jit
def polyder(p, m=1):
    """ Takes a numpy array p representing a polynomial and an integer m
        representing the number of times to take the derivative m and returns
        a numpy array representing a derivative of that polynomial. Note:
        this implementation is Numba-compatible; the NumPy version is not.
        Taken from NumPy. """
    m = int(m)
    p = NX.asarray(p)
    n = len(p) - 1
    y = p[:-1] * NX.arange(n, 0, -1)
    if m == 0:
```

```
        val = p
    else:
        val = polyder(y, m - 1)
    return val
```

Search Code, Explained

The code in search.py is a little more obtuse than usual in order to make it compatible with Numba as well as to get some performance gains when or if Numba isn't used. The data structures used and rationale behind them, in particular, should be addressed.

Numba requires homogeneous arrays. NumPy also deals in homogeneous arrays but gets around mixed types by allowing for arrays of all scalars or all objects. Arrays of objects may point to arrays of scalars or of objects, and so on. In this way NumPy can support mixed types, irregular shaped arrays, and structured arrays ('structs'). Numba can't use object arrays because the preferred runtime modes *nopython* and *nogil* can't use the python runtime to handle object manipulation.

1. If we wish to store floats (doubles) then the whole array needs to be of that type
2. If we use a multidimensional array then the contained arrays need to all be the same size and type

The second constraint is the most problematic for us because, while the storage requirements for each partition are the same, the number of partitions per orbit is not. That and other shape irregularities preclude us from using multidimensional arrays for all of our data.

A common way around this problem is to implement a multidimensional array the same way computers do - as one dimensional with all nested dimensions referenced by slice. We would then use one or more arrays to hold the offsets and lengths for each of the other dimensions. Lastly, we can store regularly-sized complex data structures based on constant offsets and ranges within the final array dimension. In our case we use one array for orbit and partition offsets and another array for all the partitions.

The third array is a denormalized copy of partition longitudes but it's there as a convenient and efficient way to bisect / binary search (vs. constructing something on-the-fly from normalized data). By using an additional array here we're able to take advantage of

additional performance through data locality[27] which translates to fewer requests to main memory.

We specified *parallel* as a Numba argument to the @jit decorator. It indicates our interest in leveraging explicit or implicit parallelism in our code. The Numba documentation has a more complete description of its capabilities but our code is using the explicit style via the *prange* function. Without the @jit decorator *prange* is interpreted by python the same as *range*. However, with @jit(parallel) the compiler attempts to arrange execution in parallel map fashion. By default it fires up a number of threads equal to the system's core count and splits work (in our case, searching orbits) among those threads. Our search process is an example of an "embarrassingly parallel"[28] problem and *prange* and other functions like it were created to work on exactly those types of problems.

One very important design consideration: how do we know this code is thread safe? By running our compiled python outside of the interpreter and the GIL[29] we leave behind both the performance limitations and the protections afforded by CPython; we are at liberty to corrupt our data and must ensure thread safety through other means.

The search code works with two main categories of data: the search arrays and the results array. The search arrays are built before searching occurs and thereafter are then read-only by convention (but not by requirement). Read-only data is always safe to use in concurrently running code. The results array requires a little more thought, however. We need to be able to take the results of a search and return the URLs for granules that match a user's AOI. Concurrent access to mutable data requires some kind of coordination, specified either with concurrency primitives, thread-safe higher level data types, etc. or by some kind of logical separation and isolated mutation. The latter is typically faster and more elegant, where possible, and achieves safety by removing the concern of shared mutation instead of trying to manage it.
We can do this with the results array by using the granule number in our search data as the offset for each element. Because a thread will never have another thread's granule (via *prange*), it will also never write to another thread's results element. The tradeoff is most of the time we'll be over-allocating space to store our results but supporting fast multithreading is easily worth this small memory cost.

The per-granule verdict of a search is either match or no-match so if a thread finds a partition that has true swath path intersection with the AOI then the thread immediately writes a '1' to the correspond granule's element in the results array and is then ready to work on another granule. If no match is found, that element in the results array remains '0'.

---

[27] https://en.wikipedia.org/wiki/Locality_of_reference
[28] https://en.wikipedia.org/wiki/Embarrassingly_parallel
[29] https://wiki.python.org/moin/GlobalInterpreterLock

After the search has gone through all orbits the parallel section of the code ends and we're free to use the results array in single-threaded fashion by reading across all of the indices and detecting which ones are no longer zero.

The rest of the code is mostly self-explanatory or has already been described in e.g. The Shape of GEDI, Part 2. There are two NumPy functions we need for the prototype that aren't Numba compatible so I had to edit the one and write the other. The first function is *polyder* and I've indicated in the comments it was a lift from NumPy. I wrote the second function, *polyval*, along with the rest of the software.

Search Performance

The search performance is very good even without Numba but when it's enabled the library+compiler makes the run time two orders of magnitude faster. It's my understanding a speed increase of that amount is typical for Numba when compared to CPython.

How fast the search runs depends on a number of variables. Those as well as some re-stated information about partitions are detailed in the table below. Note: the dataset used was the month of May, 2019. All two-year numbers are extrapolated by multiplying the data (not the final numbers but actually increasing the amount of data loaded via duplication) by 24.

Partitions specified error threshold	0.030 km	0.020 km	0.010 km	0.005 km
Partitions mean accuracy	0.0282164 km	0.0188838 km	0.0094202 km	0.0047340 km
Partitions time to generate, per month of data	17.51 hours	17.25 hours	18.71 hours	19.01 hours
Partitions size on disk, 2 year (*)	3.84 GB	5.77 GB	10.22 GB	16.10 GB
Partitions load time (*) (+)	103.782 seconds (+)	150.246 seconds (+)	282.970 seconds (+)	448.251 seconds (+)
Python, memory used (*)	1.16 GB	1.71 GB	2.94 GB	4.60 GB
Python, time to search (*)	0.325611475 seconds	0.378034381 seconds	0.550040546 seconds	0.661551114 seconds
Numba, system memory used (*)	1.15 GB	1.71 GB	2.96 GB	4.64 GB

Numba, time to search (*)	0.002471092 seconds	0.004112495 seconds	0.005550449 seconds	0.008975056 seconds

- (*) Entries with an asterisk, including all searches, used a 2-year (simulated) dataset
- (+) Data load times for a simulated 2 years can actually be as fast as < 1 second with serialization code in play (not shown; it was trivial to implement though). I'd still recommend loading the data for a long running process that could serve many requests, though.
- There were ten AOI bounding boxes used for searching and the average run-time was used as the "time to search". The timed search took place after warming up the JIT compiler with non-identical searches.
- The system used for these tests was a commodity desktop: AMD Ryzen 3900X 12-core / 24-thread, with 32GB of memory and SSD storage. Tests were conducted on bare metal (no virtualization).
- Partition generation ran with 24 + 1 processes (Python multiprocess library).
- The partitions were stored as JSON (text).
- All partitions were configured to use 7-degree polynomials.

Miscellany

Improvements for the Future

Now that we have our prototype there's still a lot of room for improvement and worthwhile exploration left. Some ideas include:

- Write a web service that integrates this prototype to an API and front-end.
- Explore other forms of regression for generalizing GEDI geo-located data.
- In many ways we've already compared the difference between spherical and ellipsoid geodesics but writing a prototype that used ellipsoid models throughout and seeing the performance differences would be worthwhile.
- Try different polynomials; using 7-degree polynomials was just a guess. This likely wouldn't impact search time very much but could improve memory requirements.
- Investigate ways this prototype could integrate with a larger body of software capable of spatial, temporal, and band subsetting for GEDI, as that is the next tool a scientist would likely reach for.
- Some code refactoring might help as well.

Code that didn’t make it

There was a fair amount of code that didn’t make it into the final version of the prototype as most of it was written and then rewritten a handful of times but I think that’s acceptable for an exploratory project such as this one. For example, here’s the first version of my bounding box intersection detection code versus the one that replaced it. In the first version I was thinking of the problem more two-dimensionally and in the second version I realized the problem can be solved more simply by projecting the two dimensions to each one-dimensional and solving them in turn. The second version is easier to follow and test:

First version (some comments removed):

```
def bbox_intersect(a_ary, b_ary):
    a_lon_min = a_ary[0][0]
    a_lon_max = a_ary[1][0]
    a_lat_min = a_ary[0][1]
    a_lat_max = a_ary[1][1]

    b_lon_min = b_ary[0][0]
    b_lon_max = b_ary[1][0]
    b_lat_min = b_ary[0][1]
    b_lat_max = b_ary[1][1]

    if (# Detect if a corner of 'b' is inside 'a'
        # Detect 'a' edge-only overlap with 'b'
        # Detect 'b' is wholly inside 'a'
        (((a_lon_min <= b_lon_min) and (a_lon_max >= b_lon_min)) or
         ((a_lon_min <= b_lon_max) and (a_lon_max >= b_lon_max))
         and
         ((a_lat_min <= b_lat_min) and (a_lat_max >= b_lat_min)) or
         ((a_lat_min <= b_lat_max) and (a_lat_max >= b_lat_max)))
        # Detect if a corner of 'a' is inside 'b'
        # Detect 'b' edge-only overlap with 'a'
        # Detect 'a' is wholly inside 'b'
        or
        (((b_lon_min <= a_lon_min) and (b_lon_max >= a_lon_min)) or
         ((b_lon_min <= a_lon_max) and (b_lon_max >= a_lon_max))
         and
         ((b_lat_min <= a_lat_min) and (b_lat_max >= a_lat_min)) or
         ((b_lat_min <= a_lat_max) and (b_lat_max >= a_lat_max)))):
        return True
    else:
        return False
```

Second and current version (some comments removed):

```
def is_overlap_sorted_values(v1, v2, w1, w2):
    if (v2 < w1) or (v1 > w2):
        return False
    else:
        return True


def bbox_intersect(a_ary, b_ary):
    # Detect longitude and latitude overlap
    if is_overlap_sorted_values(a_ary[0][0], a_ary[1][0], b_ary[0][0], b_ary[1][0]) \
            and is_overlap_sorted_values(a_ary[0][1], a_ary[1][1], b_ary[0][1],
b_ary[1][1]):
        return True
    else:
        return False
```

Summary

Over its two year mission GEDI will release coordinate data totalling 373 GB when stored as text or 149 GB when stored in a binary format. The coordinate data will contain over 10,009,000,000 points.

What we've essentially created is a form of lossy compression that not only decreases the amount of storage needed by 32:1 but also makes those 10 billion points searchable with a high degree of accuracy in 1/100th of a second, requiring only a single compute instance and commodity hardware.

I hope you enjoyed reading this as much as I did in writing it. If you would like to make any corrections or suggestions *please* contact Element 84 - I'll address it ASAP. This paper has not yet been science reviewed and any mistakes you may find (science or otherwise) are my own.

A sincere thank you to Element 84: they're a fantastic company full of capable people and it's because of them I've had the opportunity to be in an environment that made this work possible.

–Ryan Waters