### Overview

This document describes each of the important code files in the Arctic\_Carbon repository on Google Earth Engine, current as of **12/31/23**. It also briefly explains some of the significant code and data choices we made, as well as any important issues or outstanding problems in each file.

thiessen\_polys

cpixel\_generator

buffer\_generator

MASTER\_meander\_rates

Generate and export Thiessen polygons

Generate and export connected pixel images

Generate and export segment buffers

Used in Thiessen polygon creation

Outputs used in main file

LS5, 7, 8 images

Outputs used in main file

Area converted from non-water to water by study site in rolling 5 year window, 1990-2023

### 

### The Main File: MASTER\_meander\_rates

This file does all of the actual computation and analysis, and exports the tables that are used in R for further analysis/creating figures.

*One sentence summary*: Define individual study areas (Thiessen polygons) throughout the entire AOI, determine water occurrence by pixel in Landsat images, count the change in water area within each buffer in a rolling 5-year window, export results in tables.

A pseudocode outline of the file is as follows:  
*Setup*

* Imports (previously generated assets): thiessen polygons, project AOI (WesternMack\_1), high slope mask, and yearly connected pixel masks.
* Import Landsat 5, 7, and 8. Filter, merge. We use Collection 2 Surface Reflectance

*Water Identification*

* Define function **waterOnly**, which takes a single Landsat image and adds a water occurrence band, in which pixels that are water have the value 1 and all other pixels are masked.
  + Water detection relies on WRI algorithm. The land and cloud band-based masks and the slope-based mask (to remove mountain shadows in high-relief areas) are used to help mask out the surrounding landscape.
  + We also preserve the input image’s original mask, since some areas will be masked (a few pixels on edges/corners, plus the stripes from scan line error in landsat 7 images)
* Define function **getWater\_years**, which takes a year and returns a single masked image of the entire study area, where pixels that are water over a two-year window (input year plus following year) have the value 1 and all other pixels are masked.
  + Important function! All the tables that are exported are based on the output of this function and counting the water and non-water pixels in the output image for various years.
  + Take all Landsat images for the given year AND the year after. Map the waterOnly function onto each image.
  + Take the output images and select the water occurrence band of each image.
  + Unmask the image, setting unmasked pixels (which are non-water pixels) to 0. However, pixels that were masked in the original Landsat image (due to scan line error or edge misalignment) stay masked.
  + Now we have an image collection where each image has pixels that are 1 (water), 0 (non-water), or masked (no data available)
  + Determine the water status of each pixel: For each pixel, look at all images that contain that pixel for the given 2-year window. Take the mean of that pixel’s water occurrence value across all these images (images where the pixel is masked do not factor into the calculation).
  + If a pixel’s mean water occurrence is above 20%, pixel counts as water (pixel value remains 1). Otherwise, pixel is non-water and is masked out.
  + For the northern part of the study area, apply a mask generated from connected pixels analysis. This masks out any near-channel lakes above ~67°N.
  + Output the resulting image – one band named ‘water’ which captures water occurrence for a 2 year period over the entire study area. Water pixels are all 1, non-water pixels (land pixels, in almost all cases) are masked.
* Create a list (waterYear\_long) of years (1985-2023) for which we will calculate total water area in each study site (Thiessen polygon). Map the getWater\_years function onto each year in the list, generating a water-occurrence image for each year.
* Create an image collection (waterYear\_short) that has the same water-occurrence images, but only for 1990-2023. We will calculate water area change in rolling 5 year windows for this list (therefore the first year we can start with is 1990).

*Water Area Change*

* Define function **getareaChange** which takes a yearly water-occurrence image (output of getWater\_years), compares it to the water-occurrence image 5 years before, and calculates changes in land and water area over those five years.
  + Determine how much land was lost (non-water pixels that became water pixels) and gained (water pixels that became non-water pixels) by comparing the current water-occurrence image to the 5 years prior water occurrence image
  + Return an output image with two bands. The land lost band is 1 for pixels that changed from non-water to water and masked elsewhere. The land gained band is 1 for pixels that changed from water to non-water and masked elsewhere
  + Note – non-water pixels are primarily representative of land, but can also represent clouds, ice, or lack of data. Since water occurrence images use a 2-year span, we expect transient factors like clouds to be averaged out, but they may still slightly impact the data.
* Map the getareaChange function over the waterYear\_short image collection to generate maps of area change.
* Define function **areaCount** that can do two things: either count the total water area within each Thiessen polygon, or count the land gained and lost within each Thiessen polygon. Area counting relies on the ee.pixelArea method, not using the count reducer to count actual pixels in region of interest (susceptible to map projection distortion).
  + Case 1 (count total water area). Take a water occurrence image (output of getWater\_years) for a specific year. To ensure accurate area calculations, create an Earth Engine pixel area image and multiply by water occurrence image. Then use reduceRegions to count the total water area in each Thiessen polygon
  + Case 2 (count land gained and lost). Take an area change image (output of getareaChange) for a specific year. To ensure accurate area calculations, create an Earth Engine pixel area image and multiply by water occurrence image. Use reduceRegions to count the total area gained and total area lost (applying the reducer to both bands) in each Thiessen polygon
  + In both cases: return the original feature collection (the Thiessen polygons) with a number of additional properties:
    - If case 1, total water area (in square km)
    - If case 2, area gained and area lost (both in square km)
    - In both cases: start year (the beginning of either the 5 year area change window or 2 year water occurrence window), lat/long of the polygon’s centroid, the polygon’s unique ID (point\_ID), the stream order at the polygon, and a number of properties derived from the original Merit dataset
    - Temperature (temp calcs file) and permafrost status (in R?) are calculated and associated with thiessen polygons separately
* Map the areaCount function over the collection of area change images (perform case 2) and flatten the output into a single feature collection. Export the collection as a table to Drive.
* Map the areaCount function over all water-occurrence images (1985-2020) (perform case 1) to count the total water area within each segment each year. Export the resulting collection as a table to Drive.
* *Note –* for both exports, we batch the exports and only export the data for ~55 Thiessen polygons at a time. Without batching, we exceeded Earth Engine peak memory usage limits. Loop setup will be left in messy code version but taken out of clean code version

#### Resolved issues in this file

* Landsat 8 images: using in analysis now that we have moved to Collection 2 and don’t need to worry about harmonization. Initial conclusion that LS8 doesn’t do well at inland water detection turned out to not be true
* We are using all data from Landsat 7, including the striped images (May 2003 and later). The striped area is masked.
  + The masked striped area was counted as “no water” in previous versions of this code. But now we just keep that area masked and count it as “no data”
* We have moved from [Collection 1 data](https://developers.google.com/earth-engine/datasets/catalog/LANDSAT_LE07_C01_T1_SR) to [Collection 2](https://d9-wret.s3.us-west-2.amazonaws.com/assets/palladium/production/s3fs-public/atoms/files/Landsat-C1vsC2-2021-0430-LMWS.pdf), in which all of the same data is available, but some of the scaling, bands, and bitmasks have been changed. Jordan rewrote water detection code.
* Choosing cut-off values for water vs non-water pixel in get\_waterYears – messed around with it a bit but couldn’t really think of a good analytical solution. We feel fine sticking with 0.2 for now.
* Connected pixels area has high noise in results due to lake-edge fuzz and yearly variability in which lakes are connected to the main channel. We kept cpixel areas in bulk analysis of results but didn’t present them individually or as a distinct latitude bin

### 

### NDVI Calculations

This file calculates the per-pixel NDVI value across all images and years, and exports the 5-year rolling average NDVI value for each Thiessen polygon.

Pseudocode:

* Import LS 5, 7, and 8 images as in main file. Rename bands and remove speckle as in main file.
* Define function NDVI\_Calc to add NDVI band to an image, using normalized difference between NIR and Red bands
* Define function getNDVI\_years, which takes a year and returns an image with mean per-pixel NDVI value across the entire study site using all images in a two-year window
  + This is basically the same as getWater\_years but without the masking and the 20% cutoff
* Call NDVI\_Calc to add NDVI band to all images
* Call getNDVI\_years to create yearly NDVI images for 1985-2023.
* Define function ndviReduce which takes a yearly NDVI image and returns the thiessen polygon collection where each feature has been given mean NDVI value within that thiessen polygon for that year
* Map ndviReduce over the list of yearly NDVI images 1985-2023 to create a feature collection with NDVI data for each year
* Export that feature collection

### Temperature Calculations

This file (‘MASTER\_Temp\_Calcs”) calculates the number of degree days in each thiessen polygon for every 5 year period of the study. Annual values are summed in subsequent years to produce the 5 year degree day values.

* NOTE: *All code is wrapped in a large for loop because trying to export all thiessens, all years exceeds the memory limits allocated to individual Earth Engine Users. You can trying running all at once by commenting out the loop at the start and the final ‘}’ at teh very end of the code. You will also want to change the name of the exports, which are updated in every loop, as designed now.*
* Loads the ERA5 climate reanalysis data (daily mean 2m surface temp), filters it for the years of interest (1979-present), and the ice-off season (DOY 151-288).
* First function “get\_Above0Collection” calculates number of days above 0 and degree days for every year; returns 1 images for every year with bands for degDays and number of days with a mean temperature greater than 0 Celsius
* Map “get\_Above0Collection” over the list of all years, to generate an image for each year.
* Function ”get\_above0” is called and mapped over a list of all years minus 5 (1979-2018). This function calculates the degree days and number of days above 0 in each 5 year rolling window. Returns one image with the number of degree days and above 0 days in each pixel in each five year window.
* “reduceAbove0” function is mapped over the image collection produced by “get\_above0” and finds the mean value of degree days and day above 0 within each polygon in each 5 year window (i.e. every thiessen polygon end up with values for every year from 1990-2023).
* Export results.
* Future Temp: The script “MASTER\_FUTURE\_temp\_calcs” follows the same steps as above but uses CMIP6 RCP4.5 model data (specifically CanESM5 model, though the ensemble mean can also be used) rather than the ERA5 historical reanalysis data. Output is the mean value of degree days and day above 0 within each polygon in each 5 year window (i.e. every thiessen polygon end up with values for every year from 1990-2023).

### Connected Pixels: cpixel\_generator

This file uses a connected pixels analysis to generate a yearly mask that helps with filtering out lakes in the northernmost part of the study area. Once the yearly masks are generated, they are be stored as an Image Collection in Drive and imported as an asset in the main file.

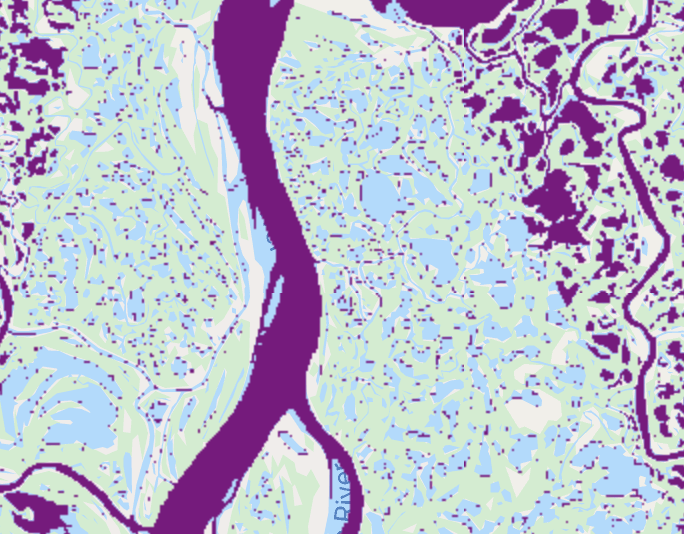
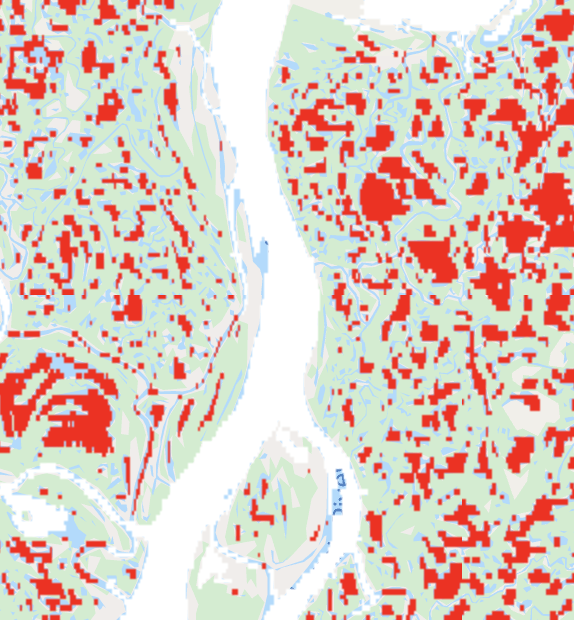
*One sentence summary:* For each year, identify water pixels in a restricted AOI of the study area, count the number of connected neighbors for each pixel, mask out any non-fully-connected water pixels, and export that mask as an image to Drive for use in the main file.

Pseudocode:

* Import Landsat 5, 7, and 8 images. Rename bands and remove speckle (as in main file), also scale raw images to surface reflectance values (not done in main file)
* Define the area in which to perform connected pixel analysis (“New\_cpixel\_AOI”) (see Imports). This is the area loosely surrounding the main channel north of 67 degrees latitude. (67°N is where the original analysis from John Perotti’s thesis stopped)
  + The lateral extent of the cpixel area is greater than the lateral extent of the Thiessen polygons in which we analyze water occurrence
* Define the area over which we will export the mask (‘extent’). This needs to cover the entire study area
* Define waterOnly and getWater\_years functions to identify water pixels in study area (same functions as in MASTER\_meander\_rates)
  + Land and cloud filter thresholds are adjusted according to the surface reflectance scaling
* Define a function, make\_cpixel, to make the connected pixel mask for each year:
  + Get the getWater\_years image for that year (water=1, non-water masked)
  + Clip that binary image to the connected pixel AOI
  + Do connected pixel analysis on the binary image. This produces a new image, where each pixel’s value is the number of connected neighbors (e.g. number of connected water pixels).
    - Max number of neighbors is 1024, which is a cap set by Earth Engine. We chose to use four-connectivity instead of eight-connectivity, but didn’t see a big difference between the two.
  + Transform this connected pixel image into a usable connected pixel mask:
    - Set all unmasked pixel values less than 1024 (water that is not completely connected) to 0
    - Expand the images footprint to cover the entire world, and set all pixels outside New\_cpixel\_AOI equal to 1. This will ensure that when we apply the cpixel masks to filter out northern lakes, we don’t accidentally mask out any other parts of the study area
    - Clip the newly-expanded image to ‘extent’ so that it only covers the study area, not the entire world (unclear if this makes a difference in computing time or not)
    - Store the year that this connected pixel mask represents
* Call this function to make a connected pixel mask for each year from 1985-2023
* Export all images to Drive at a scale of 100 (meters).
  + The scale determines the size of each pixel in the output image. Larger scales are less accurate at capturing the edges of lakes, but smaller scales are more computationally expensive. Also, lakes more easily “max out” (hit the cap of 1024 connected neighbors) on smaller scales, because the same water surface area is subdivided into more pixels.

#### Unresolved Issues

* The main issue is “lake-edge fuzz.” We do connected pixels at a 100 m pixel scale, but the water binary image has 30 m pixels. So the connected pixels image doesn’t perfectly cover all of the water pixels on each lake. When we mask out the lake area identified by connected pixels, it leaves behind those water pixels on the edges of the lakes (see image below).



Red is what will get masked out (this example is 150 m scale). Right image is after masking, showing purple (water) pixels around edges of lakes.

* + If these unmasked water pixels were always in the same place, it wouldn’t affect anything. But if they move, it will affect the analysis to some extent (since from year to year, some non-water pixels will become water pixels).
* We observed a lot of variability in which small lakes/ponds/channels were connected to the main stem from year to year. The varying connectivity of these bodies of water creates a large meander signal, which (we think) is why the results from the cpixel area are so noisy.
* We couldn’t figure out how to do the image exports in a loop. So we have a chunk of code that we just re-run and change which image we are exporting for each iteration. Not a huge deal but mildly annoying
* Other minor issues:
  + Very small/narrow streams are sometimes not recognized as connected entities, especially on larger export scales. Not a problem for the current (northern) connected pixels AOI, would be a problem over the entire watershed.

### 

### Thiessen Polygons: thiessen\_polys

The Thiessen polygon approach is an alternative method for splitting the entire study area up into smaller study sites. The original method for creating study sites was splitting the river up into 5 km long segments and wrapping each segment in a 5km by 2 km buffer (polygon). See the buffer generation section below for more on buffers.

The first generation of buffers were derived from the Hydrosheds dataset, which had inaccurate river centerlines. They also were somewhat irregular, with random gaps and overlaps between them. The second generation of buffers (derived from MERIT) follow the river more accurately and have fewer gaps, but have significantly more overlap. Buffer overlap is undesirable as it leads to double-counting some of the data.

The Thiessen polygon approach is better because it eliminates the possibility of buffer overlap. Gaps between buffers are still possible but much smaller and less frequent.

*One sentence summary:* Take the segment buffer centroids, create polygons such that the area inside each polygon is all closer to one centroid than any other, then export those polygons.

Pseudocode:

* Lots of commented-out code: old methods for filtering out large lakes, experimenting with MERIT, unifying the small MERIT features into much larger ones grouped by stream order, and two different versions of the buffer-generating function (getCutRivers).
* Define imports:
  + Single\_Unioned\_Buffer: All of the MERIT segment buffers joined together into a single polygon that buffers the entire stream network.
    - Note: we had previously created this single polygon in the script, but it was using a lot of memory, so we are loading it as an asset instead.
    - The code to create this feature is included as a block comment. It is made by buffering MERIT\_Unioned\_Streams (the streamline line segments unioned by river order) according to river order, not by directly unioning MERIT\_Segment\_Buffers
  + MERIT\_Segment\_Buffers: All individual buffers, split into roughly 5 km long segments with significant overlap.
    - The buffers used here were generated by the fall version of getCutRivers, which was applied to MERIT\_Unioned\_Streams and exported from the buffer\_generator file
    - The fall (of 2022?) version has a slightly different method for creating the buffers compared to the summer version, but it doesn’t really matter because we don’t actually use the buffers, only the centroids
* Get the centroids of each of the buffers in MERIT\_Segment\_Buffers and store them in the Feature Collection “centroids.” These centroids define the individual study sites within the project’s study area.
* Add a property to each centroid called “point\_ID,” with the value being a random integer.
* For each centroid in “centroids”:
  + Create a distance image, where the value of each pixel is the distance to the centroid (up to a maximum distance of 10 km).
  + Invert the distance image (multiply by -1) so that farther distances are lower (more negative) numbers.
  + Add a constant band to the distance image that is the value of “point\_ID.”
  + Add a constant band to the distance image that is the stream order (which is a property of the centroid).
* Reduce the Image Collection of distance images to a single image of the entire study area:
  + Do a quality mosaic using the distance band. Each pixel in the mosaiced image has the lowest possible distance value (is closest to a centroid) of any pixel in that location across all distance images.
  + Select the point\_ID (which identifies the centroid) and the order (which has the steam order) bands.
* Create vectors (polygons) from this raster image using reduceToVectors. Borders are defined by the point\_ID band, so that the area within each polygon is closer to that polygon’s centroid than to any other centroid in the study area. The geometry to reduce over is Single\_Unioned\_Buffer, so the output vectors follow the shape of the stream buffer (and this is why we didn’t need to clip earlier). We used a 300m scale.
  + The resulting features have point\_ID saved as a property. Stream order is also saved, and the name of that property is “mean” (because we reduced using the mean reducer).
* Export the Feature Collection of polygons to Drive.
* Manually review vectorized polygons, looking for any that didn’t have water or intersect streamlines or had other problems – see supplemental methods doc
  + No fancy code, just displaying stuff on map. Bad thiessen polygons were manually removed from the export table before it was used in master\_meander\_rates

#### Notes

* Not an issue but something to be aware of: when we clip the distance image to unioned\_polys, we are setting the limit for how far to either side of the stream we will look for meander. The buffers that make up unioned\_polys are buffered to different widths depending on stream order (in the past, it was a constant 2 km buffer regardless of stream order)
* We created 300m, 400m, and 500m scale thiessen polygons and decided to use the 300m ones. No real reason, just assumed that higher resolution = better
* The stream order is saved as a property called “mean,” not “stream order”

### Buffer Generation: buffer\_generator

This file generates a feature collection of buffers around the river centerlines. Each buffer represents a study site, and is supposed to be 5 km long (following the river centerline) by 2 km wide. The river centerlines are from the MERIT vectorized streamlines dataset.

Once the buffers have been generated, they can be exported to Drive and used as an asset elsewhere. The actual buffer-generating code doesn’t need to be rerun unless the centerlines or buffer generation methods change.

*Since switching to Thiessen polygons, the only parts of code here used elsewhere are:*

* *MERIT\_Segment\_Buffers – source of centroids used to create Thiessen polygons*
* *MERIT\_Unioned\_Streams – used to create the Single\_Unioned\_Buffer asset in thiessen polys code file*

*One sentence summary*: Load vectorized streamlines from MERIT, group them by stream order, slice into 5 km long by 4 km wide buffer segments across the entire study area.

Pseudocode:

*Setup*

* Use the Hydrosheds dataset to define areas that contain large lakes within the project AOI.
* Load the MERIT vectorized streamlines dataset (MERIT). Filter for streams of 7th order or larger, for the project AOI (removing some streams in the eastern area of the watershed), and to remove the lake areas defined in the first step.
  + Note that project AOI here is WesternMack, not WesternMack\_1 which is what we actually ended up using (and is a geometry import created in master\_meander\_rates). I’m not sure how big of a difference there is, but it doesn’t really matter because in the main script we only use images in WesternMack\_1 and the thiessen polygon collection has been trimmed to fit in WesternMack\_1. (It may have just been small change to cpixel area but unsure)
* (export the filtered collection, I think called MERIT\_VF\_Ord7)

*Union Segments Together*

* Convert the many short segments (features) into fewer, longer segments (features):
* Create an empty image and paint it with the river centerlines from MERIT, where the color of each centerline is given by the stream order. This image has 1 band, which is the river order.
* Do connected components on this image, which gives an output image with two bands: one band that gives a unique label to each connected area, and one band for the river order.
  + I don’t think this is actually necessary and we could just reduce to vectors from the original painted image, but it’s not really hurting anything
* Reduce this raster image to a feature collection. There is one polygon per connected area, and each feature has the river order as a property.
* Map over this feature collection a function that for each feature:
  + Find all the original MERIT segments within the geometry of this feature.
  + Combine (union) those segments together.
  + Give this new unioned feature a property that is the river order.
* The feature collection created by that mapping is what we wanted–the original MERIT dataset, but with the many short segments combined into fewer, longer segments (generally >>5 km).
* (export the results at this point as MERIT\_Unioned\_Streams)

*Create Buffers from Unioned Segments:*

* Define cutLength, the maximum length of a segment (currently 5000 m)
* For each feature:
  + Get the length of the feature
  + Define the intervals at which the feature will be cut, which is 5 km unless the feature is shorter than 5 km (then it is the length of the feature).
  + Cut the feature into smaller features at the specified intervals.
  + Take all smaller features that are longer than 3 km. For each feature:
    - Store its length (in theory, we should have lots of 5 km long features and a few edge cases).
    - Select the 2-3 km interval (this is the middle km of a 5 km long feature), and discard the rest of the feature. (theoretically resulting in a 1km long line with 2 km empty on either end)
  + Return all of these modified features in a feature collection, excluding any features less than 1 km long (which are presumably some weird edge case)
* We now have a feature collection of 1 km long lines, separated from each other by 2 km of empty space.
* Buffer each feature (currently buffering by 2 km), resulting in a 5 km long buffer segment (no longer a line) that is 4 km wide
* Export the finalized buffers as an asset to Drive.

#### Unresolved Issues

* The code is not particularly reproducible:
  + It’s not clear where the original MERIT dataset (“MERIT\_Vectors\_Mack”) comes from or how to load it. It’s imported as an asset.
    - It exists somewhere in Earth Engine–tedious to import but our final version can show the import step
* The cutlines function, which is supposed to create relatively uniform and non-overlapping buffer segments, does not do a very good job – original version had lots of gaps between segments and the revised version had lots of buffer overlap
  + I’m not really sure what a good solution to buffer overlap is. We decided to not fix it and try Thiessen polygons instead.
  + The cutlines function filters out a bunch of features that are not the expected length. It’s not clear how many of these there are or where they are–but this is probably related to the occasional gaps between buffer segments. (again, irrelevant now that we are using Thiessen polygons)