Gorzow Scaling - Midpoint Layer Derivation

This notebook identifies the source data for Gorzow scaling and explains pre-processing to output the four basic layers on which scenarios are built:

1. Simplified Land Cover
2. Simplified Land Use
3. Slope
4. Sunlight availability

To replicate this work, first open a blank project in QGIS. I have tested most of this on QGIS 3.18 and 3.24, though anything past 3.00 should suffice. When possible, I will include both a verbal explanation and a screenshot for the procedure described - this should make it easier to replicate the process in varying versions where the syntax or appearance may change slightly.

The goal of this notebook is to translate the input layers into the four basic layers described above. Poland is an incredibly data rich environment, so there are endless possible combinations of layers to get to those four layers. Below is an explanation of how I did it, though I also make note of other sources of the data.

At the end of this analysis, we will have transformed our inputs into four aligned rasters at 1m resolution. All functions will be conducted in EPSG 102173 because it works well with solar irradiance actions, has high accuracy, and is the default projection for a variety of Polish data.

# Binary Slope Layer Derivation

This layer will describe slope at 1m resolution across the city. These will be used to determine suitability for any sort of growing space. Less than 15% slope means it’s suitable. The final layer produced via this procedure will have the following codes:

* Ineligible, over 15% grade-- 0
* Flat ground – 1
* Flat roof – 2

Overview: We can use the LiDAR-derived DEM and DSM to identify flat ground and flat roofs throughout the city. We have to do these two things separately, since ground level varies across the city and the DSM is reported in feet above sea level.

We will use the following data sets:

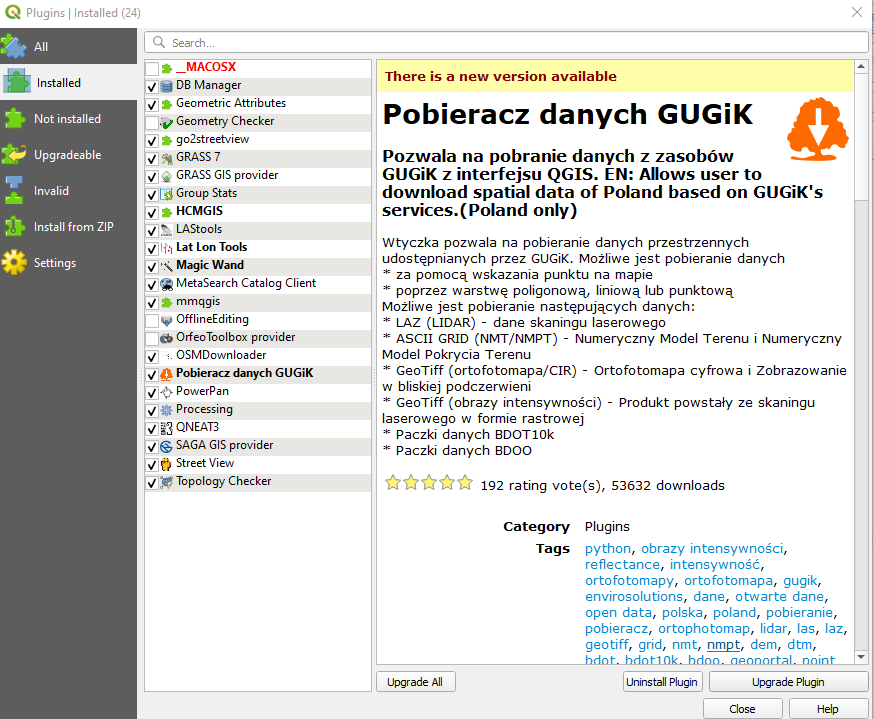
1. Gorzow boundary
2. NMT - aka DTM - [metadata](https://gis.ny.gov/elevation/metadata/2017NYC-topobath-DSM.XML)
3. NMPT - aka DSM - [metadata](https://gis.ny.gov/elevation/metadata/2017NYC-topobath-DEM-hef.XML)
4. BDOT Buildings dataset - BUBD - [metadata](https://github.com/CityOfNewYork/nyc-geo-metadata/blob/master/Metadata/Metadata_BuildingFootprints.md)

We will begin by preparing city-wide DEM and DSM layers, then we will proceed with the flat ground analysis. We will then identify flat roofs. Finally, we will mask buildings from the ground layer and add in the buildings results. In total, this should only take 30-40 minutes because so much of it is just big raster calculations. We conduct this analysis first not because we necessarily use it right away, but because these two rasters can form the foundation of aligning all our other layers.

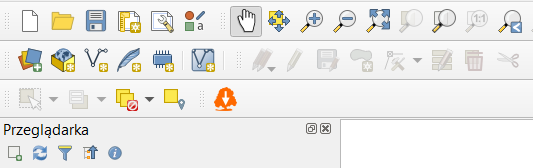
## Download the NMT (DTM), NMPT (DSM), and EGIB (building and plat map) layers

To download these files, we will use the Pobieracz danych GUGiK (Tr: “GUGiK data download”) plugin in QGIS. The plugin can be a bit difficult to use if you don’t speak Polish, so I’ll walk through it step-by-step again. Much of this writeup is based on earlier work by Konstancja Fedeńczak.

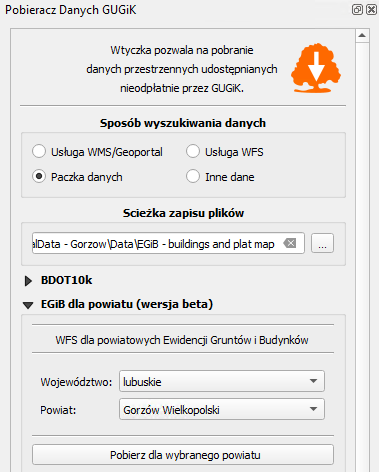
The spatial data authority has made lots of data available for download (including several we will use - BDOT data, elevation data). In order to make it faster and more efficient, they created a special plugin for QGIS, which allows you to download large amounts of data at once.



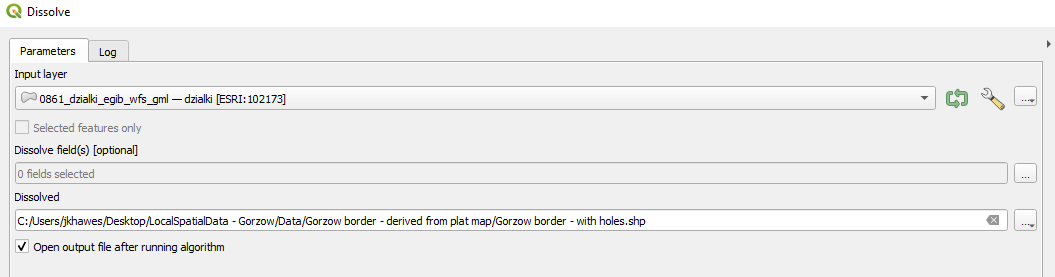
The plugin should show up on the toolbar as an orange tree with a down arrow. If it doesn’t, right click on the toolbar and select “Pobieracz danych GUGiK panel” which should bring up the download box just the same as the button would have.



At the very top of the app you have ways to search for data in WFS and WMS/Geoportal as well as a data packet service (paczka danych) and an “other data” (inne dane) portal. For the NMT and NMPT data, we can use the WFS or WMS, but for the EGiB (buildings and other data), we have to use the Paczka danych. This is convenient, though, since the WMS and WFS require you to define the boundary of your data download, while the Paczka danych has preset boundaries according to municipality borders. So we’ll download the EGiB data first, not only because it’s easier, but because it actually gives us a file we can use to set the boundary for the elevation data. So, click on Paczka danych. Set the download location - name it something with EGiB so you remember where it came from, but also include a translation so you know what it is. I use “EGiB - buildings and plat map.” From here, you can open the EGiB drop-down. The settings are simple - we select the correct location and hit download. Gorzow is in the Lubuskie vovoideship (Wojewoztwo in Polish), then we can select Gorzow Wlkp. Hit the button at the bottom to download.



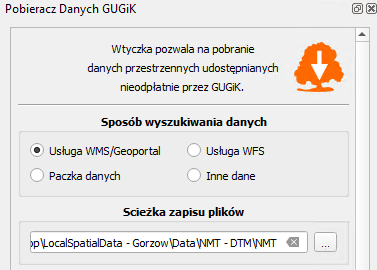
We have now downloaded a package with all the buildings in Gorzow as well as the plat map. There are also two point files, and honestly I don’t know what they include. We’ll use the building file (0861\_budynki\_egib\_wfs\_gml) directly for this procedure, and we’ll use the plat map (0861\_dzialki\_egib\_wfs\_gml) indirectly now. We will take the plat map file and use the Dissolve tool to create one large polygon that represents the entire city. While this isn’t strictly necessary with a WMS, it might help things run faster and smoother, so it’s worth the 5 minutes it takes - plus a boundary layer is always useful to have lying around. Before we can do this, we might have to tell the file what it’s projection is - sometimes the GML files import without a projection. If it does this, assign projection 102173 by right-clicking on the layer and using the Set Layer CRS option. Only one other bit of fancy settings - make sure to change the Invalid feature filtering under the wrench options to ignore invalid features. This is basically a precaution against the fact that in some versions of the plat map there are overlapping polygons.



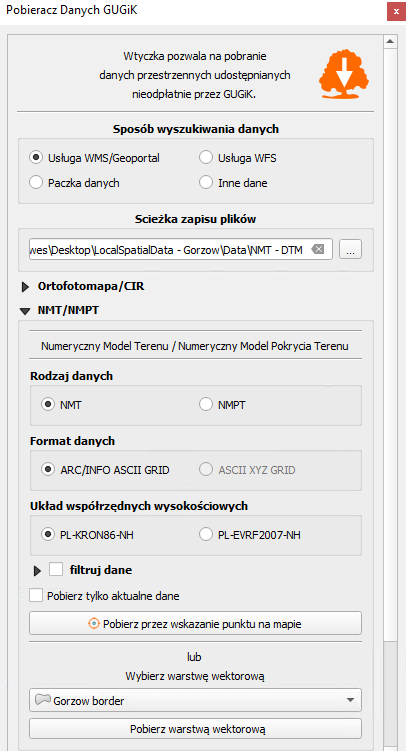


Assuming you do have a version with invalid features, there will be some holes in your new layer as well as some free-floating polygons. Turn on layer editing with the yellow pencil. Use the Delete Part tool - just click on the free-floating polygons. Ctrl-z if you accidentally get the big one. Once the free-floating polygons are all gone, you should just have holes left and the big polygon. You can fill these holes with the Fill Ring tool. Go around and delete the holes by clicking on them. All of this edits the original polygon, so if you want to have the original result of the dissolve function left at the end, you’ll want to save this separately before using the edit tools.

Now, back to the elevation data. Instructions below are for the WMS, so select that if you want my help. After you select the download server, make sure you identify the download location in the box labeled “Sciezka zapisu plikow”.



There are a variety of settings we need to set. First, of course, select the type of elevation data we’re interested in. At some point we’ll need both, obviously, but start with whichever you prefer. The format should be pre-selected on the only option. Then the third line (Uklad…) is basically just selecting the coordinate system. It doesn’t matter much. I use the default, since it’s in meters and works well for Gorzow. We’ll use the second download option, which downloads tiles that fall within the polygon we just created. Make sure the polygon is set to Gorzow border, then hit run (Pobierz wastwa wektorowa) below that. When you hit the button, it thinks for a minute, then asks you if you’re sure you want to download all the files in this area - it should be around 70. Fair warning that the pop-up sometimes hides behind another window, so be vigilant or you’ll be sitting around waiting for it forever. Hit yes and the download should begin. Keep track of this if you want by watching the download destination folder - though the new update for the plugin also shows download progress along the bottom like a processing toolbox function.



First note: There are two types of files that come when you do the downloads - I believe they’re derived from different data sources, but they both report “laser scanning” as the source and I can’t tell much difference between them. Fortunately, there’s an obvious choice in both cases. In the case of the DTM, there’s a 3712 file, and a 72777 file for each location. The 72777 files are 0.5m resolution, and we don’t need that high of resolution, so I just delete them after they’re downloaded. If you do want these for any reason, they’re the exception to the 102173 projection rule - for some reason, they’re 2176 - ETRF2000-PL/CS2000/15. In the case of the DSM, it’s a 2822 and 72776 file. Both are 0.5m, but the 72776 files don’t actually have data for all locations in Gorzow. It’s possible this is a temporary thing if they’re doing a 2023/24 update, so you may have to come up with a different reason for choosing between them, but for now, I just use 2822 because it’s the only layer that covers the whole city.

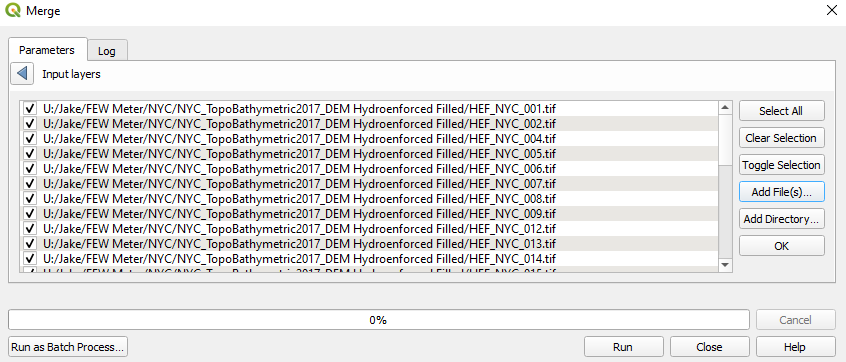
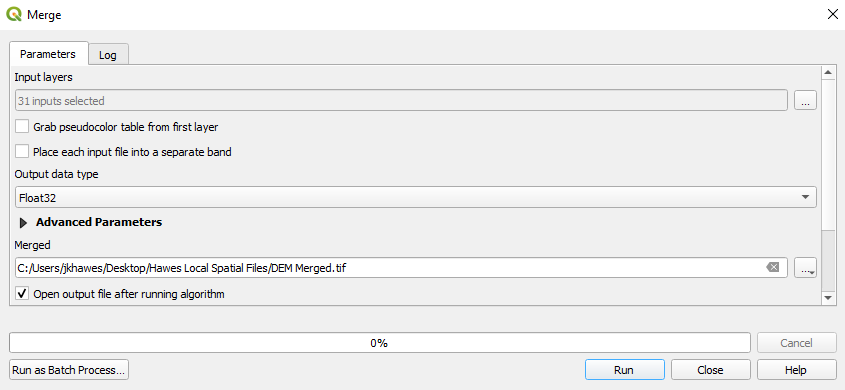
Another note: for some reason, when you download the NMT and NMPT files using the Gorzow border, it leaves out a few edge pieces. Must be something about the amount of area actually included in those tiles. Either way, you can finish the job by using the Pobierz przez wskazanie punktu na mapie tool, which allows you to point to a spot on the map and download the tile for that spot. I recommend loading all the 3712 and/or 2822 tiles that come from the first download, then use the point tool to just click on the spots where the border sticks out (see the pink below).



Once you’re done with one, you can move on to the other. At the end of all this, you should have the NMT and NMPT files in asc format, the building file in gml format, and a Gorzow boundary file you built yourself from the plat map. Don’t delete the plat map either, we’ll need that when we build a land use file.

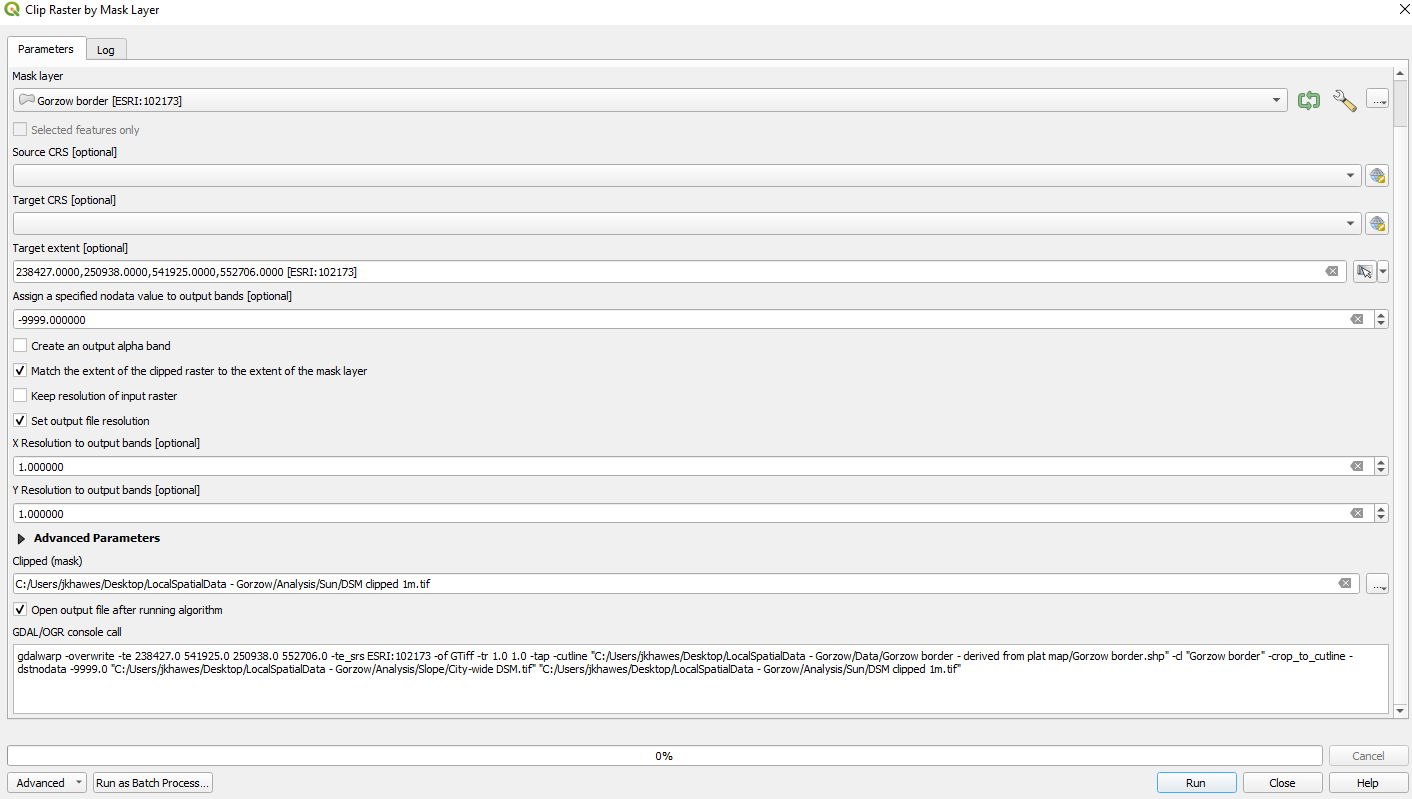
## Create city-wide DTM and DSM layers

First, as a sort of step zero, we need to stitch together the DTM and DSM layers. They are delivered in asc tiles. This is fairly straightforward with the ***merge*** tool. We don’t even need to import the rasters in Q first (and we shouldn’t, because it would take a while) - instead we can select them directly from the merge tool by clicking on Add Files on the input layers option. Ignore the fact that the files below are tif’s from NYC. This is just illustrative - the procedure is the same.



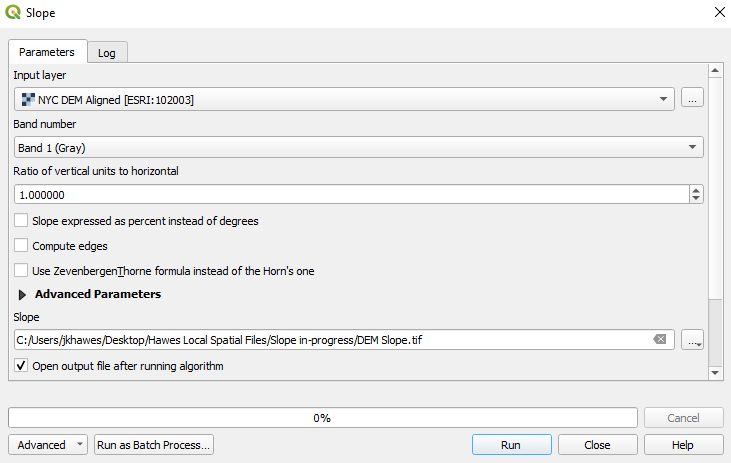
## Clip DTM/DSM to city boundaries and resample raster

Once you’ve merged all the files, I recommend clipping them to the city boundary. This is not strictly necessary, since the area is small enough that a strong computer will run everything quickly either way, but it can help with the computational time and is good practice. It’s also going to prove very useful later when you run the sun commands, since those take much longer. So before we get moving, use clip by mask layer on the two layers we just generated - DTM and DSM. When we do this, we can also convert the original DSM to 1m x 1m.

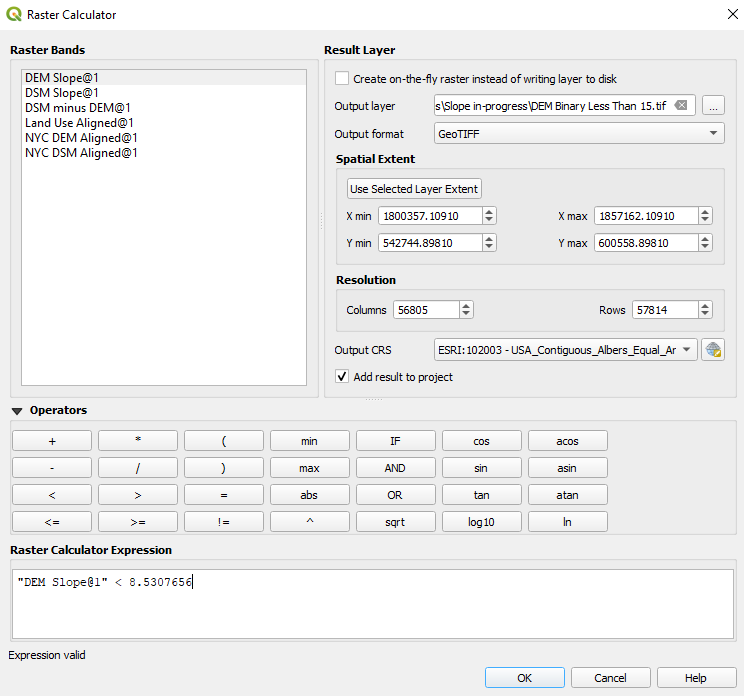


## Calculate Slope from the DTM

Now that we have clean layers to work with, we can derive slope on the ground. We can simply use the ***Slope*** function under Raster > Analysis on the merged DTM. This will yield a slope layer for all ground cells in the city. We’re interested in places where the slope is less than 15%. Unfortunately, the “Slope expressed as percent instead of degrees” function seems to return absolutely outrageous values, so I don’t recommend using that. Instead, it seems better just to convert the 15% to degrees and use that in the raster calculator in the next step. You can also run this as a batch function and derive the slope for the DSM, since that will be the first step of the next process (Identify flat roofs…).

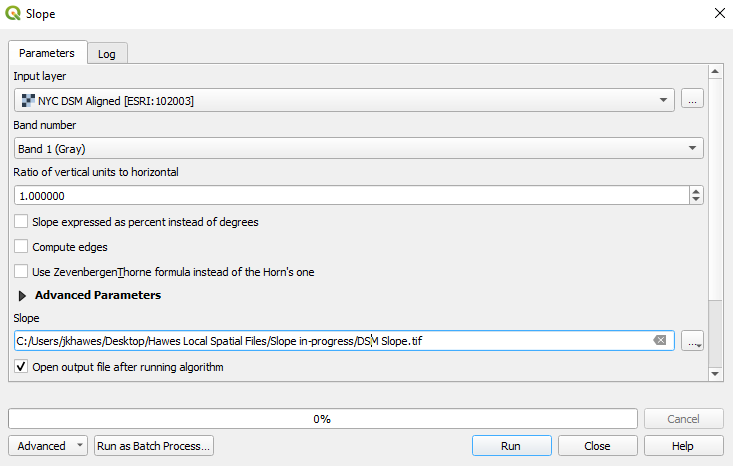


Once we have this Slope file, we can do some simple raster algebra to determine where this is greater than and less than 15%. Expressed as degrees, a 15% slope is arc-tangent of 0.15, which is 8.5307656. So we want to find places where the slope layer is less than that. You can’t also do this within the raster calculator unless you want to convert everything to radians, so it seems just as easy to use this number.

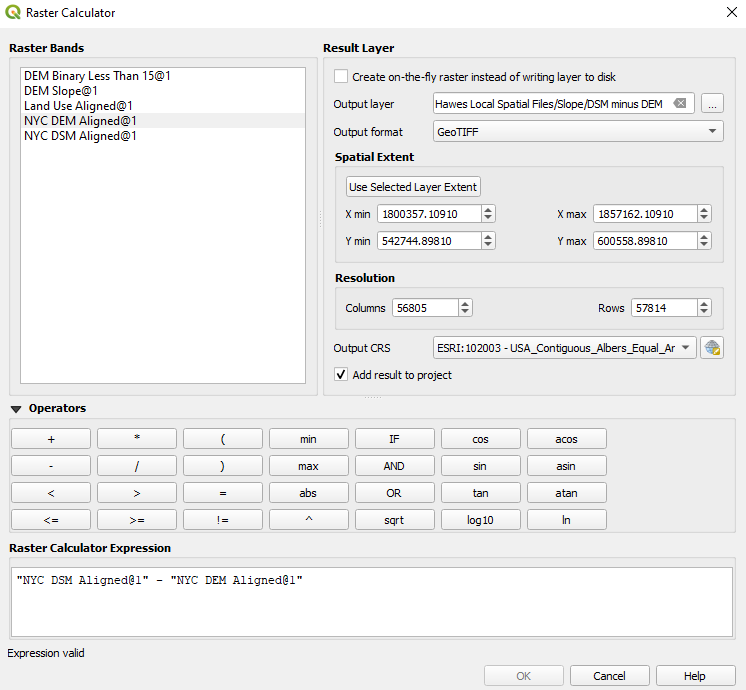


## Identify flat roofs from the DSM

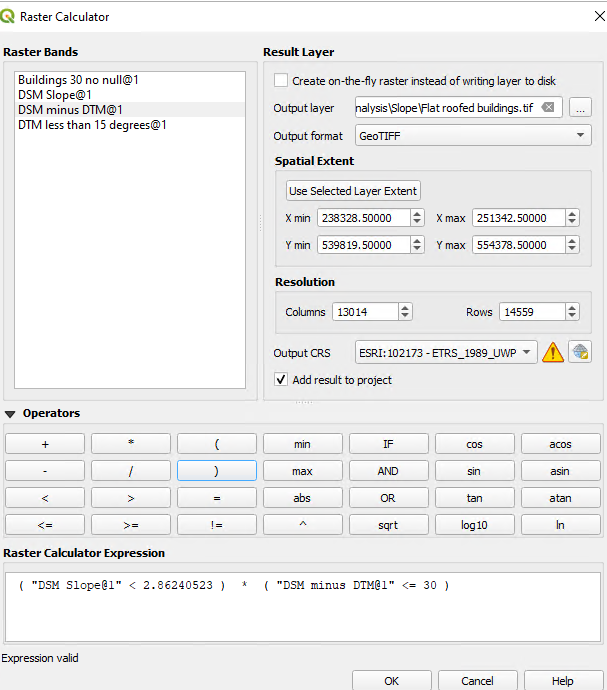
Next, we need to calculate the roof slopes - this is a bit trickier and requires several steps. First, we can run the Slope function under Raster > Analysis on the DSM.



Next, when we need to subtract the DTM from the DSM to make sure the ground level is zero all over the map.

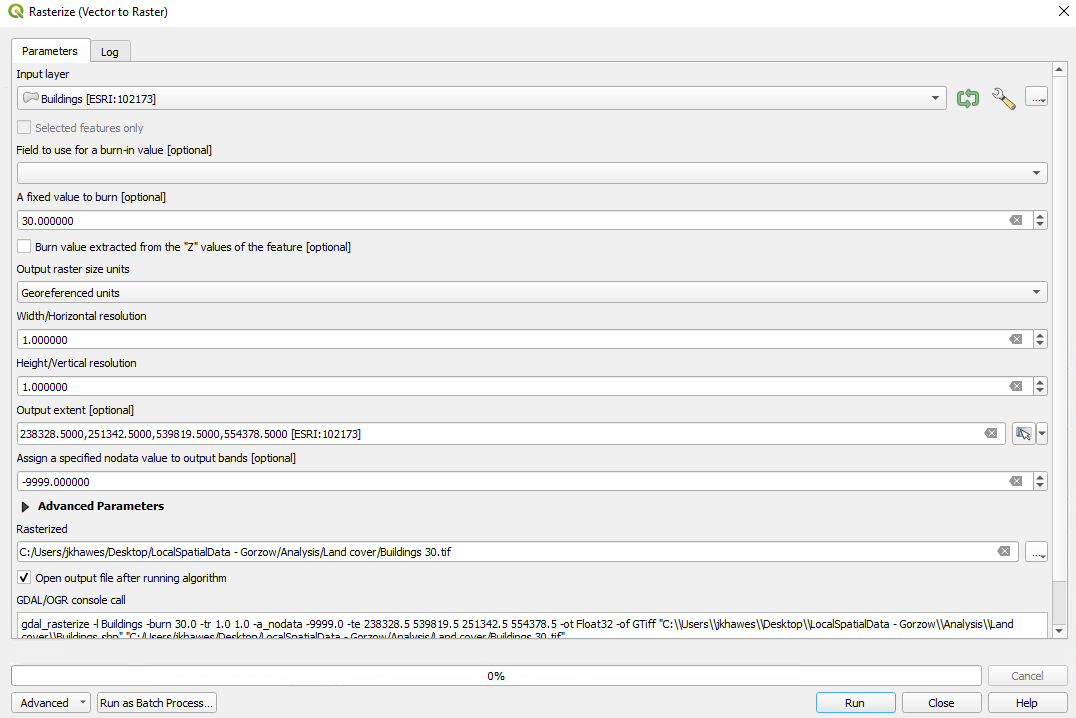


We can then run some simple raster calculations, finding the places where the Slope layer is less than 5% (less than arctangent of 0.05 = 2.86240523) and the Height of the DSM-DEM layer is less than 30m (8 stories or less) - note that this height is why we need to do the subtraction. If not, we can’t use 30 or something as a roof height, because ground level differs, so some roofs are below ground level in other places in the city. In some older versions of this, I also used 3.5 m as a minimum, but this really isn’t necessary since we use the building land cover classification below. Just need to limit the max so we’re not working on top of any skyscrapers.



## Derive a building layer

Before we can derive the final slope layer, we have to create a building layer to use as a final filter for the DTM vs. DSM. To do this, we can use the layer we downloaded earlier and create a binary raster. Instead of pure binary, I recommend making the buildings equal to 30, since this will save us a step later and it doesn’t add much complexity to the raster calculation below. We’ll use Rasterize to convert the buildings file to a raster. Before we can do this, we may have to do two things. First, we have to remind the gml that the projection is 102173, as we did before with the plat map - again, just right click on the layer and set CRS. Second, we have to convert the gml to a shapefile. The exact use of a shapefile isn’t the important part - it’s just important that we leave the gml format because the default Rasterize function doesn’t like it. We don’t need to set a field to burn anything in, since we’ll manually set it to 30 as shown below. Set the raster units to be georeferenced, since 102173 uses meters, then set the resolution to 1x1. Calculate the extent based on either the flat roofs or the flat ground files, since that will align the new raster with those. Let the no data value be -9999 as is our standard.

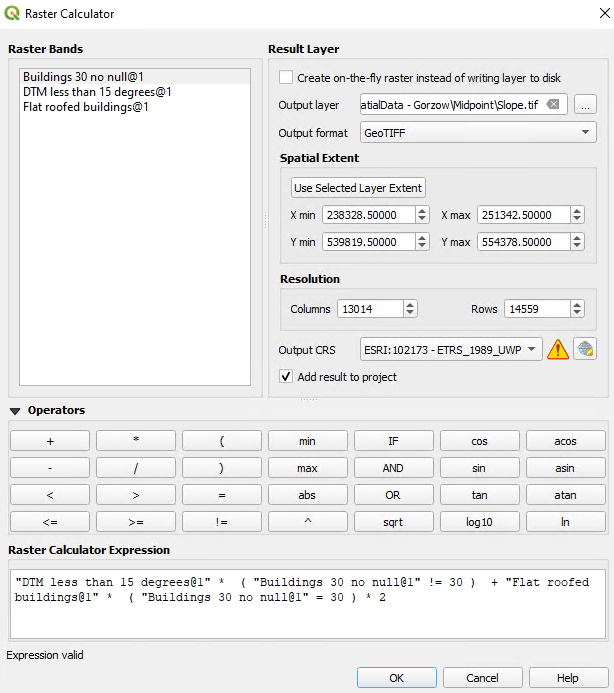


Once this runs, we’ll actually replace the null values with 0, since otherwise the raster calculation returns a bunch of null. Use r.null for this.

## Final raster calculations

Lastly, we need to combine these layers into something intelligible - we will keep flat roofs and flat ground separate for now just for the sake of preserving information. We can always reclassify later. So our goal is: 0 = > 15% slope on ground or 5% slope on roof, 1 = flat ground, 2 = flat roof. How do we get there?

First, we’ll need to make all the building footprints zero in the ground slope layer and convert everything outside of buildings to zero in the roof slope layer. To keep the information discrete, we can do one more raster calculation - RoofsLayer \* 2 + GroundLayer. In the end, we can do this all in one step. The raster calculation is exactly the same for the two layers, but inverted. See below.



After this calculation is complete, don’t move all the Slope files off the hard drive just yet - make sure to keep the aligned DSM - we will use it as our first input for the sunhours calculations later.

One other quick note - throughout this process, we have use raster calculations, which lose the no data fields. At this point, we can clean up the file considerably by cropping the raster to the city border. WE can do this with a simple Clip Raster by Mask Layer function.

# Land Cover Layer Derivation

This layer will identify open ground areas and rooftops. These open areas and rooftops will then be filtered by other qualifications from the other layers (e.g. slope). The final layer produced via this procedure will have the following codes:

* Impervious - 1
* Low vegetation, grass or dirt - 2
* Roof - 3
* Trees - 4
* Otherwise occupied - 0 (e.g., monument, water, railroad, road)

Overview: We have a land cover dataset that needs to be simplified to not have trees and needs to parcel out ineligible impervious areas like roads.

To derive land cover, we will use the following datasets:

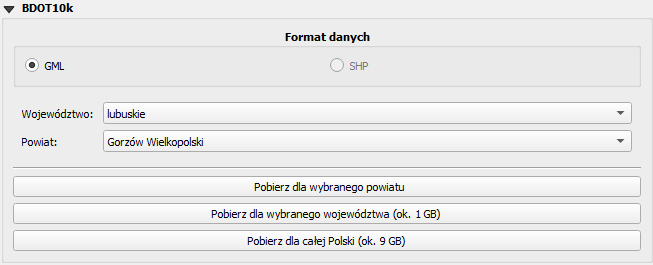
1. BDOT - download instructions below
2. Buildings - can also get this from the GUGiK Plugin (see below). EGiB or the 3D building model under Paczka danych
3. Plat map - [download](http://geoportal.wms.um.gorzow.pl/) Dzialki from WFS service - see [here](http://geoportal.wms.um.gorzow.pl/map/www/mapa.php?CFGF=wms&mylayers=+granice_miasta+ortofotomapa2019+) for other layers, names, visualization

We will reclassify and overlay these layers repeatedly to generate the final map of land cover in Gorzow. We will process each of these layers individually.

## Download BDOT

To download BDOT, we will again use the Pobieracz danych GUGiK (Tr: “GUGiK data download”) plugin in QGIS. As described above, the plugin can be a bit difficult to use if you don’t speak Polish, so I’ll walk through it step-by-step again. Again, much of this writeup is based on earlier work by Konstancja Fedeńczak.

Once again, at the top of the app you have the choice of databases. For BDOT, you download files from the Paczka Danych portal, which also means we can do it by administrative district. Gorzow Wielkopolski is part of the [Lubusz voivodeship](https://en.wikipedia.org/wiki/Lubusz_Voivodeship), and BDOT files can be downloaded specifically for the adminsitrative extent of Gorzow with the following settings:



The top button, which translates roughly to “Download for the selected county” should download only those files in the administrative extent of Gorzow (area #861 in the BDOT classification). This should run pretty quickly, since Gorzow is relatively small. The second button would download the whole voivodeship and takes substantially longer to run. The third button would download all of Poland and is quite slow.

All three options download the files in GML format. This is an open-source spatial file format that uses xml files, which QGIS interprets and generates a spatial key for (a gfs file). Don’t need to do anything fancy - can just unzip the file and drag the individual xml’s into QGIS.

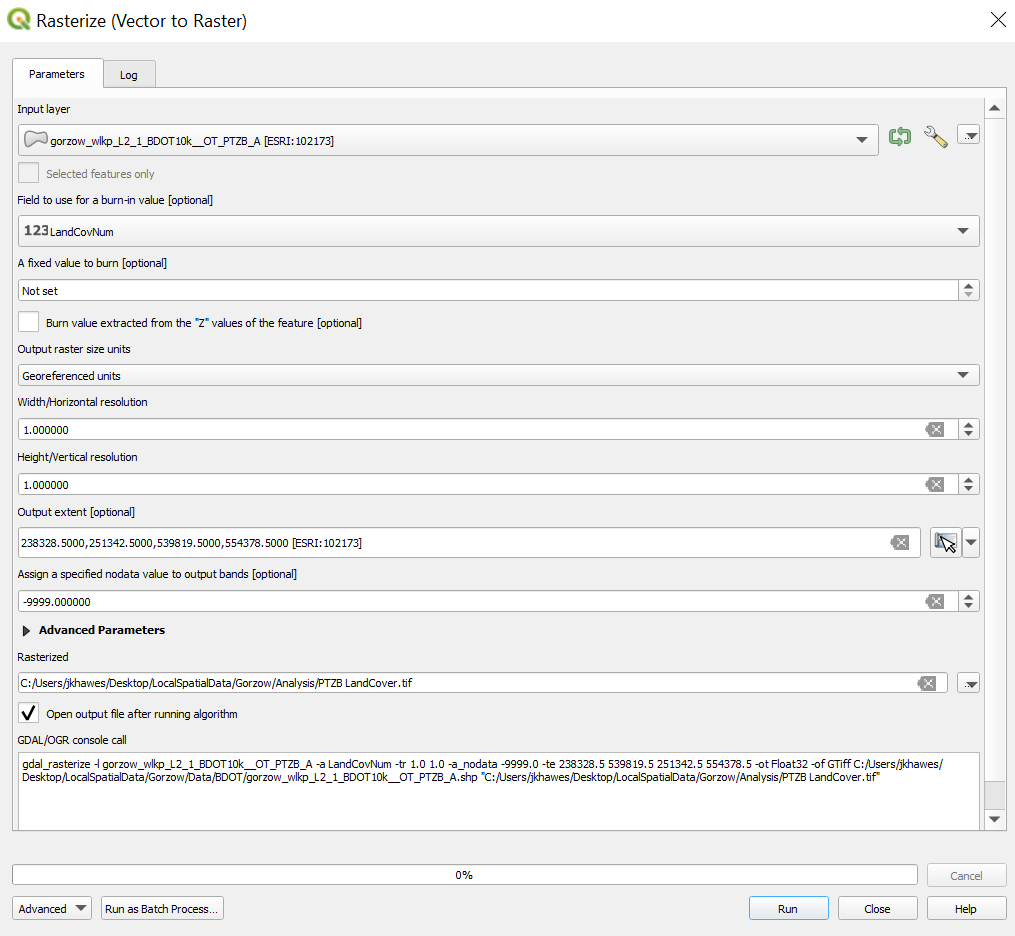
If you skipped the slope steps, go back and download the Buildings dataset from EGiB as explained above.

## Overlay PT land cover layers from BDOT

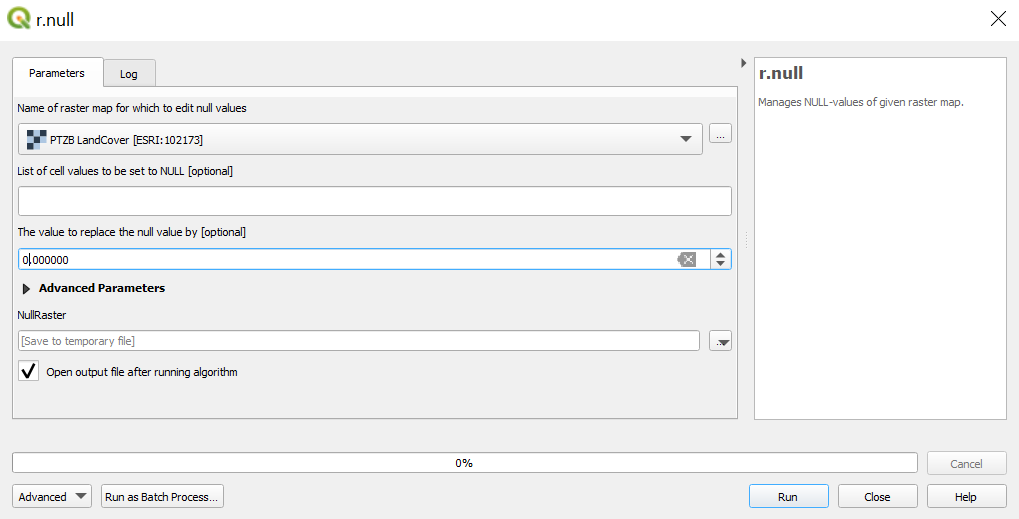
BDOT comes in kind of an odd format (similar to Dortmund if you’ve already worked through [this](https://docs.google.com/document/d/1P3FBj-uuGyviD8tYvpNZh8eToblGUd38ITeChutXnek/edit)). It is served as a series of shapefiles which each have their own classification schemes. Each shapefile is a category of land use, land cover, administrative designation, etc. and there are sub-categories within each one. I have translated the codes into English with Google Translate, which mostly gets the job done. You can find the file [here](https://drive.google.com/file/d/1FtfkSwvchFvm3sMz9rJZ3FCAyuOAg20J/view?usp=sharing). It’s a bit tricky to parse all these different layers, so I’ve detailed all the conversions. It’s worth noting that everything in this section is non-overlapping, so you can just do straight raster algebra. I do it once below as a standalone conversation with PTZB, but then you’ll notice after that I point out that you can use batch processing for a lot of this work with the PT files. Something to keep in mind. Once you get out of this PT section, things start to overlap in unexpected ways, so it’s important to follow this series of overlays in order.

I find it best to start with the land cover codes in PTZB, which is actually also a buildings file - PTZB has an attribute called roslinnosc (vegetation) with the following codes: Dr = trees, Tr = grass, Bl = Hard square (pavement?) and Br = Brak (empty?). Given that our final land cover layer should be Impervious - 1, Low vegetation, grass or dirt - 2, Roof - 3, Trees - 4, Otherwise occupied - 0 (e.g., monument, water, railroad, road), we need to turn rolinnosc into something numeric. First, we need to save the PTZB file as something editable - I tend to use shapefiles out of habit, but it’s flexible as long as it’s a vector. Right-click and Export as something of your choosing. I convert it to 102173 while doing this, but this part is optional - you can always convert the final layer later.

Once we have an editable file, we can do a field calculation to convert this land cover field to integer - copy-paste this into the field calculator: if( "roslinnosc" = 'Dr', 4, if( "roslinnosc" = 'Tr', 2, if( "roslinnosc" = 'Bl', 1, if( "roslinnosc" = 'Br', 1, 0)))). Make sure you hit save so the Rasterizer can find your field. Rasterize this field such that no value is set to -9999 and the layer is aligned with the DTM or slope layer created above. In case this is the first one of these you’re reading, to align it during the rasterization, set the output extent to the Slope raster you created with the Calculate from Layer option in the dropdown for Output Extent.



Once you have this layer, you can run r.null to produce a fully sized base layer with zeroes everywhere there wasn’t a polygon in the PTZB file.



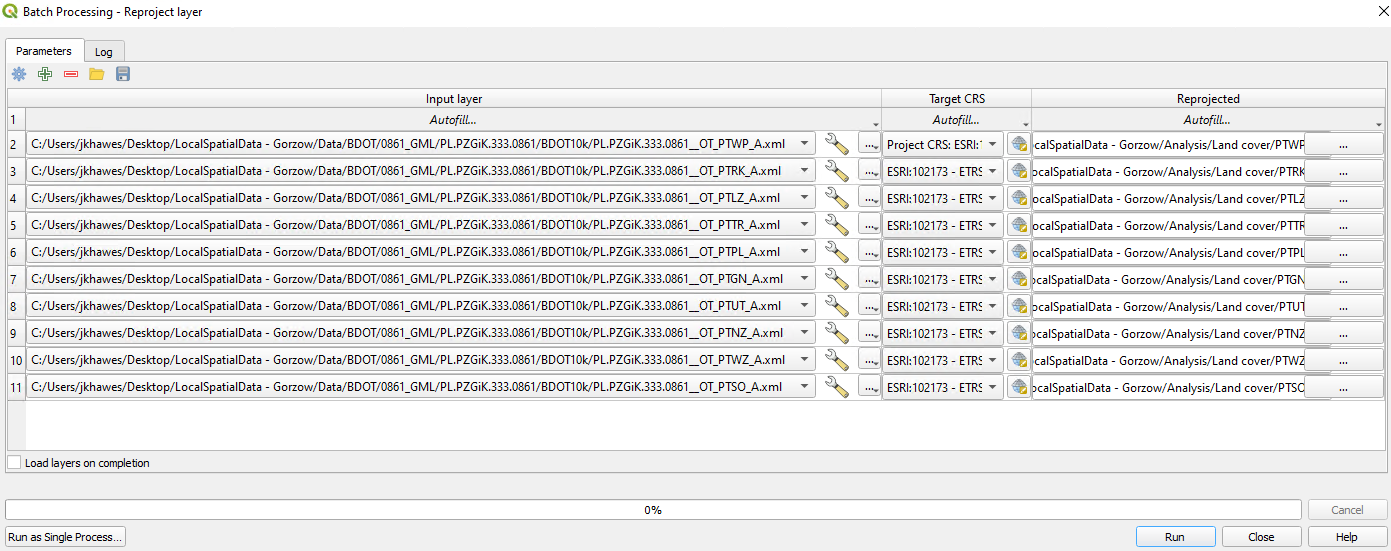
Next, we can overlay this with the actual key land cover layers - PTWP (water), PTRK (shrub - code as grass), PTLZ (forest), PTTR (grass and cultivated). The field calculations are fairly simple: the entirety of PTWP gets coded as 0, all of PTRK gets coded as 2, PTLZ gets coded as 4, and PTTR gets coded as 2. To do this, you convert those to an editable shapefile, then you run field calculator, and in the area where you would build the formula, you just put the number you want. You can’t automate the field calculator, but you can batch process the conversion to shapefile, which saves some time. To do this, use the Reproject Layer function instead of the Export As interface. Should all work the same and leave you with shapefiles of everything you want. You can also batch process the rasterization. So I recommend thinking of this whole lower section as a batch.

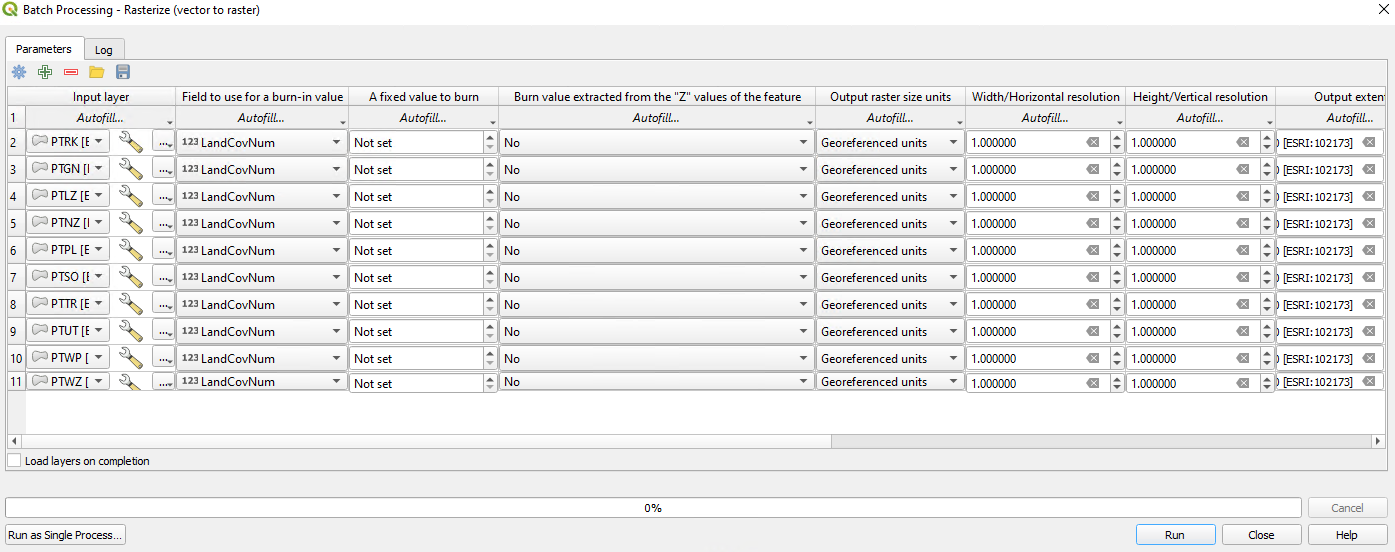
The entire equation for PPTR, for example, is just “2”. Rasterize each of these, then run r.null so that the rest of the space is zero. There should be no overlap, so we can just add these together. You can rasterize, run r.null, and add together now or wait - there are more non-overlapping layers to add if you just want to do them all at once with batch processing and less raster calculations.

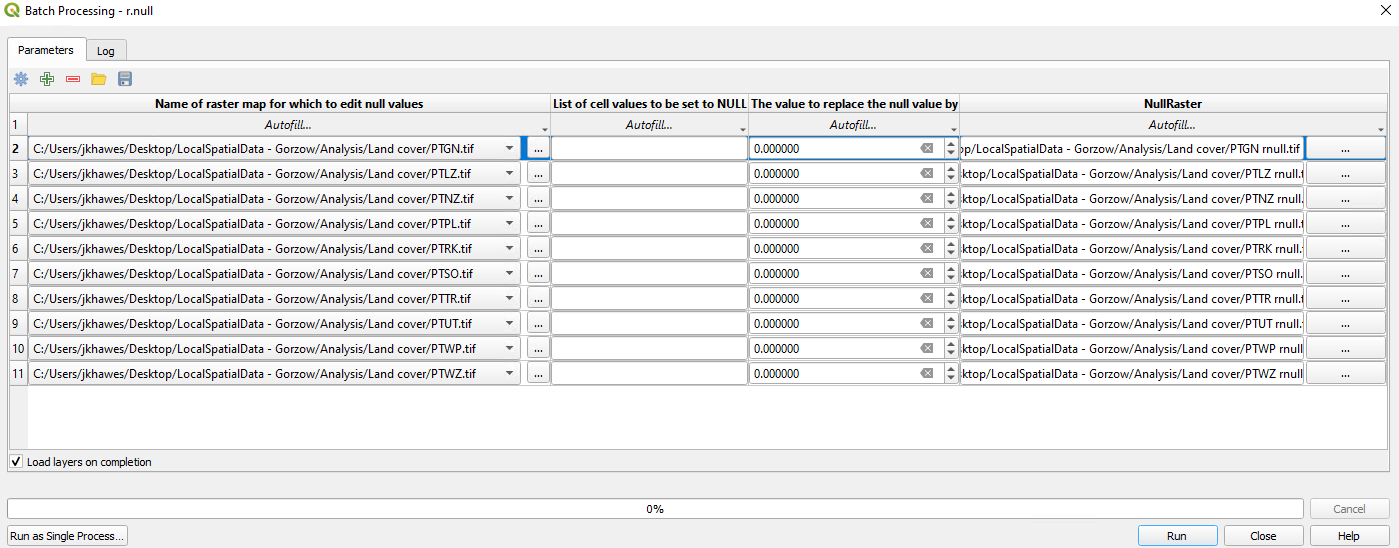
PTPL is almost all parking lots, so we can classify the whole thing as 1 and rasterize it. PTGN can be classified as dirt or low vegetation.

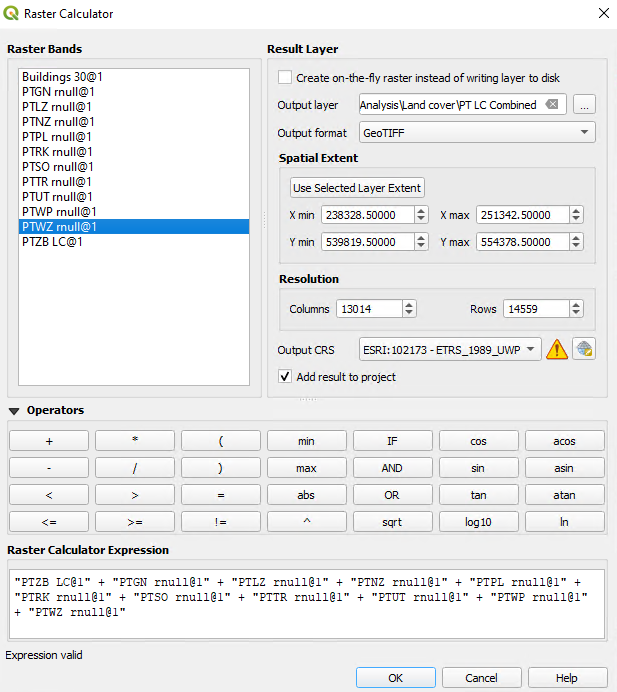
PTUT is the first one that our field calculation is more than just a number. Orchards (PTUT03) and forest nurseries (PTUT04) are trees, the rest are low vegetation: if(“x\_kod” = ‘PTUT03’, 4, if(“x\_kod” = ‘PTUT04’, 4, 2)).

PTNZ, PTWZ, and PTSO all get coded as otherwise occupied - 0. This is the end of the non-overlapping layers, so if you haven’t already rasterized, r.null’ed, and added these together, do that now. Make sure that you don’t have any values over 4, since that would indicate misalignment, then you’re good to go.





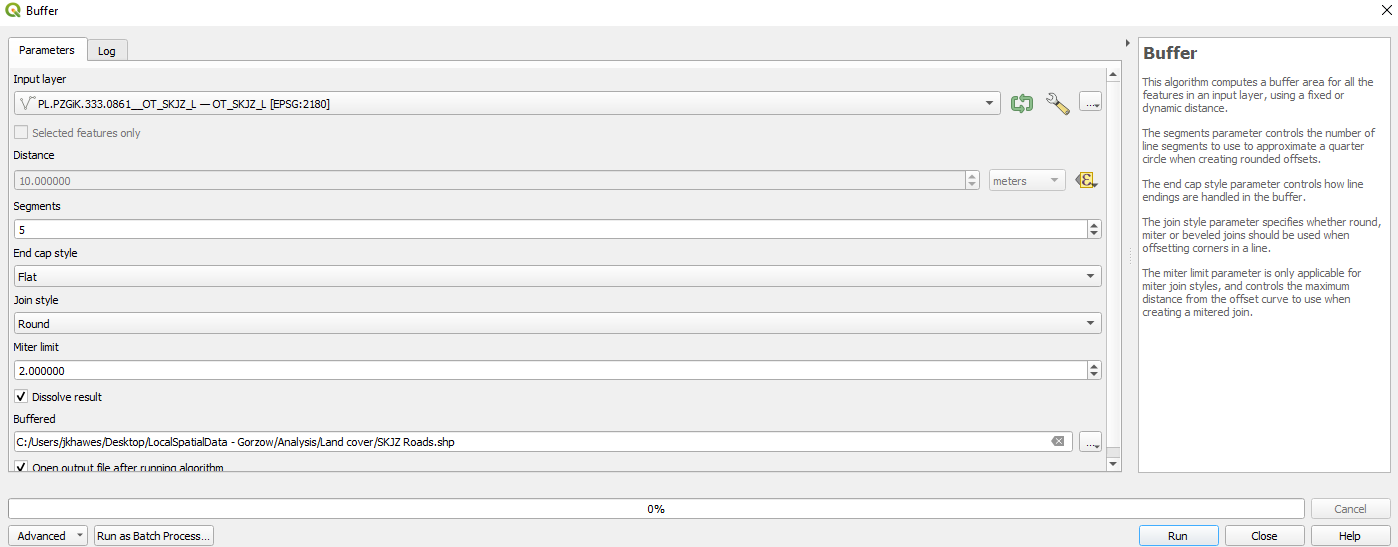




## Add in overlapping layers – roads, vegetation, and buildings from SKJZ, OIPR, and BUBD

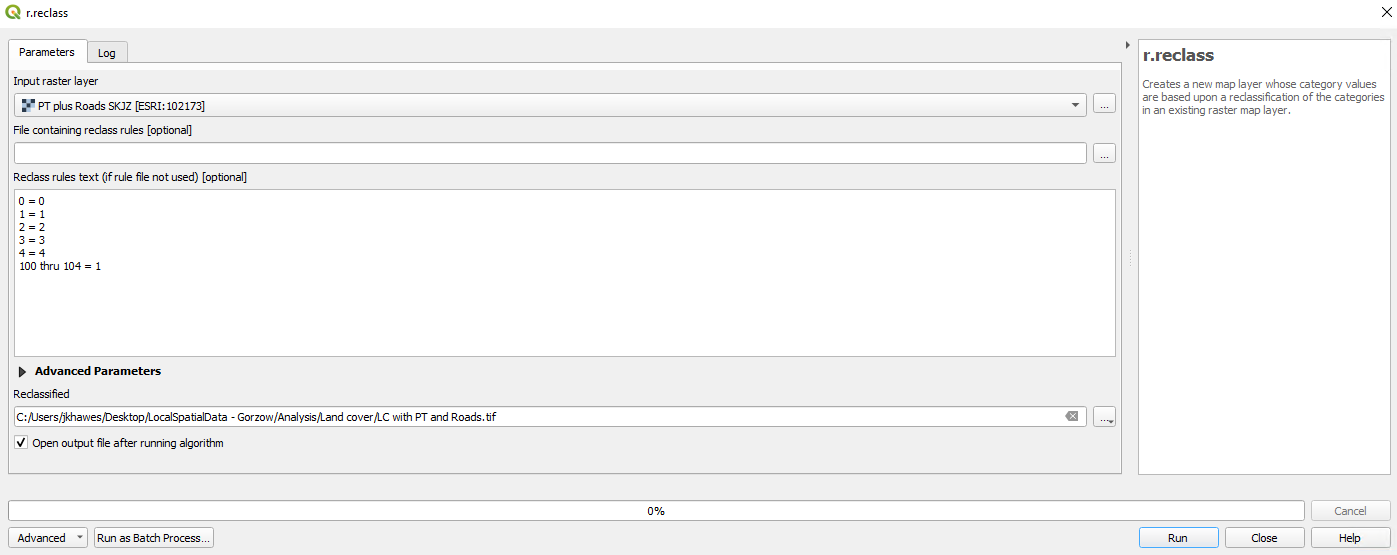
### Roads

The reason we’re out of non-overlapping layers is because we don’t use PTKM (road and rail) for the roads overlay. It is incomplete. Instead, we use the SKJZ polyline. To turn the polyline into a useful approximation of the road, we need to buffer (Vector >> Buffer) the line to the width of the road. Fortunately, there is a width parameter built into the dataset. So we can use two times the width parameter (szerNawierzchni in the codebook, szerNawier in some versions of QGIS that truncate column names) as the buffer. Use the Expression option under Distance to set it to 2 \* szerNawierzchni. See other settings below - including flat end caps and dissolve.



You also want to reproject this now so that it doesn’t have any chance of confusing the computer during raster algebra. Convert it to 102173 like all the rest. Once this is ready, rasterize it. I recommend burning in 100 so you have a simple raster calculation after you’re done. As usual, use an existing raster layer with the whole city to set the extent. You’ll need to run r.null again to make everything else zero, then add it all together. Finally, reclassify anything greater than or equal to 100 as 1 (impermeable) and the rest stays how it is:

100 thru 104 = 1  
\*=\*



Overall, this land cover layer is pretty good. It captures most of the larger areas with grass, and it represents large woodlands and paved areas well. It loses some private yards, but there isn’t much to be done about that short of a full remote sensing project, which just isn’t worth it for a relatively small improvement. We can enhance it all some by integrating larger vegetarian with some of the point layers.

### Add in trees and bushes from OIPR

The second-to-last step in polishing the land cover layer is to use the polyline and point files from OIPR to add in the trees and bushes that aren’t big enough patches to show up on the original classification scheme. Anything smaller than a small woodland - 1000m2 - is ignored in that original scheme, so lots of trees in frontyards, etc. are ignored so far. We’ll capture as many of those as we can in this step.

#### Take a random sample of each sub-classification and measure the width of the feature

In order to estimate the width of each feature in the OIPR dataset, we can take a random sample of each sub-class and calculate the average width. We can then use this width as our buffer diameter.

To take a random sample of each sub-class (x\_kod - the type of woody plant), we can use the built-in Random Extract Within Subsets tool. Since there are more than 10,000 OIPR features in total, I recommend taking a sample somewhere on the order of 5%, which should get you a nicely representative set from each sub-class.

Once you have your sample, just go through and add the width as an attribute in the attribute table. This is a case where being able to automatically zoom to the feature just by navigating the attribute table is quite useful (which also means that having two monitors for this is a big win). This is easily achieved with a little Python code, which is explained in [this StackOverflow](https://gis.stackexchange.com/questions/433516/automatically-zoom-to-feature-which-is-edited-in-attribute-table-in-qgis). Click on the layer, paste the Python code into a script, hit run (which brings up the attribute table), and run through the sample as quickly as you can. Here’s the code in case that question ever goes offline:

lyr = iface.activeLayer()

att\_tbl = iface.showAttributeTable(lyr)

tbl\_view = att\_tbl.findChildren(QTableView)[0]

sel\_mod = tbl\_view.selectionModel()

def row\_changed(current, previous):

feature\_id = current.model().data(current, QgsAttributeTableModel.FeatureIdRole)

iface.mapCanvas().zoomToFeatureIds(lyr, [feature\_id])

sel\_mod.currentRowChanged.connect(row\_changed)

Use the ruler tool in QGIS to measure the width of each feature and record that in a new field in the attribute table. It’s worth noting that this process is not perfect and requires a lot of interpretation by the coder. For example, some points are up to a few feet away from the nearest tree. Instead of coding these as zero, I simply measure the nearest tree (within 3-5 m) as long as it is not also marked independently (see below for photo notes). Shadows and season also cause a lot of variation - it can be helpful to use multiple layers, including Google, Bing, Esri, and orthoimagery if there is good, recent, well-aligned orthoimagery.

With respect to alignment - as an example, , the nearest tree on the left here can be safely associated with that point:

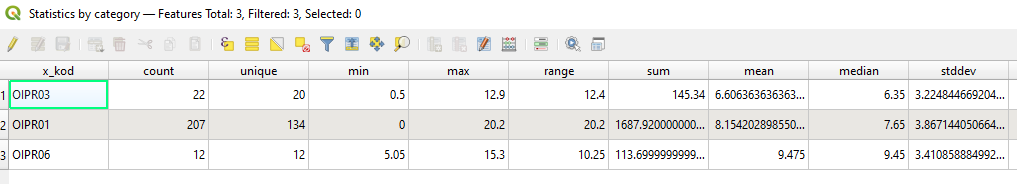


While there really is no tree here:



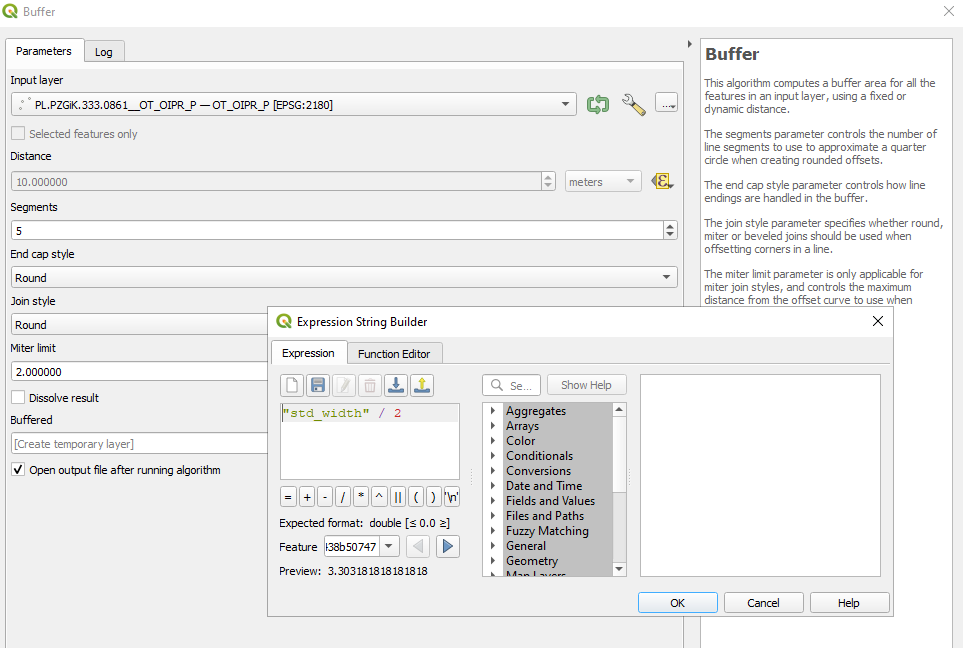
#### Use width findings to buffer lines and points, rasterize and add to land cover

Once you’re done, you can use statistics by categories to identify the average for each sub-class. When I did this, the results were as follows:



This makes the field calculation: if( "x\_kod" = 'OIPR01' , 8.154202898550, if( "x\_kod" = 'OIPR03' , 6.606363636363636, if( "x\_kod" = 'OIPR06' , 9.475, 0)))

Use a field calculator to insert this average back into the original (non-sample) dataset as a width feature - if(“x\_kod” = ‘OIPR01’, 5.2, etc.), then run the buffer tool. You’ll need the buffer expression to be half the diameter, unless you already accounted for this in your measurement or width input.



Before rasterizing, you’ll also need to give this new polygon layer the land cover codes to burn in. Similar to past overlays, the coding is as follows:

OIPR01 = 40, OIPR02 = 100, OIPR03 = 20, OIPR04 = 20, OIPR05 = 40, OIPR06 = 40, OIPR07 = 100, OIPR08 = 20, OIPR09 = 100, OIPR10 = 40, OIPR11 = 100, OIPR12 = 100, OIPR13 = 100

With the current file, since there are only three extant OIPR codes, this makes the field calculation:

CASE

WHEN "x\_kod" = 'OIPR01' THEN 40

WHEN "x\_kod" = 'OIPR03' THEN 20

WHEN "x\_kod" = 'OIPR06' THEN 40

END

Rasterize the layer with this new field burned in. Run r.null to set the rest of the map to zero. Add this to the existing land cover and set 0 = 0, 1 = 1, 2 = 2, 3 = 3, 4 = 4, 20-22 = 2, 23 = 3, 24 = 4, 40-44 = 4, 100-104 = 0:

20 thru 22 = 2

23 = 3

24 = 4

40 thru 44 = 4

100 thru 104 = 0

\* = \*

### Buildings

The last step in determining land cover in the city is to add in buildings. We already have a raster of this from the slope work. We use field calculator to give the whole thing a value - this time, though, it needs to be 30 instead of 3. This way, once we rasterize the file, run r.null (which, again, you should already have from earlier), and add it to the existing land cover data, we can just use a simple r.reclass to fix the classification:

30 thru 34 = 3  
\*=\*

# Land Use Layer Derivation This layer will describe the land use at a parcel level. These will be used primarily to sort the different types of gardens - e.g., it makes much more sense to assume an individual garden in a single-family backyard than a community garden. The final layer produced via this procedure will have the same codes as the land use layers in all the other cities, and, as usual, some information about equivalencies is available [here](https://docs.google.com/spreadsheets/d/176HUxNceFgc3QG77QbuSUSxrKWQ1HY6M/edit?usp=drive_link&ouid=102554365591872343075&rtpof=true&sd=true). I should note that these equivalencies are not quite as straightforward as some of the other cases, so it’s important to follow this procedure closely even if you understand the equivalencies in great detail.

Overview: We have PLUTO data that just need to be simplified and prepared for use in our coding scheme. We will also integrate a bit of additional information from other NYC data layers (adding in roads, sidewalks, parks, and parking lots).

To begin,we will import the following datasets:

1. BDOT - downloaded in Land Cover derivation
2. Plat map - [download](http://geoportal.wms.um.gorzow.pl/map/www/mapa.php?CFGF=wms&mylayers=+granice_miasta+ortofotomapa2019+)

All of these are updated fairly regularly, so if it’s been a while since you ran this analysis (> 6 months), it might be helpful to refresh the data being used.

## Overlay land use layers from BDOT

The land use layer follows much the same process as the land cover layer, but it’s more of a pain because the overlaps are more frequent. Easiest to just go one layer at a time.

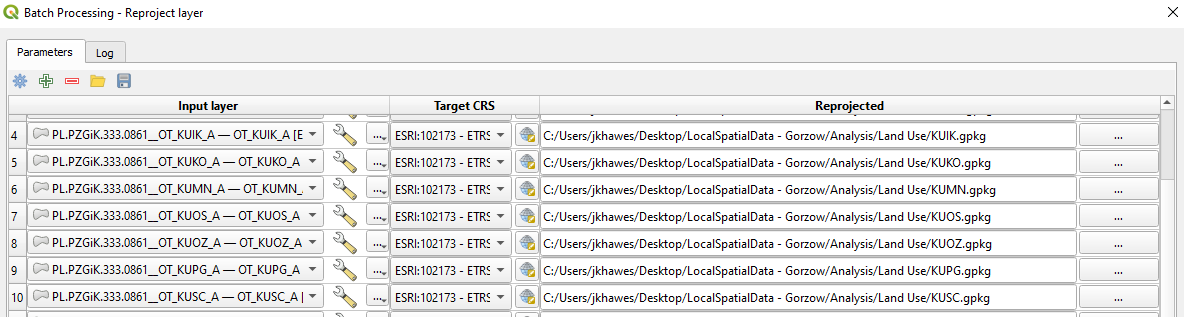
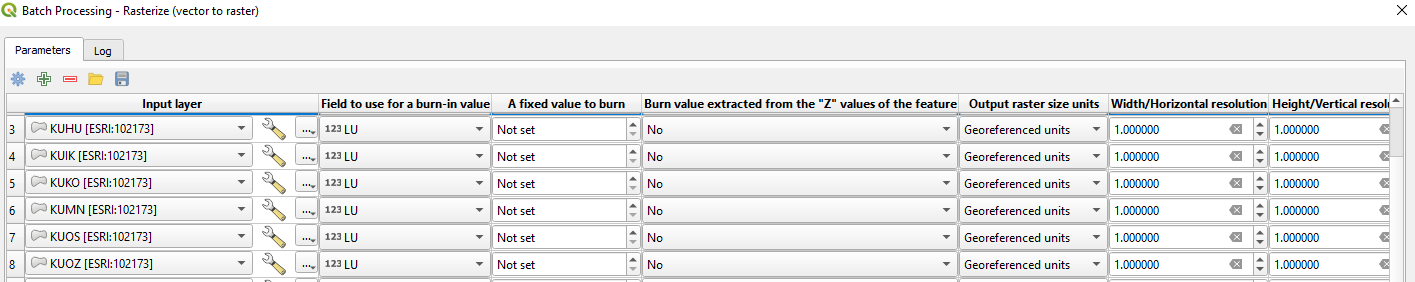
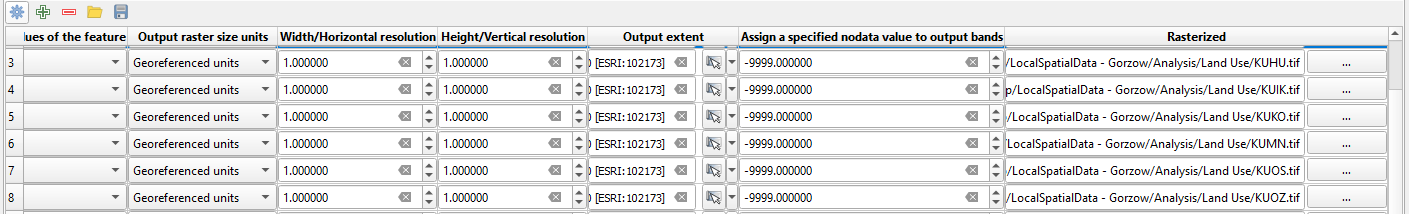
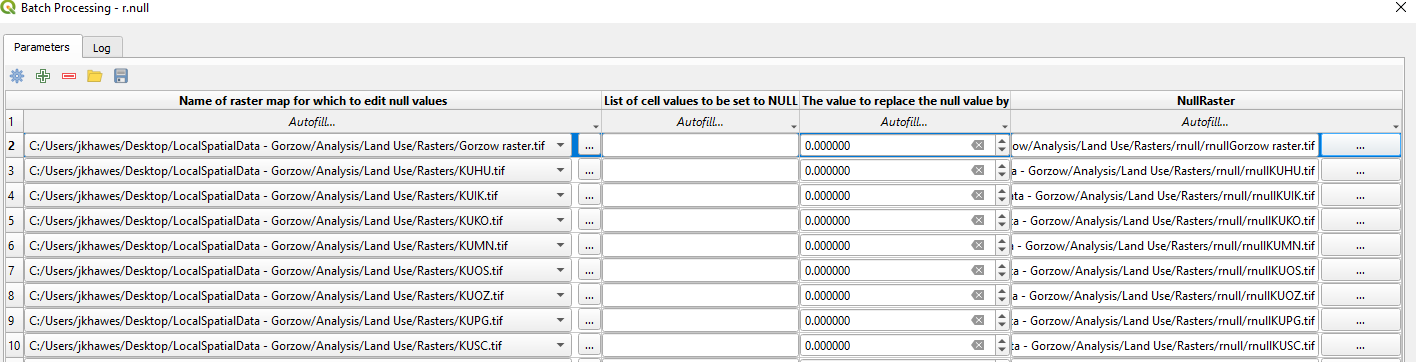
In every case, the logic flows like this:

1. Recode the layer for overlaying.
2. Rasterize and run r.null to fill everything else in with 0s.
   1. Any intermediary steps…
3. Add the new layer to the existing land use layer and reclassify.

So I’ll use that format for this section - starting with the layer name, then 1 for recoding, 2 for r.null, 3 for reclassifying.

Before we get into the one-at-a-time overlays, we can start with the basic empty raster of all unused. Theoretically, we can get this in a variety of ways. After all the layers we add below, the remaining gaps can be filled by PTLZ (trees), PTRK (shrub), and PTTR01 (grass). Since these aren’t really useful categories, we just end up coding them as insufficient information (71). Another alternative is just to use the whole city map as 71 and then overwrite this with other, more useful data as we go. I suggest the second one, just since it ensures you don’t end up with any gaps later on. Remember that sieve and slope and land cover are going to be around to make sure that you don’t have any weird little spots that pop up in gaps in between polygons. So, we begin with the Gorzow city map and go from there.

The actual process for this is:

1. Use field calculator to give these layers their LU codes all at once - I just find it easier to do all of that before using batch processing for rasterize and r.null. The code for these calculations is below.
2. You cannot rasterize with a calculated layer from an XML, so the next step is to save these as shapefiles or geopackages. I recommend using Reproject Layer in vector tools - even though it’s not absolutely necessary to have these layers in 102173, it unifies everything and allows you to use batch processing to save the new files instead of right-clicking and exporting each one individually. This really comes down to personal preference.   
   
3. Rasterize and run r.null on all of the layers - fill with 0s. Clip these back to the boundary with the original Gorzow border shapefile.  
     
     
   
4. Next, add the layers in the order below. Use r.reclassify with the included code between each addition.

So the layers, in order:

Layer: Gorzow

1. All = 71.
2. Rasterize, r.null, and clip.
3. No recode - this is the base.

Layer: KUMN -> yields LU1

1. if( "x\_kod" = 'KUMN01', 10, 0 )
2. Rasterize, r.null, and clip.
3. 81 = 10   
   \*=\*

Layer: PTTR -> yields LU2

1. if( "x\_kod" = 'PTTR02', 90, 0 )
2. Rasterize, r.null, and clip.
3. 100 = 90  
   161 = 90  
   \*=\*

Layer PTUT -> yields LU3

1. if( "x\_kod" = 'PTUT01' , 91, if( "x\_kod" = 'PTUT02' , 90, if( "x\_kod" = 'PTUT03' , 90, if( "x\_kod" = 'PTUT04' , 90, if( "x\_kod" = 'PTUT05' , 90, 0)))))
2. Rasterize, r.null, and clip.
3. 100 = 90  
   101 = 91  
   161 = 90  
   162 = 91  
   180 = 90  
   181 = 91  
   \*=\*

Layers TCON, TCRZ - if you are working in a place with TCPN or TCPK, include those here too (Gorzow doesn’t have any) -> yields LU4

1. Recode all as 3200.
2. Rasterize, r.null, and clip.
3. Add to ongoing land use raster. Anything greater than or equal to 3200 gets recoded to 32, else remains the same.   
   3200 thru 19999 = 32  
   \*=\*

\*\*\* I want to take a moment to make a special note - from this point on, you could just use the same r.reclass code every time - multiply everything by 100 and run the conversion every time. I do not do this, mostly because there are so many layers that it’s not unlikely I’ll click on the wrong one in an addition step. Only reclassifying the things we care about add a little extra security that I’m not adding the wrong thing, because if I do the reclassified layer will still have values in the thousands.

Layer PTNZ -> yields LU5

1. if( "x\_kod" = 'PTNZ01' , 2900, if( "x\_kod" = 'PTNZ02' , 2300, 0))
2. Rasterize, r.null, and clip.
3. 2300 thru 2399 = 23  
   2900 thru 2999 = 29   
   \*=\*

Layer PTWP -> yields LU6

1. All 8300
2. Rasterize, r.null, and clip.
3. 8300 thru 8399 = 83   
   \*=\*

Layer PTKM -> yields LU7

1. if( "x\_kod" = 'PTKM01' , 8000, if( "x\_kod" = 'PTKM02' , 8200, if( "x\_kod" = 'PTKM03' , 8000, if( "x\_kod" = 'PTKM04' , 4200, 0))))
2. Rasterize, r.null, and clip.
3. 4200 thru 4299 = 42  
   8000 thru 8099 = 80  
   8200 thru 8299 = 82   
   \*=\*

Layer PTZB -> yields LU8

1. if( "x\_kod" = 'PTZB01' , 1200, if( "x\_kod" = 'PTZB02' , 1100, if( "x\_kod" = 'PTZB03' , 2300, if( "x\_kod" = 'PTZB04' , 2200, if( "x\_kod" = 'PTZB05' , 0, 0)))))
2. Rasterize, r.null, and clip.
3. 1100 thru 1199 = 11  
   1200 thru 1299 = 12  
   2200 thru 2299 = 22  
   2300 thru 2399 = 23  
   \*=\*

Layer PTGN -> yields LU9

1. All 7000.
2. Rasterize, r.null, and clip.
3. 7000 thru 7099 = 70  
   \*=\*

Layers PTPL, PTWZ, and PTSO -> yields LU10

1. PTPL01 = 3100
2. if( "x\_kod" = 'PTSO01' , 4300, if( "x\_kod" = 'PTSO02' , 4300, 0))
3. if( "x\_kod" = 'PTWZ01' , 2500, if( "x\_kod" = 'PTWZ02' , 4300, 0))
4. Rasterize and r.null.
5. 2500 thru 2599 = 25  
   3100 thru 3199 = 31  
   4300 thru 4399 = 43  
   \*=\*

Layer KUPG -> yields LU11

1. if( "x\_kod" = 'KUPG01' , 4200, if( "x\_kod" = 'KUPG02' , 4200, if( "x\_kod" = 'KUPG03' , 4200, if( "x\_kod" = 'KUPG04' , 9000, if( "x\_kod" = 'KUPG05' , 2300, if( "x\_kod" = 'KUPG06' , 2500, if( "x\_kod" = 'KUPG07' , 3400, if( "x\_kod" = 'KUPG08' , 3400, if( "x\_kod" = 'KUPG09' , 3400, if( "x\_kod" = 'KUPG10' , 2300, if( "x\_kod" = 'KUPG11' , 4300, if( "x\_kod" = 'KUPG12' , 8300, if( "x\_kod" = 'KUPG13' , 2300, if( "x\_kod" = 'KUPG14' , 2300, if( "x\_kod" = 'KUPG15' , 2300, if( "x\_kod" = 'KUPG16' , 2200, 0))))))))))))))))
2. Rasterize, r.null, and clip.
3. 2200 thru 2299 = 22  
   2300 thru 2399 = 23  
   2500 thru 2599 = 25  
   3400 thru 3499 = 34  
   4200 thru 4299 = 42  
   4300 thru 4399 = 43  
   8300 thru 8399 = 83  
   9000 thru 9099 = 90  
   \*=\*

Layer KUSC -> yields LU12

1. if( "x\_kod" = 'KUSC01' , 3200, if( "x\_kod" = 'KUSC02' , 2400, 0))
2. Rasterize, r.null, and clip.
3. 2400 thru 2499 = 24  
   3200 thru 3299 = 32  
   \*=\*

Layers KUOS and KUOZ -> yields LU13

1. All KUOS values = 3500
2. All KUOZ values = 3400
3. Rasterize, r.null, and clip.
4. 3400 thru 3499 = 34  
   3500 thru 3599 = 35  
   \*=\*

Layers KUKO and KUHO-> yields LU14

1. All KUHO values = 1400
2. KUKO01 = 4200, KUKO02 = 4200, KUKO03 = 4200, KUKO04 = 4100, KUKO05 = 8300, KUKO06 = 4200, KUKO07 = 4200, KUKO08= 4200, KUKO09 = 8200, KUKO10 = 4200
3. Rasterize, r.null, and clip.
4. 1400 thru 1499 = 14  
   4100 thru 4199 = 41  
   4200 thru 4299 = 42  
   8200 thru 8299 = 82  
   8300 thru 8399 = 83  
   \*=\*

Layer KUSK -> yields LU15

1. if( "x\_kod" = 'KUSK01' , 3200, if( "x\_kod" = 'KUSK02' , 3200, if( "x\_kod" = 'KUSK03' , 2600, if( "x\_kod" = 'KUSK04' , 4100, if( "x\_kod" = 'KUSK05' , 1400, 0)))))
2. Rasterize, r.null, and clip.
3. 1400 thru 1499 = 15  
   2600 thru 2699 = 26  
   3200 thru 3299 = 32  
   4100 thru 4199 = 41  
   \*=\*

Layer KUHU-> yields LU16

1. All 2200
2. Rasterize, r.null, and clip.
3. 2200 thru 2299 = 22  
   \*=\*

Layer KUIK-> yields LU\_final

1. All 3900
2. Rasterize, r.null, and clip.
3. 3900 thru 3999 = 39  
   \*=

There should be no need to reclassify or readjust this layer after you finish with the series of overlays. Land use is ready to go.

# Sunlight Availability

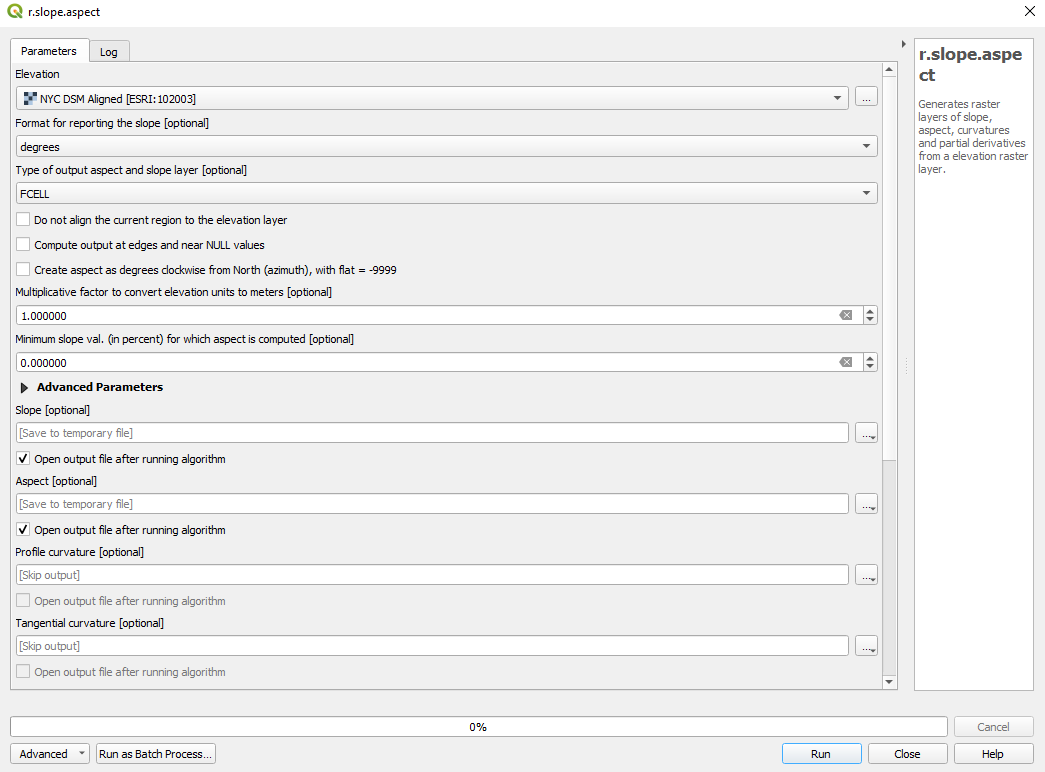
The final layer we will derive is a sunlight availability layer. This will take advantage of the r.sun package in GRASS, which takes the DSM and converts it to solar irradiance. This one is fairly complex, so I derive my process from an [example done in Canada](https://www.sciencedirect.com/science/article/pii/S0038092X10000812?via%3Dihub#fn15). Based on that paper, we will need a few inputs to make this work, including:

| Canadian layer/example | Gorzow layer/notes | Source link |
| --- | --- | --- |
| Digital Elevation Model (DEM) | DSM - include shading from buildings and trees |  |
| Slope/inclination | Derived from DSM |  |
| Aspect/orientation | Derived from DSM |  |
| Latitude | Not necessary if we use a proper projection (102003) |  |
| Albedo: the ratio of diffusely reflected radiation on a surface to its incident radiation. | Albedo can probably be calculated for each city with this function, or we can use urban averages. For i.albedo, just need landsat imagery: https://grass.osgeo.org/grass78/manuals/i.albedo.html |  |
| Mean days and corresponding angular position of the sun. | Can use the same mean days if we do want to do the calculation for every month. “ Table 1.6.1 in Duffie and Beckman (1991) readily provides the day of month, day of year and δ (sun declination) values to input into the simulation -- J.A. Duffie, W.A. Beckman, Solar Engineering of Thermal processes (second ed.), John Wiley & Sons (1991)” |  |
| Linke turbidity: a convenient approximation to model the atmospheric absorption and scattering of the solar radiation under clear skies. | If all we want is very high level stuff, we can get that from the same place the example paper did. Resolution is about the scale of NYC. Have three different raster cells for whole city, all the same value. Able to make a raster with the resolution of our DEMs and DSMs by downsampling. | <http://www.soda-pro.com/help/general-knowledge/linke-turbidity-factor> |
| Ground-measured values of global horizontal irradiation (GHI). | Available from NASA SSE POWER program - GHI is the first value (ALLSKY\_SFC\_SW\_DWN CERES SYN1deg All Sky Surface Shortwave Downward Irradiance (kW-hr/m^2/day)) while GHI under Clear-Sky conditions is the second value (CLRSKY\_SFC\_SW\_DWN CERES SYN1deg Clear Sky Surface Shortwave Downward Irradiance (kW-hr/m^2/day)) | Available at a 1x1 degree resolution. It claims to be ½ by ½ but doesn’t seem to output that for 2019 at least. – <https://power.larc.nasa.gov/data-access-viewer/> |
| Clear sky index Kc: “Ratio of the global horizontal irradiance to the global horizontal irradiance under clear sky conditions. It is important not to confuse and hence misuse this definition with those for insolation clearness index and clear sky insolation clearness index.” | Available in the POWER suite of indicators as ALLSKY\_KT. We do not use the normalized value - this transforms the Kc with the latitude - Kc = shortwave direct horizontal (GHI) / shortwave direct top-of atmos -- I haven’t figured out the use for the normalized parameter yet. Maybe comparing different locations? See [here](https://www.star.nesdis.noaa.gov/smcd/emb/radiation/solar_resource_definitions.php) for simple definitions. See [here](https://www.star.nesdis.noaa.gov/smcd/emb/radiation/documents/SRDB_1.0_Parameter_Definitions.pdf) for other details.  Definitions (rather unhelpful, except it explicitly mentions GHI): <https://power.larc.nasa.gov/#resources> | The regional data access panel at the website above allows “NetCDF” export, which can be imported as a raster in Q: <https://ereefs.aims.gov.au/ereefs-aims/help/how-to-open-a-NetCDF-file-with-ArcMap-and-QGIS>  So basically we turn this into a raster of the appropriate resolution and multiply to get a final value from r.sun. |

With this set of inputs, we should be able to calculate the shading effects in essentially any city around the world. The DSM is the hardest thing to find, and we should be able to simulate this with building height data, which is more often available.

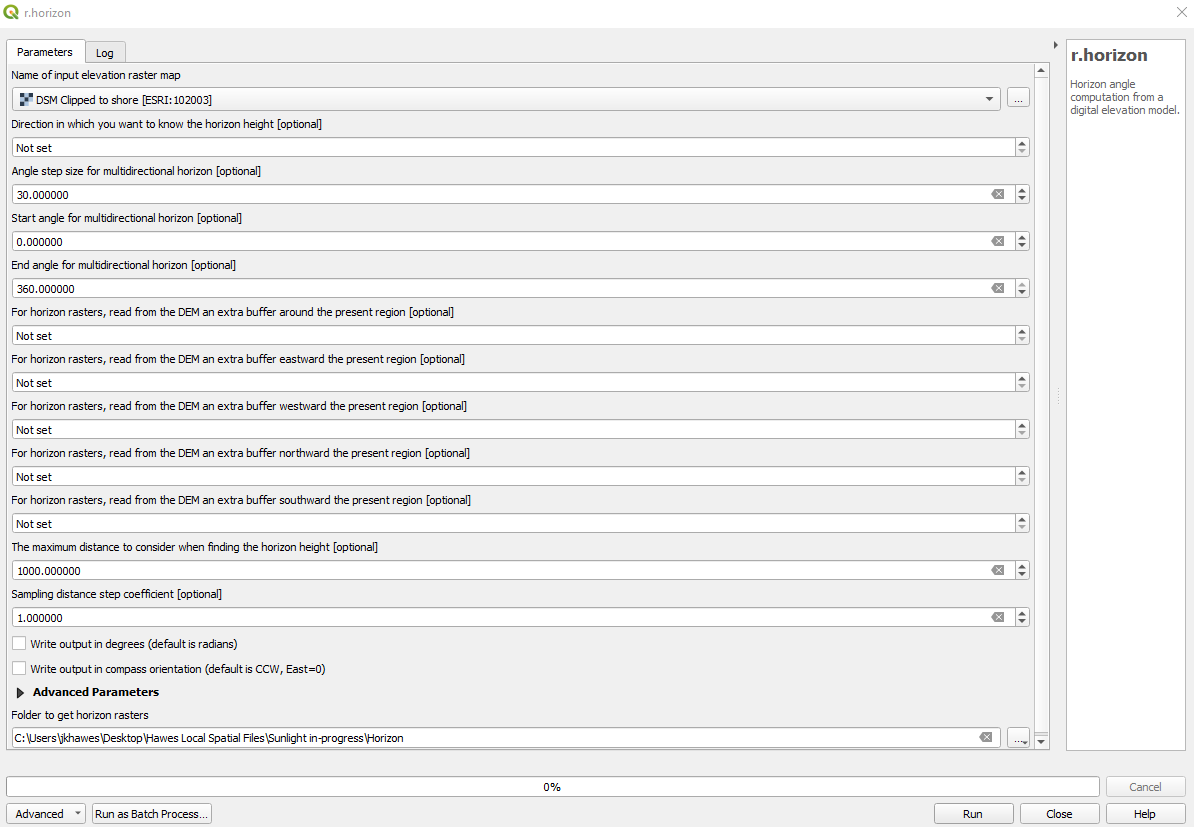
## r.slope.aspect

We need to produce maps of the slope and the aspect based on our DSM. We could use the DSM slope map we already have, but it’s just as easy to just run it all within GRASS to make sure everything is formatted the way r.sun wants it to be. This command is fairly straightforward, only a couple things need to be customized. First, we need to uncheck the box that asks about aligning with the elevation region. We do want to align all of our calculations with that region. Second, we want to suppress the outputs other than slope and aspect. No reason to spend time calculating things we won’t use.

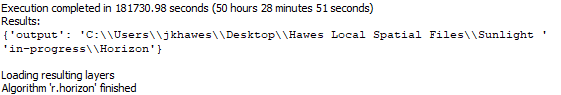


## Optional: r.horizon

If you are planning to use GRASS GIS more directly via commands, it can be very helpful to run r.horizon to determine the horizon height at all locations in the city. Unfortunately, if you plan to run the r.sun suite via QGIS, the interface does not play nice with loading an entire directory, which is the required format of the r.horizon output and the r.sun input.

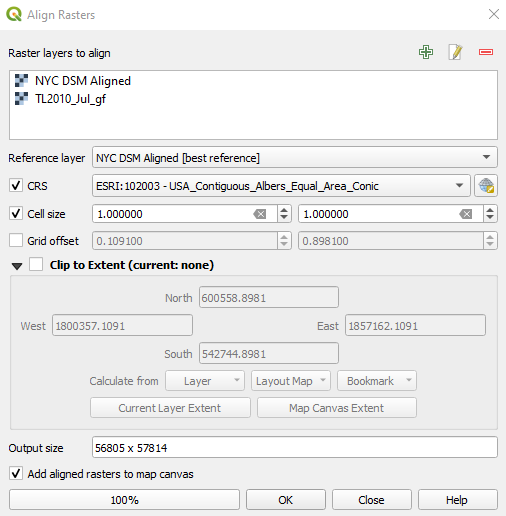


This will take a very long time to run - for NYC, it was on the order of 50 hours on the Beast at 1m resolution. In fact, the last time I ran this, it ran for 50 hours and only saved 120-360, so you may have to run it twice to convince it to save everything correctly.



## Aligning Linke

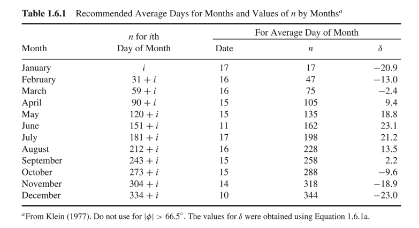
The last step before running the final command is to align the Linke values with the files we’ll be using. This will also clip the raster and will probably take 10-15 minutes.



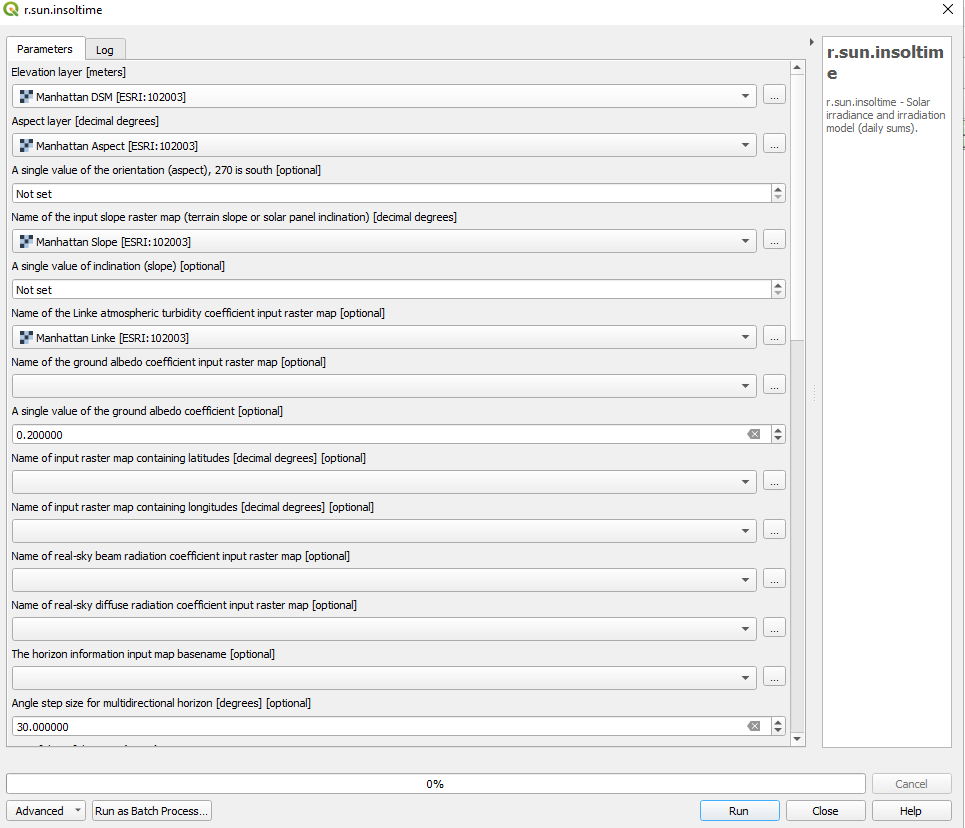
Once the Linke values have been converted to a 1x1m raster, we should be able to clip this to the borough boundaries, as we’ve done with the others.

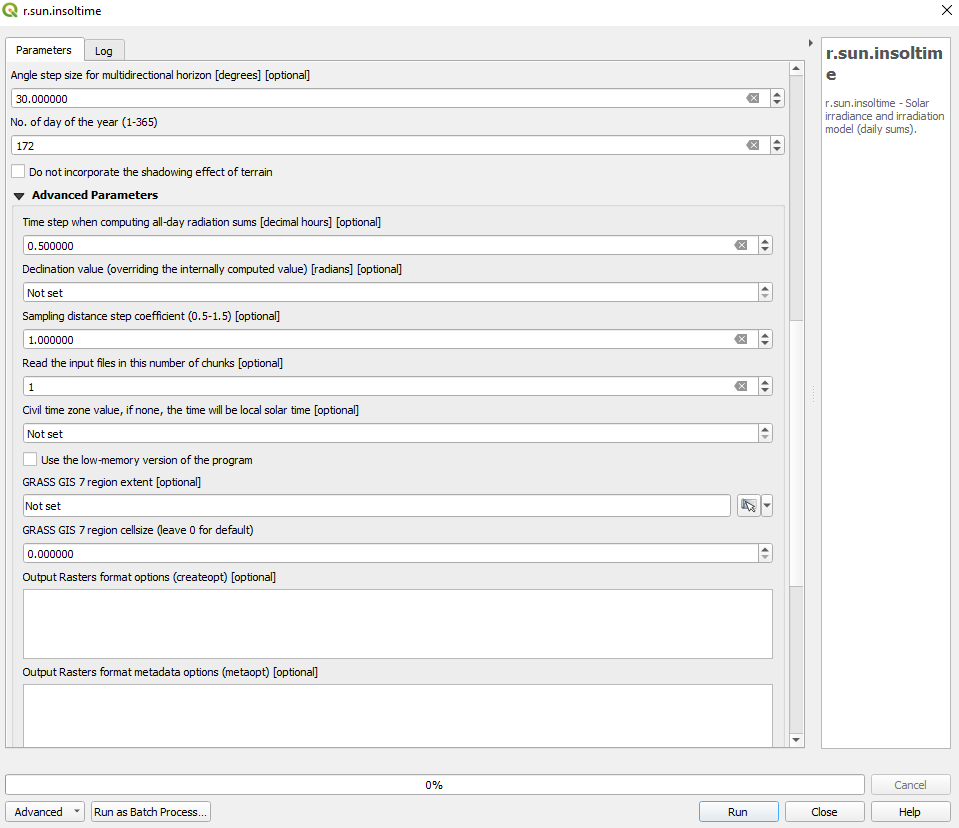
## r.sun.insoltime

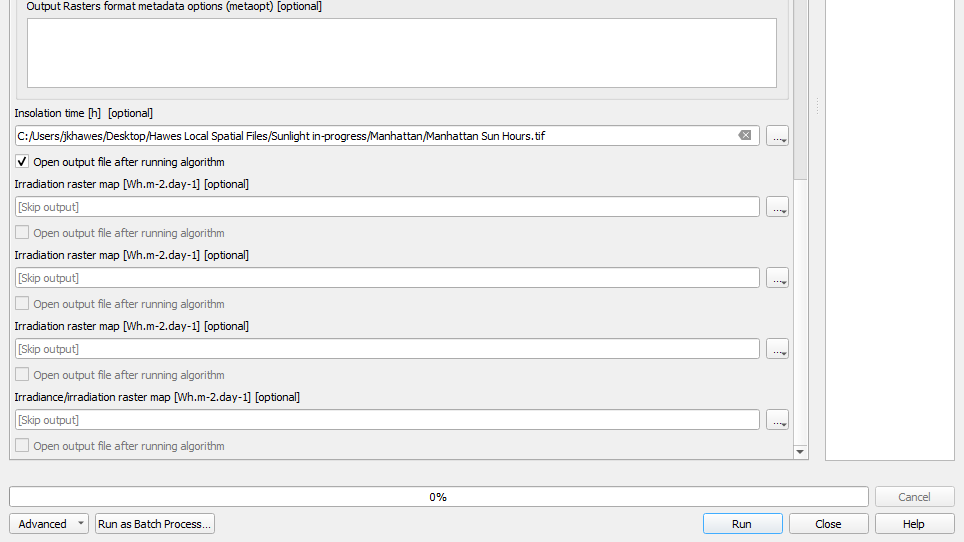
If we are strictly interested in the number of hours of sunlight, then we can simply ignore the more complex aspects like levels of radiation from the NASA data. We can retrieve average days from the book cited in the Canadian paper:



This means that the inputs look like this:





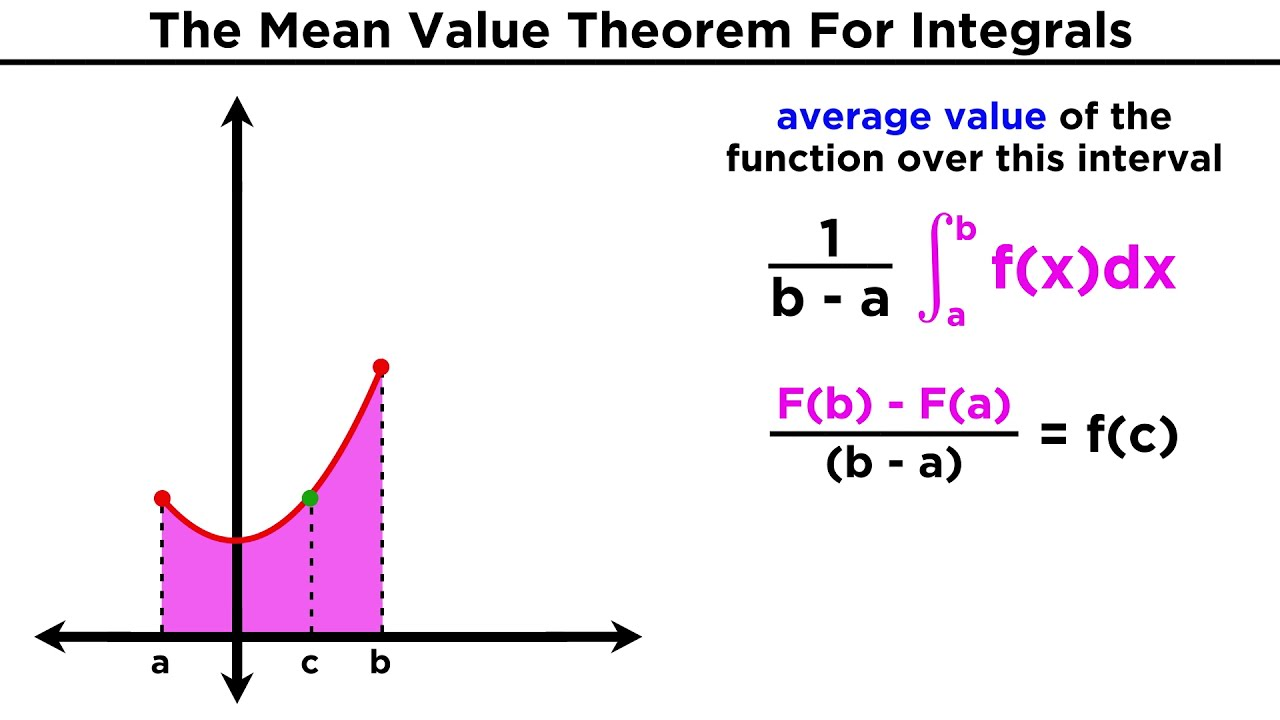


These commands run slowly, but stubbornly. I ran it on Gorzow and it took about 7 hours for each time point. Since we run it for April, July, and October before taking the growing season average, it’s about a day’s worth of simulation.

## Merge results and calculate average

First, we obviously need to recombine these tiles. This can be run as a batch process and should take several hours to run.

Once we have full sunhours layers for each month, we can take the average value of the function over the desired interval (the time between April average and October average). We can either try to fit and upside down parabola or we can assume two piecewise linear functions. The second produces more reproducible math between locations and on each cell, so we live with that simplification. The rest of the calculus in this whole process has been hidden in algorithms in QGIS, but you can actually follow along as we derive this particular equation. The mean value theorem says that we can calculate the average value of a function over any particular interval on which it is continuous, which our piecewise function is. For piecewise functions, we calculate the mean value theorem of the constituent pieces and then take a weighted mean.



We derive our equation with the standard y = mx+b, assuming that slope is linear:

For April to July, this is:

For July to October, this is:

So once we conduct the integration, the y goes away and the x gets filled in, but that still leaves us with b1 and b2. So before we can jump to the mean value theorem, we have to calculate the value of b1 and b2. We can do this by simply plugging in values we have already:

If we plug July into April to July, we have:

Solving, we end up with:

If we plug July into July to October, we have:

Solving, we end up with:

We can run both of these as raster calculations and end up with b1 and b2 as rasters. Now we can go ahead and integrate.

This is obviously a more complicated bit of math. For April to July, we end up with:

Following this same math for July to October, we end up with:

At the end, this turns out to be five raster calculations. We calculate and first with separate raster calculations, then we can directly calculate and .

The final calculation is just one final raster calculation - .

For b1, the raster calculation looks like this: "July Sunhours@1" - ( 198\* ( ( "July Sunhours@1" - "April Sunhours@1" ) / 93 ) )

For b2, the calculation looks like this: "July Sunhours@1" - ( 198 \* ( ( "October Sunhours@1" - "July Sunhours@1" ) / 90 ) )

For MeanValue1, the calculation looks like this: ( 28179 / 17298 ) \* ( "July Sunhours@1" - "April Sunhours@1" ) + "b1@1"

For MeanValue2, the calculation looks like this: ( 43740 / 16200 ) \* ( "October Sunhours@1" - "July Sunhours@1" ) + "b2@1"

For MeanValueOverall, the calculation looks like this: ( ( 93 \* "MV1@1" ) + ( 90 \* "MV2@1" ) ) / 183