# 2.8.1 点云数据处理

点云（point cloud）是使用三维扫描仪获取的资料，当然设计的三维模型也可以转换为点云数据。其中三维对象以点的形式记录，每个点为一个三维坐标，同时可能包含颜色信息（RGB），或物体反射面的强度（intensity）。强度信息是激光扫描仪接受装置采集到的回波强度，与目标的表面材质、粗糙度、入射角方向以及仪器的发射能量，激光波长有关。点云数据格式比较丰富，常用的包括.xyz(.xyzn，.xyzrgb)，.las，.ply，.pcd，.pts等，也包括一些关联格式的存储类型，例如基于numpy存储的array数组.numpy(.npu)，基于matlab格式存储的.matlab数组格式，当然也有基于文本存储的.txt文件。注意虽然有些存储类型后缀名不同，但实际上，数据格式可能相同。在地理空间数据中，常使用.las格式的数据。LAS（LASer）格式是由美国摄影测量和遥感协会（American Society for Photogrammetry and Remote Sensing，ASPRS）制定的激光雷达点云数据的交换和归档文件格式，被认为是激光雷达数据的行业标准。LAS格式点云数据包括多个版本，最近的为LAS 1.4（2011.11.14），不同的版本点云数据包括的信息也许不同，需要注意这点。LAS通常包括由整数值标识的分类信息（LAS1.1及之后的版本），其1.1-1.4LAS类别代码如下：

| 分类值/classification value | 类别 |
| --- | --- |
| 0 | 不被用于分类/Never classified |
| 1 | 未被定义/unassigned |
| 2 | 地面/ground |
| 3 | 低矮树木/low vegetation |
| 4 | 中等树木/medium vegetation |
| 5 | 高的树木/high vegetation |
| 6 | 建筑/building |
| 7 | 低的点/low point |
| 8 | 保留/reserved |
| 9 | 水体/water |
| 10 | 铁路/rail |
| 11 | 道路表面/road surface |
| 12 | 保留/reserved |
| 13 | 金属丝防护（屏蔽）/wire-guard(shield) |
| 14 | 导线（相）/wire-conductor(phase) |
| 15 | 输电杆塔/transmission tower |
| 16 | 电线连接器（绝缘子）/wire-structure connector(insulator) |
| 17 | 桥面/bridge deck |
| 18 | 高噪音/high noise |
| 19-63 | 保留/reserved |
| 64-255 | 用户定义/user definable |

处理点云数据的python库比较多，常用的包括[PDAL](https://pdal.io/)，[PCL](https://pointclouds.org/)，[open3D](http://www.open3d.org/docs/release/introduction.html)等。其中PDAL可以处理.las格式数据，读取后也可以存储为其它格式数据，使用其它库的功能处理也未尝不可。

此次实验数据为伊利诺斯州草原地质调查研究所（Illinois state geological survey - prairie research institute），发布的伊利诺伊州[.las格式的激光雷达数据](https://www.arcgis.com/apps/webappviewer/index.html?id=44eb65c92c944f3e8b231eb1e2814f4d)。研究的目标区域为芝加哥城及其周边，因为分辨率为1m，研究区域部分数据量高达1.4T，其中每一单元（tile）基本为，大约1G左右，最小的也有几百M。对于普通的计算机配置，处理大数据，通常要判断内存所能支持的容量（很多程序在处理数据时，可以不将其全部读入内存，例如h5py格式数据的批量写入和批量读取；rasterio库提供由windows功能，可以分区读取单独的栅格文件）；及CPU的计算速度，分批的处理可以避免因为处理中断造成全部数据丢失。在阐述点云数据处理时，并不处理芝加哥城所有区域数据，仅以IIT（Illinois Institute of Technology）校园为核心，一定范围内的数据处理为例。下载的点云数据包括的单元编号（文件）有(总共73个单元文件，计26.2GB)：

  | | | | ||||||

|—|—|—|—|—|||||| |LAS\_16758900.las|LAS\_17008900.las|LAS\_17258900.las|LAS\_17508900.las|LAS\_17758900.las|LAS\_18008900.las|LAS\_18258900.las|||| |LAS\_16758875.las|LAS\_17008875.las|LAS\_17258875.las|LAS\_17508875.las|LAS\_17758875.las|LAS\_18008875.las|LAS\_18258875.las|||| |LAS\_16758850.las|LAS\_17008850.las|LAS\_17258850.las|LAS\_17508850.las|LAS\_17758850.las|LAS\_18008850.las|LAS\_18258850.las|||| |LAS\_16758825.las|LAS\_17008825.las|LAS\_17258825.las|LAS\_17508825.las|LAS\_17758825.las|LAS\_18008825.las|LAS\_18258825.las|||| |LAS\_16758800.las|LAS\_17008800.las|LAS\_17258800.las|LAS\_17508800.las|LAS\_17758800.las|LAS\_18008800.las|LAS\_18258800.las|LAS\_18508800.las||| |LAS\_16758775.las|LAS\_17008775.las|LAS\_17258775.las|LAS\_17508775.las|LAS\_17758775.las|LAS\_18008775.las|LAS\_18258775.las|LAS\_18508775.las||| |LAS\_16758750.las|LAS\_17008750.las|LAS\_17258750.las|LAS\_17508750.las|LAS\_17758750.las|LAS\_18008750.las|LAS\_18258750.las|LAS\_18508750.las|LAS\_18758750.las| |LAS\_16758725.las|LAS\_17008725.las|LAS\_17258725.las|LAS\_17508725.las|LAS\_17758725.las|LAS\_18008725.las|LAS\_18258725.las|LAS\_18508725.las|LAS\_18758725.las|LAS\_19008725.las| |LAS\_16758700.las|LAS\_17008700.las|LAS\_17258700.las|LAS\_17508700.las|LAS\_17758700.las|LAS\_18008700.las|LAS\_18258700.las|LAS\_18508700.las|LAS\_18758700.las|LAS\_19008700.las|

### 2.8.1.1 点云数据处理（.las）

#### 1）查看点云数据信息

* PDAL的主要参数配置：（具体可以查看PDAL官网，或者’PDAL:Point cloud Data Abstraction Library’ 手册）

1. [Dimensions](https://pdal.io/dimensions.html)，维度，该参数给出了可能存储的不同信息，可以基于维度配置“type”类型，例如维度配置为”dimension”: “X”，可以配置”type”: “filters.sort”，即依据给出的维度，排序返回的点云。常用的包括’Classification’，分类数据；‘Density’，点密度估计；‘GpsTime’，获取该点的GPS时间；‘Intensity’，物体反射面的强度；X,Y,Z，坐标。下述代码pipeline.arrays返回的列表数组中，包含有dtype=[('X', '<f8'), ('Y', '<f8'), ('Z', '<f8'), ('Intensity', '<u2'), ('ReturnNumber', 'u1'), ('NumberOfReturns', 'u1'), ('ScanDirectionFlag', 'u1'), ('EdgeOfFlightLine', 'u1'), ('Classification', 'u1'), ('ScanAngleRank', '<f4'), ('UserData', 'u1'), ('PointSourceId', '<u2'), ('GpsTime', '<f8'), ('ScanChannel', 'u1'), ('ClassFlags', 'u1')])]，可以明确.las点云包括哪些维度。
2. [Filters](https://pdal.io/stages/filters.html)，过滤器，给定操作数据的方式，可以删除、修改、重组数据流。有些过滤器需要在对应的维度上实现，例如在XYZ坐标上实现重投影等。常用的过滤器有：create部分：filters.approximatecoplanar，基于k近邻估计点平面性；filters.cluster，利用欧氏距离度量提取和标记聚类；filters.dbscan，基于密度的空间聚类；filters.covariancefeatures，基于一个点邻域的协方差计算局部特征；filters.eigenvalues，基于k最近邻计算点特征值；filters.nndistance，根据最近邻计算距离指数;filters.radialdensity，给定距离内的点的密度。Order部分：filters.mortonorder，使用Morton排序XY数据；filters.randomize，随机化视图中的点；filters.sort，基于给定的维度排序数据。Move部分：filters.reprojection，使用GDAL将数据从一个坐标系重新投影到另一个坐标系；filters.transformation，使用4x4变换矩阵变换每个点。Cull部分：filters.crop，根据边界框或一个多边形，过滤点；filters.iqr，剔除给定维度上，四分位范围外的点；filters.locate，给定维度，通过min/max返回一个点；filters.sample，执行泊松采样并只返回输入点的一个子集；filters.voxelcenternearestneighbor，返回每个体素内最靠近体素中心的点；filters.voxelcentroidnearestneighbor，返回每个体素内最接近体素质心的点。Join部分：filters.merge，将来自两个不同读取器的数据合并到一个流中。Mesh部分：使用Delaunay三角化创建mesh; filters.gridprojection，使用网格投影方法创建mesh; filters.poisson，使用泊松曲面重建算法创建么事。Languages部分：filters.python，在pipeline中嵌入python代码。Metadata部分：filters.stats，计算每个维度的统计信息(均值、最小值、最大值等)。
3. type-[readers](https://pdal.io/stages/readers.html)-[writers](https://pdal.io/stages/writers.html)，读写类型，例如通过"type":"writers.gdal"，可以使用"gdaldriver":"GTiff驱动，使用差值算法从点云创建栅格数据。常用保存的数据类型有，writers.gdal，writers.las，writers.ogr，writers.pgpointcloud，writers.ply，writers.sqlite，writers.text等。
4. type，通常配合Filters过滤器使用。例如如果配置”type”:“filters.crop”,则可以设置”bounds”:“([0,100],[0,100])”边界边界框进行裁切。
5. output\_type, 是给出数据计算的方式，例如mean,min,max,idx,count,stdev,all,idw等。
6. resolution，指定输出栅格的精度，例如1，10等。
7. filename，指定保存文件的名称。
8. [data\_type](https://pdal.io/types.html)，保存的数据类型，例如int8,int16,unint8,float,double等。
9. limits, 数据限制，例如配置过滤器为”type”:“filters.range”, 则”limits”:“Z[0:],Classification[6:6]”，仅提取标识为6，即建筑分类的点，和建筑的Z值。

pdal是命令行工具，在Anaconda中打开对应环境的终端，输入下述命令，会获得一个点的信息。命令行操作模式可以避免大批量数据读入内存，造成溢出，只是不方便查看数据，因此采用何种方式，可以依据具体情况确定。

pdal info G:\_lidarPtClouds\_16758900.las -p 0{ “file\_size”: 549264685, “filename”: “G:\data\IIT\_lidarPtClouds\rawPtClouds\LAS\_16758900.las”, “now”: “2022-01-13T16:44:02+0800”, “pdal\_version”: “2.3.0 (git-version: Release)”, “points”: { “point”: { “ClassFlags”: 0, “Classification”: 3, “EdgeOfFlightLine”: 0, “GpsTime”: 179803760, “Intensity”: 1395, “NumberOfReturns”: 1, “PointId”: 0, “PointSourceId”: 0, “ReturnNumber”: 1, “ScanAngleRank”: 15, “ScanChannel”: 0, “ScanDirectionFlag”: 0, “UserData”: 0, “X”: 1167506.44, “Y”: 1892449.7, “Z”: 594.62 } }, “reader”: “readers.las” }

\* pipeline

通常点云数据处理过程中，包括读取、处理、写入等操作。为了方便处理流程，PDAL引入pipeline概念，可以将多个操作堆叠在一个由JSON数据格式定义的数据流中。这对于复杂的处理流程而言，具有更大的优势。同时，PDAL也提供了python模式，可以在python中调入PDAL库，以及定义pipeline操作流程。例如如下官网提供的一个简单案例，包括读取.las文件（“%s”%separate\_las），配置维度为点云x坐标（“dimension”: “X”），并依据x坐标排序返回的数组（“type”: “filters.sort”）等操作。执行pipeline（pipeline.execute()）之后，pipeline对象返回点云具有维度的值，其dtypes项返回了点云具有的维度，对应返回的数组信息。这一个单元包含点的数量为count=18721702个点。

metadata元数据，可以打印查看，包括有坐标投影信息。

import util\_misc  
dirpath=r"G:\data\IIT\_lidarPtClouds\rawPtClouds"  
fileType=["las"]  
las\_paths=util\_misc.filePath\_extraction(dirpath,fileType)  
  
s\_t=util\_misc.start\_time()  
import pdal,os  
separate\_las=os.path.join(list(las\_paths.keys())[0],list(las\_paths.values())[0][32]).replace("\\","/") #注意文件名路径中"\"和"/"，不同库支持的类型可能有所不同，需自行调整  
  
json="""  
[  
 "%s",  
 {  
 "type": "filters.sort",  
 "dimension": "X"  
 }  
]  
"""%separate\_las  
  
pipeline=pdal.Pipeline(json)  
count=pipeline.execute()  
print("pts count:",count)  
arrays=pipeline.arrays  
print("arrays:",arrays)  
metadata=pipeline.metadata  
log=pipeline.log  
print("complete .las reading ")  
util\_misc.duration(s\_t)

start time: 2022-01-13 16:51:29.329062  
pts count: 16677942  
arrays: [array([(1175000., 1884958.18, 634.81, 9832, 1, 1, 0, 0, 5, 15., 0, 0, 1.78265216e+08, 0, 0),  
 (1175000., 1884941.11, 644.75, 18604, 1, 1, 0, 0, 5, 15., 0, 0, 1.78265216e+08, 0, 0),  
 (1175000., 1884931.3 , 641.59, 4836, 1, 1, 0, 0, 5, 15., 0, 0, 1.78265232e+08, 0, 0),  
 ...,  
 (1177500., 1882882.43, 597.43, 50337, 1, 1, 0, 0, 2, 15., 0, 0, 1.78265200e+08, 0, 0),  
 (1177500., 1882865.16, 597.49, 44165, 1, 1, 0, 0, 2, 15., 0, 0, 1.78239520e+08, 0, 0),  
 (1177500., 1882501.12, 596.34, 44193, 1, 1, 0, 0, 2, 15., 0, 0, 1.78239520e+08, 0, 0)],  
 dtype=[('X', '<f8'), ('Y', '<f8'), ('Z', '<f8'), ('Intensity', '<u2'), ('ReturnNumber', 'u1'), ('NumberOfReturns', 'u1'), ('ScanDirectionFlag', 'u1'), ('EdgeOfFlightLine', 'u1'), ('Classification', 'u1'), ('ScanAngleRank', '<f4'), ('UserData', 'u1'), ('PointSourceId', '<u2'), ('GpsTime', '<f8'), ('ScanChannel', 'u1'), ('ClassFlags', 'u1')])]  
complete .las reading   
end time: 2022-01-13 16:51:57.974012  
Total time spend:0.47 minutes

PDAL处理后，可以读取’pipeline’对象的属性，因为一个点包含多个信息，为方便查看，可以将点云数组转换为DataFrame格式数据。

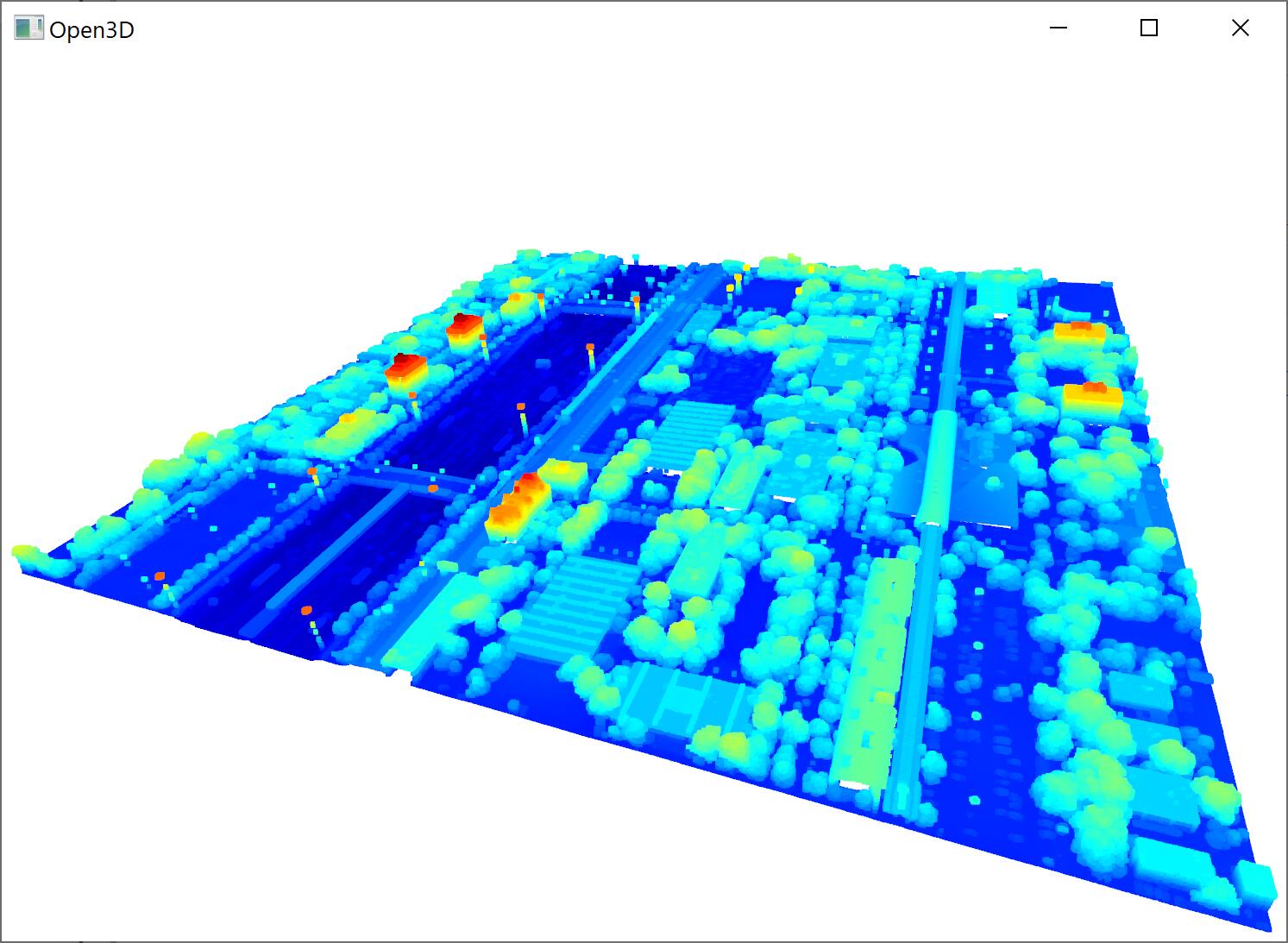
import pandas as pd  
pts\_df=pd.DataFrame(arrays[0])  
util\_misc.print\_html(pts\_df)



除了直接查看数据内容，可以借助open3d库打印三维点云，互动观察。但是因为最初使用PDAL读取.las点云（open3d目前不支持读取.las格式点云数据），需要将读取的点云数据转换为open3d支持的格式。显示的色彩代表点云的高度信息。

import open3d as o3d  
o3d\_pts=o3d.geometry.PointCloud()  
o3d\_pts.points=o3d.utility.Vector3dVector(pts\_df[['X','Y','Z']].to\_numpy())  
o3d.visualization.draw\_geometries([o3d\_pts])

Jupyter environment detected. Enabling Open3D WebVisualizer.  
[Open3D INFO] WebRTC GUI backend enabled.  
[Open3D INFO] WebRTCWindowSystem: HTTP handshake server disabled.



#### 2）建立DSM(Digital Surface Model)，与分类栅格

三维点云数据是三维格式数据，可以提取信息将其转换为对应的二维栅格数据，方便数据分析。点云数据转二维栅格数据最为常用的包括生成分类栅格数据，即地表覆盖类型；二是，提取地物高度，例如提取建筑物高度信息，植被高度信息等；三是生成DEM（Digital Elevation Model），DTM（Digital Terrain Model）等。

DEM-数字高程模型，为去除自然和建筑对象的裸地表面高程；

DTM-数字地形（或地面）模型，在DEM基础上增加自然地形的矢量特征，如河流和山脊。DEM和DTM很多时候并不明确区分，具体由数据所包含的内容来确定；

DSM-数字表面模型，同时捕捉地面，自然（例如树木），以及人造物（例如建筑）特征。

将对点云数据所要处理的内容定义在一个函数中，每一处理内容为一个pipeline，由json格式定义。下述代码定义了三项内容，一是，提取地物覆盖分类信息，用于建立分类栅格；二是，提取高程信息，用于建立DSM；三是，仅提取ground地表类型的高程，可以建立DEM。为了方便函数内日后不断增加新的提取内容，定义输入参数json\_combo管理和判断所要计算的pipleline，灵活处理函数，当增加新pipeline时避免较大的改动。

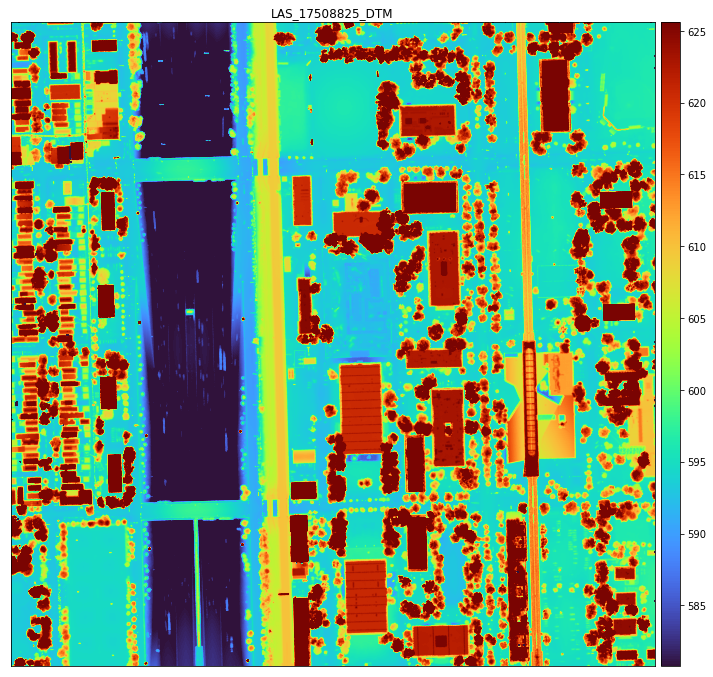
对于文件很大的地理空间信息数据，通常在处理过程中，完成一个主要的数据处理后就将其置于硬盘，使用时再读取。因此处理完一个点云单元（tile）之后，即刻将其保存到硬盘中，并不驻留于内存里，避免内存溢出。

def las\_info\_extraction(las\_fp,json\_combo):  
 import pdal  
 '''  
 function - 转换单个.las点云数据为分类栅格数据，和DSM栅格数据等  
   
 Paras:  
 las\_fp - .las格式文件路径  
 save\_path - 保存路径列表，分类DSM存储与不同路径下   
 '''  
 pipeline\_dict={}  
 if 'json\_classification' in json\_combo.keys():  
 #pipeline-用于建立分类栅格  
 json\_classification="""  
 {  
 "pipeline": [  
 "%s",  
 {  
 "filename":"%s",  
 "type":"writers.gdal",  
 "dimension":"Classification",  
 "data\_type":"uint16\_t",  
 "output\_type":"mean",   
 "resolution": 1  
 }   
 ]   
 }"""%(las\_fp,json\_combo['json\_classification'])  
 pipeline\_dict['json\_classification']=json\_classification  
   
 if "json\_DSM" in json\_combo.keys():  
 #pipeline-用于建立DSM栅格数据  
 json\_DSM="""  
 {  
 "pipeline": [  
 "%s",  
 {  
 "filename":"%s",  
 "gdaldriver":"GTiff",  
 "type":"writers.gdal",  
 "output\_type":"mean",   
 "resolution": 1  
 }   
 ]   
 }"""%(las\_fp,json\_combo['json\_DSM'])   
 pipeline\_dict['json\_DSM']=json\_DSM  
   
 if 'json\_ground' in json\_combo.keys():  
 #pipelin-用于提取ground地表  
 json\_ground="""  
 {  
 "pipeline": [  
 "%s",  
 {  
 "type":"filters.range",  
 "limits":"Classification[2:2]"   
 },  
 {  
 "filename":"%s",  
 "gdaldriver":"GTiff",  
 "type":"writers.gdal",  
 "output\_type":"mean",   
 "resolution": 1  
 }   
 ]   
 }"""%(las\_fp,json\_combo['json\_ground'])   
 pipeline\_dict['json\_ground']=json\_ground  
   
 for k,json in pipeline\_dict.items():   
 pipeline=pdal.Pipeline(json)  
 try:  
 pipeline.execute()  
 except:  
 print("\n An exception occurred,the file name:%s"%las\_fp)   
 print("finished conversion...")  
   
import os   
dirpath=r"G:\data\IIT\_lidarPtClouds\rawPtClouds"   
las\_fp=os.path.join(dirpath,'LAS\_17508825.las').replace("\\","/")  
workspace=r'G:\data\IIT\_lidarPtClouds'  
json\_combo={"json\_classification":os.path.join(workspace,'classification\_DSM\LAS\_17508825\_classification.tif').replace("\\","/"),  
 "json\_DSM":os.path.join(workspace,'classification\_DSM\LAS\_17508825\_DSM.tif').replace("\\","/")} #配置输入参数  
las\_info\_extraction(las\_fp,json\_combo)

finished conversion...

读取保存的DSM栅格文件，用earthpy库打印查看，因为数据中可能存在异常值，造成显示上的灰度，因此可以用分位数（np.quantile）的方法配置vmin和vmax参数。

import rasterio as rio  
import os  
import numpy as np  
import earthpy.plot as ep  
  
workspace=r'G:\data\IIT\_lidarPtClouds'  
with rio.open(os.path.join(workspace,'classification\_DSM\LAS\_17508825\_DSM.tif')) as DSM\_src:  
 DSM\_array=DSM\_src.read(1)  
titles = ["LAS\_17508825\_DTM"]  
ep.plot\_bands(DSM\_array, cmap="turbo", cols=1, title=titles, vmin=np.quantile(DSM\_array,0.1), vmax=np.quantile(DSM\_array,0.9))



同样读取保存的分类栅格数据，但是需要自行定义打印显示函数，根据整数指示的类别打印。其中类别由LAS格式给定的分类标识确定，颜色可以根据显示所要达到的效果自行定义。并增加了图例，方便查看颜色所对应的分类。

def las\_classification\_plotWithLegend(las\_fp):   
 import rasterio as rio  
 import pandas as pd  
 import numpy as np  
 import matplotlib.pyplot as plt  
 from matplotlib.colors import ListedColormap  
 from matplotlib import colors  
 from matplotlib.patches import Rectangle  
 '''  
 function - 显示由.las文件生成的分类栅格文件，并显示图例  
   
 Paras:  
 las\_fp - 分类文件路径  
 '''   
 with rio.open(las\_fp) as classi\_src:  
 classi\_array=classi\_src.read(1)  
  
 las\_classi\_colorName={0:'black',1:'white',2:'beige',3:'palegreen',4:'lime',5:'green',6:'tomato',7:'silver',8:'grey',9:'lightskyblue',10:'purple',11:'slategray',12:'grey',13:'cadetblue',14:'lightsteelblue',15:'brown',16:'indianred',17:'darkkhaki',18:'azure',9999:'white'}  
 las\_classi\_colorRGB=pd.DataFrame({key:colors.hex2color(colors.cnames[las\_classi\_colorName[key]]) for key in las\_classi\_colorName.keys()})  
 classi\_array\_color=[pd.DataFrame(classi\_array).replace(las\_classi\_colorRGB.iloc[idx]).to\_numpy() for idx in las\_classi\_colorRGB.index]  
 classi\_array\_color\_=np.concatenate([np.expand\_dims(i,axis=-1) for i in classi\_array\_color],axis=-1)  
 fig, ax=plt.subplots(figsize=(12, 12))  
 im=ax.imshow(classi\_array\_color\_, )  
 ax.set\_title(  
 "LAS\_classification",  
 fontsize=14,  
 )  
  
 #增加图例  
 color\_legend=pd.DataFrame(las\_classi\_colorName.items(),columns=["id","color"])  
 las\_classi\_name={0:'never classified',1:'unassigned',2:'ground',3:'low vegetation',4:'medium vegetation',5:'high vegetation',6:'building',7:'low point',8:'reserved',9:'water',10:'rail',11:'road surface',12:'reserved',13:'wire-guard(shield)',14:'wire-conductor(phase)',15:'transimission',16:'wire-structure connector(insulator)',17:'bridge deck',18:'high noise',9999:'null'}  
 color\_legend['label']=las\_classi\_name.values()  
 classi\_lengend=[Rectangle((0, 0), 1, 1, color=c) for c in color\_legend['color']]  
  
 ax.legend(classi\_lengend,color\_legend.label,mode='expand',ncol=3)  
 plt.tight\_layout()  
 plt.show()  
  
import os  
workspace=r'G:\data\IIT\_lidarPtClouds'  
las\_fp=os.path.join(workspace,'classification\_DSM\LAS\_17508825\_classification.tif')  
las\_classification\_plotWithLegend(las\_fp)



#### 3）批量处理.las点云单元

通过一个点云单元的代码调试，完成对一个单元的点云数据处理，为了能够批量处理所有点云单元，建立批量处理点云数据的函数。该函数直接调用上述单个点云单元处理函数，仅梳理所有点云单元文件的读取和保存路径。为了查看计算进度，使用tqdm库可以将循环计算过程以进度条的方式显示，明确完成所有数据计算大概所要花费的时间。同时以调用自定义的start\_time()和duration(s\_t)方法，计算具体的时长。

def las\_info\_extraction\_combo(las\_dirPath,json\_combo\_):  
 import util\_misc,util\_A,os,re  
 from tqdm import tqdm  
 '''  
 function - 批量转换.las点云数据为DSM和分类栅格  
   
 Paras:  
 las\_dirPath - LAS文件路径  
 save\_path - 保存路径   
 '''  
 file\_type=['las']  
 las\_fn=util\_misc.filePath\_extraction(las\_dirPath,file\_type)  
 '''展平列表函数'''  
 flatten\_lst=lambda lst: [m for n\_lst in lst for m in flatten\_lst(n\_lst)] if type(lst) is list else [lst]  
 las\_fn\_list=flatten\_lst([[os.path.join(k,las\_fn[k][i]) for i in range(len(las\_fn[k]))] for k in las\_fn.keys()])  
 pattern=re.compile(r'[\_](.\*?)[.]', re.S)   
 for i in tqdm(las\_fn\_list):   
 fn\_num=re.findall(pattern, i.split("\\")[-1])[0] #提取文件名字符串中的数字  
 #注意文件名路径中"\"和"/"，不同库支持的类型可能有所不同，需自行调整  
 json\_combo={key:os.path.join(json\_combo\_[key],"%s\_%s.tif"%(os.path.split(json\_combo\_[key])[-1],fn\_num)).replace("\\","/") for key in json\_combo\_.keys()}   
 util\_A.las\_info\_extraction(i.replace("\\","/"),json\_combo)  
   
dirpath=r"G:\data\IIT\_lidarPtClouds\rawPtClouds"   
json\_combo\_={"json\_classification":r'G:\data\IIT\_lidarPtClouds\classification'.replace("\\","/"),"json\_DSM":r'G:\data\IIT\_lidarPtClouds\DSM'.replace("\\","/")} #配置输入参数  
  
import util\_misc  
s\_t=util\_misc.start\_time()  
las\_info\_extraction\_combo(dirpath,json\_combo\_)   
util\_misc.duration(s\_t)

start time: 2022-01-13 19:07:54.440621  
  
  
100%|██████████| 73/73 [19:20<00:00, 15.89s/it]  
  
end time: 2022-01-13 19:27:14.586610  
Total time spend:19.33 minutes

* 合并栅格数据

以点云单元形式批量处理完所有的点云数据，生成同点云单元数量的多个DSM文件，和多个分类文件后，需要将其合并成一个完整的栅格文件。合并的方法主要使用rasterio库提供的merge方法。同时需要注意，要配置压缩，及保存类型，否则合并后的栅格文件可能非常大，例如本次合并所有的栅格后，文件大小约为4.5GB，但是配置"compress":'lzw',和"dtype":get\_minimum\_int\_dtype(mosaic),后，文件大小仅为201MB，大幅度压缩了文件，有利于节约硬盘空间，及存读速度。

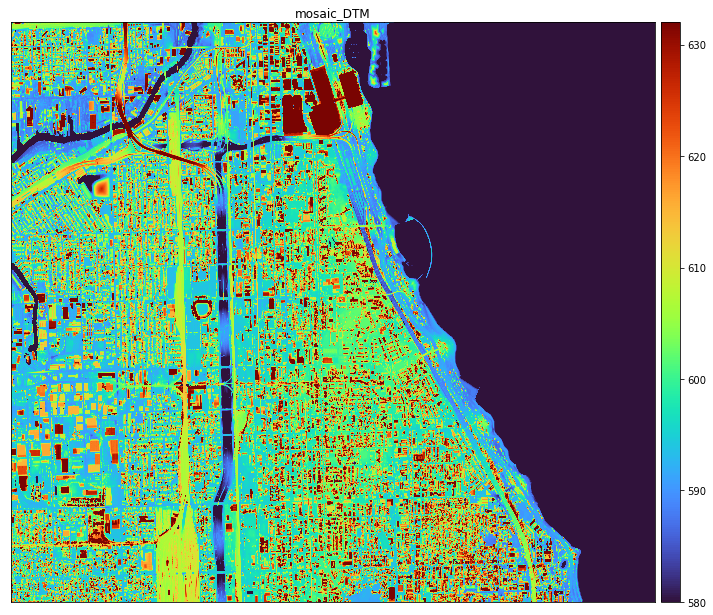
在配置文件保存类型时，迁移了rasterio库给出的函数get\_minimum\_int\_dtype(values)，自行依据数组数值确定所要保存的文件类型，从而避免了自行定义。

def raster\_mosaic(dir\_path,out\_fp,):  
 import rasterio,glob,os  
 from rasterio.merge import merge  
 '''  
 function - 合并多个栅格为一个  
   
 Paras:  
 dir\_path - 栅格根目录  
 out-fp - 保存路径  
   
 return:  
 out\_trans - 返回变换信息  
 '''  
   
 #迁移rasterio提供的定义数组最小数据类型的函数  
 def get\_minimum\_int\_dtype(values):  
 """  
 Uses range checking to determine the minimum integer data type required  
 to represent values.  
  
 :param values: numpy array  
 :return: named data type that can be later used to create a numpy dtype  
 """  
  
 min\_value = values.min()  
 max\_value = values.max()  
  
 if min\_value >= 0:  
 if max\_value <= 255:  
 return rasterio.uint8  
 elif max\_value <= 65535:  
 return rasterio.uint16  
 elif max\_value <= 4294967295:  
 return rasterio.uint32  
 elif min\_value >= -32768 and max\_value <= 32767:  
 return rasterio.int16  
 elif min\_value >= -2147483648 and max\_value <= 2147483647:  
 return rasterio.int32  
   
 search\_criteria = "\*.tif" #搜寻所要合并的栅格.tif文件  
 fp\_pattern=os.path.join(dir\_path, search\_criteria)  
 fps=glob.glob(fp\_pattern) #使用glob库搜索指定模式的文件  
 src\_files\_to\_mosaic=[]  
 for fp in fps:  
 src=rasterio.open(fp)  
 src\_files\_to\_mosaic.append(src)   
 mosaic,out\_trans=merge(src\_files\_to\_mosaic) #merge函数返回一个栅格数组，以及转换信息   
   
 #获得元数据  
 out\_meta=src.meta.copy()  
 #更新元数据  
 data\_type=get\_minimum\_int\_dtype(mosaic)  
 out\_meta.update({"driver": "GTiff",  
 "height": mosaic.shape[1],  
 "width": mosaic.shape[2],  
 "transform": out\_trans,  
 #通过压缩和配置存储类型，减小存储文件大小  
 "compress":'lzw',  
 "dtype":get\_minimum\_int\_dtype(mosaic),   
 }  
 )   
 with rasterio.open(out\_fp, "w", \*\*out\_meta) as dest:  
 dest.write(mosaic.astype(data\_type))   
   
 return out\_trans  
DSM\_dir\_path=r'G:\data\IIT\_lidarPtClouds\DSM'  
DSM\_out\_fp=r'G:\data\IIT\_lidarPtClouds\mosaic\DSM\_mosaic.tif'  
  
import util\_misc  
s\_t=util\_misc.start\_time()  
out\_trans=raster\_mosaic(DSM\_dir\_path,DSM\_out\_fp)  
util\_misc.duration(s\_t)

start time: 2022-01-13 19:34:23.680848  
end time: 2022-01-13 19:35:13.451225  
Total time spend:0.82 minutes

依据上述同样的方法，读取、打印和查看合并后的DSM栅格。

import rasterio as rio  
import earthpy.plot as ep  
import numpy as np  
DSM\_fp=r'G:\data\IIT\_lidarPtClouds\mosaic\DSM\_mosaic.tif'  
with rio.open(DSM\_fp) as DSM\_src:  
 mosaic\_DSM\_array=DSM\_src.read(1)  
titles = ["mosaic\_DTM"]  
ep.plot\_bands(mosaic\_DSM\_array, cmap="turbo", cols=1, title=titles, vmin=np.quantile(mosaic\_DSM\_array,0.25), vmax=np.quantile(mosaic\_DSM\_array,0.95))



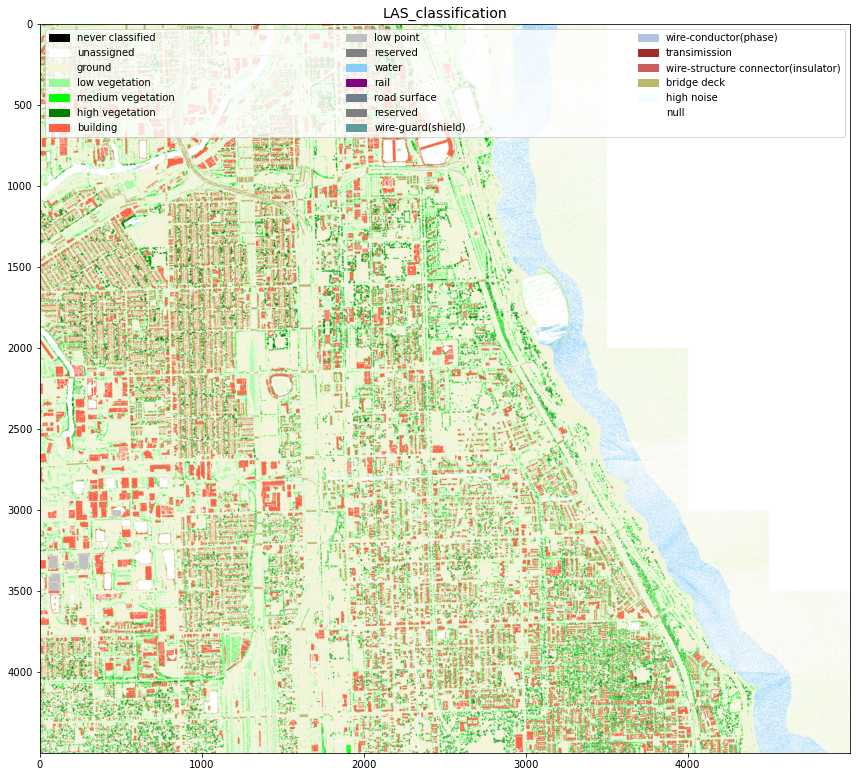
同样合并单个的分类栅格为一个文件。

classi\_dir\_path=r'G:\data\IIT\_lidarPtClouds\classification'  
classi\_out\_fp=r'G:\data\IIT\_lidarPtClouds\mosaic\classification\_mosaic.tif'  
  
import util\_misc,util\_A  
s\_t=util\_misc.start\_time()  
out\_trans=util\_A.raster\_mosaic(classi\_dir\_path,classi\_out\_fp)  
util\_misc.duration(s\_t)

start time: 2022-01-13 19:37:32.662113  
end time: 2022-01-13 19:37:49.891427  
Total time spend:0.28 minutes

import rasterio as rio  
import earthpy.plot as ep  
import numpy as np  
classi\_fp=r'G:\data\IIT\_lidarPtClouds\mosaic\classification\_mosaic.tif'  
with rio.open(classi\_fp) as classi\_src:  
 mosaic\_classi\_array=classi\_src.read(1)  
  
from skimage.transform import rescale  
mosaic\_classi\_array\_rescaled=rescale(mosaic\_classi\_array, 0.2, anti\_aliasing=False,preserve\_range=True)  
print("original shape:",mosaic\_classi\_array.shape)  
print("rescaled shape:",mosaic\_classi\_array\_rescaled.shape)  
  
import util\_A  
util\_A.las\_classification\_plotWithLegend\_(mosaic\_classi\_array\_rescaled)

original shape: (22501, 25001)  
rescaled shape: (4500, 5000)  
  
  
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).



### 2.8.1.2 建筑高度提取

建筑高度提取的流程为：将DSM栅格重投影为该区域（芝加哥）Landsat所定义的坐标投影系统，统一投影坐标系；–>从[Chicago Data Portal，CDP](https://data.cityofchicago.org/)获取’Building Footprints (current)’.shp格式的Polygon建筑分布–>依据DSM栅格的范围（extent）裁切.shp格式的建筑分布矢量数据，并定义投影同重投影后的DSM栅格文件；–>PSAL提取地面（ground）信息，并存储为栅格；–>插值所有单个的ground栅格，并合并，重投影同DSM投影后保存；–>根据分类栅格数据，从DSM中提取建筑区域的高程数据–>用裁切后的建筑矢量数据，使用’rasterstats’库提供的’zonal\_stats’方法，分别提取DSM和ground栅格数据高程信息，统计方式为median（中位数）；–>用区域统计提取的DSM-ground，即为建筑高度数据；–>将建筑高度数据写入GeoDataFrame，并保存为.shp文件，备日后分析使用。

#### 1） 定义获取栅格投影函数，及栅格重投影函数

投影和重投影的方法在Landsat遥感影像处理部分使用过，可以结合查看。

def get\_crs\_raster(raster\_fp):  
 import rasterio as rio  
 '''  
 function - 获取给定栅格的投影坐标-crs.  
   
 Paras:  
 raster\_fp - 给定栅格文件的路径  
 '''  
 with rio.open(raster\_fp) as raster\_crs:  
 raster\_profile=raster\_crs.profile  
 return raster\_profile['crs']  
   
ref\_raster=r'G:\data\Landsat\data\_processed\DE\_Chicago.tif' # 使用的为Landsat部分处理的遥感影像，可以自行下载对应区域的Landsat，作为参数输入，获取其投影  
dst\_crs=get\_crs\_raster(ref\_raster)  
print("dst\_crs:",dst\_crs)

dst\_crs: EPSG:32616

def raster\_reprojection(raster\_fp,dst\_crs,save\_path):  
 from rasterio.warp import calculate\_default\_transform, reproject, Resampling  
 import rasterio as rio  
 '''  
 function - 转换栅格投影  
   
 Paras:  
 raster\_fp - 待转换投影的栅格  
 dst\_crs - 目标投影  
 save\_path - 保存路径  
 '''  
 with rio.open(raster\_fp) as src:  
 transform, width, height = calculate\_default\_transform(src.crs, dst\_crs, src.width, src.height, \*src.bounds)  
 kwargs = src.meta.copy()  
 kwargs.update({  
 'crs': dst\_crs,  
 'transform': transform,  
 'width': width,  
 'height': height  
 })  
 with rio.open(save\_path, 'w', \*\*kwargs) as dst:  
 for i in range(1, src.count + 1):  
 reproject(  
 source=rio.band(src, i),  
 destination=rio.band(dst, i),  
 src\_transform=src.transform,  
 src\_crs=src.crs,  
 dst\_transform=transform,  
 dst\_crs=dst\_crs,  
 resampling=Resampling.nearest)   
 print("finished reprojecting...")  
   
DTM\_fp=r'G:\data\IIT\_lidarPtClouds\mosaic\DSM\_mosaic.tif'  
DTM\_reprojection\_fp=r'G:\data\IIT\_lidarPtClouds\mosaic\DSM\_mosaic\_reprojection.tif'  
dst\_crs=dst\_crs  
raster\_reprojection(DTM\_fp,dst\_crs,DTM\_reprojection\_fp)

finished reprojecting...

#### 2）按照给定的栅格，获取栅格的范围来裁切.shp格式文件（Polygon）

在使用’gpd.clip(vector\_projection\_,poly\_gdf) ’方法裁切矢量数据时，需要清理数据，包括vector.dropna(subset=["geometry"], inplace=True)清理空值；以及polygon\_bool=vector\_projection.geometry.apply(lambda row:True if type(row)==type\_Polygon and row.is\_valid else False)清理无效的Polygon对象，和不为’shapely.geometry.polygon.Polygon’格式的数据。只有清理完数据后才能够执行裁切，否则会提示错误。

def clip\_shp\_withRasterExtent(vector\_shp\_fp,reference\_raster\_fp,save\_path):  
 import rasterio as rio  
 from rasterio.plot import plotting\_extent  
 import geopandas as gpd   
 import pandas as pd  
 from shapely.geometry import Polygon  
 import shapely  
 '''  
 function - 根据给定栅格的范围，裁切.shp格式数据，并定义投影同给定栅格  
   
 Paras:  
 vector\_shp\_fp - 待裁切的vector文件路劲  
 reference\_raster\_fp - 参考栅格，extent及投影  
 save\_path - 保存路径  
   
 return:  
 poly\_gdf - 返回裁切边界  
 '''  
 vector=gpd.read\_file(vector\_shp\_fp)  
 print("before dropna:",vector.shape)  
 vector.dropna(subset=["geometry"], inplace=True)  
 print("after dropna:",vector.shape)  
 with rio.open(reference\_raster\_fp) as src:  
 raster\_extent=plotting\_extent(src)  
 print("extent:",raster\_extent)  
 raster\_profile=src.profile  
 crs=raster\_profile['crs']  
 print("crs:",crs)   
 extent=raster\_extent  
 polygon=Polygon([(extent[0],extent[2]),(extent[0],extent[3]),(extent[1],extent[3]),(extent[1],extent[2]),(extent[0],extent[2])])  
 #poly\_gdf=gpd.GeoDataFrame([1],geometry=[polygon],crs=crs)   
 poly\_gdf=gpd.GeoDataFrame({'name':[1],'geometry':[polygon]},crs=crs)   
 vector\_projection=vector.to\_crs(crs)  
   
 #移除非Polygon类型的行，和无效的Polygon(用.is\_valid验证)，否则无法执行.clip  
 type\_Polygon=shapely.geometry.polygon.Polygon  
 polygon\_bool=vector\_projection.geometry.apply(lambda row:True if type(row)==type\_Polygon and row.is\_valid else False)  
 vector\_projection\_=vector\_projection[polygon\_bool]  
  
 vector\_clip=gpd.clip(vector\_projection\_,poly\_gdf)   
 vector\_clip.to\_file(save\_path)  
 print("finished clipping and projection...")  
 return poly\_gdf  
   
DTM\_reprojection\_fp=r'G:\data\IIT\_lidarPtClouds\mosaic\DSM\_mosaic\_reprojection.tif'   
vector\_shp\_fp=r'G:\data\Building Footprints\Building Footprints.shp'  
save\_path=r'G:\data\building\_footprints\_clip\_projection\building\_footprints\_clip\_projection.shp'  
poly\_gdf=clip\_shp\_withRasterExtent(vector\_shp\_fp,DTM\_reprojection\_fp,save\_path)

before dropna: (820606, 50)  
after dropna: (820600, 50)  
extent: (445062.87208577903, 452785.6657144039, 4627534.041486925, 4634507.0108712595)  
crs: EPSG:32616  
finished clipping and projection...

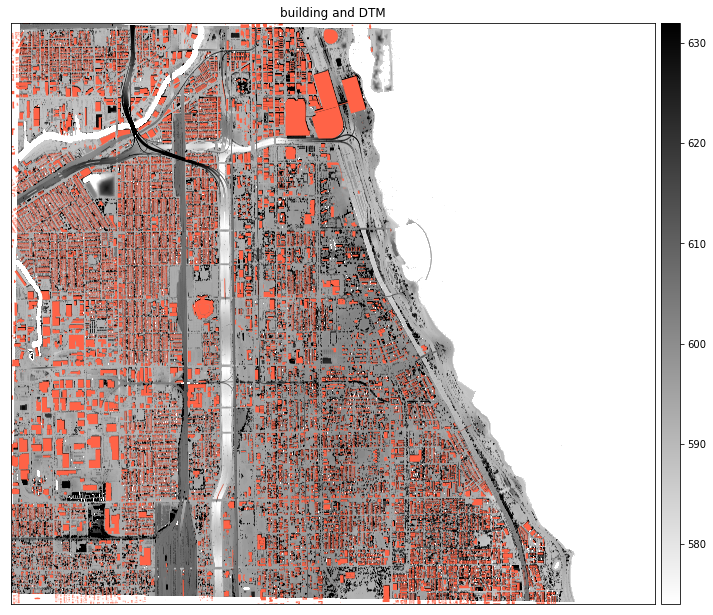
* 查看处理后的.shp格式建筑矢量数据

可以叠加打印DSM的栅格数据，和建筑矢量数据，确定二者在地理空间坐标保持一致的条件下，相互吻合。说明数据处理正确，否则需要返回查看之前的代码，确定出错的位置，调整代码重新计算。

import matplotlib.pyplot as plt  
import geopandas as gpd  
#fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 8))  
vector=gpd.read\_file(vector\_shp\_fp)  
vector.plot(figsize=(12,12))



import matplotlib.pyplot as plt  
import earthpy.plot as ep  
import rasterio as rio  
from rasterio.plot import plotting\_extent  
import numpy as np  
import geopandas as gpd  
  
fig, ax=plt.subplots(figsize=(12, 12))  
  
DTM\_reprojection\_fp=r'G:\data\IIT\_lidarPtClouds\mosaic\DSM\_mosaic\_reprojection.tif'  
with rio.open(DTM\_reprojection\_fp) as DTM\_src:  
 mosaic\_DTM\_array=DTM\_src.read(1)  
 plot\_extent=plotting\_extent(DTM\_src)  
   
titles = ["building and DTM"]  
ep.plot\_bands(mosaic\_DTM\_array, cmap="binary", cols=1, title=titles, vmin=np.quantile(mosaic\_DTM\_array,0.25), vmax=np.quantile(mosaic\_DTM\_array,0.95),ax=ax,extent=plot\_extent)  
  
building\_clipped\_fp=r'G:\data\building\_footprints\_clip\_projection\building\_footprints\_clip\_projection.shp'  
vector=gpd.read\_file(building\_clipped\_fp)  
vector.plot(ax=ax,color='tomato')  
plt.show()



#### 3）根据分类栅格数据，从DSM中提取建筑区域的高程数据

因为建筑矢量数据每个建筑polygon并不一定仅包括分类为建筑的DSM栅格，可能包括其它分类数据，因此需要DSM仅保留建筑高程信息，避免计算误差。配合使用np.where()实现。

import util\_A  
classi\_fp=r'G:\data\IIT\_lidarPtClouds\mosaic\classification\_mosaic.tif'  
classi\_reprojection\_fp=r'G:\data\IIT\_lidarPtClouds\mosaic\classi\_mosaic\_reprojection.tif'  
dst\_crs=dst\_crs  
util\_A.raster\_reprojection(classi\_fp,dst\_crs,classi\_reprojection\_fp)

finished reprojecting...

import util\_misc  
s\_t=util\_misc.start\_time()   
  
classi\_reprojection\_fp=r'G:\data\IIT\_lidarPtClouds\mosaic\classi\_mosaic\_reprojection.tif'  
with rio.open(classi\_reprojection\_fp) as classi\_src:  
 classi\_reprojection=classi\_src.read(1)  
 out\_meta=classi\_src.meta.copy()  
   
building\_DSM=np.where(classi\_reprojection==6,mosaic\_DTM\_array,np.nan) #仅保留建筑高程信息  
building\_DSM\_fp=r'G:\data\IIT\_lidarPtClouds\mosaic\building\_DSM.tif'  
with rio.open(building\_DSM\_fp, "w", \*\*out\_meta) as dest:  
 dest.write(building\_DSM.astype(rio.uint16),1)   
util\_misc.duration(s\_t)

start time: 2022-01-13 20:22:29.961764  
end time: 2022-01-13 20:22:38.113300  
Total time spend:0.13 minutes

* 提取ground，并插值，合并、以及重投影，查看数据

-提取

import util\_A  
las\_dirPath=r"G:\data\IIT\_lidarPtClouds\rawPtClouds"  
json\_combo\_={"json\_ground":r'G:\data\IIT\_lidarPtClouds\ground'}  
util\_A.las\_info\_extraction\_combo(las\_dirPath,json\_combo\_)

100%|██████████| 73/73 [06:59<00:00, 5.74s/it]

-插值

插值使用了rasterio库提供的fillnodata方法。该方法是对每个像素，在四个方向上以圆锥形搜索值，根据反向距离加权计算插值。一旦完成所有插值，可以使用插值像素上的3x3平均过滤器迭代，平滑数据。这种算法通常适宜于连续变化的栅格，例如DEM，以及填补小的空洞。

def rasters\_interpolation(raster\_path,save\_path,max\_search\_distance=400,smoothing\_iteration=0):  
 import rasterio,os  
 from rasterio.fill import fillnodata  
 import glob  
 from tqdm import tqdm  
 '''   
 function - 使用rasterio.fill的插值方法，补全缺失的数据  
   
 Paras:  
 raster\_path - 栅格根目录  
 save\_path - 保持的目录  
 '''  
 search\_criteria = "\*.tif" #搜寻所要合并的栅格.tif文件  
 fp\_pattern=os.path.join(raster\_path, search\_criteria)  
 fps=glob.glob(fp\_pattern) #使用glob库搜索指定模式的文件  
   
 for fp in tqdm(fps):  
 with rasterio.open(fp,'r') as src:  
 data=src.read(1, masked=True)  
 msk=src.read\_masks(1)   
 #配置max\_search\_distance参数，或者多次执行插值，补全较大数据缺失区域  
 fill\_raster=fillnodata(data,msk,max\_search\_distance=max\_search\_distance,smoothing\_iterations=0)   
 out\_meta=src.meta.copy()   
 with rasterio.open(os.path.join(save\_path,"interplate\_%s"%os.path.basename(fp)), "w", \*\*out\_meta) as dest:   
 dest.write(fill\_raster,1)  
   
raster\_path=r'G:\data\IIT\_lidarPtClouds\ground'  
save\_path=r'G:\data\IIT\_lidarPtClouds\ground\_interpolation'   
import util\_misc  
s\_t=util\_misc.start\_time()   
rasters\_interpolation(raster\_path,save\_path,max\_search\_distance=400,smoothing\_iteration=0)   
util\_misc.duration(s\_t)

start time: 2022-01-13 20:32:00.967135  
  
  
100%|██████████| 73/73 [04:56<00:00, 4.06s/it]  
  
end time: 2022-01-13 20:36:58.039148  
Total time spend:4.95 minutes

-合并

ground\_dir\_path=r'G:\data\IIT\_lidarPtClouds\ground\_interpolation'   
ground\_out\_fp=r'G:\data\IIT\_lidarPtClouds\mosaic\ground\_mosaic.tif'  
  
import util\_misc,util\_A  
s\_t=util\_misc.start\_time()  
out\_trans=util\_A.raster\_mosaic(ground\_dir\_path,ground\_out\_fp)  
util\_misc.duration(s\_t)

start time: 2022-01-13 20:38:01.332643  
end time: 2022-01-13 20:38:22.115885  
Total time spend:0.33 minutes

-重投影

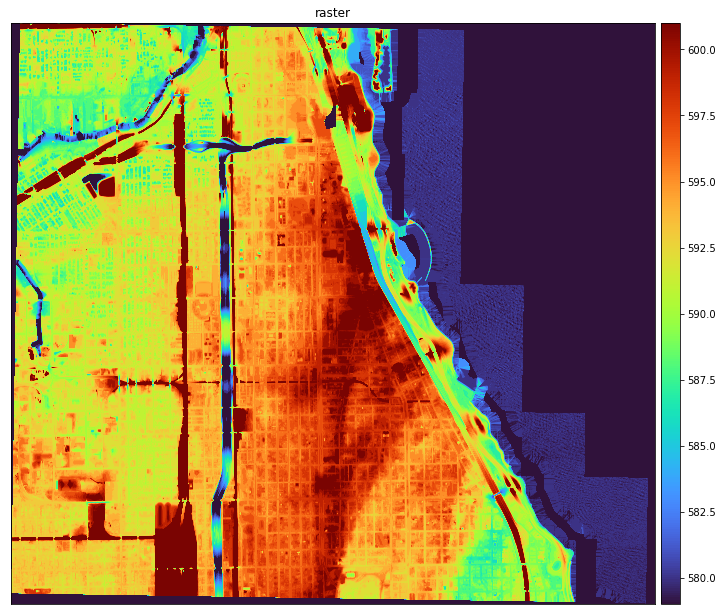
import util\_A  
ref\_raster=r'G:\data\Landsat\data\_processed\DE\_Chicago.tif' # 使用的为Landsat部分处理的遥感影像，可以自行下载对应区域的Landsat，作为参数输入，获取其投影  
dst\_crs=util\_A.get\_crs\_raster(ref\_raster)  
print("dst\_crs:",dst\_crs)  
  
ground\_fp=r'G:\data\IIT\_lidarPtClouds\mosaic\ground\_mosaic.tif'  
ground\_reprojection\_fp=r'G:\data\IIT\_lidarPtClouds\mosaic\ground\_mosaic\_reprojection.tif'  
util\_A.raster\_reprojection(ground\_fp,dst\_crs,ground\_reprojection\_fp)

dst\_crs: EPSG:32616  
finished reprojecting...

-查看数据

为了方便栅格数据的打印查看，将其定义为一个函数，方便调用。

def raster\_show(raster\_fp,title='raster',vmin\_vmax=[0.25,0.95],cmap="turbo"):  
 import rasterio as rio  
 import earthpy.plot as ep  
 import numpy as np  
 '''  
 function - 使用earthpy库显示遥感影像（一个波段）  
   
 Paras:  
 raster\_fp - 输入栅格路径  
 vmin\_vmax -调整显示区间  
 '''   
   
 with rio.open(raster\_fp) as src:  
 array=src.read(1)  
 titles=[title]  
 ep.plot\_bands(array, cmap=cmap, cols=1, title=titles, vmin=np.quantile(array,vmin\_vmax[0]), vmax=np.quantile(array,vmin\_vmax[1]))  
  
raster\_fp=r'G:\data\IIT\_lidarPtClouds\mosaic\ground\_mosaic\_reprojection.tif'  
raster\_show(raster\_fp)



#### 3）区域统计，计算建筑高度

使用rasterstats库的zonal\_stats方法，提取DSM和ground栅格高程数据。

from rasterstats import zonal\_stats  
import util\_misc  
building\_clipped\_fp=r'G:\data\building\_footprints\_clip\_projection\building\_footprints\_clip\_projection.shp'  
  
s\_t=util\_misc.start\_time()   
building\_DTM\_fp=r'G:\data\IIT\_lidarPtClouds\mosaic\building\_DSM.tif'  
stats\_DTM=zonal\_stats(building\_clipped\_fp, building\_DTM\_fp,stats="median")  
ground\_mosaic\_fp=r'G:\data\IIT\_lidarPtClouds\mosaic\ground\_mosaic\_reprojection.tif'  
stats\_ground=zonal\_stats(building\_clipped\_fp, ground\_mosaic\_fp,stats="median")  
util\_misc.duration(s\_t)

start time: 2022-01-13 20:45:20.199333  
end time: 2022-01-13 20:53:11.112515  
Total time spend:7.83 minutes

建筑高度=DSM提取的高程-ground提取的高程。为方便计算将其转换为pandas的DataFrame数据格式，应用.apply及lambda函数进行计算。并将计算结果增加到建筑矢量数据的GeoDataFrame中，另存为.shp格式数据。

import numpy as np  
import pandas as pd  
import geopandas as gpd  
building\_height\_df=pd.DataFrame({'dtm':[k['median'] for k in stats\_DTM],'ground':[k['median'] for k in stats\_ground]})  
building\_height\_df['height']=building\_height\_df.apply(lambda row:row.dtm-row.ground if row.dtm>row.ground else -9999,axis=1)  
print(building\_height\_df[:10])  
  
building\_clipped\_fp=r'G:\data\building\_footprints\_clip\_projection\building\_footprints\_clip\_projection.shp'  
vector=gpd.read\_file(building\_clipped\_fp)  
vector['height']=building\_height\_df['height']  
vector.to\_file(r'G:\data\building\_footprints\_height\building\_footprints\_height.shp')  
print("finished computation and save...")

dtm ground height  
0 625.0 593.0 32.0  
1 625.0 591.0 34.0  
2 626.0 595.0 31.0  
3 625.0 593.0 32.0  
4 625.0 591.0 34.0  
5 625.0 591.0 34.0  
6 625.0 591.0 34.0  
7 0.0 594.0 -9999.0  
8 630.0 592.0 38.0  
9 626.0 592.0 34.0  
finished computation and save...

打开，与打印查看计算结果。

import geopandas as gpd  
import matplotlib.pyplot as plt  
from mpl\_toolkits.axes\_grid1 import make\_axes\_locatable  
import numpy as np  
  
building\_footprints\_height\_fp=r'G:\data\building\_footprints\_height\building\_footprints\_height.shp'  
building\_footprints\_height=gpd.read\_file(building\_footprints\_height\_fp)  
  
fig, ax=plt.subplots(figsize=(12, 12))  
divider=make\_axes\_locatable(ax)  
cax\_1=divider.append\_axes("right", size="5%", pad=0.1)  
building\_footprints\_height.plot(column='height',ax=ax,cax=cax\_1,legend=True,cmap='OrRd',vmin=np.quantile(building\_footprints\_height.height,0.25), vmax=np.quantile(building\_footprints\_height.height,0.95)) #'OrRd','PuOr'

