City-scale Building Occupancy Prediction Using Geographic Information System Data

Jing Wang[1], Yunyang Ye[2], Wangda Zuo[1], Joshua New[3], Amy Rose[3]

1 University of Colorado Boulder, Department of Civil, Environmental, and Architectural Engineering, Boulder, CO, USA;
2 Pacific Northwest National Laboratory, Richland, WA, USA[a];
3 Oak Ridge National Laboratory, Oak Ridge, TN, USA.

Disclaimer

This documentation is a joint project report developed by CU Boulder and Oak Ridge National Laboratory. It is undergoing the transformation to be turned into a conference paper.

1 Introduction

The goal of this research is to assist in quantifying the impact of enhanced occupancy information on the accuracy of building energy models compared to actual 15-minute data. Through gridded population data extraction and cross-reference with building location and area information, daytime and nighttime occupant density data that could be leveraged to modify the occupancy schedules are generated. This work will improve the ability to quantitatively estimate and empirically validate reductions in energy use intensity of individual buildings within a large area. Further, this work will test the value for sources of occupancy data to improve utility-scale models and create measures to assess the energy, demand, emissions, and cost-saving opportunities for a utility. This paper provides the methodology and detailed techniques that are used to achieve this aim.

2 Technology and Workflow

The raw data include daytime and nighttime population data in the format of raster datasets, as well as building footprint and conditioned area data in the format of csv file. In order to finally obtain the daytime and nighttime population data in each provided building, we followed the workflow below:

1. **Extracting population data from the raster datasets:** In this part, we first used a geographical software *ArcGIS* to visualize the grid and population data and get an overall impression of the data. Also, we transferred the format of the image to *TIFF* so that it can be further processed with python. Then, we used python package *rasterio* to extract the data points embedded in each pixel of the TIFF image. The extracted information includes

[a] PNNL is Yunyang Ye's current affiliation whereas this work was done when he was affiliated with University of Colorado Boulder.

longitude and latitude of each grid center and daytime/nighttime population in each grid cell.

2. **Cleaning population and building data:** Since the raw population data includes datapoints that have no population information (represented by number -32,768), in this part we deleted those datapoints to save unnecessary computing time. Further, in the original building data (EPB_G2.csv), there exist repeated building records. Hence, we deleted the repeated records and reduced the building number from 178,355 to 120,372. This will also save extra computational time.
3. **Finding intersected area between each building and the grids:** In this part, we swept all buildings over every grid cell to find the intersected area using python package *shapely*. This helps distributing the building area into each cell that it has overlap with and thus prepares for the next step. As required by the package, we also converted the longitude and latitude of grid cells and building footprints into an x-y coordinate system. Due to the large amount of data, we used *parallel computing* and the *supercomputer* to conduct the computation.
4. **Distributing population data into each building:** Knowing the building area inside each grid cell, in this part, we distributed the population in each cell into the buildings inside it. The distribution is conducted based on the building conditioned floor area and the "people spend 87 percent of time indoor" principle. Besides the results of daytime/nighttime population, we also obtained the occupant density data by dividing the occupant number by total floor area.
5. **Building re-indexing and final results:** In this final part, we correlated our results with the originally provided building indices (178,355 buildings) so that the generated results can be directly leveraged by the user.

The following is a diagram summarizing our entire workflow.

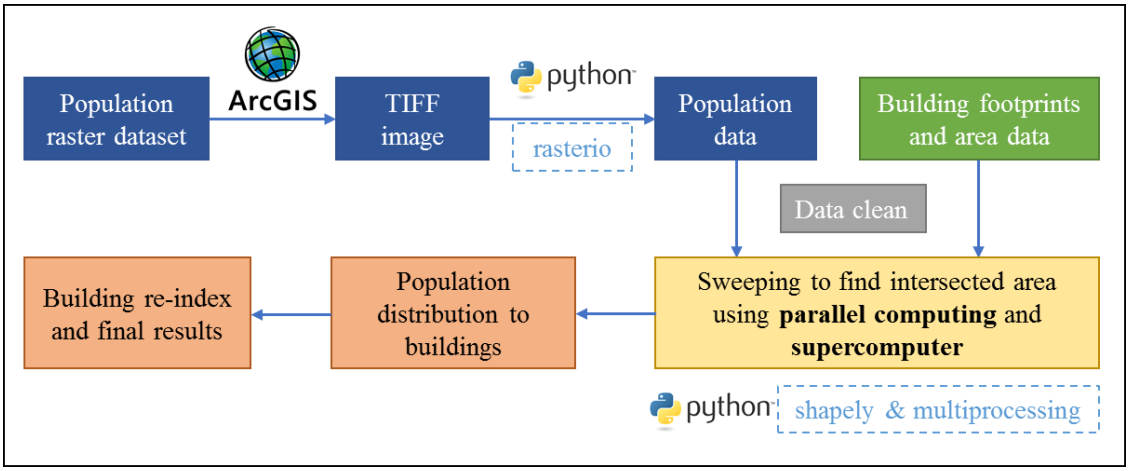

Figure 1 Diagram of the workflow

3 Technical Details

This section provides technical details in processing the population and building data to help better understand the code and use them. Key equations and explanations of the codes (e.g., inputs and outputs, sample of results) will be discussed. In the folder *Task0b_submission*, five python code files are included:

1_population.py
2_clean_data.py
3_sweep.py
4_distribution.py
5_reindex.py

Each code relates to one corresponding subsection below. All coding is in python 2.7 with comment explanations. They can be run in either Linux or Windows system given that the required python packages are installed. All intermediate result files are all put in the same folder.

3.1 Extract population data

- **Visualization**

Using the software ArcGIS, we visualized the population raster datasets as follows.

Figure 2 Day population dataset visualization

Figure 3 Night population dataset visualization

Each dataset consists of 725 columns and 717 rows (Figure 4). In total, there are 725*717=519,825 cells. Each cell has the size of 0.000833333x0.000833333 (longitude x latitude). Both of the raster datasets use the same global coordinate system: WGS_1984.

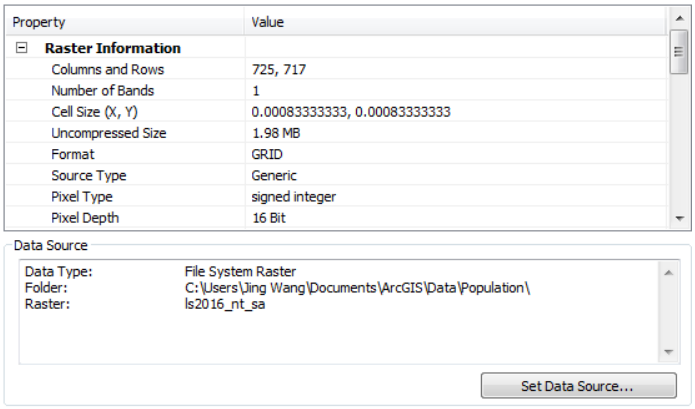

Figure 4 Raster dataset information

- **Cell size check**

To double check the grid cell size, we used Harversine formula[b], which is a formula for calculating great-circle distance between two points on a sphere, to convert the grid size from longitude/latitude into meters (see code *check_distance.py*). The results show that the grid size is 92.66 * 92.66 = 8585.88 m^2, close to the provided number of "90 meters". The difference might be caused by the fact that Harversine formula is for spheres and the earth is a spheroid.

```
from math import cos, asin, sqrt
import pandas as pd

# implementation of the Harversine formula
# calculates great-circle distance between two points on a sphere (not accounting for the Earth being a spheroid)
def distance(lat1, lon1, lat2, lon2):
    p = 0.017453292519943295     # Pi/180
    a = 0.5 - cos((lat2 - lat1) * p)/2 + cos(lat1 * p) * cos(lat2 * p) * (1 - cos((lon2 - lon1) * p)) / 2
    return 12742 * asin(sqrt(a)) * 1000 # 2*R*asin...

df1=pd.read_csv('total.csv', index_col=0)

for i in range(len(df1)):
    print(distance(df1['lat'][i],df1['lon'][i],df1['lat'][i+1],df1['lon'][i+1]))
```

Figure 5 Code: check_distance.py

- **Information extraction**

To extract the information from the raster datasets, we first converted them into TIFF image files with ArcGIS (day.tif, night.tif). As the next step, we used the python package for dealing with raster datasets: rasterio[c]. The function rasterio.transform returns the *center coordinates* of a pixel at each row and column. The value embedded in each pixel represents the corresponding day/night population. Using the following code, we managed to record the longitude, latitude, daytime population and nighttime population of every grid cell. The results are exported in "total.csv".

[b] https://en.wikipedia.org/wiki/Haversine_formula

[c] https://geohackweek.github.io/raster/04-workingwithrasters/

```
# processing the data
lon = [] #list of longitude
lat = [] # list of latitude
day_pop = [] # list of daytime population
night_pop = [] # list of nighttime population

for row in range(0,725):
    for col in range(0,717):
        location = src1.transform * (row,col)
        lon.append(location[0])
        lat.append(location[1])
        day_pop.append(array1[row,col])
        night_pop.append(array2[row,col])
        print(row,col)
temp=pd.DataFrame()
temp['lon']=lon
temp['lat']=lat
temp['day_pop']=day_pop
temp['night_pop']=night_pop
temp.to_csv('total.csv')
```

Figure 6 Code for extracting population data

- **Sample results**

The population data is in the following format.

	lon	lat	day_pop	night_pop
0	-85.5592	35.47083	-32768	-32768
1	-85.5592	35.47	-32768	-32768
2	-85.5592	35.46917	-32768	-32768
3	-85.5592	35.46833	-32768	-32768
4	-85.5592	35.4675	-32768	-32768
5	-85.5592	35.46667	-32768	-32768

Figure 7 Sample results of population data

3.2 Clean population and building data

As shown in Figure 7, some grid cells do not contain actual population data, which is represented by the number -32768. To save unnecessary computational time, we removed those cells and generated the cleaned population data file “population_new.csv”.

Similarly, in the original building data file “EPB_G2.csv”, there exist repeated building records (buildings with the same footprint as well as area). To save extra computational time, we also removed those repeated records and generated the new building data file “building_new.csv”. Also, we converted the area unit from sqft to sq.m in this part.

- **Sample results**

The cleaned population and building data are in the following format.

-85.5592	35.015	0	0
-85.5592	35.01417	0	0
-85.5583	35.01583	0	0
-85.5583	35.015	0	0
-85.5583	35.01417	0	0
-85.5583	35.01333	0	0
-85.5583	35.0125	0	0
-85.5583	35.01167	0	0
-85.5583	35.01083	0	0

Figure 8 Sample results of cleaned population data

	footprint	area
0	35.07876/-85.186735_35.078804/-85.18	1277.598
1	35.078733/-85.187749_35.078814/-85.1	225.8144
2	35.034369/-85.155029_35.034369/-85.1	218.6781
3	35.148437/-85.251216_35.148456/-85.2	39839.08
4	35.148437/-85.251216_35.148456/-85.2	19919.54
5	35.010815/-85.288927_35.010905/-85.2	227.8553
6	35.005796/-85.216401_35.00546/-85.21	1700.152
7	35.009975/-85.293461_35.009988/-85.2	120.1164

Figure 9 Sample results of cleaned building data

3.3 Find intersected area

This part deals with finding the intersected area of each building with each grid. We first converted latitude and longitude of each vertex into the same x-y coordinate system and then used python shapely.geomatry package to determine the intersected areas. Due to the high computational demand, we used parallel computing on a supercomputer to conduct the computation process. The results of this part are saved in the "gridcell.csv" file.

- **Convert to x-y coordinate system**

As required by the shapely package, the coordinates of building footprints and each grid cell should be in the same two-dimensional coordinate system. Hence, we pointed the lower left corner of all the grids as the origin of the x-y coordinate system and then projected all other points into this system. The code is shown below.

```
# convert grids and footprints lon/lat into x/y, the lower left corner is the origin of the coordinate system
## read the cleaned population data
population=[]
with open('population_new.csv','r') as csvfile:
    csvreader = csv.reader(csvfile)
    for row in csvreader:
        population.append(map(float,row))
## get the lon/lat of the lower left corner
lon0=min(column(population, 0))
lat0=min(column(population, 1))
print(min(column(population, 0)), min(column(population, 1)))
## read the cleaned footprint data
footprint=[]
with open('building_new.csv') as csvfile:
    csvreader = csv.reader(csvfile)
    csvreader.next()   # optional
    for row in csvreader:
        a=row[1]
        b=[]
        for z in a.replace('/','_').split('_'):
            b.append(float(z))
        footprint.append(b)
## calculate the x, y coordinates of all footprint vertices (unit: meter)
x_footprint=[]
y_footprint=[]
x_rowlist=[]
y_rowlist=[]
for i in range(len(footprint)):
    for j in range(len(footprint[i])):
        if j%2==0:
            y_rowlist.append(distance(lon0, lat0, lon0, footprint[i][j]))
        else:
            x_rowlist.append(distance(lon0, lat0, footprint[i][j], lat0))
    x_footprint.append(x_rowlist)
    y_footprint.append(y_rowlist)
    x_rowlist=[]
    y_rowlist=[]
```

Figure 10 Code of converting latitude and longitude into x-y coordinate system

- **Use shapely to find intersections**

After the conversion, we used shapely.geometry package to find the intersections. First, we create Polygon 1 with the building footprints. Then, Polygon 2 is generated based on the coordinates of the four corners of each grid cell. The intersects function determines if the two polygons have intersections and if so, the index of the intersected grid cell will be recorded in a list, as well as the corresponding intersected area.
The code for finding intersections is shown below.

```
# sweep all cells to find intersections between building footprints and the grids
# search for the intersection coordinates and calculate the area inside each grid cell
def runModel(i,x_footprint,y_footprint,population,lon0,lat0,output):
    writeFile=open('gridcell.csv', 'ab')
    writer = csv.writer(writeFile)
    rowlist1=[]
    rowlist2=[]
    vertex1=[]
    for j in range(len(x_footprint)):
        vertex1.append((x_footprint[j],y_footprint[j]))
    p1=Polygon(vertex1) # polygon of building footprint
    for j in range(len(population)):
        vertex2=[]
        # calculate the x, y coordinates of all grid cell vertices (unit: meter)
        lon1,lat1=population[j][0]-0.00083333333/2.0,population[j][1]-0.00083333333/2.0
        lon2,lat2=population[j][0]-0.00083333333/2.0,population[j][1]+0.00083333333/2.0
        lon3,lat3=population[j][0]+0.00083333333/2.0,population[j][1]+0.00083333333/2.0
        lon4,lat4=population[j][0]+0.00083333333/2.0,population[j][1]-0.00083333333/2.0
        vertex2.append((distance(lon0, lat0, lon1, lat0),distance(lon0, lat0, lon0, lat1)))
        vertex2.append((distance(lon0, lat0, lon2, lat0),distance(lon0, lat0, lon0, lat2)))
        vertex2.append((distance(lon0, lat0, lon3, lat0),distance(lon0, lat0, lon0, lat3)))
        vertex2.append((distance(lon0, lat0, lon4, lat0),distance(lon0, lat0, lon0, lat4)))
        p2=Polygon(vertex2) # polygon of grids
        res=p1.intersection(p2)
        if (p1.intersects(p2)) == True:
            print(str(j)+': Yes!')
            rowlist1.append(j)
            if (isinstance(res,multipolygon.MultiPolygon)):
                temp_area=0
                for re in res:
                    array = list(re.exterior.coords.xy)
                    x = array[0].tolist()
                    y = array[1].tolist()
                    temp_area=temp_area+PolyArea(x,y)
                rowlist2.append(temp_area)
            else:
                array = list(res.exterior.coords.xy)
                x = array[0].tolist()
                y = array[1].tolist()
                rowlist2.append(PolyArea(x,y))
    row=[i,rowlist1,rowlist2]
    writer.writerow(row)
    writeFile.close()
    output.put(row)
    return output
```

Figure 11 Code of finding intersected grid cells and area

- **Multiprocessing on supercomputer**

The 120,372 buildings need to sweep the 220,637 grid cells and it takes approximately 50 days to run the simulation with a normal workstation without multiprocessing. Hence, we used multiprocessing and separated the simulation into 12 jobs to run on CU Boulder's RMACC Summit Supercomputer[d]. It took less than 10 hours to finish the computation.
The code for multiprocessing is shown below.

[d] https://www.colorado.edu/rc/resources/summit

```
## record the start time
start = time.time()

## multi-processing
output = mp.Queue()
processes = [mp.Process(target=runModel,args=(i,x_footprint[i],y_footprint[i],population,lon0,lat0,output)) for i in range(len(x_footprint))]

## count the number of cpu
cpu = mp.cpu_count() #record the results including inputs and outputs
print cpu

model_results = []

run_times = math.floor(len(processes)/cpu)
if run_times > 0:
    for i in range(int(run_times)):
        for p in processes[i*int(cpu):(i+1)*int(cpu)]:
            p.start()
        print(1)
        for p in processes[i*int(cpu):(i+1)*int(cpu)]:
            p.join()

        #get the outputs
        temp = [output.get() for p in processes[i*int(cpu):(i+1)*int(cpu)]]

        for x in temp:
            model_results.append(x)
print(2)
for p in processes[int(run_times)*int(cpu):len(processes)]:
    p.start()
print(3)
for p in processes[int(run_times)*int(cpu):len(processes)]:
    p.join()
print(4)
## get the outputs
temp = [output.get() for p in processes[int(run_times)*int(cpu):len(processes)]]
for x in temp:
    model_results.append(x)

## record the end time
end = time.time()

return model_results,end-start
```

Figure 12 Code of multiprocessing

- **Sample results**

The generated results are in the following format. The first column shows the building index. The second column shows the grid cell indices that have intersections with this building. The third column shows the distributed building area in each of the intersected area.

0	[130449, 130450, 130903, 130904]	[63.76295232772827, 338.6571252346039, 107.19553661346436, 771.1611108779907]
1	[129994, 129995, 130449, 130450]	[51.701316833496094, 38.66186189651489, 22.14273762702942, 113.87061882019043]
2	[147962]	[109.55172681808472]
3	[93932, 93933, 93934, 94363, 94364	[132.76432752609253, 2633.3255207538605, 4634.679136276245, 1671.5942468643188,
4	[93932, 93933, 93934, 94363, 94364	[132.76432752609253, 2633.3255207538605, 4634.679136276245, 1671.5942468643188,
5	[75749]	[114.11651074886322]
6	[114272, 114759]	[416.93603253364563, 434.49438762664795]
7	[73784]	[120.31430947780609]
8	[73784]	[120.31430947780609]
9	[67344, 67345, 67744, 67745]	[173.35358428955078, 432.2187428474426, 137.39282727241516, 679.3933386802673]

Figure 13 Sample results of "gridcell.csv"

3.4 Distribute population to buildings

Based on the intersected grid cells found in the last step, this part deals with proportioning population into each building according to the total floor area. As shown in Figure 14, the whole procedure consists of three steps:

1. Assign buildings to its intersected grids based on building footprint and distribute building area into each grid (refer to Section 3.3). It is to note that, the total area calculated based on footprint does not equal to the floor area given by ORNL as floor number is not considered. Hence, we scaled the area and distributed the actual total floor area based on the area distribution coefficients in each grid.
2. Proportion daytime/nighttime population to each building inside each grid according to floor area. We adopted the number that "people spend 87% of their time in buildings". The equation used to proportion the population is as follows. Take Bldg 1 in Grid 1 in Figure 14 as an example. Assume only Bldg 1 and Bldg 2 have intersections with Grid 1.

$$People\ number\ in\ Bldg\ 1 = \frac{Total\ people\ in\ Grid\ 1 * 0.87}{\sum_{i=1}^{2} Floor\ area\ of\ Bldg\ i\ in\ Grid\ 1} * Floor\ area\ of\ Bldg\ 1\ in\ Grid\ 1$$

3. Add up the total population of each building inside its intersected grid cells. We also calculated the building people density by dividing the population by the total floor area.

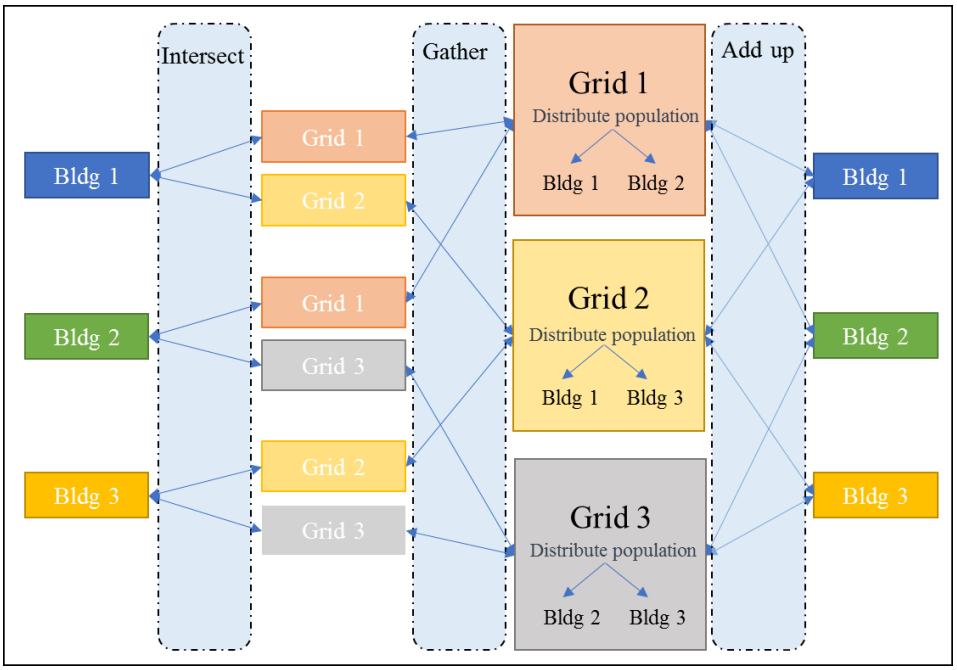

Figure 14 Diagram of the procedure to distribute population to buildings

The results of this part are put in the file "final.csv". The unit of population density is people/sq.m.

	day_pop	night_pop	day_density	night_density
0	111.8553	0	0.0875513	0
1	22.99466	17.4	0.10182993	0.077054437
2	1.74	0	0.0079569	0
3	183.86	0	0.00461507	0
4	91.93	0	0.00461507	0
5	1.74	0	0.00763643	0
6	12.18	4.35	0.00716407	0.002558595
7	24.07	0	0.20038891	0
8	48.14	0	0.20038891	0
9	33.06	21.75	0.01165261	0.007666192
10	31.32	0	0.00520549	0
11	134.85	0	0.81867275	0
12	1.74	6.96	0.01285216	0.051408643
13	5.22	16.53	0.04460853	0.141260332
14	2.61	8.7	0.00957856	0.031928545
15	2.61	8.7	0.01863945	0.062131503

Figure 15 Sample results of "final.csv"

3.5 Building re-index and final results

As we cleaned repeated building records in Section 3.2, in order to make the final results match the original building indices in "EPB_G2" file, we re-indexed the buildings in "final.csv". The re-indexed building data is saved in "reindex.csv" and has the following format.

	cent_lat	cent_lon	footprint	area[sq m]	day_pop	night_pop	day_density[people/sqm]	night_density[people/sqm]
0	35.07866	-85.187	35.07876/-	1277.597632	38.46407	0	0.030106561	0
1	35.07872	-85.1879	35.078733	225.8143802	8.568597	0.598934147	0.03794531	0.00265233
2	35.03442	-85.155	35.034369	218.6780842	0.454484	0	0.002078325	0
3	35.03442	-85.155	35.034369	218.6780842	0.454484	0	0.002078325	0
4	35.03442	-85.155	35.034369	218.6780842	0.454484	0	0.002078325	0
5	35.14712	-85.252	35.148437	39839.07622	178.6122	0	0.004483342	0
6	35.14712	-85.252	35.148437	39839.07622	178.6122	0	0.004483342	0

Figure 16 Sample results of "reindex.csv"

4 Conclusion

In this report, the methodology and detailed techniques of conducting Virtual EPB Task 0b: Enhanced Occupancy are introduced. The final deliverables of this task include 5 main codes, 1 assistive code, 3 input files, 7 output csv files, 1 technical report, and 1 README file. The detailed descriptions of the above files are available in the README file.

The final results are presented in a csv file ("reindex.csv") and include the daytime/nighttime occupant number, as well as occupant density of every provided building in the utility area. With the generated results of this task, readers should be able to further input these data into their building models and conduct quantitative analysis on the impact of occupancy-enhanced schedules on building energy simulation results.

Author Contributions: Conceptualization, J.N., W.Z.; methodology, J.W., Y.Y., J.N.; software, J.W., Y.Y.; validation, J.W., Y.Y.; resources, W.Z., J.N.; data curation, J.W., Y.Y., J.N., A.R.; writing—original draft preparation, J.W.; writing—review and editing, J.W., Y.Y., W.Z., J.N., A.R.; visualization, J.W.; supervision, W.Z., J.N.; project administration, J.N.; funding acquisition, J.N. All authors have read and agreed to the published version of the manuscript.

Funding: This work was funded by field work proposal CEBT105 under US Department of Energy Building Technology Office Activity Number BT0305000, as well as Office of Electricity Activity Number TE1103000.

Acknowledgments: The authors would like to thank Amir Roth and Madeline Salzman for their support and review of this project. The authors would also like to thank Mark Adams for his contributions to the AutoBEM software for model generation.