UBEM

Fatima Johari

Vesrion: May 2022

Contents

1 Installation and dependencies

This tool has been developed around the building simulation software EnergyPlus using the programming language Python. To run the tool for the first time, it is necessary to make sure that all the required dependencies of the tool have been already installed properly. EnergyPlus is an open-source building energy simulation software for estimating the energy use in buildings. As of Jun 2022, the developed tool has been tested for EnergyPlus versions of 9.1.0 to 9.6.0. Although the user has the possibility to choose the version of the EnergyPlus in the input of the tool, it is recommended to use the EnergyPlus version 9.1.0 that is available to download from this link:

- EnergyPlus 9.1.0

The main python packages that are used in this tool are,

- eppy,
- geopandas,
- io,
- itertools,
- math,
- matplotlib,
- multiprocessing,
- numpy,
- pandas,
- shapely,
- time.

Additionally, there are more packages that are possibly required.These packages include,

- fiona,
- openpyxl,
- pyproj,
- rtree.

2 Tutorials

After the installation is completed, from the "UBEM" folder, open the "main.py" file. By simply running the file, the input data are read, the underlying functions are called and the building models are made and simulated for the given weather and location.

Should the model starts running, this text is seen on the console:

```
SIMULATION IS RUNNING ...

Building 0/50
C:\EnergyPlusV9-1-0\energyplus.exe
– weather-input data\epwWeatherData.epw
– output-directory \UBEM \
– idd C:\EnergyPlusV9-1-0 \Energy+.idd \UBEM \in.idf
```

It is going to take about a few seconds to run the simulation for each building. When the simulation is done, the total computation time as well as the computation time for each building is given on the console. For example:

```
SIMULATION IS DONE!

—– Running time: 0 minutes and 47 seconds
—– Running time per building: 5.00 seconds
```

With completion of the simulation, the results for energy use of the building is written in an excel file. The file is saved under the UBEM \output folder.

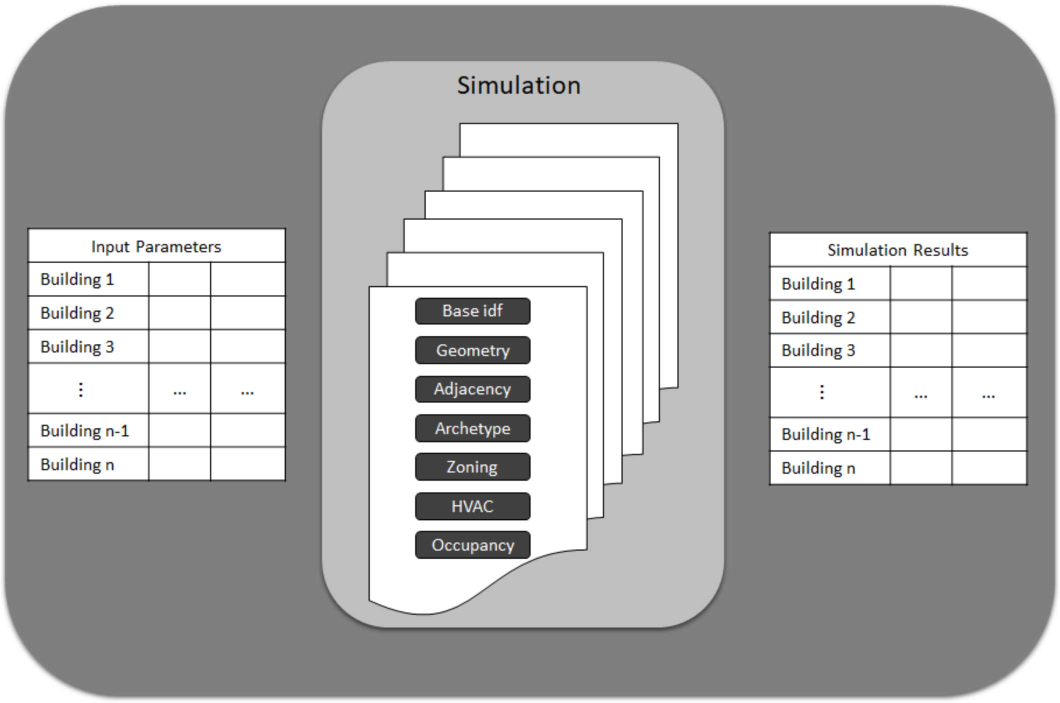
3 Model framework

The following sections describe the method for development of an UBEM which can be used for modeling and simulation of buildings at different spatial and temporal resolutions. This model makes use of Python to automate the process of energy modeling and simulation of buildings using EnergyPlus.

As of May 2022, the model works based on EnergyPlus version 9.1 and Python 3.8. The model makes use of different packages in Python, in particular, **eppy**, which is used for working with EnergyPlus idf files from the Python environment. In order to run the model, in addition to EnergyPlus, all the required packages including, eppy, geopandas, shaply have to be installed.

In general, in building energy modeling, a model is comprised of four main components, including ”geometry”, ”construction and material”, ”system and schedule” and finally, ”occupancy and use”. In this model, the geometry of buildings, i.e., building footprints and height, are read from GIS data and converted to 3D surfaces with x,y,z vertices. These surfaces form boundaries of thermal zones of a building. Depending on the type and year of construction of the buildings, i.e., archetypes, information about construction and material is exported from a predefined dataset. This information is used for characterizing the surfaces of the thermal zones. Type of heating, ventilation and air conditioning (HVAC) systems are given in the EPC dataset. For buildings without EPC, standard systems are added to the building model. Regarding the occupancy and use of the building, probabilistic profiles made from urban mobility data and stochastic occupancy models are used.

In this model, defining and processing all the mentioned component of a building energy model is done using a model structure that is illustrated in Figure 1. Description of the all different components of this model is found as follows.



3.1 Base idf

Every simulation in EnergyPlus requires a set of initial parameters, without which the simulation fails to start. These parameters are defined in a file called "base_idf.py". This file includes a series of functions belong to a python class, "baseIDF". This class generates an intermediate data format (IDF) file including all the initial parameters to be used as a basis for the UBEM simulation using EnergyPlus. This class includes the following functions:

- blank_idf
  This function initializes an empty EnergyPlus IDF file, based on the version of the EnergyPlus, and the EnergyPlus weather data (EPW) that are going to be used. In this version of the work, the model is compatible with EnergyPlus versions 9.1.0 to 9.6.0.

```
def blank_idf(epw, version = 9.1):
    """
    Creates a blank IDF.

    parameters
    ----------
    epw: epw
        EnergyPlus weather file.
    version: float
        Version of the EnergyPlus software.

    output
    ----------
    idf
        An empty idf file.
    """

    idftxt = "" # Empty string.
    fhandle = StringIO(idftxt) # Handles the empty string.
    idf = IDF(fhandle, epw) # Converts the empty string to an empty idf.

    # Specifies and adds the version of EnergyPlus to the empty idf.
    idf.newidfobject("VERSION",
                     Version_Identifier = version)
    return idf
```

- Simulation parameters
  This function initializes two important parts of the simulation. One is to conduct the auto-sizing of the thermal zone, systems or plants, when it is necessary (depends on the heating system). As default, for an ideal heating system, it is deactivated. The other describes the parameters that are used during the simulation of buildings, For example, the basis for computation of solar radiation distribution in buildings.

```
def simulation_parameters(idf, zoneSizing = 'No', systemSizing = 'No', plantSizing = 'No',
                          epwRunPeriod = 'Yes'):

    """
    Sets up the simulation parameters for sizing systems and modeling building.

    parameters
    ------
    idf
    zoneSizing: str
    systemSizing: str
    plantSizing: str

    output
    ------
    idf
    """
    idf.newidfobject("SIMULATIONCONTROL",
                     Do_Zone_Sizing_Calculation = zoneSizing,
                     Do_System_Sizing_Calculation = systemSizing,
                     Do_Plant_Sizing_Calculation = plantSizing,
                     Run_Simulation_for_Sizing_Periods = "No",
                     Run_Simulation_for_Weather_File_Run_Periods = epwRunPeriod )

    idf.newidfobject("BUILDING",
                     Name = "Building",
                     North_Axis = 0,
                     Terrain = "City",
                     Solar_Distribution = "FullExterior")

    return idf
```

- Simulation period and time step
  This function specifies the simulation period and time step. The simulation time step, known as ”zone time step”, is generally used in performing the heat balance algorithms of the thermal zone. The suggested value from EnergyPlus is 6, meaning 6 time steps in every hour, i.e, 10 min. This number can very from 1 to 60. As defult, the simulation runs for a whole year starting from 1 Jan to 31 Dec. The simulation time step is also set to be 6.

```
def simulation_period (idf, timeStep = 6, startMonth = 1, startDay = 1, endMonth = 12, endDay = 31,
                               firstDayWeek = 'Monday')

    """
    Sets the simulation time and time step.

    parameters
    ------
    idf

    timeStep: 1< int <60
       Sub-hour simulation time steps.
    startMont: 1< int <12
        Begin month of the simulation.
    startDay: 1< int <31
        Begin day of the simulation.
    endMonth: 1< int <12
        End month of the simulation.
    endDay: 1< int <31
        End day of the simulation.
    firstDayWeek: str
        Begin day of the week.

    output
    ------
    idf
    """

    idf.newidfobject('RUNPERIOD',
                      Name = 'RUNPERIOD',
                      Begin_Month = startMonth,
                      Begin_Day_of_Month = startDay,
                      End_Month = endMonth,
                      End_Day_of_Month = endDay,
                      Day_of_Week_for_Start_Day = firstDayWeek)

    idf.newidfobject('TIMESTEP',
                      Number_of_Timesteps_per_Hour = timeStep)
    return idf
```

- Geographical location of the building.
  Information about the geographical location of the building, including, city, latitude and longitude, time zone and height above the sea level are automatically exported from the available data in the weather data file. Therefore, it is not related to specific buildings but rather for the whole city.

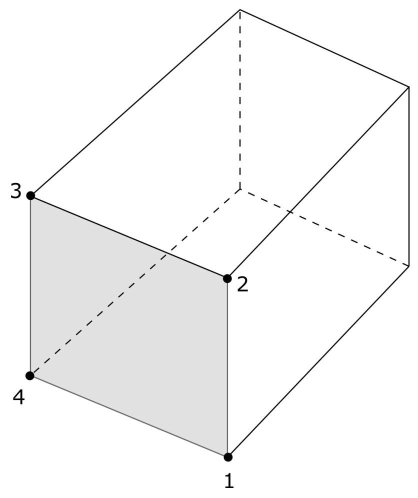
```
def site_location(idf, epw):

    """
    Sets up information of the location using the weather data file.

    parameters
    ----------
    idf
    epw
        EnergyPlus weather data file.

    output
    ----------
    idf
    """

    location = pd.read_csv(epw , nrows=0) # Reads the weather data file.

    city = location.columns[1] # City or location of the site.
    latitude = location.columns[6] # Latitude of the site.
    longitude = location.columns[7] # Longitude of the site.
    timeZone = location.columns[8] # Time zone.
    elevation = location.columns[9] # Elevation of the site.

    idf.newidfobject('SITE:LOCATION',
                     Name = city,
                     Latitude = latitude,
                     Longitude = longitude,
                     Time_Zone = timeZone,
                     Elevation = elevation)

    return idf
```

- Simulation output data.
  Here, the output parameters, type of report and reporting time step for the results are defined. In this version of the model, annual and hourly energy demand for household electricity, space heating and hot water use are found in the output from the model. In addition to the energy related parameters, the mean air temperature of the room as well as the site air temperature are also reported in the model output. These data are later used in the model for designing the HVAC systems.

```
def simulation_output(idf)

    """
    Chooses the simulation output parameters.

    parameters
    ------
    idf

    output
    ------
    html, csv:
       Annual as well as hourly results for household electricity, space heating and hot water use in buildings.
       Mean air temperature of the zone as well as ambient temperature of the site.
    """

    idf.newidfobject('OUTPUT:VARIABLEDICTIONARY',
                     Key_Field = 'Regular')

    idf.newidfobject('OUTPUT:TABLE:SUMMARYREPORTS',
                     Report_1_Name = 'AllSummary')

    idf.newidfobject('OUTPUTCONTROL:TABLE:STYLE',
                     Column_Separator= 'comma')

    idf.newidfobject('OUTPUT:METER',
                     Key_Name= 'Heaing:DistrictHeating',
                     Reporting_Frequency = 'Hourly')

    idf.newidfobject('OUTPUT:METER',
                     Key_Name= 'WaterSystems:DistrictHeating',
                     Reporting_Frequency = 'Hourly')

    idf.newidfobject('OUTPUT:METER',
                     Key_Name= 'Electricity:Building',
                     Reporting_Frequency = 'Hourly')

    idf.newidfobject('OUTPUT:METER',
                     Key_Name= 'Zone Mean Air Temperature',
                     Reporting_Frequency = 'Hourly')

    idf.newidfobject('OUTPUT:METER',
                     Key_Name= 'Site Outdoor Air Drybulb Temperature',
                     Reporting_Frequency = 'Hourly')

    return idf
```

3.2 Geometry

In EnergyPlus, it is crucial to follow the global geometric rules. These rules specify the starting vertex and vertex entry direction of the building surfaces. In this model, it is chosen to read the polygons from their lower right corner and continue counterclockwise (as illustrated in figure below).



The information about the chosen global geometry rules has be written into the idf file. This is done using a function for writing an EnergyPlus idf object for the global geometry rules.

```
def global_geometry_rules(idf):

"""
Defines the global geometry rule for the idf file.

parameters
-----
idf

output
-----
idf
"""

idf.newidfobject ('GLOBALGEOMETRYRULES',
                  Starting_Vertex_Position = 'LowerRightCorner',
                  Vertex_Entry_Direction = 'Counterclockwise',
                  Coordinate_System = 'Relative')

return idf
```

However, not all the building polygons obtained from the GIS data follow these rules. The function below, assures that all the buildings imported to the model meet these requirements. This function rearranges the building polygon according to the specified geometry rules.

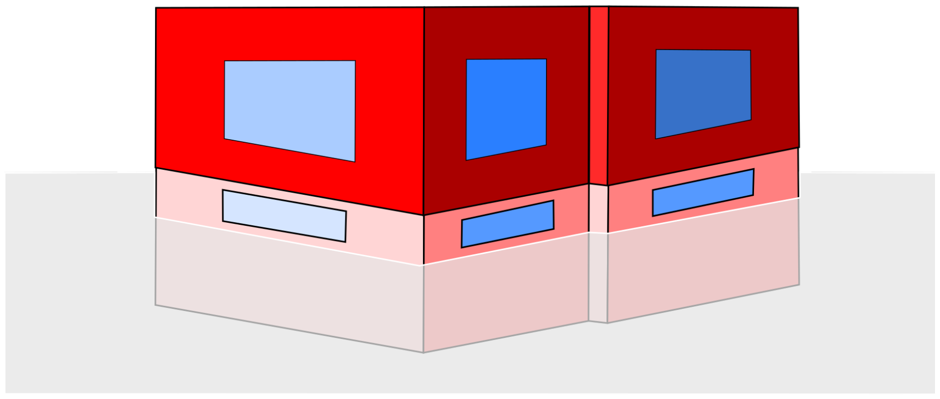
```
def global_geometry(polygon):

"""
Reconstructs the polygon according to the geometry rules specified by EnergyPlus.

parameters
----------
polygon: shapely polygon
   Surfaces of the building.

output
----------
polygon: shapely polygon
   Reconstructed surfaces of the building.
"""

# Gets x and y coordinates of the polygon.
x, y = polygon.exterior.xy

# Finds the lowest point of a polygon, based on min (y).
for a, b in zip(x,y):
   if b == min(y):
      firstVertex = Point(a,b)

# Reconstructs the polygon counter clockwise starting from the lowest point.
perimeter = polygon.exterior.coords
twoRounds = itertools.chain(perimeter, perimeter)

newCoordinates = []
for v in twoRounds:
   if Point(v) == firstVertex:
         newCoordinates.append(v)
         while len(newCoordinates) < len(perimeter):
               newCoordinates.append(next(twoRounds))
         break

polygon = Polygon(newCoordinates)

# Simplifies the polygon in case of repeated points.
polygon = polygon.simplify(0.5, preserve_topology= True)

return polygon
```

3.3 Zoning

In this version of the model, each building is divided into one thermal zone. Geometry of the thermal zone is therefore, equivalent to the geometry of the building. With this assumption, every thermal zone enclosed by external walls, roof and floor. The external walls and roof are all exposed to the ambient, i.e., wind and sun, unless otherwise stated. For attached or semi-detached buildings, the adjacent wall(s) is considered as unexposed. In this case, it is assumed that no heat is transferred between the two buildings and therefor the adjacent wall is labeled as adiabatic. Adjacency of the buildings is found from their geospatial data and a predefined class with the same name. Description of this class is found in Section 3.4. The external floor is also in contact with the ground. If, according to the EPCs, the building has a heated basement, one extra thermal zone is added to the model. In this case, one internal floor/ceiling separates the two thermal zones from each other. By default, the height of the basement was set to 2.3 m of which 1 m is on the ground and 1.3 m is below the ground. The underground surfaces of the basement are assumed to be adiabatic while the on ground share of the walls are exposed to the ambient.

The external windows of a building (both for the basement and the main zone) are defined as a ratio of the floor area. In general, to avoid unrealistic window areas on external walls of a building with complex constructions, no window is assigned to the walls with area of less than 2 m^2.

Figure 3 illustrated the method for zoning configuration of building models.



In order to write information of the building thermal zone in the idf file, functions and a class found in "zoning.py" is used. This file includes as follows:

- Zoning initialization

In EnergyPlus, a thermal zone first needs to be initialized. This is done based on a limited number of variables such as a name that is given to the zone, i.e., an identifier for later calling back the zone in the other EnergyPlus objects, the coordinates of the origin, etc.

```
def zone (idf, zoneName):
    """
    Initializes the calculations of the thermal zone in EnergyPlus.

    parameters
    ——
    idf
    zoneName:str
       Name of the zone, it can be either 'ZONE', or 'BASEMENT'.

    output
    ——
    idf
    """

    idf.newidfobject('ZONE',
                     Name = zoneName,
                     Ceiling_Height = 'autocalculate',
                     Volume = 'autocalculate',
                     Floor_Area = 'autocalculate',
                     Part_of_Total_Floor_Area = 'yes')

    return idf
```

- Thermal Zone development

This function develops a thermal zone of each building based on the described assumptions. The building surfaces, e.g., walls, windows, etc., that is used in this function are defined separately and just called here.

```
def onezone_building_surfaces(buildingPolygon, buildingHeight, basement, wwr, adjacency):
    """
    Gets the geometry of the building and write in idf file for the building surfaces.

    parameters
    ——
    buildingPolygon: shapely polygon
      Building footprint.
    buildingHeight: int
      Height of the building.
    basement: int
      Number of under ground floors from the EPC.
    wwr: int
      Window to wall ratio.
    adjacency: gpd.GeoDatFrame
      A dataframe with the adjacent buildings.
    output
    ——
    idf
      An idf for building surfaces.
    """
    height0 = 0 # Height of the ground level.
    heightbExposed = 1  height of basement exposed walls on ground.
    heightbAdiabatic = -1.3  height of the under ground floors.
    wwrb = wwr

    # Convert the geometry of the building polygon to arrays of x and y.
    x,y = buildingPolygon.exterior.xy

    # Reverse the order for adjacent floor and ceiling.
    xx,yy = x[::-1],y[::-1]

    idftxt = ""
    # If there is one or more adjacent walls.
    if len(adjacency) > 0:
      for i in range(len(x)-1):

        # Find the center of the walls.
        point = LineString([(x[i],y[i]),(x[i+1],y[i+1])]).centroid

        adj = []
        for j in range (len(adjacency)):

          p = Polygon(adjacency.geometry[j])
          # Calculate the distance of adjacency.
          adj.append( point.distance(p) )

        # If the distance of adjacency is less than 20 cm, it is an adiabatic wall,
        # otherwise a a normal heat transmittance external wall.
```

continues ...

```
        if np.array(adj).any() < 0.2:
          # Adiabatic walls with no windows.
          idftxt += (adiabatic_wall(x[i],x[i+1],y[i],y[i+1],height0,buildingHeight,i))
        else:
          # External Walls
          idftxt += (external_wall(x[i],x[i+1],y[i],y[i+1],height0,buildingHeight,i))

          # Get the length of a wall
          line = LineString([Point((x[i],y[i])), Point((x[i+1],y[i+1]))])

          # Add windows if the length of the wall is greater than 2 meters.
          # Avoid having windows on smaller walls of complex building shapes.
          if line.length > 2:
            # Calculate the geometry of windows based on the geometry of the walls and WWR.
            a = line.interpolate(wwr, normalized=True)
            b = line.interpolate(1-wwr, normalized=True)
            R = (buildingHeight - buildingHeight * math.sqrt(wwr))/2

            # External Windows
            idftxt += (external_win(a.x,b.x,a.y,b.y, height0+R, buildingHeight-R, i))

  # If there is NO adjacent walls.
  if len(adjacency) == 0:
    for i in range(len(x)-1):
      # External Walls
      idftxt += (external_wall(x[i],x[i+1],y[i],y[i+1],height0,buildingHeight,i))

      # Get the length of a wall
      line = LineString([Point((x[i],y[i])), Point((x[i+1],y[i+1]))])

      # Add windows if the length of the wall is greater than 2 meters.
      # Avoid having windows on smaller walls of complex building shapes.
      if line.length > 2:
        # Calculates the geometry of windows based on the geometry of the walls and WWR.
        a = line.interpolate(wwr, normalized=True)
        b = line.interpolate(1-wwr, normalized=True)
        R = (buildingHeight - buildingHeight * math.sqrt(wwr))/2
        # External Windows
        idftxt += (external_win(a.x,b.x,a.y,b.y, height0+R, buildingHeight-R, i))

  # External Floor
  idftxt += floor(x, y, height0, basement)

  # External Roof
  idftxt += roof(xx, yy, buildingHeight)

  # If the building has a basement floor.
  if basement == 1:
    for i in range(len(x)-1):
      # Basement adiabatic walls (underground walls)
      idftxt += basement_adiabatic_wall(x[i], x[i+1], y[i], y[i+1], height0, heightbAdiabatic, i)

      # Basement exposed walls (onground walls)
      idftxt += basement_external_wall(x[i], x[i+1], y[i], y[i+1], height0, heightbExposed, i)
```

continues ...

```
            # Gets the length of a wall
            line = LineString([Point((x[i],y[i])), Point((x[i+1],y[i+1]))])

            # Adds windows if the length of the wall is greater than 2 meters.
            # Avoids having windows on smaller walls of complex building shapes.
            if line.length > 2:

                # Calculates the geometry of windows based on the geometry of the walls and WWR.
                a = line.interpolate(wwrb, normalized=True)
                b = line.interpolate(1-wwrb, normalized=True)
                Rb = (heightbExposed - heightbExposed * math.sqrt(wwrb))/2

                # External Windows
                idftxt += (external_win(a.x,b.x,a.y,b.y, height0+Rb, heightbExposed-Rb, i,'Basement'))

        # Basement ceiling and floor.
        idftxt += basement_ceiling(xx, yy, height0)
        idftxt += basement_floor(x, y, heightb)

    # Convert the text file to an idf.
    fhandle = io.StringIO(idftxt)
    idf = IDF(fhandle)

    return idf
```

- Building surfaces

A set of functions defines the building surfaces, including their geometry (coordinates), construction and exposure to the ambient. These surfaces are used in the previous function to form the thermal zone.

Here, all the information is written as strings (normal text). Later, these texts are compiled together and converted to an idf. However, it is still important to write the text with the style that is written in an EnrgyPlus idf file.

```
def external_wall(x0,x1,y0,y1,z0,z1, i):
    """
    Writes the information of the external walls.

    parameters
    ----
    x0,x1,y0,y1,z0,z1: int
       The vertices of the wall as illustrated below:

          (x1,y1,z1)        (x0,y0,z1)
                * * * * * * * **
                *               *
                *               *
                *               *
                * * * * * * * **
          (x1,y1,z0)        (x0,y0,z0)

    i: int
       Number of external wall in the building

    output
    ----
    str
    """

    wallVertices = 4
    wall = ""
    wall += ('BUILDINGSURFACE:DETAILED,'+'\'+
                'Wall'+str(i)+',\'+     #!- Name
                'Wall, \'+     #!- Surface Type
                'Exterior Wall, \'+     #!- Construction Name
                'ZONE'+',\'+    #!- Zone Name
                'Outdoors, \'+    #!- Outside Boundary Condition
                ', \'+    #!- Outside Boundary Condition Object
                'SunExposed, \'+    #!- Sun Exposure
                'WindExposed, \'+    #!- Wind Exposure
                ',\' +    #!- View Factor to Ground
                str(wallVertices)+',\')    #!- Number of Vertices

    # Vertices of the wall.
    wall += (str(x0)+',\' + str(y0) + ',\' + str(z0) + ',\' +
                str(x0)+',\' + str(y0) + ',\' + str(z1) + ',\' +
                str(x1)+',\' + str(y1) + ',\' + str(z1) + ',\' +
                str(x1)+',\' + str(y1) + ',\' + str(z0) + ';\n' )

    return wall
```

```
def adiabatic_wall(x0, x1, y0, y1, z0, z1, i):
"""
Writes the information of the adjacent walls. Assumed to be adiabatic.

parameters
——
x0,x1,y0,y1,z0,z1: int
   The vertices of the wall.
i: int
   Number of external wall in the building

output
——
str
"""

wallVertices = 4
wall = ""
wall += ('BUILDINGSURFACE:DETAILED,'+'\n'+
              'Wall'+str(i)+',\n'+    #!- Name
              'Wall, \n'+    #!- Surface Type
              'Exterior Wall, \n'+    #!- Construction Name
              'ZONE'+',\n'+    #!- Zone Name
              'Adiabatic, \n'+    #!- Outside Boundary Condition
              ', \n'+    #!- Outside Boundary Condition Object
              'NoSun, \n'+    #!- Sun Exposure
              'NoWind, \n'+    #!- Wind Exposure
              ',\n' +    #!- View Factor to Ground
              str(wallVertices)+',\n')    #!- Number of Vertices

# Vertices of the wall.
wall += (str(x0)+',\n' + str(y0) + ',\n' + str(z0) + ',\n' +
              str(x0)+',\n' + str(y0) + ',\n' + str(z1) + ',\n' +
              str(x1)+',\n' + str(y1) + ',\n' + str(z1) + ',\n' +
              str(x1)+',\n' + str(y1) + ',\n' + str(z0) + ';\n' )

return wall
```

```
def external_win(x0, x1, y0, y1, z0, z1, i, zoneName):
 """
 Writes the information of the external windows.

 parameters
 ——
 x0,x1,y0,y1,z0,z1: int
 The vertices of the windows on external walls.
 i: int
    Number of external windows in the building.
 zoneName:str
    Name of the zone, it can be either 'ZONE', or 'BASEMENT'.

 output
 ——
 str
 """
 winVertices = 4
 win = ""
 win += ('FENESTRATIONSURFACE:DETAILED,'+'\n'+
               'Window'+zoneName+str(i)+',\n'+    #!- Name
               'Window, \n'+    # !- Surface Type
               'Exterior Window, \n'+    # !- Construction Name
               'Wall'+ str(i)+',\n'+    # !-Building Surface Name
               ', \n'+
               ', \n'+
               ', \n'+
               ', \n'+
               str(winVertices)+',\n')

 win += (str(x0) + ',\n' + str(y0) + ',\n' + str(z0) + ',\n' +
               str(x0) + ',\n' + str(y0) + ',\n' + str(z1) + ',\n' +
               str(x1) + ',\n' + str(y1) + ',\n' + str(z1) + ',\n' +
               str(x1) + ',\n' + str(y1) + ',\n' + str(z0) + ';\n' )

 return win
```

```
def roof(x,y,z):
 """
 Writes information of the external roof.

 Parameters
 ——
 x,y,z: int
    The vertices of the external roof.

 output
 ——
 str
 """

 roofVertices = len(x)-1
 roof = ""
 roof += ('BUILDINGSURFACE:DETAILED,'+'\n'+
               'Roof'+',\n'+      #!- Name
               'Roof, \n'+      #!- Surface Type
               'Exterior Roof, \n'+      #!- Construction Name
               'ZONE'+',\n'+      #!- Zone Name
               'Outdoors, \n'+      #!- Outside Boundary Condition
               ', \n'+      #!- Outside Boundary Condition Object
               'SunExposed, \n'+      #!- Sun Exposure
               'WindExposed, \n'+      #!- Wind Exposure
               ',\n' +      #!- View Factor to Ground
               str(roofVertices)+',\n')      #!- Number of Vertices

 for i in range(roofVertices-1):
    roof += (str(x[i]) + ',\n '+
                  str(y[i]) + ',\n' +
                  str(z) + ',\n' )

 roof += (str(x[-2]) + ',\n' +
               str(y[-2]) + ',\n' +
               str(z) + ';\n' )

 return roof
```

```
def floor(x, y, z0, basement):
    """
    Writes information of the floor.
    It is an external floor unless the building has a basement.
    Then, the floor is regarded as the internal floor.

    parameters
    ----
    x,y: nd.array
    z: int
        The vertices of the floor.
    basement: int, 0 or 1        Information on the basement.
    output
    ----
    str
    """
    floorVertices = len(x)-1
    floor = ""

    if basement == 1:
        floor += ('BUILDINGSURFACE:DETAILED,'+'\n'+
                    'Interior Floor'+ ',\n'+     #!- Name
                    'Floor, \n'+     #!- Surface Type
                    'Interior Floor, \n'+     #!- Construction Name
                    'ZONE'+',\n'+     #!- Zone Name
                    'Surface, \n'+     #!- Outside Boundary Condition
                    'Interior Ceiling Basement'+',\n'+     #!- Outside Boundary Condition Object
                    'NoSun, \n'+     #!- Sun Exposure
                    'NoWind, \n'+     #!- Wind Exposure
                    ',\n' +     #!- View Factor to Ground
                    str(floorVertices)+',\n')     #!- Number of Vertices
    else:
        floor += ('BUILDINGSURFACE:DETAILED,'+'\n'+
                    'Floor'+',\n'+     #!- Name
                    'Floor, \n'+     #!- Surface Type
                    'Exterior Floor, \n'+     #!- Construction Name
                    'ZONE'+',\n'+     #!- Zone Name
                    'Ground, \n'+     #!- Outside Boundary Condition
                    ', \n'+     #!- Outside Boundary Condition Object
                    'NoSun, \n'+     #!- Sun Exposure
                    'NoWind, \n'+     #!- Wind Exposure
                    ',\n' +     #!- View Factor to Ground
                    str(floorVertices)+',\n')     #!- Number of Vertices

    for i in range(floorVertices-1):
        floor += (str(x[i]) + ',\n' +
                    str(y[i]) + ',\n' +
                    str(z0) + ',\n')

    floor += (str(x[-2]) + ',\n' +
                str(y[-2]) + ',\n' +
                str(z0) + ';\n' )

    return floor
```

```
def basement_ceiling(x, y, z0):
    """
    Writes information of the basement ceiling.

    parameters
    ——
    x,y: nd.array
    z0: int
       The vertices of the ceiling of the basement.

    output
    ——
    str
    """
    basementCeilingVertices = len(x)-1

    basementCeiling = ""
    basementCeiling += ('BUILDINGSURFACE:DETAILED,'+'\n'+
                              'Interior Ceiling Basement'+',\n'+     #!- Name
                              'Ceiling, \n'+     #!- Surface Type
                              'Interior Ceiling, \n'+     #!- Construction Name
                              'BASEMENT,\n'+     #!- Zone Name
                              'Surface, \n'+     #!- Outside Boundary Condition
                              'Interior Floor'+', \n'+     #!- Outside Boundary Condition Object
                              'NoSun, \n'+     #!- Sun Exposure
                              'NoWind, \n'+     #!- Wind Exposure
                              ',\n' +     #!- View Factor to Ground
                              str(basementCeilingVertices)+',\n')     #!- Number of Vertices

    for i in range(basementCeilingVertices-1):
       basementCeiling += (str(x[i]) + ',\n' +
                                str(y[i]) + ',\n' +
                                str(z0) + ',\n' )

    basementCeiling += (str(x[-2]) + ',\n' +
                                str(y[-2]) + ',\n' +
                                str(z0) + ';\n' )

    return basementCeiling
```

```
def basement_external_wall(x0, x1, y0, y1, z0, zb, i):
    """
    Writes information of the basement external floor.

    parameters
    ——
    x0, x1, y0, y1, z0, zb: int
       The vertices of the external walls of the basement.
    i: int
       Number of the walls.

    output
    ——
    str
    """
    basementWall = ""
    basementWall =('BUILDINGSURFACE:DETAILED,'+'\n'+
                        str('BasementWall'+ str(i))+ ',\n'+      #!- Name
                        'Wall, \n'+      #!- Surface Type
                        'Exterior Wall, \n'+      #!- Construction Name
                        'BASEMENT, \n'+      #!- Zone Name
                        'OtherSideCoefficients, \n'+      #!- Outside Boundary Condition
                        'bWall, \n'+      #!- Outside Boundary Condition Object
                        'NoSun, \n'+      #!- Sun Exposure
                        'NoWind, \n'+      #!- Wind Exposure
                        ',\n' +      #!- View Factor to Ground
                        '4,\n')      #!- Number of Vertices

    basementWall += (str(x0) + ',\n' + str(y0) + ',\n' + str(zb) + ',\n' +
                          str(x0) + ',\n' + str(y0) + ',\n' + str(z0) + ',\n' +
                          str(x1) + ',\n' + str(y1) + ',\n' + str(z0) + ',\n' +
                          str(x1) + ',\n' + str(y1) + ',\n' + str(zb) + ';\n' )

    return basementWall
```

```
def basement_floor(x, y, zb):
    """
    Writes information of the basement external floor.

    parameters
    -----
    x,y: nd.array
    zb: int
       The vertices of the ceiling of the basement.

    output
    -----
    str
    """
    basementFloorVertices = len(x)-1

    basementFloor = ""
    basementFloor += ('BUILDINGSURFACE:DETAILED,'+'\n'+
                      'Exterior Floor, \n'+ #!- Name
                      'Floor, \n'+ #!- Surface Type
                      'Exterior Floor, \n'+ #!- Construction Name
                      'BASEMENT, \n'+ #!- Zone Name
                      'OtherSideCoefficients, \n'+ #!- Outside Boundary Condition
                      'bFloor, \n'+ #!- Outside Boundary Condition Object
                      'NoSun, \n'+ #!- Sun Exposure
                      'NoWind, \n'+ #!- Wind Exposure
                      ',\n' + #!- View Factor to Ground
                      str(basementFloorVertices)+',\n') #!- Number of Vertices

    for i in range(basementFloorVertices-1):
    basementFloor += (str(x[i]) + ',\n' +
                      str(y[i]) + ',\n' +
                      str(zb) + ',\n' )

    basementFloor += (str(x[-2]) + ',\n' +
                      str(y[-2]) + ',\n' +
                      str(zb) + ';\n' )

    return basementFloor
```

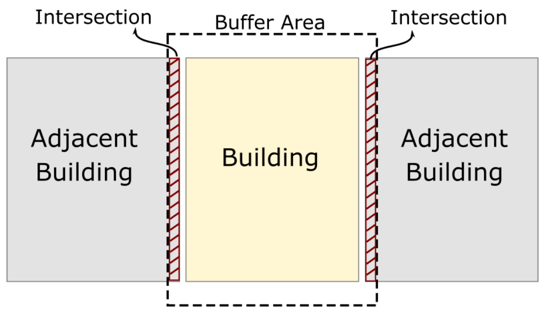
```
def basement_adiabatic_wall(x0, x1, y0, y1, z0, zb, i):
    """
    Writes information of the basement adiabatic external wall.

    parameters
    ---------
    x0,x1,y0,y1,z0,zb: int
       The vertices of the adiabatic walls of the basement.
    i: int
       Number of the walls.

    output
    ------
    str
    """

    basementWall = ""
    basementWall =('BUILDINGSURFACE:DETAILED,'+'\n'+
                        str('WallBasementAdiabatic'+ str(i))+ ',\n'+ #!- Name
                        'Wall, \n'+ #!- Surface Type
                        'Exterior Wall Basement, \n'+ #!- Construction Name
                        'BASEMENT, \n'+ #!- Zone Name
                        'Adiabatic, \n'+ #!- Outside Boundary Condition
                        ',\n'+ #!- Outside Boundary Condition Object
                        'NoSun, \n'+ #!- Sun Exposure
                        'NoWind, \n'+ #!- Wind Exposure
                        ',\n' + #!- View Factor to Ground
                        '4,\n') #!- Number of Vertices

    basementWall += (str(x0) + ',\n' + str(y0) + ',\n' + str(zb) + ',\n' +
                         str(x0) + ',\n' + str(y0) + ',\n' + str(z0) + ',\n' +
                         str(x1) + ',\n' + str(y1) + ',\n' + str(z0) + ',\n' +
                         str(x1) + ',\n' + str(y1) + ',\n' + str(zb) + ';\n' )

    return basementWall
```

```
def basement_external_wall(x0, x1, y0, y1, z0, z1, i):
    """
    Writes information of the basement exposed external wall.

    parameters
    ----------
    x0,x1,y0,y1,z0,z1: int
       The vertices of the exposed walls of the basement.
    i: int
       Number of the walls.

    output
    ------
    str
    """

    basementWall = ""
    basementWall =('BUILDINGSURFACE:DETAILED,'+'\n'+
                        str('WallBasement'+ str(i))+ ',\n'+ #!- Name
                        'Wall, \n'+ #!- Surface Type
                        'Exterior Wall Basement, \n'+ #!- Construction Name
                        'BASEMENT, \n'+ #!- Zone Name
                        'Outdoors, \n'+ #!- Outside Boundary Condition
                        ',\n'+ #!- Outside Boundary Condition Object
                        'SunExposed, \n'+ #!- Sun Exposure
                        'WindExposed, \n'+ #!- Wind Exposure
                        ',\n' + #!- View Factor to Ground
                        '4,\n') #!- Number of Vertices

    basementWall += (str(x0) + ',\n' + str(y0) + ',\n' + str(z0) + ',\n' +
                          str(x0) + ',\n' + str(y0) + ',\n' + str(z1) + ',\n' +
                          str(x1) + ',\n' + str(y1) + ',\n' + str(z1) + ',\n' +
                          str(x1) + ',\n' + str(y1) + ',\n' + str(z0) + ';\n' )

    return basementWall
```

3.4 Adjacency

In general, if a building surface is exposed to the ambient, it is regarded as a heat transfer surface contributing to the the heat balance of a building. However, not all building surfaces, i.e., external building surfaces, are exposed. Adjacent buildings may have one or more surfaces that are not exposed and do not contribute to the heat transmission losses or gains either. In this model, these surfaces are labelled as adiabatic. In order to find an adiabatic surface of a building, the surface should be in contact or in adjacency of a neighbouring surface or building. This means that the distance between the two adjacent surfaces of the two building should be negligible. The maximum distance, in this model, is assumed to be 0.2 m. The choice 0.2 m distance is made quite arbitrarily.

With this assumptions, in order to find the adjacency of a building, first a small buffer of 0.2 m is drawn around the building polygon. This new polygon, which is slightly bigger than the main building polygon, represents the adjacency threshold. Any polygon, or surfaces that falls into this area is assumed to be in adjacency with the main building polygon. An illustration of the described method is presented in the figure below.



In technical terms, by intersection the buffer polygon with the main GIS layer, consisting all the buildings of the city, it is possible to reach a list of intersecting polygons. These polygons are just a slice of the neighbouring building polygons. Yet, they can specify whether the building is adjacent with the other buildings or not. The process of finding a list of adjacent buildings, if any, is done in a function presented as follows.

```
def adjacent_polygons (buildingPolygon, buildingId, df):
    ”””
    Finds adjacent buildings.

    parameters
    ——
    buildingPolygon: shapely polygon
    buildingId: str
      Building ID
    df: gpd.GeoDataFrame
      GIS data of the city

    output
    ——
    intersection: gpd.GeoDataFrame
      Intersecting or adjacent polygons
    ”””

    #Increase the area of the building polygon.
    geoB = gpd.GeoDataFrame([buildingPolygon.buffer(0.2,
                                                    join_style = 2,
                                                    mitre_limit = 2,
                                                    cap_style = 2 )],
                             columns=['geometry'],
                             geometry='geometry').set_crs('EPSG:3006')
```
Continues ...

```
    #Finds any intersection between the building polygon buffer and GIS data.
    intersection = gpd.overlay(geoB, df, how = 'intersection')

    #Exclueds the intersection of the building with itself.
    # The building ID is used as a key.
    intersection = intersection[intersection.buildingId != buildingId]
    intersection = intersection.reset_index(drop = True)

    return intersection
```

Nonetheless, with this function only a list of adjacent buildings polygons is presented. For finding the exact adjacent surfaces, i.e., walls, of the building, further algorithms has to be developed. This part of the method is found in Section 3.3 where the zoning is done and the zone surfaces are defined for each building. In summary, to find the adjacent surfaces of the building, the distance between the surface and the neighboring buildings, i.e., the slice of the adjacent building found in the list, is calculated. If the calculated distance is less than 0.2 m, the surface is assumed to be in contact with the neighboring building, i.e., it is labeled as an adjacent surface, and therefore considered to be adiabatic. This part of the calculations is done where the zoning is presented.

3.5 Archetype

To define the construction and material of buildings, deterministic categories of building archetypes were developed as follows.

Archetype	Building Type	Construction Period
1	SFB	2010-2021
2	SFB	2000-2010
3	SFB	1990-2000
4	SFB	1980-1990
5	SFB	1970-1980
6	SFB	1960-1970
7	SFB	1950-1960
8	SFB	before 1950
9	MFB	2010-2021
10	MFB	2000-2010
11	MFB	1990-2000
12	MFB	1975-1990
13	MFB	1960-1975
14	MFB	1945-1960
15	MFB	before 1945

This list of archetypes is read from a text file, ”archetypeClasses.txt”, using the function below and then imposed to the model.

```
def archetype (buildingType, buildingYear):
    ”””
    Reads information of pre-defined building archetypes and labels the buildings accordingly.

    parameters
    ——–
    buildingType: str
        Type of building given in the GIS file.
    buildingYear:int
        Construction year of the building.

    output
    ——–
    int
        Given category of the building based on its archetype.
    ”””

    # Reads the list of predefined archetypes.
    listOfArchetypes = pd.read_csv('archetypes/archetypeClasses.txt', sep = ”,”,
                                   encoding = 'ANSI', header= 'infer')

    # Filters the list of archetypes based on type and construction year of the building.
    archetype = listOfArchetypes[(listOfArchetypes['buildingType'] == buildingType) &
                                 (listOfArchetypes['buildingYearFrom'] < buildingYear) &
                                 (buildingYear <= listOfArchetypes['buildingYearTo'])

    # Gets the category of the building based on the filtered list.
    arch = archetype['archetype'].values[0]

    return arch
```

As seen the table, buildings are categorized based on their main type, i.e., single-family (SFB) or multi-family (MFB). However, buildings and in particular, single-family buildings, are more likely to be labelled with more categories. For instance, single family buildings can be categorized at detached, semi-detached, etc. Regardless of the detailed types of buildings, here the categorization is done solely on the main categories of building. The function below, therefore, translates all the different types into the two man types of single or multi-family buildings.

```
def building_type (buildingType):
    """
    Returns the main type of buildings, i.e., Apartment or House.

    parameters
    ——
    buildingType: str
        Type of building given in the GIS file.

    output
    ——
    str:
        Given type of buildings: 'Apartment', 'House'
    """

    if buildingType == 'multiFamilyBuilding':
        return 'Apartment'
    else:
        return 'House'
```

3.6 HVAC system

In this version of the model, the HVAC system includes an ideal heating system with an exhaust ventilation system. The choice of ideal system is made according to what is done in the existing UBEMs, such as UMI or CityBest. For district heated buildings, this method has proven to be acceptable. In this case, the room temperature is set to 21°C and the ventilation flow rate (which is proportional to the volume of the building) is set to 0.5 ACH. The HVAC system of the building is defined using a Python class with the same name ("HVAC"). This class includes the functions for an ideal heating and a ventilation systems which are as follows.

```
def ventilation (idf, ventFlow, ventType, zone):
    """
    Makes the model for the ventilation system.

    parameters
    -----
    idf
    ventFlow:float
       Ventilation flow rate(ACH).
    ventType: str
       Main type of the ventilation system.
       (exhaust, intake, balanced)

    output
    -----
    idf
    """

    # EnergyPlus idf object for the ventilation system.
    idf.newidfobject('ZONEVENTILATION:DESIGNFLOWRATE',
                        Name = 'Ventilation',
                        Zone_or_ZoneList_Name = 'Zone',
                        Schedule_Name = 'Always 1',
                        Design_Flow_Rate_Calculation_Method = 'AirChangesHour',
                        Air_Changes_per_Hour = ventFlow,
                        Ventilation_Type = ventType)

    # The schedule for operation if the ventilation system.
    # Here, the ventilation is always on.
    idf.newidfobject('SCHEDULE:COMPACT',
                        Name = 'Always 1',
                        Schedule_Type_Limits_Name = 'Any Number',
                        Field_1 = 'Through: 12/31',
                        Field_2 = 'For: AllDays',
                        Field_3 = 'Until:24:00',
                        Field_4 = 1)

    # Schedule_Type_Limit     idf.newidfobject('SCHEDULETYPELIMITS',
    Name = 'Any Number')

    return idf
```

```
def ideal (idf, roomTemp):
    """
    Makes the model for the ideal heating system.

    parameters     ——
    idf
    roomTemp: float
    Indoor set temperature for heating.

    output
    ——
    idf
    """

    # Define a dual setpoint control for thermostat.
    idf.newidfobject('THERMOSTATSETPOINT:DUALSETPOINT',
                        Name = 'DualSetpoint',
                        Heating_Setpoint_Temperature_Schedule_Name = 'HeatingON',
                        Cooling_Setpoint_Temperature_Schedule_Name = 'CoolingON')

    # Schedule for the heating set temperature of the thermostat.
    idf.newidfobject ('SCHEDULE:COMPACT',
                        Name = 'HeatingON',
                        Schedule_Type_Limits_Name = 'Any Number',
                        Field_1 = 'Through: 12/31',
                        Field_2 = 'For: AllDays',
                        Field_3 = 'Until: 24:00',
                        Field_4 = roomTemp)

    # Schedule for the cooling set temperature of the thermostat.
    idf.newidfobject ('SCHEDULE:COMPACT',
                        Name = 'CoolingON',
                        Schedule_Type_Limits_Name = 'Any Number',
                        Field_1 = 'Through: 12/31',
                        Field_2 = 'For: AllDays',
                        Field_3 = 'Until: 24:00',
                        Field_4 = 45 ) # To limit the cooling demand in Swedish buildings.

    # Schedule for the dual Setpoint.
    idf.newidfobject ('SCHEDULE:COMPACT',
                        Name = 'Always 4',
                        Schedule_Type_Limits_Name = ' Any Number',
                        Field_1 = ' Through : 12/31',
                        Field_2 = ' For : AllDays',
                        Field_3 = ' Until : 24 : 00',
                        Field_4 = 4)

    # Define the thermostat settings.
    idf.newidfobject ('ZONECONTROL:THERMOSTAT',
                        Name = 'Thermostat',
                        Zone_or_ZoneList_Name = 'Zone',
                        Control_Type_Schedule_Name = 'Always 4',
                        Control_1_Object_Type = 'ThermostatSetpoint:DualSetpoint',
                        Control_1_Name = 'DualSetpoint')
```
Continues ...

```
# Ideal system
idf.newidfobject ('ZONEHVAC:IDEALLOADSAIRSYSTEM',
                  Name = 'PurchasedAir',
                  Zone_Supply_Air_Node_Name = 'SupplyInlet')

# Ideal system equipment connections.
idf.newidfobject ('ZONEHVAC:EQUIPMENTCONNECTIONS',
                  Zone_Name = 'Zone',
                  Zone_Conditioning_Equipment_List_Name = 'Equipment',
                  Zone_Air_Inlet_Node_or_NodeList_Name = 'SupplyInlet',
                  Zone_Air_Node_Name = 'ZoneAirNode',
                  Zone_Return_Air_Node_or_NodeList_Name = 'ReturnOutlet')

# Ideal system equipment.
idf.copyidfobject( IDF( io.StringIO ('ZONEHVAC:EQUIPMENTLIST'+','+
                                     'Equipment'+','+
                                     'SequentialLoad'+','+
                                     'ZoneHVAC:IdealLoadsAirSystem'+','+
                                     'PurchasedAir'+','+
                                     '1'+','+
                                     '1'+';')
                      ).idfobjects['ZONEHVAC:EQUIPMENTLIST'][0])

return idf
```

3.7 Occupancy and internal gain

To incorporate the internal heat gain from occupants related activities, including occupants' metabolic heat, use of electrical appliances and lighting and finally use of domestic hot water, a set of functions and schedules have been added to the model. All can be found in the file named as "Occupancy.py". This file includes two python classes, one for initialization of the variables and the other for defining the EnergyPlus schedules and EnergyPlus idf objects. Detailed description of these functions are presented as follows.

- User profile generator

Occupants related internal heat gain is highly dependant on absence or presence of occupants and their level of activities as well as use of appliances, and lighting. In this model, using annual average daily occupancy profiles (for weekdays and weekends separately), it is tried to provide a good estimation these stochastic parameters. More information on the average profiles is found in Section **??**. To reach annual occupancy profiles based on given daily profiles, it is sufficient to simply replicate the daily profiles for the whole year. However, considering the weekday weekend variation of the data, in the duplication process, exact order of the days and weeks in a year should be followed. The function below assures that the specific order of the days for each calendar year (or user-defined year) is followed.

```
def days (firstDayOfYear, leapYear ):

    """
    Generates an array of days of a given calendar year with a specific start day.

    parameters
    ——
    firstDayOfYear: str
       First day of the calendar year, i.e., Monday, Tuesday, or etc.
    leapYear: int
       1 if calendar year is a leap year, 0 if it is not.

    output
    ——
    list
       A list of 365 days of calendar year, staring from the specific start day.
    """
    # A list of days of a week starting from Monday.
    daysOfWeek = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']

    # Duplicate the days of a week for a year with 53 weeks starting from Monday.
    arrayOfDays = list(np.tile(daysOfWeek, 53))

    # Get the index of the specific start day in the list of days of a year.
    index = arrayOfDays.index(firstDayOfYear)

    # Rearrange the list of days of a year based on the specific start day and with considering the leap year.
    daysOfYear = list( arrayOfDays[index:] + arrayOfDays )[:365 + leapYear]

    return daysOfYear
```

After receiving a list of the days of the year, the second step is then to concatenate the daily profiles corresponding to each day of the week, i.e., weekday or weekend, and generate the annual occupancy profiles for occupants presence, metabolic heat gain, use of appliances as well as domestic hot water use. For generating the lighting profile, however, the seasonal dependencies also play an import role. Therefore, based on the given data for monthly average daily lighting, it is possible to make a close to reality lighting profile which is not only affected by the days, but also by the seasons. After conducting this function, the occupancy profiles are written into text files that can be used in the model later.

```
def profile_generator (occupancy, personHeat, appliances, hotWater, lighting, firstDayOfYear,
                       leapYear, building):
  """
  Generates hourly user profile for occupancy, metabolic heat, domestic hot water,
  use of electrical appliances and lighting.

  parameters
  ----
  occupancy: dataframe
     Annual average daily occupancy profile.
  personHeat: dataframe
     Annual average daily metabolic heat gain profile.
  appliances: dataframe
     Annual average daily electricity use for electrical appliances.
  hotWater: dataframe
     Annual average daily domestic hot water use.
  lighting: dataframe
     Monthly average daily electricity use for lighting.
  firstDayOfYear: str
     First day of the calendar year, i.e., Monday, Tuesday, or etc.
  leapYear: int
     1 if calendar year is a leap year, 0 if it is not.
  building: str
     Main type of buildings, i.e., Apartment or House.

  output
  ----
  occupancyProfile***.txt
  personHeatProfile***.txt
  appliancesProfile***.txt
  hotWaterProfile***.txt
  lightingProfile***.txt:
     Stand-alone text files including hourly user profiles for both apartments and houses.
  """

  # Get the list of days of the year.
  daysOfYear = Initials.days (firstDayOfYear, leapYear)

  # Define an array of the months based on the day numbers.
  months = np.array([31, 59+leapYear, 90+leapYear, 120+leapYear,151+leapYear,
                     181+leapYear, 212+leapYear, 243+leapYear, 273+leapYear,
                     304+leapYear, 334+leapYear, 365+leapYear+1])
```

continues ...

```
# Write an hourly array for annual user profiles based on days (weekday or weekend), and leap year.
occupancyProfile = []
personHeatProfile = []
appliancesProfile = []
hotWaterProfile = []
lightingProfile = []
d, m = 0, 0
for day in daysOfYear:
    d += 1

    # For profiles with no seasonal dependencies.
    occupancyProfile.append(occupancy.iloc[:,1].values if day == 'Saturday' or day == 'Sunday'
                                    else occupancy.iloc[:,0].values)
    personHeatProfile.append(personHeat.iloc[:,1].values if day == 'Saturday' or day == 'Sunday'
                                    else personHeat.iloc[:,0].values)
    appliancesProfile.append(appliances.iloc[:,1].values if day == 'Saturday' or day == 'Sunday'
                                    else appliances.iloc[:,0].values)
    hotWaterProfile.append(hotWater.iloc[:,1].values if day == 'Saturday' or day == 'Sunday'
                                    else hotWater.iloc[:,0].values)

    # For lighting profile with seasonal dependency.
    if d < months[m]:
        lightingProfile.append(lighting.iloc[:,m+12].values if day == 'Saturday' or day == 'Sunday'
                                    else lighting.iloc[:,m].values)
    else:
        m +=1
        lightingProfile.append(lighting.iloc[:,m+12].values if day == 'Saturday' or day == 'Sunday'
                                    else lighting.iloc[:,m].values)
        continue

# Convert lists to numy arrays and reshape them.
occupancyProfile = np.array(occupancyProfile).flatten()
personHeatProfile = np.array(personHeatProfile).flatten() # Watts/person
appliancesProfile = np.array(appliancesProfile).flatten() # Watts/person
hotWaterProfile = np.array(hotWaterProfile).flatten()/3600000 #liter/h.person to m3/s.person
# 25% reduction in lighting power from 2009 to 2016.
lightingProfile = np.array(lightingProfile).flatten()* 0.75 # Watts/person

# Get hot water peak flow rate from the profile.
maxFlowHotWater = np.max(hotWaterProfile)

return (pd.DataFrame(occupancyProfile).to_csv('occupancyProfile%s.txt'%building,
                                                index=False, header=False),
        pd.DataFrame(personHeatProfile).to_csv('personHeatProfile%s.txt'%building,
                                                index=False, header=False),
        pd.DataFrame(appliancesProfile).to_csv('appliancesProfile%s.txt'%building,
                                                index=False, header=False),
        # The flow rate should be a fraction of peak flow rate.
        pd.DataFrame(hotWaterProfile/maxFlowHotWater).to_csv('hotWaterProfile%s.txt'%building,
                                                             index=False, header=False),
        pd.DataFrame(lightingProfile).to_csv('lightingProfile%s.txt'%building,
                                              index=False, header=False),
        maxFlowHotWater)
```

- Number of occupants

Estimation of the occupant is directly influenced by the number of people leaving in buildings. According to the Statistics Sweden (SCB), in Sweden, the average building floor area per person is 42 m^2. Using this value, it is possible to make a rough estimation of the number of occupants in each building. With the lack of knowledge on the living area of buildings, using the given building footprint area and height, the overall area is approximated. These calculation are initiated in this function.

```
def occupants(area, height):
    """
    Estimates the number of building occupants.

    parameters
    ----------
    area: int
        Area of the building footprint.
    height: int
        Height of the building.

    output
    ------
    occupants: int
        Number of occupants in buildings.
    """

    # Calculates the number of floors.
    # Floor height (floor to floor) is assumed to be 2.8.
    numberOfFloors = round(height // 2.8)
    floorArea = area * numberOfFloors

    # Calulates the building occupants for the floor area.
    # In Sweden, on average, floor area per person is 42 m2.
    occupants = round(floorArea // 42)

    return occupants
```

- Internal heat gain from occupants

This function sets the metabolic heat gain from occupants in the zone. The metabolic heat gain is dependant on the absence or presence of the occupants as well as their levels of activities. These data is what is found in the text files generated from the annual average daily profiles. These data are defined as schedules to be read by the EnergyPlus object for calculating the internal gain from people.

```
def people (idf, area, height):
    """
    Sets occupants' related heat gain.
    input
    ——
    idf: idf

    area: int
      Area of the building footprint.

    height: int
      Height of the building.

    output
    ——
    idf
    """

    # Sets the number of occupants.
    occ = Initials.occupants(area, height)

    # Sets internal heat gain for occupants in the zone.
    idf.newidfobject('PEOPLE',
                     Name = 'People',
                     Number_of_People = str(occ),
                     Zone_or_ZoneList_Name = 'Zone',
                     Number_of_People_Schedule_Name = 'OccupancyProfile',
                     Activity_Level_Schedule_Name = 'PersonHeatProfile')
    # Activity_Level_Schedule.
    idf.newidfobject ('SCHEDULE:FILE',
                     Name = 'PersonHeatProfile',
                     Schedule_Type_Limits_Name = 'Any Number',
                     File_Name = 'personHeatProfile%s.txt' %building,
                     Column_Number = '1',
                     Rows_to_Skip_at_Top = '0')

    # Number_of_People_Schedule.
    idf.newidfobject ('SCHEDULE:FILE',
                     Name = 'OccupancyProfile',
                     Schedule_Type_Limits_Name = 'Any Number',
                     File_Name = 'occupancyProfile%s.txt' %building,
                     Column_Number = '1',
                     Rows_to_Skip_at_Top = '0')

    return idf
```

- Internal heat gain from electric equipment and lighting

The internal heat gain linked to energy use in electrical equipment and lighting is a function of user profiles and electric power of equipment and lighting. With a knowledge on the total electric power used by appliances and lights (this is what is given in the generated user profiles for electrical appliances and lighting), it is possible to write an Energy plus object that handles the related calculations.

```
def equipment (idf, area, height, building):
    """
    Sets the heat gain from using equipment and lighting.

    parameters
    ——
    idf: idf

    area: int
       Area of building footprint.
    height: int
       Height of building.
    building: str
       Main type of buildings, i.e., Apartment or House.

    output
    ——
    idf
    """

    # Sets the number of occupants
    occ = Initials.occupants(area, height)

    # Sets internal gains for electric equipment in the zone.
    .newidfobject ('ELECTRICEQUIPMENT',
                    Name = 'ElectricEquipment',
                    Zone_or_ZoneList_Name = 'Zone',
                    Schedule_Name = 'AppliancesProfile',
                    Design_Level_Calculation_Method = 'EquipmentLevel',
                    Design_Level = str(occ* 1)) #power/person

    # Use profile schedule
    idf.newidfobject ('SCHEDULE:FILE',
                    Name = 'AppliancesProfile',
                    Schedule_Type_Limits_Name = 'Any Number',
                    File_Name = 'appliancesProfile%s.txt' %building,
                    Column_Number = '1',
                    Rows_to_Skip_at_Top = '0')

    return idf
```

```
def lights (idf, area, height, building):
    """
    Sets the heat gain from using equipment and lighting.

    parameters
    ——
    idf: idf
    area: int
       Area of building footprint.
    height: int
       Height of building.
    building: str
       Main type of building, i.e., Apartment or House.

    output
    ——
    idf
    """

    # Sets the number of occupants.
    occ = Initials.occupants(area, height)

    # Sets internal gains for lighting in the zone.
    idf.newidfobject ('LIGHTS',
                    Name = 'Lights',
                    Zone_or_ZoneList_Name = 'Zone',
                    Schedule_Name = 'LightingProfile',
                    Design_Level_Calculation_Method = 'LightingLevel',
                    Design_Level = str(occ* 1)) #power/person

    # Use profile schedule
    idf.newidfobject ('SCHEDULE:FILE',
                    Name = 'LihtingProfile',
                    Schedule_Type_Limits_Name = 'Any Number',
                    File_Name = 'lightingProfile%s.txt' %building,
                    Column_Number = '1',
                    Rows_to_Skip_at_Top = '0')
    return idf
```

- Energy use for domestic hot water

Energy demand for domestic hot water use in buildings is a function of hot water flow rate and cold and hot water temperatures. In EnergyPlus, each of these parameters can be set to constant or variable values. For the sake of simplicity, in this version of the model, it is assumed that the hot and cold water temperatures are constant values that can be defined by the user. For the hot water flow rate, however, the hourly use profile also initially obtained from the annual average daily profiles is used. This profile which is a fraction of the peak flow rate is defined as a schedule in EnergyPlus. This schedule for hot water flow rate is then used in an EnergyPlus idf object which serves the model with calculations of the required energy for domestic hot water use in buildings.

```
def dhw (idf, area, height, hotWaterTemp, coldWaterTemp, maxFlowHotWater, building):
    """
    Calculates the hot water use in the zone.
    parameters
    ——
    idf: idf
    area: int
       Area of building footprint.
    height: int
       Height of building.
    hotWaterTemp: int
       Temperature of hot water
    coldWaterTemp: int
       Temperature of cold water.
    maxFlowHotWater: float
       Peak flow rate for hot water use.
    building: str
       Main type of buildings, i.e., Apartment or House.
    output
    ——
    idf
    """

    # Sets the number of occupants.
    occ = Initials.occupants(area, height)

    # Heat demand for hot water use, based on the given use profile and hot and cold water temperatures.
    idf.newidfobject ('WATERUSE:EQUIPMENT',
                    Name = 'WaterUse',
                    Peak_Flow_Rate = str(float(maxFlowHotWater)*occ),
                    Flow_Rate_Fraction_Schedule_Name= 'HotWaterUse',
                    Hot_Water_Supply_Temperature_Schedule_Name = 'HotWaterTemp',
                    Cold_Water_Supply_Temperature_Schedule_Name = 'ColdWaterTemp',
                    Zone_Name = 'Zone')

    # Schedule for hot water temperature.
    idf.newidfobject ('SCHEDULE:COMPACT',
                    Name = 'HotWaterTemp',
                    Schedule_Type_Limits_Name = 'Any Number',
                    Field_1 = 'Through: 12/31',
                    Field_2 = 'For: AllDays',
                    Field_3 = 'Until: 24:00',
                    Field_4 = str(hotWaterTemp)
```

continues ...

```
# Schedule for cold water temperature.
idf.newidfobject ('SCHEDULE:COMPACT',
                    Name = 'ColdWaterTemp',
                    Schedule_Type_Limits_Name = 'Any Number',
                    Field_1 = 'Through: 12/31',
                    Field_2 = 'For: AllDays',
                    Field_3 = 'Until: 24:00',
                    Field_4 = str(coldWaterTemp) #cold water temperature)

# Schedule for hot water use.
idf.newidfobject ('SCHEDULE:FILE',
                    Name = 'HotWaterUse',
                    Schedule_Type_Limits_Name = 'Any Number',
                    File_Name = 'hotWaterProfile%s.txt' %building,
                    Column_Number = '1',
                    Rows_to_Skip_at_Top = '0')

return idf
```

3.8 Model simulation

Using describes classed and functions, it is possible to develop different parts of a building model (or idf file). However, in order to run the simulation for each building of the city, all the information from separated idf files has to be compiled in one. This process is conducted in ”simulation.py”. In this file, a python class is responsible for compiling a unique idf file, running the simulation and reporting the output results for each building of the city. The two main functions of this class are as follows.

- Model development

In this function, first an empty idf file is initiated and then based on the input variables and previously developed functions is step by step completed . The output of this is a complete idf file that can be sent to EnergyPlus and run.

```
def building_idf (buildingPolygon, buildingArea, buildingHeight, buildingId, basementInfo, epw, wwr,
                  material_idf, basement_idf, df, hotWaterTemp, coldWaterTemp, maxFlowHotWater,
                  buildingMainType):
    """
    Creates a complete IDF from all the input variables and functions.

    parameters
    ----
    buildingPolygon: Shapely.Polygon
      Building footprint from GIS data.
    buildingArea: int
      Calculated area of the building.
    buildingHeight: int
      Estimated height of the building from GIS data.
    buildingId:str
      Unique ID that is given to each building found in the GIS data.
    epw: epw
      The energyPlus weather file.
    wwr: float
      Window to wall ratio of buildings.
    material_idf: idf
      Building construction and material found for each archetype.
    basement_idf: idf
      Predefined specifications of the basement properties.
    df: gpd.GeoDataFrame
      GIS dataset
    buildingMainType: str
      Main type of building, i.e., house or apartment.
    basementInfo: int
      Basement info from the EPC (number of underground floor).
    output
    -----
    idf
      Completed idf for simulation.
    """

    # Modify the geometry rules of the building polygon.
    buildingPolygon = Geometry.global_geometry(buildingPolygon)
                                                                  continues ...
```

```
#Activate the idf of the basement.
if basementInfo > 0:
   basement = 1
else:
   basement = 0

# Initialize the idf file.
idf = baseIDF.blank_idf(epw)

# Add the main simulation parameters.
baseIDF.simulation_parameters(idf)
baseIDF.site_location(idf,epw)
baseIDF.simulation_period(idf)
baseIDF.simulation_output(idf)

# Specify the geometry rules.
Geometry.global_geometry_rules (idf)

# Write the idf for a singlezone model.
oneZone.zone(idf)

# Find the adjacent buildings
adj = Adjacency.adjacent_polygons(buildingPolygon, buildingId, df)

# Define the idf for construction and surfaces.
surfaceidf = oneZone.onezone_building_surfaces(buildingPolygon, buildingHeight, basement, wwr, adj)

# Copy the information from "surfaceidf" to the idf file.
for s in range (len(surfaceidf.idfobjects["BUILDINGSURFACE:DETAILED"])):
   idf.copyidfobject(surfaceidf.idfobjects["BUILDINGSURFACE:DETAILED"][s])

for w in range (len(surfaceidf.idfobjects["FENESTRATIONSURFACE:DETAILED"])):
   idf.copyidfobject(surfaceidf.idfobjects["FENESTRATIONSURFACE:DETAILED"][w])

# Copy the information from the predefined basement idf to the idf file.
if basement == 1:
   idf.newidfobject('ZONE', Name = 'BASEMENT', Ceiling_Height = '2.3')
   for schedule in range (len(basement_idf.idfobjects["SCHEDULETYPELIMITS"])):
      idf.copyidfobject(basement_idf.idfobjects["SCHEDULETYPELIMITS"][schedule])
   for schedule in range (len(basement_idf.idfobjects["SCHEDULE:COMPACT"])):
      idf.copyidfobject(basement_idf.idfobjects["SCHEDULE:COMPACT"][schedule])
   for surface in range (len(basement_idf.idfobjects
                                        ["SURFACEPROPERTY:OTHERSIDECOEFFICIENTS"])):
      idf.copyidfobject(basement_idf.idfobjects
                                     ["SURFACEPROPERTY:OTHERSIDECOEFFICIENTS"][surface])

# Copy the construction and material from predefined archetype idfs.
for m in range (len(material_idf.idfobjects["Material"])):
   idf.copyidfobject(material_idf.idfobjects["Material"][m])

for ma in range (len(material_idf.idfobjects["MATERIAL:AIRGAP"])):
   idf.copyidfobject(material_idf.idfobjects["MATERIAL:AIRGAP"][ma])
```

continues ...

```
for mgl in range (len(material_idf.idfobjects["WINDOWMATERIAL:GLAZING"])):
   idf.copyidfobject(material_idf.idfobjects["WINDOWMATERIAL:GLAZING"][mgl])

for mgs in range (len(material_idf.idfobjects["WINDOWMATERIAL:GAS"])):
   idf.copyidfobject(material_idf.idfobjects["WINDOWMATERIAL:GAS"][mgs])

for c in range (len(material_idf.idfobjects["CONSTRUCTION"])):
   idf.copyidfobject(material_idf.idfobjects["CONSTRUCTION"][c])

# Add information of the HVAC system to the idf file.
HVAC.ventilation(idf, 0.5, 'Exhaust')
HVAC.ideal(idf, 21)

# Add the infomation of the occupancy, gain and DHW to the idf file.
Occupancy.people(idf,buildingArea, buildingHeight, buildingMainType)
Occupancy.equipment (idf, buildingArea, buildingHeight, buildingMainType)
Occupancy.dhw (idf, buildingArea, buildingHeight, hotWaterTemp, coldWaterTemp, maxFlowHotWater,
                    buildingMainType)

return idf
```

- Model simulation

To simulate the idf file of a building, EnergyPlus has to be called and forced to run through Python. This is done using the python package, EPPY. After running the simulation the results of the model simulation is collected from the EnergyPlus output files. The results include the annual and hourly energy use for space heating and hot water use of a building. Running the simulation and reporting the results are done using below function.

```
def building_idf (buildingPolygon, buildingArea, buildingHeight, buildingId,basementInfo,epw, wwr,
                  material_idf, basement_idf, df,hotWaterTemp, coldWaterTemp, maxFlowHotWater,
                  buildingMainType):

    """
    Creates a complete IDF from all the input variables and functions.

    parameters
    ——
    buildingPolygon: Shapely.Polygon
       Building footprint from GIS data.
    buildingArea: int
       Calculated area of the building.
    buildingHeight: int
       Estimated height of the building from GIS data.
    buildingId:str
       Unique ID that is given to each building found in the GIS data.
    epw: epw
       The energyPlus weather file.
    wwr: float
       Window to wall ratio of buildings.
    material_idf: idf
       Building construction and material found for each archetype.
    basement_idf: idf
       Predefined specifications of the basement properties.
    df: gpd.GeoDataFrame
       GIS dataset
    buildingMainType: str
       Main type of building, i.e., house or apartment.
    basementInfo: int
       Basement info from the EPC (number of underground floor).

    output
    ——
    idf
       Completed idf for simulation.
    """

    # Modify the geometry rules of the building polygon.
    buildingPolygon = Geometry.global_geometry(buildingPolygon)

                                                                  continues ...
```

```
#Activate the idf of the basement.
if basementInfo > 0:
    basement = 1
else:
    basement = 0

# Initialize the idf file.
idf = baseIDF.blank_idf(epw)

# Add the main simulation parameters.
baseIDF.simulation_parameters(idf)
baseIDF.site_location(idf,epw)
baseIDF.simulation_period(idf)
baseIDF.simulation_output(idf)

# Specify the geometry rules.     Geometry.global_geometry_rules (idf)

# Write the idf for a singlezone model.
oneZone.zone (idf)

# Find the adjacent buildings
adj = Adjacency.adjacent_polygons(buildingPolygon, buildingId, df)

# Define the idf for construction and surfaces.
surfaceidf = oneZone.onezone_building_surfaces(buildingPolygon,buildingHeight, basement, wwr, adj)

# Copy the information from "surfaceidf" to the idf file.
for s in range (len(surfaceidf.idfobjects["BUILDINGSURFACE:DETAILED"])):
    idf.copyidfobject(surfaceidf.idfobjects["BUILDINGSURFACE:DETAILED"][s])

for w in range (len(surfaceidf.idfobjects["FENESTRATIONSURFACE:DETAILED"])):
    idf.copyidfobject(surfaceidf.idfobjects["FENESTRATIONSURFACE:DETAILED"][w])

# Copy the information from the predefined basement idf to the idf file.
if basement == 1:
    idf.newidfobject('ZONE', Name = 'BASEMENT', Ceiling_Height = '2.3')
    for schedule in range (len(basement_idf.idfobjects["SCHEDULETYPELIMITS"])):
        idf.copyidfobject(basement_idf.idfobjects["SCHEDULETYPELIMITS"][schedule])
    for schedule in range (len(basement_idf.idfobjects["SCHEDULE:COMPACT"])):
        idf.copyidfobject(basement_idf.idfobjects["SCHEDULE:COMPACT"][schedule])
    for surface in range (len(basement_idf.idfobjects[
                                        "SURFACEPROPERTY:OTHERSIDECOEFFICIENTS"])):
        idf.copyidfobject(basement_idf.idfobjects[
                                    "SURFACEPROPERTY:OTHERSIDECOEFFICIENTS"][surface])

# Copy the construction and material from predefined archetype idfs.
for m in range (len(material_idf.idfobjects["Material"])):
    idf.copyidfobject(material_idf.idfobjects["Material"][m])

for ma in range (len(material_idf.idfobjects["MATERIAL:AIRGAP"])):
    idf.copyidfobject(material_idf.idfobjects["MATERIAL:AIRGAP"][ma])

for mgl in range (len(material_idf.idfobjects["WINDOWMATERIAL:GLAZING"])):
    idf.copyidfobject(material_idf.idfobjects["WINDOWMATERIAL:GLAZING"][mgl])
```

continues ...

```
for mgs in range (len(material_idf.idfobjects["WINDOWMATERIAL:GAS"])):
   idf.copyidfobject(material_idf.idfobjects["WINDOWMATERIAL:GAS"][mgs])

for c in range (len(material_idf.idfobjects["CONSTRUCTION"])):
   idf.copyidfobject(material_idf.idfobjects["CONSTRUCTION"][c])

# Add information of the HVAC system to the idf file.
HVAC.ventilation(idf, 0.5, 'Exhaust')
HVAC.ideal(idf, 21)

# Add the information of the occupancy, gain and DHW to the idf file
Occupancy.people(idf,buildingArea, buildingHeight, buildingMainType)
Occupancy.equipment (idf, buildingArea, buildingHeight, buildingMainType)
Occupancy.dhw (idf, buildingArea, buildingHeight, hotWaterTemp, coldWaterTemp, maxFlowHotWater,
                    buildingMainType)

return idf
```

```
def simulation (idf):
    ”””
    Runs the idf file and reports the results in the output.

    parameters
    ——
    idf:
    Completed idf file.

    output
    ——
    spaceHeat:
    Hourly space heating demand.

    hotWater:
    Hourly heat demand for DHW

    heatDemand:
    Total hourlry energy demand.
    ”””

    # Runs the idf file using eppy.run() function.
    idf.run()

    # Read hourly results of the space heating.
    fout = ’eplusout.mtr’
    hourly = pd.read_csv(fout , sep = ”[,]”, skiprows=11, skipfooter = 2, engine=’python’)

    spaceHeat = np.array(hourly.iloc[0::3,1])
    spaceHeat = spaceHeat/ 3600000 # kWh/h

    # Reads the hourly hot water use.
    hotWater = np.array(hourly.iloc[1::3,1])
    hotWater = hotWater/3600000 # kWh/h

    # Calculates the total hourly heat demand.
    HeatDemand = spaceHeat + hotWater

    return spaceHeat, hotWater, HeatDemand
```

- Execution of the model

```
””” Initializing the simulation tool EnergyPlus ”””

# Call the idd (EnergyPlus exe file)
iddfile = ”EnergyPlus/Energy+.idd”

# Initialize the idf file processing.
IDF.setiddname(iddfile)

””” Importing the input parameters ”””

# EnergyPlus weather file.
epw = ”inputs/*.epw”

# The Geocoded building data.
df = gpd.read_file('inputs/*.shp')
df = df.set_crs('EPSG:3006') # Swedish GIS system coordinates.

# idf for the basement properties.
basement_idf = IDF (”inputs/basement_idf.idf”)

# Input parameters from a user defined text file.
parameters = pd.read_csv('inputs/parameters.txt', sep = ',', header = 'infer', encoding= 'ANSI')

””” Initializing the input parameters ”””

# window to wall ratio.
wwr = parameters.wwr[0]

# Hot and cold water temperatures for domestic hot water use.
hotWaterTemp = parameters.hotWaterTemp[0]
coldWaterTemp = parameters.coldWaterTemp[0]
```