geomeppy Documentation

Release 0.11.8

Jamie Bull

Nov 11, 2022

Contents

Contents:

CHAPTER 1

Tutorial

A walk-through introduction to *geomeppy* for beginners to the library.

1.1 Start here

1.1.1 Introduction

This tutorial is intended as a walk-through for complete beginners to *geomeppy*.

At the end of the tutorial, you will have created and run an EnergyPlus model without once having to manually edit an *.idf* file, without creating geometry in SketchUp, and without any other such hassles.

In outline, you will:

1. create a virtual environment and install *geomeppy*
2. install EnergyPlus (if you haven't already)
3. create a simple test building
4. add some windows
5. set the constructions and boundary conditions
6. view the model in 3D
7. run a simulation in EnergyPlus

1.1.2 Installation

First we need to set up a virtual environment to keep this Python environment separate from any other installations on your machine. This tutorial assumes you're using Python 3 so make sure you have that installed.

```
$ mkdir tutorial
$ cd tutorial
$ python3 -m venv venv
```

Now we activate the virtual environment.

```
$ source venv/bin/activate
```

And install *geomeppy*.

```
(venv)$ pip3 install geomeppy
```

This will take a while to install the *geomeppy* package and its dependencies. While it's installing, we can move on to installing EnergyPlus if you don't already have it installed.

The EnergyPlus installers can be found on the energyplus.net site. Choose the correct package for your platform and install it.

1.1.3 Creating a model

Next we want to create our basic model, so we'll open a Python interpreter.

```
(venv)$ python3
```

From now on, we're working in the Python interpreter, indicated by the >>> prompt.

Be aware that this tutorial is written for the Mac and for EnergyPlus 9.1.0.

If you are on Windows or Linux machine or have a different version of EnergyPlus installed, you'll need to replace the path to the EnergyPlus installation folder with the appropriate path.

For example, on Windows, this is usually `C:/EnergyPlusV9-1-0`, and on Linux, `/usr/local/EnergyPlus-9-1-0`.

```
>>> from geomeppy import IDF
>>> IDF.setiddname('/Applications/EnergyPlus-9-1-0/Energy+.idd')
>>> idf = IDF('/Applications/EnergyPlus-9-1-0/ExampleFiles/Minimal.idf')
```

Now we have imported an EnergyPlus IDF file which contains the bare minimum required to be able to run successfully. We need to add a weather file to our IDF object first though.

```
>>> idf.epw = "USA_CO_Golden-NREL.724666_TMY3.epw"
```

Now we have assigned an EPW format weather file to the IDF. This will tell EnergyPlus about the weather conditions to simulate.

So far, this is the same as using *eppy*, but now we are getting to the *geomeppy* part where we can generate and manipulate the IDF geometry. First off, let's add a simple square block.

```
>>> idf.add_block(
        name='Boring hut',
        coordinates=[(10,0),(10,10),(0,10),(0,0)],
        height=3.5)
```

Next we can set some default constructions for the IDF surfaces.

```
>>> idf.set_default_constructions()
```

Now we need to tell *geomeppy* to match up our surfaces with the correct outside conditions. In this case, it will just set the outside boundary conditions for all the walls and roof to "outside", and that of the floor to "ground".

```
>>> idf.intersect_match()
```

Last step, let's add some nice big windows. We'll set window to wall ratio (WWR) of 0.6 for all external walls.

```
>>> idf.set_wwr(0.6)
```

So what have we built? We can export the IDF geometry as an OBJ file, a format that can be imported into a 3D geometry viewer.

```
>>> idf.to_obj('boring_hut.obj')
```

Try dragging the 'boring_hut.obj' file and 'boring_hut.mtl' file to 3D Viewer to see your "Boring hut" in glorious interactive 3D.

And now, the moment you've been working towards all this time...

```
>>> idf.run()
```

You should see an output something like the following:

```
C:\EnergyPlusV9-1-0\energyplus.exe --idd C:/EnergyPlusV9-1-0/Energy+.idd --output-
↪directory C:\Users\jamie\geomeppy --weather C:\EnergyPlusV9-1-0\WeatherData\USA_CA_
↪San.Francisco.Intl.AP.724940_TMY3.epw C:\Users\jamie\geomeppy\in.idf

EnergyPlus Starting
EnergyPlus, Version 9.1.0-08d2e308bb, YMD=2019.09.17 10:11
Adjusting Air System Sizing
Adjusting Standard 62.1 Ventilation Sizing
Initializing Simulation
Reporting Surfaces
Beginning Primary Simulation
Initializing New Environment Parameters
Warming up {1}
Warming up {2}
Warming up {3}
Warming up {4}
Warming up {5}
Warming up {6}
Starting Simulation at 12/21 for DENVER_STAPLETON ANN HTG 99.6% CONDNS DB
Initializing New Environment Parameters
Warming up {1}
Warming up {2}
Warming up {3}
Warming up {4}
Warming up {5}
Warming up {6}
Starting Simulation at 07/21 for DENVER_STAPLETON ANN CLG .4% CONDNS DB=>MWB
Writing tabular output file results using HTML format.
Writing final SQL reports
EnergyPlus Run Time=00hr 00min  0.53sec
EnergyPlus Completed Successfully.
```

This indicates that EnergyPlus has run successfully.

1.1.4 Summary

In this tutorial you learned how to install geomeppy, create some simple geometry, visualise it using both the built in *idf.view_model* method, and also by producing files for use in external viewers using the *idf.to_obj* method, and finally run the IDF in EnergyPlus.

It's not very exciting, since we haven't added any heating or cooling systems, or output variables. Geomeppy is a geometry package after all!

But in the next tutorial, we'll add a heating system, while also learning about some of the other features of *geomeppy*.

Next steps

1.2 Next steps

1.2.1 Introduction

This tutorial continues on from the *Start here* guide to *geomeppy*. In it you will learn how to add a simple heating system, simulate multiple runs, and view the results of your simulation.

In outline, you will:

1. create a test building with two blocks and three zones
2. add a heating system to service the zones
3. add output variables to measure the energy performance
4. run several simulations while varying the glazing orientation
5. extract and explore the simulation outputs

1.2.2 The building

First we'll build a couple of blocks with default constructions and window-to-wall ratio of 25%.

```
>>> from geomeppy import IDF
>>> IDF.setiddname("C:/EnergyPlusV9-1-0/Energy+.idd")
>>> idf = IDF("C:/EnergyPlusV9-1-0/ExampleFiles/Minimal.idf")
>>> idf.epw = "USA_CO_Golden-NREL.724666_TMY3.epw"
>>> idf.add_block(
      name='Two storey',
      coordinates=[(10,0),(10,5),(0,5),(0,0)],
      height=6,
      num_stories=2,
    )
>>> idf.add_block(
      name='One storey',
      coordinates=[(10,5),(10,10),(0,10),(0,5)],
      height=3,
    )
>>> idf.intersect_match()
>>> idf.set_default_constructions()
```

1.2.3 Heating systems

This is all good, but we're really interested in the energy performance of our building. Lets add a heating system.

```
>>> stat = idf.newidfobject(
      "HVACTEMPLATE:THERMOSTAT",
      Name="Zone Stat",
      Constant_Heating_Setpoint=20,
      Constant_Cooling_Setpoint=25,
    )
>>> for zone in idf.idfobjects["ZONE"]:
      idf.newidfobject(
          "HVACTEMPLATE:ZONE:IDEALLOADSAIRSYSTEM",
          Zone_Name=zone.Name,
          Template_Thermostat_Name=stat.Name,
    )
```

We'll also add some output variables.

```
>>> idf.newidfobject(
        "OUTPUT:VARIABLE",
        Variable_Name="Zone Ideal Loads Supply Air Total Heating Energy",
        Reporting_Frequency="Hourly",
    )
>>> idf.newidfobject(
        "OUTPUT:VARIABLE",
        Variable_Name="Zone Ideal Loads Supply Air Total Cooling Energy",
        Reporting_Frequency="Hourly",
    )
```

1.2.4 A small experiment

To give us something to test out geomeppy, we'll design a test where we explore the effect of moving the glazing from mainly on

```
>>> north_wwr = [i / 10 for i in range(1, 10)]
>>> print(north_wwr)
[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
>>> south_wwr = [1-wwr for wwr in north_wwr]
>>> print(south_wwr)
[0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1]
```

These are our window-to-wall ratios for the North and South facades.

```
>>> for north, south in zip(north_wwr, south_wwr):
        idf.set_wwr(north, construction="Project External Window", orientation="north
↪")
        idf.set_wwr(south, construction="Project External Window", orientation="south
↪")
        idf.run(
            output_prefix=f"{north}_{south}_",  # so we can tell which files belong␣
↪to which simulation
            expandobjects=True,  # because we're using HVAC:Template objects
        )
```

1.2.5 Results

Gathering EnergyPlus results can be a little bit tricky, so here's a little class that we can use to extract total energy use in kWh. It uses the *esoreader* package, and I highly recommend using the *pandas* interface for anything more involved than this little demo.

```
>>> class ESO:
        def __init__(self, path):
            self.dd, self.data = esoreader.read(path)
```

```
>>>     def read_var(self, variable, frequency="Hourly"):
            return [
                {"key": k, "series": self.data[self.dd.index[frequency, k, variable]]}
                for _f, k, _v in self.dd.find_variable(variable)
            ]
```

```
>>>     def total_kwh(self, variable, frequency="Hourly"):
            j_per_kwh = 3_600_000
            results = self.read_var(variable, frequency)
            return sum(sum(s["series"]) for s in results) / j_per_kwh
```

Now we'll use the *ESO* class to read in the results of the simulations.

```
>>> results = []
>>> for north, south in zip(north_wwr, south_wwr):
        eso = ESO(f"tests/tutorial/{north}_{south}_out.eso")
        heat = eso.total_kwh("Zone Ideal Loads Supply Air Total Heating Energy")
        cool = eso.total_kwh("Zone Ideal Loads Supply Air Total Cooling Energy")
        results.append([north, south, heat, cool, heat + cool])
        idf.run(output_prefix=f"{north}_{south}_", expandobjects=True)
```

And print out a table of results.

```
>>> headers = ["WWR-N", "WWR-S", "Heat", "Cool", "Total"]
>>> header_format = "{:>10}" * (len(headers))
>>> row_format = "{:>10.1f}" * (len(headers))
>>> print(header_format.format(*headers))
>>> for row in results:
        print(row_format.format(*row))
```

```
WWR-N      WWR-S       Heat       Cool      Total
  0.1        0.9      346.3       81.5      427.9
  0.2        0.8      347.4       79.3      426.7
  0.3        0.7      348.1       77.1      425.2
  0.4        0.6      348.5       75.1      423.5
  0.5        0.5      348.6       73.1      421.7
  0.6        0.4      348.5       71.1      419.6
  0.7        0.3      348.2       69.1      417.3
  0.8        0.2      347.5       67.3      414.8
  0.9        0.1      346.5       65.5      412.0
```

You can see the annual energy use is lower with more glazing on the North facade and less on the South facade. This is driven by the increase in energy required for cooling when there is more South-facing glazing.

1.2.6 Summary

This tutorial has shown you how to add a heating system to your model, run a set of simulations while changing geometry between each run, and process the results.

So far you've used the default constructions. In the next tutorial you'll find out how to change these and run yet more interesting simulations.

Constructions

1.3 Constructions

1.3.1 Introduction

TODO: add an outline of the tutorial

1.3.2 The building

We'll use the same building as in the previous tutorial.

```
>>> from geomeppy import IDF
>>> IDF.setiddname("C:/EnergyPlusV9-1-0/Energy+.idd")
>>> idf = IDF("C:/EnergyPlusV9-1-0/ExampleFiles/Minimal.idf")
>>> idf.epw = "USA_CO_Golden-NREL.724666_TMY3.epw"
>>> idf.add_block(
      name='Two storey',
      coordinates=[(10,0),(10,5),(0,5),(0,0)],
      height=6,
      num_stories=2,
    )
>>> idf.add_block(
      name='One storey',
      coordinates=[(10,5),(10,10),(0,10),(0,5)],
      height=3,
    )
>>> idf.intersect_match()
>>> idf.set_default_constructions()
>>> idf.set_wwr(0.25, construction="Project External Window")
```

1.3.3 Viewing constructions

Next we'll take a look at some of the constructions.

```
>>> for c in idf.idfobjects["CONSTRUCTION"]:
        print(c)
CONSTRUCTION,
    Project Wall,             !- Name
    DefaultMaterial;          !- Outside Layer
CONSTRUCTION,
    Project Partition,        !- Name
    DefaultMaterial;          !- Outside Layer
CONSTRUCTION,
    Project Floor,            !- Name
```

(continues on next page)

(continued from previous page)

```
    DefaultMaterial;           !- Outside Layer
CONSTRUCTION,
    Project Flat Roof,         !- Name
    DefaultMaterial;           !- Outside Layer
CONSTRUCTION,
    Project Ceiling,           !- Name
    DefaultMaterial;           !- Outside Layer
CONSTRUCTION,
    Project Door,              !- Name
    DefaultMaterial;           !- Outside Layer
CONSTRUCTION,
    Project External Window,    !- Name
    DefaultGlazing;            !- Outside Layer
```

Ah. So what do these "DefaultMaterial" and "DefaultGlazing" look like?

```
>>> idf.getobject("MATERIAL", "DefaultMaterial")
MATERIAL,
    DefaultMaterial,           !- Name
    Rough,                     !- Roughness
    0.1,                       !- Thickness
    0.1,                       !- Conductivity
    1000,                      !- Density
    1000,                      !- Specific Heat
    0.9,                       !- Thermal Absorptance
    0.7,                       !- Solar Absorptance
    0.7;                       !- Visible Absorptance
```

```
>>> idf.getobject("WINDOWMATERIAL:SIMPLEGLAZINGSYSTEM", "DefaultGlazing")
WINDOWMATERIAL:SIMPLEGLAZINGSYSTEM,
DefaultGlazing,            !- Name
2.7,                       !- UFactor
0.763,                     !- Solar Heat Gain Coefficient
0.8;                       !- Visible Transmittance
```

Well the window looks OK, but those "DefaultMaterial"'s should definitely be replaced with something better.

1.3.4 Importing constructions

We can import constructions from another IDF. For this tutorial, we'll fetch the ones from "WindowTestsSimple.idf".

```
>>> src_idf = IDF("C:/EnergyPlusV9-1-0/ExampleFiles/WindowTestsSimple.idf")
>>> copy_constructions(source_idf=src_idf, target_idf=idf)
```

Now we can assign some of those constructions to our IDF.

```
>>> for wall in idf.getsubsurfaces("wall"):
        wall.Construction_Name = "EXTERIOR"
>>> for roof in idf.getsubsurfaces("roof"):
        roof.Construction_Name = "ROOF31"
>>> for floor in idf.getsubsurfaces("floor"):
        floor.Construction_Name = "FLOOR38"
```

And run the simulation.

```
>>> idf.run(output_directory="tests/tutorial")
```

1.3.5 Summary

TODO: add a summary of the contents

CHAPTER 2

How-to guides

How to perform common tasks with *geomeppy*.

2.1 Basics

The API for *geomeppy* depends on replacing an *eppy* IDF object with a *geomeppy* IDF object. To use this, you will need to import IDF from *geomeppy* rather than directly from *eppy*.

In your scripts, use `from geomeppy import IDF` instead of `from eppy.modeleditor import IDF`. All other features of the current release version of *eppy* will still be available.

geomeppy then provides a simple Python API for actions on the IDF object:

2.1.1 Intersect and match surfaces

Intersect all surfaces:

```
IDF.intersect()
```

Set boundary conditions of surfaces:

```
IDF.match()
```

Intersect surfaces then set/update boundary conditions:

```
IDF.intersect_match()
```

2.1.2 Move an IDF

Move the whole IDF close to 0,0 on the x, y axes:

```
IDF.translate_to_origin()
```

Move the whole IDF to x + 50, y + 20:

```
IDF.translate([50, 20])
```

Move the whole IDF to z + 10:

```
IDF.translate([0, 0, 10])
```

2.1.3 Rotate

Rotate the IDF 90 degrees counterclockwise around the centre of its bounding box:

```
IDF.rotate(90)
```

2.1.4 Scale

Scale the IDF to double its size (default is on the xy axes):

```
IDF.scale(2)
```

Scale the IDF to double its size (in the xy axes):

```
IDF.scale(2, axes='xy')
```

Scale the IDF to double its size (in the z axis):

```
IDF.scale(2, axes='z')
```

2.1.5 Add windows to external walls

Set a WWR of 20% (the default) for all external walls:

```
IDF.set_wwr()
```

Set a WWR of 25% for all external walls:

```
IDF.set_wwr(wwr=0.25)
```

Set no windows on all external walls with azimuth of 90, and WWR of 20% on other walls:

```
IDF.set_wwr(wwr_map={90: 0})
```

Set a WWR of 30% for all external walls with azimuth of 90, and no windows on other walls:

```
IDF.set_wwr(wwr=0, wwr_map={90: 0.3})
```

If `wwr_map` is passed, it overrides any value passed to `wwr`, including the default of 0.2. However it only overrides it on walls which have an azimuth in the `wwr_map`. Any omitted walls' WWR will be set to the value in `wwr`. If you want to specify no windows for walls which are not specified in `wwr_map`, you must also set `wwr=None`.

2.1.6 Set constructions

Set a name for each construction in the model:

```
IDF.set_default_constructions()
```

2.1.7 View a simple 3D model

Show a zoomable, rotatable transparent model using *matplotlib*:

```
IDF.view_model()
```

2.1.8 Export a 3D OBJ file

Generate a model which can be viewed in external programs:

```
IDF.to_obj('mymodel.obj')
```

You can view the exported model here. Just drag the .obj file and .mtl file into the browser window.

2.1.9 Get all surfaces

```
IDF.getsurfaces()
```

2.1.10 Get all surfaces of a given type

```
IDF.getsurfaces('wall')
```

This only works if the surface type has been set in the IDF.

2.1.11 Get all subsurfaces

```
IDF.getsubsurfaces()
```

2.1.12 Get all subsurfaces of a given type

```
IDF.getsubsurfaces('window')
```

This only works if the surface type has been set in the IDF.

2.1.13 Add a block automatically

Automatically add a building block to the IDF:

```
IDF.add_block(...)
```

This method requires some explanation. The parameters required are:

```
name : str
    A name for the block.
coordinates : list
    A list of (x, y) tuples representing the building outline.
height : float
    The height of the block roof above ground level.
num_stories : int, optional
    The total number of stories including basement stories. Default : 1.
below_ground_stories : int, optional
    The number of stories below ground. Default : 0.
below_ground_storey_height : float, optional
    The height of each basement storey. Default : 2.5.
zoning : str, optional
    The rules to use in creating zones. Currently two options are available:
    - `by_storey`: sets each storey in the block as a Zone.
    - `core/perim`: creates core and perimeter Zones for each storey (see perim_
↪depth).
perim_depth : float, optional
    Depth of the perimeter zones if the core/perim zoning pattern is requested.␣
↪Default : 3.0.
```

The block generated will have boundary conditions set correctly and any intersections with adjacent blocks will be handled automatically. The surface type will be set to `wall`, `floor`, `ceiling` or `roof` for each surface. Constructions are not set automatically so these will need to be added afterwards in the normal way for Eppy.

2.1.14 Set surface coordinates

```
surface.setcoords(...)
```

For example:

```
wall = idf.newidfobject(
    'BUILDINGSURFACE:DETAILED',
    Name='awall',
    Surface_Type = 'wall',
    )
wall.setcoords([(0,0,1),(0,0,0),(1,0,0),(1,0,1)])
```

CHAPTER 3

Reference

Detailed documentation for geomeppy.

3.1 geomeppy

This package contains the main API for geomeppy, the *IDF* class. This class is a drop-in substitute for the *IDF* class in *eppy*. The functions documented here are not only for *geomeppy*. All the existing functions of *eppy.modeleditor.IDF* are also still available.

Any functions in this class can be considered as stable. Changes in the future will only add functionality (or fix bugs, should any arise).

class geomeppy.**IDF**(*idfname=None*, *epw=None*)

Bases: *geomeppy.patches.PatchedIDF*

Geometry-enabled IDF class, usable in the same way as Eppy's IDF.

This adds geometry functionality to Eppy's IDF class.

add_block(**args*, ***kwargs*)

Add a block to the IDF.

Parameters

- **name** – A name for the block.
- **coordinates** – A list of (x, y) tuples representing the building outline.
- **height** – The height of the block roof above ground level.
- **num_stories** – The total number of stories including basement stories. Default : 1.
- **below_ground_stories** – The number of stories below ground. Default : 0.
- **below_ground_storey_height** – The height of each basement storey. Default : 2.5.
- **zoning** – The zoning pattern of the block. Default : by_storey

- **perim_depth** – Depth of the perimeter zones if the core/perim zoning pattern is requested. Default : 3.0.

add_shading_block(**args*, ***kwargs*)
Add a shading block to the IDF.

Parameters

- **name** – A name for the block.
- **coordinates** – A list of (x, y) tuples representing the building outline.
- **height** – The height of the block roof above ground level.
- **num_stories** – The total number of stories including basement stories. Default : 1.
- **below_ground_stories** – The number of stories below ground. Default : 0.
- **below_ground_storey_height** – The height of each basement storey. Default : 2.5.

add_zone(*zone*)
Add a zone to the IDF.

Parameters **zone** – A Zone object holding details about the zone.

block = None

bounding_box()
Calculate the site bounding box.

Returns A polygon of the bounding box.

centroid
Calculate the centroid of the site bounding box.

Returns The centroid of the site bounding box.

copyidfobject(*idfobject*)
Add an IDF object to the IDF.

This has been monkey-patched to add the return value.

Parameters **idfobject** – The IDF object to copy. Usually from another IDF, or it can be used to copy within this IDF.

Returns EpBunch object.

getextensibleindex(*key*, *name*)
Get the index of the first extensible item.

Only for internal use. # TODO : hide this

key [str] The type of IDF object. This must be in ALL_CAPS.

name [str] The name of the object to fetch.

int

getiddgroupdict()
Return a idd group dictionary sample: {'Plant-Condenser Loops': ['PlantLoop', 'CondenserLoop'],

'Compliance Objects': ['Compliance:Building'], 'Controllers': ['Controller:WaterCoil',

'Controller:OutdoorAir', 'Controller:MechanicalVentilation', 'AirLoopHVAC:ControllerList'],

. . . }

dict

classmethod getiddname()
Get the name of the current IDD used by eppy.

str

getobject(*key*, *name*)
Fetch an IDF object given key and name.

key [str] The type of IDF object. This must be in ALL_CAPS.

name [str] The name of the object to fetch.

EpBunch object.

getshadingsurfaces(*surface_type=''*)
Return all subsurfaces in the IDF.

Parameters **surface_type** – Type of surface to get. Defaults to all.

Returns IDF surfaces.

getsubsurfaces(*surface_type=''*)
Return all subsurfaces in the IDF.

Parameters **surface_type** – Type of surface to get. Defaults to all.

Returns IDF surfaces.

getsurfaces(*surface_type=''*)
Return all surfaces in the IDF.

Parameters **surface_type** – Type of surface to get. Defaults to all.

Returns IDF surfaces.

idd_info = None

iddname = None

idfstr()
String representation of the IDF.

str

initnew(*fname*)
Use the current IDD and create a new empty IDF. If the IDD has not yet been initialised then this is done first.

fname [str, optional] Path to an IDF. This does not need to be set at this point.

initread(*idfname*)
Use the current IDD and read an IDF from file. If the IDD has not yet been initialised then this is done first.

idf_name [str] Path to an IDF file.

initreadtxt(*idftxt*)
Use the current IDD and read an IDF from text data. If the IDD has not yet been initialised then this is done first.

idftxt [str] Text representing an IDF file.

intersect()
Intersect all surfaces in the IDF.

intersect_match()
Intersect all surfaces in the IDF, then set boundary conditions.

match()
Set boundary conditions for all surfaces in the IDF.

new(*fname=None*)
Create a blank new idf file. Filename is optional.

fname [str, optional] Path to an IDF. This does not need to be set at this point.

newidfobject(*key*, *aname=''*, ***kwargs*)
Add a new idfobject to the model.

If you don't specify a value for a field, the default value will be set.

For example

```
newidfobject("CONSTRUCTION")
newidfobject("CONSTRUCTION",
    Name='Interior Ceiling_class',
    Outside_Layer='LW Concrete',
    Layer_2='soundmat')
```

Parameters

- **key** – The type of IDF object. This must be in ALL_CAPS.
- **aname** – This parameter is not used. It is left there for backward compatibility.
- **kwargs** – Keyword arguments in the format *field=value* used to set fields in the EnergyPlus object.

Returns EpBunch object.

popidfobject(*key*, *index*)
Pop an IDF object from the IDF.

key [str] The type of IDF object. This must be in ALL_CAPS.

index [int] The index of the object to pop.

EpBunch object.

printidf()
Print the IDF.

read()
Read the IDF file and the IDD file.

If the IDD file had already been read, it will not be read again.

Populates the following data structures:

```
- idfobjects : list
- model : list
- idd_info : list
- idd_index : dict
```

removeextensibles(*key*, *name*)
Remove extensible items in the object of key and name.

Only for internal use. # TODO : hide this

key [str] The type of IDF object. This must be in ALL_CAPS.

name [str] The name of the object to fetch.

EpBunch object

removeidfobject(*idfobject*)
Remove an IDF object from the IDF.

idfobject [EpBunch object] The IDF object to remove.

rotate(*angle*, *anchor=None*)
Rotate the IDF counterclockwise by the angle given.

Parameters

- **angle** – Angle (in degrees) to rotate by.
- **anchor** – Point around which to rotate. Default is the centre of the the IDF's bounding box.

run(***kwargs*)
This method wraps the following method:

rruunn(idf=None, weather=None, output_directory='', annual=False, design_day=False, idd=None, epmacro=False,
Wrapper around the EnergyPlus command line interface.

idf [str] Full or relative path to the IDF file to be run, or an IDF object.

weather [str] Full or relative path to the weather file.

output_directory [str, optional] Full or relative path to an output directory (default: 'run_outputs)

annual [bool, optional] If True then force annual simulation (default: False)

design_day [bool, optional] Force design-day-only simulation (default: False)

idd [str, optional] Input data dictionary (default: Energy+.idd in EnergyPlus directory)

epmacro [str, optional] Run EPMacro prior to simulation (default: False).

expandobjects [bool, optional] Run ExpandObjects prior to simulation (default: False)

readvars [bool, optional] Run ReadVarsESO after simulation (default: False)

output_prefix [str, optional] Prefix for output file names (default: eplus)

output_suffix [str, optional]

Suffix style for output file names (default: L) L: Legacy (e.g., eplustbl.csv) C: Capital (e.g., eplusTable.csv) D: Dash (e.g., eplus-table.csv)

version [bool, optional] Display version information (default: False)

verbose: str

Set verbosity of runtime messages (default: v) v: verbose q: quiet

ep_version: str EnergyPlus version, used to find install directory. Required if run() is called with an IDF file path rather than an IDF object.

str : status

CalledProcessError

AttributeError If no ep_version parameter is passed when calling with an IDF file path rather than an IDF object.

save (*filename=None*, *lineendings='default'*, *encoding='latin-1'*)
Save the IDF as a text file with the optional filename passed, or with the current idfname of the IDF.

filename [str, optional] Filepath to save the file. If None then use the IDF.idfname parameter. Also accepts a file handle.

lineendings [str, optional] Line endings to use in the saved file. Options are 'default', 'windows' and 'unix' the default is 'default' which uses the line endings for the current system.

encoding [str, optional] Encoding to use for the saved file. The default is 'latin-1' which is compatible with the EnergyPlus IDFEditor.

saveas (*filename*, *lineendings='default'*, *encoding='latin-1'*)
Save the IDF as a text file with the filename passed.

filename [str] Filepath to to set the idfname attribute to and save the file as.

lineendings [str, optional] Line endings to use in the saved file. Options are 'default', 'windows' and 'unix' the default is 'default' which uses the line endings for the current system.

encoding [str, optional] Encoding to use for the saved file. The default is 'latin-1' which is compatible with the EnergyPlus IDFEditor.

savecopy (*filename*, *lineendings='default'*, *encoding='latin-1'*)
Save a copy of the file with the filename passed.

filename [str] Filepath to save the file.

lineendings [str, optional] Line endings to use in the saved file. Options are 'default', 'windows' and 'unix' the default is 'default' which uses the line endings for the current system.

encoding [str, optional] Encoding to use for the saved file. The default is 'latin-1' which is compatible with the EnergyPlus IDFEditor.

scale (*factor*, *anchor=None*, *axes='xy'*)
Scale the IDF by a scaling factor.

Parameters

- **factor** – Factor to scale by.
- **anchor** – Point to scale around. Default is the centre of the the IDF's bounding box.
- **axes** – Axes to scale on. Default 'xy'.

set_default_constructions ()

set_wwr (*wwr=0.2*, *construction=None*, *force=False*, *wwr_map={}*, *orientation=None*)
Add strip windows to all external walls.

Different WWR can be applied to specific wall orientations using the *wwr_map* keyword arg. This map is a dict of wwr values, keyed by *wall.azimuth*, which overrides the default passed as *wwr*.

They can also be applied to walls oriented to a compass point, e.g. north, which will apply to walls which have an azimuth within 45 degrees of due north.

Parameters

- **wwr** – Window to wall ratio in the range 0.0 to 1.0.
- **construction** – Name of a window construction.
- **force** – True to remove all subsurfaces before setting the WWR.

- **wwr_map** – Mapping from wall orientation (azimuth) to WWR, e.g. {180: 0.25, 90: 0.2}.
- **orientation** – One of "north", "east", "south", "west". Walls within 45 degrees will be affected.

classmethod setidd(*iddinfo*, *iddindex*, *block*, *idd_version*)
Set the IDD to be used by eppy.

iddinfo [list] Comments and metadata about fields in the IDD.

block [list] Field names in the IDD.

classmethod setiddname(*iddname*, *testing=False*)
Set the path to the EnergyPlus IDD for the version of EnergyPlus which is to be used by eppy.

iddname [str] Path to the IDD file.

testing [bool] Flag to use if running tests since we may want to ignore the *IDDAlreadySetError*.

IDDAlreadySetError

to_obj(*fname=None*, *mtllib=None*)
Export an OBJ file representation of the IDF.

This can be used for viewing in tools which support the .obj format.

Parameters

- **fname** – A filename for the .obj file. If None we try to base it on IDF.idfname and change the filetype.
- **mtllib** – The name of a .mtl file to be referenced from the .obj file. If None, we use default.mtl.

translate(*vector*)
Move the IDF in the direction given by a vector.

Parameters **vector** – A vector to translate by.

translate_to_origin()
Move an IDF close to the origin so that it can be viewed in SketchUp.

view_model(*test=False*)
Show a zoomable, rotatable representation of the IDF.

3.1.1 geomeppy.recipes module

Recipes for making changes to EnergyPlus IDF files.

These are generally exposed and methods on the IDF object, e.g. *set_default_constructions(idf)* can be called on an existing *IDF* object like `myidf.set_default_constructions()`.

geomeppy.recipes.**rotate**(*surfaces*, *angle*)
Rotate all surfaces by an angle.

Parameters

- **surfaces** – A list of EpBunch objects or a mutable sequence.
- **angle** – An angle in degrees.

geomeppy.recipes.**rotate_coords**(*coords*, *radians*)
Rotate a set of coords by an angle in radians.

Parameters

- **coords** – A list of points.
- **radians** – The angle to rotate by.

Returns List of Vector3D objects.

geomeppy.recipes.**scale**(*surfaces*, *factor*, *axes*)
Scale all surfaces by a factor.

Parameters

- **surfaces** – A list of EpBunch objects.
- **factor** – Factor to scale the surfaces by.
- **axes** – Axes to scale on.

geomeppy.recipes.**scale_coords**(*coords*, *factor*, *axes='xy'*)
Scale a set of coords by a factor.

Parameters

- **coords** – A list of points.
- **factor** – Factor to scale the surfaces by.
- **axes** – Axes to scale on.

Returns A scaled polygon.

geomeppy.recipes.**set_default_construction**(*surface*)
Set default construction for a surface in the model.

Parameters **surface** – A model surface.

geomeppy.recipes.**set_default_constructions**(*idf*)
Set default constructions for surfaces in the model.

Parameters **idf** – The IDF object.

geomeppy.recipes.**set_wwr**(*idf*, *wwr=0.2*, *construction=None*, *force=False*, *wwr_map=None*, *orientation=None*)
Set the window to wall ratio on all external walls.

Parameters

- **idf** – The IDF to edit.
- **wwr** – The window to wall ratio.
- **construction** – Name of a window construction.
- **force** – True to remove all subsurfaces before setting the WWR.
- **wwr_map** – Mapping from wall orientation (azimuth) to WWR, e.g. {180: 0.25, 90: 0.2}.
- **orientation** – One of "north", "east", "south", "west". Walls within 45 degrees will be affected.

geomeppy.recipes.**translate**(*surfaces*, *vector*)
Translate all surfaces by a vector.

Parameters

- **surfaces** – A list of EpBunch objects.
- **vector** – Representation of a vector to translate by.

geomeppy.recipes.**translate_coords**(*coords*, *vector*)
Translate a set of coords by a direction vector.

Parameters

- **coords** – A list of points.
- **vector** – Representation of a vector to translate by.

Returns List of translated vectors.

geomeppy.recipes.**translate_to_origin**(*idf*)
Move an IDF close to the origin so that it can be viewed in SketchUp.

Parameters **idf** – The IDF to edit.

geomeppy.recipes.**window_vertices_given_wall**(*wall*, *wwr*)
Calculate window vertices given wall vertices and glazing ratio.

:: For each axis:

1) Translate the axis points so that they are centred around zero
2) **Either:**
   a) Multiply the z dimension by the glazing ratio to shrink it vertically
   b) Multiply the x or y dimension by 0.995 to keep inside the surface
3) Translate the axis points back to their original positions

Parameters

- **wall** – The wall to add a window on. We expect each wall to have four vertices.
- **wwr** – Window to wall ratio.

Returns Window vertices bounding a vertical strip midway up the surface.

3.1.2 geomeppy.view_geometry module

Tool for visualising geometry.

geomeppy.view_geometry.**main**(*fname=None*, *polygons=None*)

geomeppy.view_geometry.**view_idf**(*fname=None*, *idf_txt=None*, *test=False*, *idf=None*)
Display an IDF for inspection.

Parameters

- **fname** – Path to the IDF.
- **idf_txt** – The string representation of an IDF.

geomeppy.view_geometry.**view_polygons**(*polygons*)
Display a collection of polygons for inspection.

Parameters **polygons** – A dict keyed by colour, containing Polygon3D objects to show in that colour.

3.1.3 geomeppy.builder module

Build IDF geometry from minimal inputs.

class geomeppy.builder.**Block**(*name*, *coordinates*, *height*, *num_stories=1*, *below_ground_stories=0*, *below_ground_storey_height=2.5*, *zoning='by_storey'*, *perim_depth=3.0*)

Bases: object

ceiling_heights
Ceiling height for each storey in the block.

Returns A list of ceiling heights.

ceilings
Coordinates for each ceiling in the block.

Returns Coordinates for all ceilings.

floor_heights
Floor height for each storey in the block.

Returns A list of floor heights.

floors
Coordinates for each floor in the block.

Returns Coordinates for all floors.

footprint
Ground level outline of the block.

Returns A 2D outline of the block.

lowest_floor_level
Floor level of the lowest basement storey.

Returns Lowest floor height.

roofs
Coordinates for each roof of the block.

This returns a list with an entry for each floor for consistency with the other properties of the Block object, but should only have roof coordinates in the list in the final position.

Returns Coordinates for all roofs.

storey_height
Height of above ground stories.

Returns Average storey height.

stories
A list of dicts of the surfaces of each storey in the block.

Returns list of dicts

Example dict format::

{'floors': [...], 'ceilings': [...], 'walls': [...], 'roofs': [...], }

surfaces
Coordinates for all the surfaces in the block.

Returns Coordinates for all surfaces.

walls
Coordinates for each wall in the block.

These are ordered as a list of lists, one for each storey.

Returns Coordinates for all walls.

class geomeppy.builder.**Zone**(*name*, *surfaces*)
Bases: object

Represents a single zone for translation into an IDF.

3.1.4 geomeppy.extractor module

Module for extracting the geometry from an existing IDF.

There is the option to copy:

- thermal zone description and geometry
- surface construction elements

geomeppy.extractor.**copy_constructions**(*source_idf*, *target_idf=None*, *fname='new.idf'*)
Extract construction objects from a source IDF and add them to a target IDF or a new IDF.

Parameters

- **source_idf** – An IDF to source objects from.
- **target_idf** – An optional IDF to add objects to. If none is passed, a new IDF is returned.
- **fname** – A name for the new IDF created if no target IDF is passed in. Default: "new.idf".

Returns Either the target IDF or a new IDF containing the construction objects.

geomeppy.extractor.**copy_geometry**(*source_idf*, *target_idf=None*, *fname='new.idf'*)
Extract geometry objects from a source IDF and add them to a target IDF or a new IDF..

Parameters

- **source_idf** – An IDF to source objects from.
- **target_idf** – An optional IDF to add objects to. If none is passed, a new IDF is returned.
- **fname** – A name for the new IDF created if no target IDF is passed in. Default: "new.idf".

Returns Either the target IDF or a new IDF containing the geometry objects.

geomeppy.extractor.**copy_group**(*source_idf*, *target_idf*, *group*)
Extract a group of objects from a source IDF and add them to a target IDF.

Parameters

- **source_idf** – An IDF to source objects from.
- **target_idf** – An IDF to add objects to.
- **group** – The name of the group of objects to copy.

Returns A new IDF containing the objects which belong to the group.

3.1.5 geomeppy.patches module

Monkey patches for changes to classes and functions in Eppy. These include fixes which have not yet made it to the released version of Eppy. These will be removed if/when they are added to Eppy.

class geomeppy.patches.**EpBunch**(*obj*, *objls*, *objidd*, **args*, ***kwargs*)
Bases: eppy.bunch_subclass.EpBunch

Monkeypatched EpBunch to add the setcoords function.

setcoords(*poly*, *ggr=None*)
Set the coordinates of a surface.

Parameters

- **poly** – Either a Polygon3D object of a list of (x,y,z) tuples.
- **ggr** – A GlobalGeometryRules IDF object. Defaults to None.

class geomeppy.patches.**PatchedIDF**(*idfname=None*, *epw=None*)
Bases: eppy.modeleditor.IDF

Monkey-patched IDF.

Patched to add read (to add additional functionality) and to fix copyidfobject and newidfobject.

copyidfobject(*idfobject*)
Add an IDF object to the IDF.

This has been monkey-patched to add the return value.

Parameters **idfobject** – The IDF object to copy. Usually from another IDF, or it can be used to copy within this IDF.

Returns EpBunch object.

newidfobject(*key*, *aname=''*, ***kwargs*)
Add a new idfobject to the model.

If you don't specify a value for a field, the default value will be set.

For example

```
newidfobject("CONSTRUCTION")
newidfobject("CONSTRUCTION",
    Name='Interior Ceiling_class',
    Outside_Layer='LW Concrete',
    Layer_2='soundmat')
```

Parameters

- **key** – The type of IDF object. This must be in ALL_CAPS.
- **aname** – This parameter is not used. It is left there for backward compatibility.
- **kwargs** – Keyword arguments in the format *field=value* used to set fields in the EnergyPlus object.

Returns EpBunch object.

read()
Read the IDF file and the IDD file.

If the IDD file had already been read, it will not be read again.

Populates the following data structures:

```
- idfobjects : list
- model : list
- idd_info : list
- idd_index : dict
```

geomeppy.patches.**addthisbunch**(*bunchdt*, *data*, *commdct*, *thisbunch*, *_idf*)
Add an object to the IDF. Monkeypatched to return the object.

thisbunch usually comes from another idf file or it can be used to copy within the idf file.

Parameters

- **bunchdt** – Dict of lists of idf_MSequence objects in the IDF.
- **data** – Eplusdata object containing representions of IDF objects.
- **commdct** – Descriptions of IDF fields from the IDD.
- **thisbunch** – The object to add to the model.
- **_idf** – The IDF object. Not used either here or in Eppy but kept for consistency with Eppy.

Returns The EpBunch object added.

geomeppy.patches.**idfreader1**(*fname*, *iddfile*, *theidf*, *conv=True*, *commdct=None*, *block=None*)
Read idf file and return bunches.

Parameters

- **fname** – Name of the IDF file to read.
- **iddfile** – Name of the IDD file to use to interpret the IDF.
- **conv** – If True, convert strings to floats and integers where marked in the IDD. Defaults to None.
- **commdct** – Descriptions of IDF fields from the IDD. Defaults to None.
- **block** – EnergyPlus field ID names of the IDF from the IDD. Defaults to None.

Returns bunchdt Dict of lists of idf_MSequence objects in the IDF.

Returns block EnergyPlus field ID names of the IDF from the IDD.

Returns data Eplusdata object containing representions of IDF objects.

Returns commdct List of names of IDF objects.

Returns idd_index A pair of dicts used for fast lookups of names of groups of objects.

Returns versiontuple Version of EnergyPlus from the IDD.

geomeppy.patches.**makeabunch**(*commdct*, *obj*, *obj_i*)
Make a bunch from the object.

Parameters

- **commdct** – Descriptions of IDF fields from the IDD.
- **obj** – List of field values in an object.
- **obj_i** – Index of the object in commdct.

Returns EpBunch object.

geomeppy.patches.**makebunches**(*data*, *commdct*, *theidf*)
Make bunches with data.

Parameters

- **data** – Eplusdata object containing representions of IDF objects.
- **commdct** – Descriptions of IDF fields from the IDD.
- **theidf** – The IDF object.

Returns Dict of lists of idf_MSequence objects in the IDF.

geomeppy.patches.**obj2bunch**(*data*, *commdct*, *obj*)
Make a new bunch object using the data object.

Parameters

- **data** – Eplusdata object containing representions of IDF objects.
- **commdct** – Descriptions of IDF fields from the IDD.
- **obj** – List of field values in an object.

Returns EpBunch object.

geomeppy.patches.**readdatacommdct1**(*idfname*, *iddfile='Energy+.idd'*, *commdct=None*, *block=None*)
Read the idf file.

This is patched so that the IDD index is not lost when reading a new IDF without reloading the modeleditor module.

Parameters

- **idfname** – Name of the IDF file to read.
- **iddfile** – Name of the IDD file to use to interpret the IDF.
- **commdct** – Descriptions of IDF fields from the IDD. Defaults to None.
- **block** – EnergyPlus field ID names of the IDF from the IDD. Defaults to None.

Returns block EnergyPlus field ID names of the IDF from the IDD.

Returns data Eplusdata object containing representions of IDF objects.

Returns commdct List of names of IDF objects.

Returns idd_index A pair of dicts used for fast lookups of names of groups of objects.

3.1.6 geomeppy.utilities module

Utilities for use in geomeppy.

geomeppy.utilities.**almostequal**(*first*, *second*, *places=7*)
Tests a range of types for near equality.

3.1.7 Subpackages

geomeppy.geom package

Submodules

geomeppy.geom.clippers module

Perform clipping operations on Polygons

PyClipper is used for clipping. It's a wrapper for the C++ version of the Clipper library.

We implement a few of the functions of PyClipper here as *.difference*, *.intersect*, and *.union* methods of the *Clipper2D* and *Clipper3D* classes. These are then used as mixins for the *Polygon2D* and *Polygon3D* classes.

class geomeppy.geom.clippers.**Clipper2D**
Bases: object

This class is used to add clipping functionality to the Polygon2D class.

difference(*poly*)
Difference from another polygon.

Parameters **poly** – The clip polygon.

Returns A list of Polygons representing the difference.

intersect(*poly*)
Intersect with another polygon.

Parameters **poly** – The clip polygon.

Returns False if no intersection, otherwise a list of Polygons representing each intersection.

union(*poly*)
Union with another polygon.

Parameters **poly** – The clip polygon.

Returns A list of Polygons.

class geomeppy.geom.clippers.**Clipper3D**
Bases: *geomeppy.geom.clippers.Clipper2D*

This class is used to add clipping functionality to the Polygon3D class.

geomeppy.geom.intersect_match module

Intersect and match all surfaces in an IDF.

geomeppy.geom.intersect_match.**intersect_idf_surfaces**(*idf*)
Intersect all surfaces in an IDF.

Parameters **idf** – The IDF.

geomeppy.geom.intersect_match.**match_idf_surfaces**(*idf*)
Match all surfaces in an IDF.

Parameters **idf** – The IDF.

geomeppy.geom.intersect_match.**sorted_tuple**(*m*, *s*)
Used as a key for the matches.

geomeppy.geom.polygons module

Heavy lifting geometry for IDF surfaces.

class geomeppy.geom.polygons.**Polygon**(*vertices*)

Bases: *geomeppy.geom.clippers.Clipper2D*, collections.abc.MutableSequence

Base class for 2D and 3D polygons.

area

bounding_box

buffer(*distance=None*, *join_style=2*)

Returns a representation of all points within a given distance of the polygon.

Parameters **join_style** – The styles of joins between offset segments: 1 (round), 2 (mitre), and 3 (bevel).

centroid

The centroid of a polygon.

edges

A list of edges represented as Segment objects.

insert(*key*, *value*)

S.insert(index, value) – insert value before index

invert_orientation()

Reverse the order of the vertices.

This can be used to create a matching surface, e.g. the other side of a wall.

Returns A polygon.

is_convex

n_dims

normal_vector

points_matrix

Matrix representing the points in a polygon.

Format:: [[x1, x2,... xn] [y1, y2,... yn] [z1, z2,... zn] # all 0 for 2D polygon

vector_class

vertices_list

A list of the vertices in the format required by pyclipper.

Returns A list of tuples like [(x1, y1), (x2, y2),... (xn, yn)].

xs

ys

zs

class geomeppy.geom.polygons.**Polygon2D**(*vertices*)

Bases: *geomeppy.geom.polygons.Polygon*

Two-dimensional polygon.

n_dims = 2

normal_vector

project_to_3D(*example3d*)

Project the 2D polygon rotated into 3D space.

This is used to return a previously rotated 3D polygon back to its original orientation, or to to put polygons generated from pyclipper into the desired orientation.

Parameters **example3D** – A 3D polygon in the desired plane.

Returns A 3D polygon.

vector_class
alias of *geomeppy.geom.vectors.Vector2D*

zs

class geomeppy.geom.polygons.**Polygon3D**(*vertices*)
Bases: *geomeppy.geom.clippers.Clipper3D*, *geomeppy.geom.polygons.Polygon*

Three-dimensional polygon.

distance
Distance from the origin to the polygon.

Where v[0] * x + v[1] * y + v[2] * z = a is the equation of the plane containing the polygon (and where v is the polygon normal vector).

Returns The distance from the origin to the polygon.

from_wkt(*wkt_poly*)
Convert a wkt representation of a polygon to GeomEppy.

This also accounts for the possible presence of inner rings by linking them to the outer ring.

Parameters **wkt_poly** – A text representation of a polygon in well known text (wkt) format.

Returns A polygon.

is_clockwise(*viewpoint*)
Check if vertices are ordered clockwise

This function checks the vertices as seen from the viewpoint.

Parameters **viewpoint** – A point from which to view the polygon.

Returns True if vertices are ordered clockwise when observed from the given viewpoint.

is_coplanar(*other*)
Check if polygon is in the same plane as another polygon.

This includes the same plane but opposite orientation.

Parameters **other** – Another polygon.

Returns True if the two polygons are coplanar, else False.

is_horizontal
Check if polygon is in the xy plane.

Returns True if the polygon is in the xy plane, else False.

n_dims = 3

normal_vector
Unit normal vector perpendicular to the polygon in the outward direction.

We use Newell's Method since the cross-product of two edge vectors is not valid for concave polygons. https://www.opengl.org/wiki/Calculating_a_Surface_Normal#Newell.27s_Method

normalize_coords(*ggr*)
Order points, respecting the global geometry rules

Parameters **ggr** – EnergyPlus GlobalGeometryRules object.

Returns The normalized polygon.

order_points(*starting_position*)
Reorder the vertices based on a starting position rule.

Parameters **starting_position** – The string that defines vertex starting position in EnergyPlus.

Returns The reordered polygon.

outside_point(*entry_direction='counterclockwise'*)
Return a point outside the zone to which the surface belongs.

The point will be outside the zone, respecting the global geometry rules for vertex entry direction.

Parameters **entry_direction** – Either "clockwise" or "counterclockwise", as seen from outside the space.

Returns A point vector.

project_to_2D()
Project the 3D polygon into 2D space.

This is so that we can perform operations on it using pyclipper library.

Project onto either the xy, yz, or xz plane. (We choose the one that avoids degenerate configurations, which is the purpose of proj_axis.)

Returns A 2D polygon.

projection_axis
An axis which will not lead to a degenerate surface.

Returns The axis index.

vector_class
alias of *geomeppy.geom.vectors.Vector3D*

zs

geomeppy.geom.polygons.**bounding_box**(*polygons*)
The bounding box which encompasses all of the polygons in the x,y plane.

Parameters **polygons** – A list of polygons.

Returns A 2D polygon.

geomeppy.geom.polygons.**break_polygons**(*poly*, *hole*)
Break up a surface with a hole in it.

This produces two surfaces, neither of which have a hole in them.

Parameters

- **poly** – The surface with a hole in.
- **hole** – The hole.

Returns Two Polygon3D objects.

geomeppy.geom.polygons.**intersect**(*poly1*, *poly2*)
Calculate the polygons to represent the intersection of two polygons.

Parameters

- **poly1** – The first polygon.

- **poly2** – The second polygon.

Returns A list of unique polygons.

geomeppy.geom.polygons.**is_convex_polygon**(*polygon*)

Return True if the polynomial defined by the sequence of 2D points is 'strictly convex': points are valid, side lengths non- zero, interior angles are strictly between zero and a straight angle, and the polygon does not intersect itself.

See: https://stackoverflow.com/a/45372025/1706564

:: NOTES:

1. Algorithm: the signed changes of the direction angles from one side to the next side must be all positive or all negative, and their sum must equal plus-or-minus one full turn (2 pi radians). Also check for too few, invalid, or repeated points.
2. No check is explicitly done for zero internal angles (180 degree direction-change angle) as this is covered in other ways, including the $n < 3$ check.

geomeppy.geom.polygons.**is_hole**(*surface*, *possible_hole*)

Identify if an intersection is a hole in the surface.

Check the intersection touches an edge of the surface. If it doesn't then it represents a hole, and this needs further processing into valid EnergyPlus surfaces.

Parameters

- **surface** – The first surface.
- **possible_hole** – The intersection into the surface.

Returns True if the possible hole is a hole in the surface.

geomeppy.geom.polygons.**normalize_coords**(*poly*, *outside_pt*, *ggr=None*)

Put coordinates into the correct format for EnergyPlus dependent on Global Geometry Rules (GGR).

Parameters

- **poly** – Polygon with new coordinates, but not yet checked for compliance with GGR.
- **outside_pt** – An outside point of the new polygon.
- **ggr** – EnergyPlus GlobalGeometryRules object.

Returns The normalized polygon.

geomeppy.geom.polygons.**project**(*pt*, *proj_axis*)

Project point pt onto either the xy, yz, or xz plane

We choose the one that avoids degenerate configurations, which is the purpose of proj_axis. See http://stackoverflow.com/a/39008641/1706564

geomeppy.geom.polygons.**project_inv**(*pt*, *proj_axis*, *a*, *v*)

Returns the vector w in the surface's plane such that project(w) equals x.

See http://stackoverflow.com/a/39008641/1706564

Parameters

- **pt** – A two-dimensional point.
- **proj_axis** – The axis to project into.
- **a** – Distance to the origin for the plane to project into.
- **v** – Normal vector of the plane to project into.

Returns The transformed point.

geomeppy.geom.polygons.**project_to_2D**(*vertices*, *proj_axis*)
Project a 3D polygon into 2D space.

Parameters

- **vertices** – The three-dimensional vertices of the polygon.
- **proj_axis** – The axis to project into.

Returns The transformed vertices.

geomeppy.geom.polygons.**project_to_3D**(*vertices*, *proj_axis*, *a*, *v*)
Project a 2D polygon into 3D space.

Parameters

- **vertices** – The two-dimensional vertices of the polygon.
- **proj_axis** – The axis to project into.
- **a** – Distance to the origin for the plane to project into.
- **v** – Normal vector of the plane to project into.

Returns The transformed vertices.

geomeppy.geom.polygons.**section**(*first*, *last*, *coords*)

geomeppy.geom.polygons.**set_entry_direction**(*poly*, *outside_pt*, *ggr=None*)
Check and set entry direction for a polygon.

Parameters

- **poly** – A polygon.
- **outside_pt** – A point beyond the outside face of the polygon.
- **ggr** – EnergyPlus global geometry rules

Returns A polygon with the vertices correctly oriented.

geomeppy.geom.polygons.**set_starting_position**(*poly*, *ggr=None*)
Check and set starting position.

geomeppy.geom.segments module

Segment class, representing a line segment.

class geomeppy.geom.segments.**Segment**(**vertices*)
Bases: object

Line segment in 3D.

geomeppy.geom.surfaces module

A collection of functions which act on surfaces or lists of surfaces.

geomeppy.geom.surfaces.**get_adjacencies**(*surfaces*)
Create a dictionary mapping surfaces to their adjacent surfaces.

Parameters **surfaces** – A mutable list of surfaces.

Returns Mapping of surfaces to adjacent surfaces.

geomeppy.geom.surfaces.**getidfplanes**(*surfaces*)
Fast access data structure for potentially matched surfaces.

Get a data structure populated with all the surfaces in the IDF, keyed by their distance from the origin, and their normal vector.

Parameters **surfaces** – List of all the surfaces.

Returns Mapping to look up IDF surfaces.

geomeppy.geom.surfaces.**minimal_set**(*polys*)
Remove overlaps from a set of polygons.

Parameters **polys** – List of polygons.

Returns List of polygons with no overlaps.

geomeppy.geom.surfaces.**populate_adjacencies**(*adjacencies*, *s1*, *s2*)
Update the adjacencies dict with any intersections between two surfaces.

Parameters

- **adjacencies** – Dict to contain lists of adjacent surfaces.
- **s1** – Object representing an EnergyPlus surface.
- **s2** – Object representing an EnergyPlus surface.

Returns An updated dict of adjacencies.

geomeppy.geom.surfaces.**set_coords**(*surface*, *coords*, *ggr*)
Update the coordinates of a surface.

Parameters

- **surface** – The surface to modify.
- **coords** – The new coordinates as lists of [x,y,z] lists.
- **ggr** – Global geometry rules.

geomeppy.geom.surfaces.**set_matched_surfaces**(*surface*, *matched*)
Set boundary conditions for two adjoining surfaces.

Parameters

- **surface** – The first surface.
- **matched** – The second surface.

geomeppy.geom.surfaces.**set_unmatched_surface**(*surface*, *vector*)
Set boundary conditions for a surface which does not adjoin another one.

Parameters

- **surface** – The surface.
- **vector** – The surface normal vector.

geomeppy.geom.transformations module

A module to handle translations, using Christopher Gohlke's transforms3d as far as possible, but also trying to respect the intent of the algorithms used in OpenStudio for the sake of consistency between tools based on EnergyPlus.

class geomeppy.geom.transformations.**Transformation**(*mat=None*)
Bases: object

geomeppy.geom.transformations.**align_face**(*polygon*)
Transformation to align face with z-axis.

Parameters **polygon** – Polygon to be aligned.

Returns Polygon3D aligned with the z-axis.

geomeppy.geom.transformations.**invert_align_face**(*original*, *poly2*)
Transformation to align face with original position.

Parameters

- **original** – Polygon in the desired orientation.
- **poly2** – Polygon previously aligned with *align_face*.

Returns Polygon returned to the original orientation.

geomeppy.geom.vectors module

2D and 3D vector classes.

These are used to represent points in 2D and 3D, as well as directions for translations.

class geomeppy.geom.vectors.**Vector2D**(**args*)
Bases: collections.abc.Sized, collections.abc.Iterable, typing.Generic

Two dimensional point.

as_array(*dims=3*)
Convert a point to a numpy array.

Converts a Vector3D to a numpy.array([x,y,z]) or a Vector2D to a numpy.array([x,y]). Ensures all values are floats since some other types cause problems in pyclipper (notably where sympy.Zero is used to represent 0.0).

Parameters

- **pt** – The point to convert.
- **dims** – Number of dimensions {default : 3}.

Returns Vector as a Numpy array.

as_tuple(*dims=3*)
Convert a point to a numpy array.

Convert a Vector3D to an (x,y,z) tuple or a Vector2D to an (x,y) tuple. Ensures all values are floats since some other types cause problems in pyclipper (notably where sympy.Zero is used to represent 0.0).

Parameters

- **pt** – The point to convert.
- **dims** – Number of dimensions {default : 3}.

Returns Vector as a tuple.

closest(*poly*)
Find the closest vector in a polygon.

Parameters **poly** – Polygon or Polygon3D

cross(*other*)

dot(*other*)

invert()

length
The length of a vector.

normalize()

relative_distance(*v2*)
A distance function for sorting vectors by distance.

This only provides relative distance, not actual distance since we only use it for sorting.

Parameters **v2** – Another vector.

Returns Relative distance between two point vectors.

set_length(*new_length*)

class geomeppy.geom.vectors.**Vector3D**(*x*, *y*, *z=0*)
Bases: *geomeppy.geom.vectors.Vector2D*

Three dimensional point.

geomeppy.geom.vectors.**inverse_vector**(*v*)
Convert a vector to the same vector but in the opposite direction

Parameters **v** – The vector.

Returns The vector reversed.

geomeppy.io package

This module contains code to interface with other geometry file formats. At present this is limited to .obj files, though we hope to include other formats in the future.

Submodules

geomeppy.io.obj module

Module for export of .obj files

The OBJ file format developed by Wavefront Technologies is used in programs like Blender to represent 3D models.

Geomeppy is able to output simple .obj files, and accompanying .mtl files to represent materials. This module contains the source code for this operation. It should be treated as developers' reference only. As a user, if you want to create a .obj file it is best to use the *IDF.to_obj()* function.

The generated files can be viewed online at https://3dviewer.net/. Drag the .obj file and the .mtl file into the browser and you will be able to interact with a zoomable, rotatable model of your IDF.

OBJ file specifications

MTL file specifications

Example polygon:

```
# vertices
v 0.0 0.0 0.0
v 1.0 0.0 0.0
v 1.0 0.0 1.0
v 0.0 0.0 1.0

# face
f 1// 2// 3// 4//
```

class geomeppy.io.obj.**ObjWriter**
Bases: object

Container class holding the data needed to generate the .obj file.

add_face(*coords*, *mtl*, *test=True*)

build_simple_surface(*surface*)

build_surface_with_subsurface(*surface*, *subsurface*)
Work around the perimeter of the outer surface, triangulating the surface.

faces = []

from_surfaces(*surfaces*, *subsurfaces*, *shading_surfaces*)

prepare_shadingsurfaces(*shading_surfaces*)

prepare_surfaces(*surfaces*, *subsurfaces*)

v_set = {}

vertices = []

write(*fname*, *mtllib*)
Write the .obj file.

Parameters

- **fname** – A filename for the .obj file.
- **mtllib** – The name of a .mtl file to be referenced from the .obj file.

geomeppy.io.obj.**export_to_obj**(*idf*, *fname=None*, *mtllib=None*)
Export an OBJ file representation of the IDF.

This can be used for viewing in tools which support the .obj format.

Parameters

- **idf** – An IDF to export.
- **fname** – A filename for the .obj file. If None we try to base it on IDF.idfname and change the filetype.
- **mtllib** – The name of a .mtl file to be referenced from the .obj file. If None, we use default.mtl.

Python Module Index

g

Index

I

L

M

N

O

P

R

S