GSSHAPY Documentation

Release 2.3.7

Nathan Swain, Alan D. Snow, and Scott D. Christensen

Aug 30, 2017

Contents

GsshaPy is an object relational model (ORM) for the Gridded Surface Subsurface Hydrologic Analysis (GSSHA) model and a toolkit to convert gridded input into GSSHA input.

CHAPTER 1

Contents

1.1 Introduction

Last Updated: April 10, 2017

GsshaPy is an object relational model (ORM) for the Gridded Surface Subsurface Hydrologic Analysis (GSSHA) model and a toolkit to convert gridded input into GSSHA input. The purpose of GsshaPy is to expose GSSHA files to a web development environment by reading them into an SQL database. The files can also be written back to file for model execution. GsshaPy is built on top of the powerful SQLAlchemy ORM.

1.1.1 What is GSSHA?

GSSHA is a physically-based, distributed hydrologic model. GSSHA is developed and maintained by Coastal and Hydraulics Laboratory (CHL) which is a member of the Engineer Research & Development Center of the United States Army Corps of Engineers (USACE). GSSHA is used to predict soil moisture as well as runoff and flooding on watersheds.

Note: For more information about GSSHA please visit the the gsshawiki .

1.1.2 Installation

Note: The spatial components of GsshaPy can rely heavily on the PostGIS spatial extension for the PostgreSQL database. To work with spatial data in GsshaPy you will need to use a PostgreSQL database with PostGIS 2.1 or greater enabled.

Linux/Mac

Download Miniconda & Install Miniconda

See: https://conda.io/miniconda.html

Install Miniconda

```
$ chmod +x miniconda.sh
$ ./miniconda.sh -b
$ export PATH=$HOME/miniconda2/bin:$PATH
$ conda update --yes conda python
```

Install gsshapy

```
$ conda create --name gssha python=2
$ source activate gssha
$ conda config --add channels conda-forge
(gssha)$ conda install --yes gsshapy
```

Install gsshapy for development:

```
$ git clone https://github.com/CI-WATER/gsshapy.git
$ cd gsshapy
$ conda env create -f conda_env.yml
$ source activate gssha
(gssha)$ conda config --add channels conda-forge
(gssha)$ conda install --yes pynio
(gssha)$ python setup.py develop
```

Note: When using a new terminal, always type *source activate gssha* before using GsshaPy.

Windows

Note: pynio installation instructions are not provided for Windows, so `HRRRtoGSSHA()` will not work.

Download & Install Miniconda

- Go to: http://conda.pydata.org/miniconda.html
- Download and run Windows Python 2 version installer
- Install at C:\Users\YOUR_USERNAME\Miniconda2 or wherever you want
- During installation, make Miniconda the default python and export to path

Install gsshapy:

Open up the CMD program. Then, enter each line separately.

```
> conda update --yes conda python
> conda create --name gssha python=2
> activate gssha
(gssha)> conda config --add channels conda-forge
(gssha)> conda install --yes gsshapy
```

Install gsshapy for development:

Download the code for gsshapy from https://github.com/CI-WATER/gsshapy or clone it using a git program.

Open up the CMD program. Then, enter each line separately.

```
> cd gsshapy
> conda env create -f conda_env.yml
> activate gssha
(gssha)> conda config --add channels conda-forge
(gssha)> python setup.py develop
```

Note: When using a new CMD terminal, always type *activate gssha* before using GsshaPy.

1.1.3 License

GsshaPy is released under the BSD 3-Clause license.

1.1.4 Source

The source code is available on GitHub: https://github.com/CI-WATER/gsshapy.git

1.1.5 Authors

Nathan Swain, Alan D. Snow, and Scott D. Christensen.

1.1.6 NSF Grant

GsshaPy was developed at Brigham Young University with support from the National Science Foundation (NSF) under Grant No. 1135482. GsshaPy is part of a larger effort known as CI-Water. The purpose of CI-Water is to develop cyber infrastructure for water resources decision support.

1.2 Logging

GsshaPy uses the default Python logging module. By default, nothing is logged anywhere. Here is how to configure your instance.

1.2.1 Print to console

To use the default logging:

```
import gsshapy
gsshapy.log_to_console()

# then use gsshapy
```

To set custom level:

```
import gsshapy
gsshapy.log_to_console(level='INFO')

# then use gsshapy
```

gsshapy.**log_to_console**(*status=True*, *level=None*)

Log events to the console.

Parameters

- **status** (*bool, Optional, Default=True*) – whether logging to console should be turned on(True) or off(False)
- **level** (*string, Optional, Default=None*) – level of logging; whichever level is chosen all higher levels will be logged. See: https://docs.python.org/2/library/logging.html#levels

1.2.2 Log to file

To use the default logging:

```
import gsshapy
gsshapy.log_to_file(filename='gsshapy_run.log')

# then use gsshapy
```

To set custom level:

```
import gsshapy
gsshapy.log_to_file(filename='gsshapy_run.log', level='INFO')

# then use gsshapy
```

gsshapy.**log_to_file**(*status=True*, *filename='/home/docs/.cache/gsshapy/log/gsshapy.log'*, *level=None*)

Log events to a file.

Parameters

- **status** (*bool, Optional, Default=True*) – whether logging to file should be turned on(True) or off(False)
- **filename** (*string, Optional, Default=None*) – path of file to log to
- **level** (*string, Optional, Default=None*) – level of logging; whichever level is chosen all higher levels will be logged. See: https://docs.python.org/2/library/logging.html#levels

1.3 Getting Started

Last Updated: July 30, 2014

This tutorial is provided to help you get started using GsshaPy. In this tutorial you will learn important GsshaPy concepts and how to:

1.3.1 Read Files to a Database

Last Updated: July 30, 2014

This page will provide an example of how GsshaPy can be used to read single a GSSHA model file into an SQL database. We will read in the project file from the Park City model that you downloaded on the previous page.

Initiate GsshaPy Database

The first step is to create a database and populate it with all of the GsshaPy tables. The database tools API for creating databases is located here: *Database Tools*

For this tutorial you will need to create a new database in a PostgreSQL database and enable the PostGIS extension. This can be done by following the instructions on the PostGIS website: http://postgis.net/docs/manual-2.1/postgis_installation.html#create_new_db_extensions .

Create a database user with password and a PostGIS enabled database with the following credentials:

- Username: gsshapy
- Password: pass
- Database: gsshapy_tutorial

Open a Python console and execute the following commands to populate the database with GsshaPy tables:

```
>>> from gsshapy.lib import db_tools as dbt
>>> sqlalchemy_url = dbt.init_postgresql_db(username='gsshapy', host='localhost',␣
↪database='gsshapy_tutorial', port='5432', password='pass')
```

This method returns an SQLAlchemy url. This url is used to create SQLAlchemy session objects for interacting with the database. In the Python console:

```
>>> session_maker = dbt.get_sessionmaker(sqlalchemy_url)
>>> session = session_maker()
```

Create a GsshaPy Object

We need to create an instance of the GsshaPy **ProjectFile** file class to be able to read the project file into the database. In the python console, import the **ProjectFile** file class and instantiate it to create new **ProjectFile** object:

```
>>> from gsshapy.orm import ProjectFile
>>> projectFile = ProjectFile()
```

Read the File into the Database

Next, define a few variables that will define the directory where the files are located and the name of the project file. Be sure to enter the path to where you unzipped the tutorial data as the directory variable. Invoke the `read()` method on `projectFile` to read the contents of the file into the database:

```
>>> readDirectory = '/path_to/tutorial-data'
>>> filename = 'parkcity.prj'
>>> projectFile.read(directory=readDirectory, filename=filename, session=session)
```

The contents of the project file has now been read into the database. The next tutorial will illustrate how you can query the data in the database using the GsshaPy objects.

Inspect Supporting Objects

As was mentioned in the introduction, GsshaPy file objects are often supported by other supporting objects. In the case of the project file, there is only one supporting object called *gsshapy.orm.ProjectCard*. The project file consists of a set of key value pairs called cards. Each card is stored using one of these project card objects. When you executed the `read()` method, it created an instance of *gsshapy.orm.ProjectCard* for each project card in the project file. These project file objects are accessible via the `projectCards` property of the project file object. To illustrate this concept, execute the following lines in the Python console:

```
>>> projectCards = projectFile.projectCards
>>> for card in projectCards:
...     print card
...

<ProjectCard: Name=WMS, Value=WMS 9.1 (64-Bit)>
<ProjectCard: Name=WATERSHED_MASK, Value="parkcity.msk">
<ProjectCard: Name=PROJECT_PATH, Value="">
<ProjectCard: Name=#LandSoil, Value="parkcity.lsf">
<ProjectCard: Name=#PROJECTION_FILE, Value="parkcity_prj.pro">
<ProjectCard: Name=NON_ORTHO_CHANNELS, Value=None>
<ProjectCard: Name=FLINE, Value="parkcity.map">
<ProjectCard: Name=METRIC, Value=None>
<ProjectCard: Name=GRIDSIZE, Value=90.000000>
<ProjectCard: Name=ROWS, Value=72>
<ProjectCard: Name=COLS, Value=67>
...........
```

Each project card object is summarized similar to the sampling above. You can access the card name and value using the properties of the project card:

```
>>> for card in projectCards:
...     print card.name, card.value
...

WMS WMS 9.1 (64-Bit)
WATERSHED_MASK "parkcity.msk"
PROJECT_PATH ""
#LandSoil "parkcity.lsf"
#PROJECTION_FILE "parkcity_prj.pro"
NON_ORTHO_CHANNELS None
FLINE "parkcity.map"
METRIC None
GRIDSIZE 90.000000
```

```
ROWS 72
COLS 67
..........
```

GsshaPy eliminates the need for you to manually parse the file. Instead, you can work with each file using an object oriented approach. Behind the scenes, SQLAlchemy issues queries to the database tables to populate objects with data. This will be illustrated more concretely in the next tutorial.

1.3.2 Query using GsshaPy Objects

Last Updated: July 30, 2014

Explore the Database

To prove that the exercise has actually done something, let's explore database. Before we do this using the GsshaPy objects, lets explore a little using the **psql** commandline utility. Open a new terminal or command prompt (leave the terminal with your Python prompt running) and issue the following commands:

```
$ psql -U gsshapy -d gsshapy_tutorial
```

Enter the password if prompted, which should be "pass" if you set up the database using the credentials in the last tutorial. You should now have an SQL prompt to your database. Issue the following command:

```
gsshapy_tutorial=> \dt

                    List of relations
 Schema |             Name              | Type  |  Owner
--------+-------------------------------+-------+---------
 public | cif_bcs_points                | table | gsshapy
 public | cif_breakpoint                | table | gsshapy
 public | cif_channel_input_files       | table | gsshapy
 public | cif_culverts                  | table | gsshapy
 public | cif_links                     | table | gsshapy
 public | cif_nodes                     | table | gsshapy
 public | cif_reservoir_points          | table | gsshapy
 public | cif_reservoirs                | table | gsshapy
 public | cif_trapezoid                 | table | gsshapy
 public | cif_upstream_links            | table | gsshapy
 public | cif_weirs                     | table | gsshapy

 ...

 public | prj_project_cards             | table | gsshapy
 public | prj_project_files             | table | gsshapy

 ...

 public | wms_dataset_files             | table | gsshapy
 public | wms_dataset_rasters           | table | gsshapy
(61 rows)
```

This will list all the tables in the gsshapy_tutorial database. If the database was initialized correctly, you should see a list of 60+ or so tables. The three letter prefix on the filename is associted with the file extension or in some cases the type of file. For example, there are two tables used to store project files: `prj_project_files` and

prj_project_cards. The project file table is not very interesting, so, we will query the prj_project_cards table. This can be done as follows:

```
gsshapy_tutorial=> SELECT * FROM prj_project_cards;

 id | projectFileID |        name        |                          value
----+---------------+--------------------+-------------------------------------------
↪--------
  1 |             1 | WMS                | WMS 9.1 (64-Bit)
  2 |             1 | WATERSHED_MASK     | "parkcity.msk"
  3 |             1 | PROJECT_PATH       | ""
  4 |             1 | #LandSoil          | "parkcity.lsf"
  5 |             1 | #PROJECTION_FILE   | "parkcity_prj.pro"
  6 |             1 | NON_ORTHO_CHANNELS |
  7 |             1 | FLINE              | "parkcity.map"
  8 |             1 | METRIC             |
  9 |             1 | GRIDSIZE           | 90.000000
 10 |             1 | ROWS               | 72
 11 |             1 | COLS               | 67

 ...

 37 |             1 | IN_HYD_LOCATION    | "parkcity.ihl"
 38 |             1 | OUT_HYD_LOCATION   | "parkcity.ohl"
 39 |             1 | CHAN_DEPTH         | "parkcity.cdp"
(39 rows)
```

Each record in the prj_project_cards table stores the name and value of one card in the project file. The prj_project_cards table is related to the prj_project_files table through a foreign key column called projectFileID (the column with all 1's).

Execute the following command to quit the **psql** program:

```
gsshapy_tutorial=> \q
```

Querying Using GsshaPy Objects

The **ProjectCard** *table* class maps to the prj_project_cards table and the **ProjectFile** class maps to the prj_project_files table. Instances of these classes can be used to query the database. Suppose we need to retrieve all of the project cards from a project file. We can use SQLAlchemy session object and SQL expression language to do this. Back in the Python console, execute the following:

```
>>> from gsshapy.orm import ProjectCard
>>> cards = session.query(ProjectCard).all()
>>> for card in cards:
...     print card
...
```

See also:

For an overview of the SQLAlchemy SQL expression language see the following tutorials: Object Relational Tutorial and SQL Expression Language.

As in the previous tutorial, the query returns a list of *gsshapy.orm.ProjectCard* objects that represent the records in the prj_project_cards table. The *gsshapy.orm.ProjectCard* class also has a relationship property called *projectFile* that maps to the associated *gsshapy.orm.ProjectFile* class. If we wanted to ensure

that we only queried for project cards that belong to the project file we read in during the first exercise, we could use the `filter()` method of the `query` object:

```
>>> cards = session.query(ProjectCard).filter(ProjectCard.projectFile == projectFile).
↪all()
>>> for card in cards:
...     print card
...
```

The result is the same as before, because we only have one project file read into the database. As illustrated in the previous tutorial, we could also use the relationship properties to issue the queries to the database:

```
>>> cards = projectFile.projectCards
>>> for card in cards:
...     print card
...
```

The later two methods are equivilent. This is only a micro tasting of the power of the SQLAlchemy query language. Please review the SQLAlchemy documentation for a more detailed explanation of querying.

1.3.3 Write Files from a Database

Last Updated: July 30, 2014

Reading GSSHA files is only half the story. GsshaPy is also able read a GsshaPy database and write the data back to the proper file formats. It is necessary to write the data back to the original file format to be able to execute the GSSHA simulation after modifying some input file in the database.

Retrieve Object from Database

Like reading to the database, we need and instance of a *gsshapy.orm.ProjectFile* class to call the `write()` method on. When writing, the **ProjectFile** instance is created by querying the database for the project file we wish to write. Issue the following command in the Python prompt:

```
>>> newProjectFile = session.query(ProjectFile).first()
```

The query instantiates the new project file object with the data from the database query.

Write to File

Now we call the `write()` method on the new instance of *gsshapy.orm.ProjectFile*, `newProjectFile`. This method requires three arguments: an SQLAlchemy session object, a directory to write to, and the name you wish the file to be saved as. Define these attributes and call the write method as follows:

```
>>> writeDirectory = '/path_to/tutorial-data/write'
>>> name = 'mymodel'
>>> newProjectFile.write(session=session, directory=writeDirectory, name=name)
```

If all has gone well, you will find a copy of the project file in the **write** directory. If you compare the file with the original you will notice some differences. Notice that most of the path prefixes have been changed to match the name of the project file. This is a GSSHA convention that is preserved by GsshaPy. If you change only the project file using GsshaPy, be sure it is written out with the same name as the original. Paths are stored as relative in the GsshaPy database. Consequently, all the paths will be written out again as relative paths.

Tip: If you need to prepend a directory to the paths in the project file, use the `appendDirectory()` method of a *gsshapy.orm.ProjectFile* object.

1.3.4 Read and Write Entire GSSHA Projects

Last Updated: July 30, 2014

Each of the GSSHA model files can be read or written in the same manner that was illustrated with the project file. Each has a `read()` and a `write()` method. However, the *gsshapy.orm.ProjectFile* class has several additional methods that can be used to read and write all or some of the GSSHA model files simultaneously. These methods are:

- readProject()
- readInput()
- readOutput()
- writeProject()
- writeInput()
- wrtieOutput()

The names are fairly self explanatory, but more detailed explanations are provided in the API documentation for the *gsshapy.orm.ProjectFile* class. In this tutorial we will learn how to read an entire project and write it back to file.

Read All Files

Create a new session for this part of the tutorial, but use the same database:

```
>>> from gsshapy.lib import db_tools as dbt
>>> sqlalchemy_url = dbt.sqlalchemy_url = dbt.init_postgresql_db(username='gsshapy',␣
↪host='localhost', database='gsshapy_tutorial', port='5432', password='pass')
>>> session_maker = dbt.get_sessionmaker(sqlalchemy_url)
>>> all_session = session_maker()
```

Instantiate a new *gsshapy.orm.ProjectFile* object:

```
>>> from gsshapy.orm import ProjectFile
>>> projectFileAll = ProjectFile()
```

Invoke the `readProject()` method to read all supported GSSHA input and output files into the database:

```
>>> readDirectory = '/path_to/tutorial-data'
>>> filename = 'parkcity.prj'
>>> projectFileAll.readProject(directory=readDirectory, projectFileName=filename,␣
↪session=all_session)
```

The process of reading all the files into the database takes a moment, so be patient. The `readInput()` and `readOutput` methods can be used to read only input files or output files, respectively. If the task you are using GsshaPy for is related to pre-processing the input files, you may want to use the `readInput()` method to save a little time on overhead. Similarly, if the task you are performing is related to post-processing the output files you may find the `readOutput()` method useful.

If you feel adventurous, you could use **psql** or PGAdminIII to investigate the database. Many of the tables will now be populated with data.

Write All Files

Now that all of the files have been read into the database, we can write them back out to file. Retrieve the project file from the database and invoke the `writeProject()` method:

```
>>> newProjectFileAll = all_session.query(ProjectFile).filter(ProjectFile.id == 2).
↪one()
>>> writeDirectory = '/path_to/tutorial-data/write'
>>> name = 'mymodel'
>>> newProjectFileAll.writeProject(session=all_session, directory=writeDirectory,␣
↪name=name)
```

Note: We filter our query using the project file id of 2, because this is the second project file we have read in during this set of tutorials. If you end up reading in several projects, you can easily change this to another id to retrieve the GSSHA project you desire.

All of the files that were read into the database should be written to file in the *write* directory of the tutorial files. For the files that use the project name prefix as filename convention, the prefix has been changed to match the name supplied by the user ('mymodel' if you followed the tutorial exactly). Like the read methods, there are two other write methods that can be used to write only the input files or only the output files: `writeInput()` and `writeOutput()`, respectively. Use `writeInput()` when you want to write the model out to execute a simulation.

1.3.5 Work with Spatial Data

Last Updated: August 1, 2014

Up to this point in the tutorial, you have not been using the spatial functionality in GsshaPy. This is where most of the progress occurred in version 2.0. In this tutorial you will learn how to read a project using spatial database objects. This means that instead of storing the points, lines, polygons, and rasters as plain text, they will be stored using the spatial field types provided by PostGIS (raster and geometry). Once stored in PostGIS spatial fields, the data is exposed to over 1000 spatial database functions that can be used in queries to convert the data to different formats (e.g.: KML, WKT, GeoJSON, PNG), transform the coordinate reference system, and perform geoprocessing tasks (e.g.: buffer, intersect, union).

Spatial Project Read

To read the GSSHA files into spatial fields in the database we setup as we did before, creating a session and an instance of the *gsshapy.orm.ProjectFile* class:

```
>>> from gsshapy.lib import db_tools as dbt
>>> from gsshapy.orm import ProjectFile
>>> sqlalchemy_url =  dbt.init_postgresql_db(username='gsshapy', host='localhost',␣
↪database='gsshapy_tutorial', port='5432', password='pass')
>>> session_maker = dbt.get_sessionmaker(sqlalchemy_url)
>>> spatiaSession = session_maker()
>>> spatialProjectFile = ProjectFile()
```

Then we call the `readProject()` method enabling spatial objects by setting the `spatial` argument to `True`. Invoking the `readProject()` method looks like this:

```
>>> readDirectory = '/path_to/tutorial-data'
>>> filename = 'parkcity.prj'
>>> spatialProjectFile.readProject(directory=readDirectory, projectFileName=filename,␣
↪session=spatialSession, spatial=True)
```

You will probably notice that reading the project into the database using spatial objects takes a little more time than when reading it without spatial objects. This delay is caused primarily by the conversion of the native GSSHA spatial formats to the PostGIS spatial formats. For example, GSSHA stores raster files in GRASS ASCII format while PostGIS stores rasters in the binary version of the Well Known Text format (Well Known Binary).

The `readProject()`, `readInput()`, and `readOutput()` methods attempt to automatically lookup the spatial reference ID using the projection file that can be included with GSSHA models. This uses a web service, so an internet connection is required. If the projection file is not included or no id can be found, the default spatial reference ID will be used: 4236 WGS 84. Warnings will be thrown in this case, but GsshaPy will not throw an error. If you want to force the spatial reference ID, pass it in manually using the `spatialReferenceID` parameter.

Spatial Read for Individual Files

You can also apply the spatial read methodology to individual files. Instantiate the file object for the file you wish to read into the database with spatial objects and call the `read()` method with the same spatial arguments as illustrated above. For example, let's read in the mask map file.

The file object that is used to read in the mask map is the *gsshapy.orm.RasterMapFile* object (use the *Supported Files* page to determine what objects are used to read each file). Automatic spatial reference id lookup is not a feature for the read methods of individual files. Use the *gsshapy.orm.ProjectionFile* class method, `lookupSpatialReferenceID`, too look up the srid or enter it manually. Create a new instance of this object and invoke the `read()` method pointing it at the mask map file (`parkcity.msk`):

```
>>> from gsshapy.orm import RasterMapFile, ProjectionFile
>>> filename = 'parkcity.msk'
>>> srid = ProjectionFile.lookupSpatialReferenceID('readDirectory, 'parkcity_prj.pro')
>>> maskMap = RasterMapFile()
>>> maskMap.read(directory=readDirectory, filename=filename, session=spatialSession,␣
↪spatial=True, spatialReferenceID=srid)
```

Note: Not all files have spatial data, so passing in the spatial arguments to the read methods of these objects has no effect on the reading process. Refer to the *API Documentation* for a file object to determine if it supports spatial objects.

Spatial Project Write

There is no change needed to write a project that has been read in spatially. Use the same write methods as illustrated in the previous tutorial. This will not be demonstrated here.

Spatial Visualizations

After a project or file object has been read into the database with spatial objects, it is exposed to a number of spatial methods that can be used to generate visualizations of the data in various formats.

To demonstrate how these methods can be used to generate spatial visualizations, we will use the `getModelSummaryAsKml()` method of the *gsshapy.orm.ProjectFile*. This method uses the mask map

and the stream network to generate a summary visualization of the GSSHA model. Define the path where you want to write the kml file to. Then, query the database for the project file that was read in as part of the spatial reading (id = 3 if you have been following the tutorial) and invoke the `getModelSummaryAsKml()` method on it:

```
>>> from gsshapy.orm import ProjectFile
>>> import os
>>> kml_path = os.path.join(writeDirectory, 'model_summary.kml')
>>> newSpatialProjectFile = spatialSession.query(ProjectFile).filter(ProjectFile.id␣
↪== 3).one()
>>> newSpatialProjectFile.getModelSummaryAsKml(session=spatialSession, path=kml_path)
```

You will find the `model_summary.kml` file in your **write** directory. If you have the Google Earth Desktop application, you can view the visualization. KML can also be loaded into the Google Maps and Google Earth web viewers to embed it in a website. You can experiment with the other spatial methods to understand how they work. Refer to the *API Documentation* for details in how to use each method.

Spatial Methods Available

File objects that include spatial methods include:

gsshapy.orm.WMSDatasetFile:

- getAsKmlGridAnimation()
- getAsKmlPngAnimation()

gsshapy.orm.ChannelInputFile:

- streamNetworkAsKml()
- streamNetworkAsWkt()
- streamNetworkAsGeoJson()

gsshapy.orm.LinkNodeDatasetFile:

- getAsKmlAnimation()

gsshapy.orm.ProjectFile:

- getModelSummaryAsKml()
- getModelSummaryAsWkt()
- getModelSummaryAsGeoJson()

The *gsshapy.base.GeometricObjectBase* offers several general purpose methods for objects that inherit from it:

- getAsKml()
- getAsWkt()
- getAsGeoJson()
- getSpatialReferenceId()

The *gsshapy.base.RasterObjectBase* offers several general purpose methods for objects that inherit from it:

- getAsKmlGrid()
- getAsKmlClusters()
- getAsKmlPng()

- getAsGrassAsciiGrid()

The full tutorial script can be downloaded here: `tutorial-script.py`

1.3.6 Requirements

Download the example GSSHA model files here: `tutorial-data.zip`.

Unzip the contents of the file into a safe location. This file will become the working directory for the tutorial. The *write* directory is purposely empty. The other files in this directory make up the input and output files for a GSSHA model of the Park City, Utah watershed.

This tutorial makes use of a PostGIS enabled PostgreSQL database. GsshaPy uses the PostGIS to store spatial features of the models and it uses several PostGIS database functions to generate the spatial visualizations. To learn how to install PostGIS, visit their website: http://postgis.net/documentation . If you are using a Mac, an excellent option for easily testing with PostGIS is the Postgres App: http://postgresapp.com/ .

After installing PostgreSQL with PostGIS, create a database called "gsshapy_tutorial" and enable the PostGIS extension on the database. Refer to the documentation on the PostGIS docs for how this is to be done. Managing roles and databases is made much simpler using the PGAdminIII graphical user interface for PostgreSQL. You can find PGAdminIII here: http://www.pgadmin.org/ .

The tutorial also requires that you are using some version of Python 2.7. GsshaPy has not ported to Python 3 at this time.

Summary of Requirements

- Tutorial Files: `tutorial-data.zip`
- GsshaPy 2.0+
- PostgreSQL 9.3+
- PostGIS 2.1+
- Python 2.7.x

1.3.7 Key Concepts

The key abstraction of GsshaPy are the GSSHA model files. Most GSSHA model files are text files and many of them use a card system for assigning model parameters. Some of the files are GRASS ASCII maps and some of the data in other files are spatial in nature (e.g.: Link node and WMS datasets).

File Objects

Each file is represented in GsshaPy by an object. The file objects are defined by classes that inherit from the `gsshapy.base.file_base.GsshaPyFileObjectBase`. This class defines the `read()` and `write()` methods that are used by all file objects to read the file into an SQL database and write them back out to file.

Supporting Objects

Most file objects are supported with several supporting objects. The purpose of these objects is to provide the contents of the files at a higher level abstraction to make them easier to work with. For example, the precipitation file is decomposed into three other objects including an object representing precipitation events, another representing the

rain gages, and another object representing each value in the precipitation time series. This makes modifying and working with precipitation files easier than worrying about individual lines in the text file.

Mapping Objects to Database Tables

Both the file classes and supporting object classes inherit from from the SQLAlchemy `declarative_base` class. The `declarative_base` class maps each class to a table in the database, among other things. The properties of the file and supporting classes define the columns and relationships of the corresponding tables in the database. Instances of these classes, then, represent individual records in the tables.

In most cases, a majority of the information in each file is stored in the database tables associated with the supporting classes. The file class `read()` and `write()` methods orchestrate the reading of data into the database and writing it back out by using the supporting classes.

See also:

For an explanation of the SQLAlchemy ORM see http://docs.sqlalchemy.org/en/rel_0_9/orm/tutorial.html . If you are not familiar with SQLAlchemy, it strongly recommended that you follow this tutorial before you continue, because GsshaPy relies heavily on SQLAlchemy ORM concepts.

1.4 Changes in Version 2.2

Last Updated: March 2, 2017

The release of GsshaPy 2.2.0 constitutes many changes which will be summarized here. Changes vary from minor bug fixes to completely new file objects for files not previously supported. Several non-reverse compatible changes were also made to make using GsshaPy more convenient. The following list covers the major changes:

1.4.1 New Modeling Methods

To run existing GSSHA models and couple output from Land Surface Models or RAPID, the *gsshapy.modeling.GSSHAFramework* class was created.

To create basic GSSHA models, the *gsshapy.modeling.GSSHAModel* class was created.

1.4.2 Methods to connect with Land Surface Models

Land Surface Model output to GSSHA input (GRIDtoGSSHA)

GRIDtoGSSHA

class gsshapy.grid.**GRIDtoGSSHA**(*gssha_project_folder*, *gssha_project_file_name*, *lsm_input_folder_path*, *lsm_search_card*, *lsm_lat_var='lat'*, *lsm_lon_var='lon'*, *lsm_time_var='time'*, *lsm_lat_dim='lat'*, *lsm_lon_dim='lon'*, *lsm_time_dim='time'*, *output_timezone=None*, *pangaea_loader=None*)

Bases: `object`

This class converts the LSM output data to GSSHA formatted input.

gssha_project_folder
`str` – Path to the GSSHA project folder

gssha_project_file_name
str – Name of the GSSHA elevation grid file.

lsm_input_folder_path
str – Path to the input folder for the LSM files.

lsm_search_card
str – Glob search pattern for LSM files. Ex. "*.nc".

lsm_lat_var
Optional[str] – Name of the latitude variable in the LSM netCDF files. Defaults to 'lat'.

lsm_lon_var
Optional[str] – Name of the longitude variable in the LSM netCDF files. Defaults to 'lon'.

lsm_time_var
Optional[str] – Name of the time variable in the LSM netCDF files. Defaults to 'time'.

lsm_lat_dim
Optional[str] – Name of the latitude dimension in the LSM netCDF files. Defaults to 'lat'.

lsm_lon_dim
Optional[str] – Name of the longitude dimension in the LSM netCDF files. Defaults to 'lon'.

lsm_time_dim
Optional[str] – Name of the time dimension in the LSM netCDF files. Defaults to 'time'.

output_timezone
Optional[tzinfo] – This is the timezone to output the dates for the data. Default is the timezone of your GSSHA model. This option does NOT currently work for NetCDF output.

pangaea_loader
Optional[str] – String to define loader used when opening pangaea dataset (Ex. 'hrrr'). Default is None.

Example:

```
from gsshapy.grid import GRIDtoGSSHA

g2g = GRIDtoGSSHA(gssha_project_folder='E:/GSSHA',
                  gssha_project_file_name='gssha.prj',
                  lsm_input_folder_path='E:/GSSHA/lsm-data',
                  lsm_search_card="*.nc",
                 )
```

lsm_precip_to_gssha_precip_gage(*out_gage_file*, *lsm_data_var*, *precip_type='RADAR'*)
This function takes array data and writes out a GSSHA precip gage file. See: http://www.gsshawiki.com/Precipitation:Spatially_and_Temporally_Varied_Precipitation

---

Note:

GSSHA CARDS:

- PRECIP_FILE card with path to gage file
- RAIN_INV_DISTANCE or RAIN_THIESSEN

Parameters

- **out_gage_file** (*str*) – Location of gage file to generate.

- **lsm_data_var** (*str or list*) – This is the variable name for precipitation in the LSM files. If there is a string, it assumes a single variable. If it is a list, then it assumes the first element is the variable name for RAINC and the second is for RAINNC (see: http://www.meteo.unican.es/wiki/cordexwrf/OutputVariables).
- **precip_type** (*Optional[str]*) – This tells if the data is the ACCUM, RADAR, or GAGES data type. Default is 'RADAR'.

GRIDtoGSSHA Example:

```
from gsshapy.grid import GRIDtoGSSHA

#STEP 1: Initialize class
g2g = GRIDtoGSSHA(gssha_project_folder='/path/to/gssha_project',
                  gssha_project_file_name='gssha_project.prj',
                  lsm_input_folder_path='/path/to/wrf-data',
                  lsm_search_card='*.nc',
                  lsm_lat_var='XLAT',
                  lsm_lon_var='XLONG',
                  lsm_time_var='Times',
                  lsm_lat_dim='south_north',
                  lsm_lon_dim='west_east',
                  lsm_time_dim='Time',
                  )

#STEP 2: Generate GAGE data (from WRF)
g2g.lsm_precip_to_gssha_precip_gage(out_gage_file="E:/GSSHA/wrf_gage_1.gag",
                                    lsm_data_var=['RAINC', 'RAINNC'],
                                    precip_type='ACCUM')
```

HRRRtoGSSHA Example:

```
from gsshapy.grid import HRRRtoGSSHA

#STEP 1: Initialize class
h2g = HRRRtoGSSHA(
                  #YOUR INIT PARAMETERS HERE
                 )

#STEP 2: Generate GAGE data
g2g.lsm_precip_to_gssha_precip_gage(out_gage_file="E:/GSSHA/hrrr_gage_1.gag",
                                    lsm_data_var='prate',
                                    precip_type='RADAR')
```

lsm_data_to_arc_ascii (*data_var_map_array*, *main_output_folder=''*)

Writes extracted data to Arc ASCII file format into folder to be read in by GSSHA. Also generates the HMET_ASCII card file for GSSHA in the folder named 'hmet_file_list.txt'.

Warning: For GSSHA 6 Versions, for GSSHA 7 or greater, use lsm_data_to_subset_netcdf.

Note:

GSSHA CARDS:

- HMET_ASCII pointing to the hmet_file_list.txt
- LONG_TERM (see: http://www.gsshawiki.com/Long-term_Simulations:Global_parameters)

Parameters

- **data_var_map_array** (*list*) – Array to map the variables in the LSM file to the matching required GSSHA data.
- **main_output_folder** (*Optional[str]*) – This is the path to place the generated ASCII files. If not included, it defaults to os.path.join(self.gssha_project_folder, "hmet_ascii_data").

GRIDtoGSSHA Example:

```
from gsshapy.grid import GRIDtoGSSHA

#STEP 1: Initialize class
g2g = GRIDtoGSSHA(gssha_project_folder='/path/to/gssha_project',
                  gssha_project_file_name='gssha_project.prj',
                  lsm_input_folder_path='/path/to/wrf-data',
                  lsm_search_card='*.nc',
                  lsm_lat_var='XLAT',
                  lsm_lon_var='XLONG',
                  lsm_time_var='Times',
                  lsm_lat_dim='south_north',
                  lsm_lon_dim='west_east',
                  lsm_time_dim='Time',
                  )

#STEP 2: Generate ASCII DATA

#SEE: http://www.meteo.unican.es/wiki/cordexwrf/OutputVariables

#EXAMPLE DATA ARRAY 1: WRF GRID DATA BASED
data_var_map_array = [
                      ['precipitation_acc', ['RAINC', 'RAINNC']],
                      ['pressure', 'PSFC'],
                      ['relative_humidity', ['Q2', 'PSFC', 'T2']], #MUST BE␣
↪IN ORDER: ['SPECIFIC HUMIDITY', 'PRESSURE', 'TEMPERATURE']
                      ['wind_speed', ['U10', 'V10']], #['U_VELOCITY', 'V_
↪VELOCITY']
                      ['direct_radiation', ['SWDOWN', 'DIFFUSE_FRAC']], #MUST␣
↪BE IN ORDER: ['GLOBAL RADIATION', 'DIFFUSIVE FRACTION']
                      ['diffusive_radiation', ['SWDOWN', 'DIFFUSE_FRAC']],
↪#MUST BE IN ORDER: ['GLOBAL RADIATION', 'DIFFUSIVE FRACTION']
                      ['temperature', 'T2'],
                      ['cloud_cover' , 'CLDFRA'], #'CLOUD_FRACTION'
                     ]

g2g.lsm_data_to_arc_ascii(data_var_map_array)
```

HRRRtoGSSHA Example:

```
from gsshapy.grid import HRRRtoGSSHA

#STEP 1: Initialize class
h2g = HRRRtoGSSHA(
                   #YOUR INIT PARAMETERS HERE
                  )
```

```
#STEP 2: Generate ASCII DATA

#EXAMPLE DATA ARRAY 1: HRRR GRID DATA BASED
data_var_map_array = [
                      ['precipitation_rate', 'prate'],
                      ['pressure', 'sp'],
                      ['relative_humidity', '2r'],
                      ['wind_speed', ['10u', '10v']],
                      ['direct_radiation_cc', ['dswrf', 'tcc']],
                      ['diffusive_radiation_cc', ['dswrf', 'tcc']],
                      ['temperature', 't'],
                      ['cloud_cover_pc' , 'tcc'],
                     ]

h2g.lsm_data_to_arc_ascii(data_var_map_array)
```

lsm_data_to_subset_netcdf(*netcdf_file_path*, *data_var_map_array*, *resample_method=None*)
Writes extracted data to the NetCDF file format

> **Warning:** The NetCDF GSSHA file is only supported in GSSHA 7 or greater.

Note:

GSSHA CARDS:

- HMET_NETCDF pointing to the netcdf_file_path
- LONG_TERM (see: http://www.gsshawiki.com/Long-term_Simulations:Global_parameters)

Parameters

- **netcdf_file_path** (*string*) – Path to output the NetCDF file for GSSHA.
- **data_var_map_array** (*list*) – Array to map the variables in the LSM file to the matching required GSSHA data.
- **resample_method** (*Optional[gdalconst]*) – Resample input method to match hmet data to GSSHA grid for NetCDF output. Default is None.

GRIDtoGSSHA Example:

```
from gsshapy.grid import GRIDtoGSSHA

#STEP 1: Initialize class
g2g = GRIDtoGSSHA(gssha_project_folder='/path/to/gssha_project',
                  gssha_project_file_name='gssha_project.prj',
                  lsm_input_folder_path='/path/to/wrf-data',
                  lsm_search_card='*.nc',
                  lsm_lat_var='XLAT',
                  lsm_lon_var='XLONG',
                  lsm_time_var='Times',
                  lsm_lat_dim='south_north',
                  lsm_lon_dim='west_east',
                  lsm_time_dim='Time',
                  )
```

```
#STEP 2: Generate NetCDF DATA

#EXAMPLE DATA ARRAY 1: WRF GRID DATA BASED
#SEE: http://www.meteo.unican.es/wiki/cordexwrf/OutputVariables

data_var_map_array = [
                      ['precipitation_acc', ['RAINC', 'RAINNC']],
                      ['pressure', 'PSFC'],
                      ['relative_humidity', ['Q2', 'PSFC', 'T2']], #MUST BE␣
↪IN ORDER: ['SPECIFIC HUMIDITY', 'PRESSURE', 'TEMPERATURE']
                      ['wind_speed', ['U10', 'V10']], #['U_VELOCITY', 'V_
↪VELOCITY']
                      ['direct_radiation', ['SWDOWN', 'DIFFUSE_FRAC']], #MUST␣
↪BE IN ORDER: ['GLOBAL RADIATION', 'DIFFUSIVE FRACTION']
                      ['diffusive_radiation', ['SWDOWN', 'DIFFUSE_FRAC']],
↪#MUST BE IN ORDER: ['GLOBAL RADIATION', 'DIFFUSIVE FRACTION']
                      ['temperature', 'T2'],
                      ['cloud_cover' , 'CLDFRA'], #'CLOUD_FRACTION'
                     ]

g2g.lsm_data_to_subset_netcdf("E/GSSHA/gssha_wrf_data.nc",
                              data_var_map_array)
```

HRRRtoGSSHA Example:

```
from gsshapy.grid import HRRRtoGSSHA

#STEP 1: Initialize class
h2g = HRRRtoGSSHA(
                  #YOUR INIT PARAMETERS HERE
                 )

#STEP 2: Generate NetCDF DATA

#EXAMPLE DATA ARRAY 2: HRRR GRID DATA BASED
data_var_map_array = [
                      ['precipitation_rate', 'prate'],
                      ['pressure', 'sp'],
                      ['relative_humidity', '2r'],
                      ['wind_speed', ['10u', '10v']],
                      ['direct_radiation_cc', ['dswrf', 'tcc']],
                      ['diffusive_radiation_cc', ['dswrf', 'tcc']],
                      ['temperature', 't'],
                      ['cloud_cover_pc' , 'tcc'],
                     ]

h2g.lsm_data_to_subset_netcdf("E:/GSSHA/gssha_wrf_data.nc",
                              data_var_map_array)
```

HMET ASCII UPDATE

gsshapy.grid.grid_to_gssha.**update_hmet_card_file**(*hmet_card_file_path*, *new_hmet_data_path*)

This function updates the paths in the HMET card file to the new location of the HMET data. This is necessary because the file paths are absolute and will need to be updated if moved.

Parameters

- **hmet_card_file_path** (*str*) – Location of the file used for the HMET_ASCII card.
- **new_hmet_data_path** (*str*) – Location where the HMET ASCII files are currently.

Example:

```
new_hmet_data_path = "E:\GSSHA\new_hmet_directory"
hmet_card_file_path = "E:\GSSHA\hmet_card_file.txt"

update_hmet_card_file(hmet_card_file_path, new_hmet_data_path)
```

HRRR output to GSSHA input (HRRRtoGSSHA)

http://rapidrefresh.noaa.gov/

HRRRtoGSSHA

class gsshapy.grid.**HRRRtoGSSHA**(*gssha_project_folder*, *gssha_project_file_name*, *lsm_input_folder_path*, *lsm_search_card*, *lsm_lat_var='gridlat_0'*, *lsm_lon_var='gridlon_0'*, *lsm_time_var='time'*, *lsm_lat_dim='ygrid_0'*, *lsm_lon_dim='xgrid_0'*, *lsm_time_dim='time'*, *output_timezone=None*)

Bases: gsshapy.grid.grid_to_gssha.GRIDtoGSSHA

This class converts the HRRR output data to GSSHA formatted input. This class inheris from class:*GRIDtoGSSHA*.

gssha_project_folder
str – Path to the GSSHA project folder

gssha_project_file_name
str – Name of the GSSHA elevation grid file.

lsm_input_folder_path
str – Path to the input folder for the LSM files.

lsm_search_card
str – Glob search pattern for LSM files. Ex. "*.grib2".

lsm_lat_var
Optional[str] – Name of the latitude variable in the LSM netCDF files. Defaults to 'lat'.

lsm_lon_var
Optional[str] – Name of the longitude variable in the LSM netCDF files. Defaults to 'lon'.

lsm_time_var
Optional[str] – Name of the time variable in the LSM netCDF files. Defaults to 'time'.

lsm_lat_dim
Optional[str] – Name of the latitude dimension in the LSM netCDF files. Defaults to 'lat'.

lsm_lon_dim
Optional[str] – Name of the longitude dimension in the LSM netCDF files. Defaults to 'lon'.

lsm_time_dim
Optional[str] – Name of the time dimension in the LSM netCDF files. Defaults to 'time'.

output_timezone
Optional[tzinfo] – This is the timezone to output the dates for the data. Default is the timezone of your GSSHA model. This option does NOT currently work for NetCDF output.

Example:

```
from gsshapy.grid import HRRRtoGSSHA

l2g = HRRRtoGSSHA(gssha_project_folder='E:\GSSHA',
                  gssha_project_file_name='gssha.prj',
                  lsm_input_folder_path='E:\GSSHA\hrrr-data',
                  lsm_search_card="*.grib2",
                  )

# example data var map
data_var_map_array = [
                        ['precipitation_rate', 'PRATE_P0_L1_GLC0'],
                        ['pressure', 'PRES_P0_L1_GLC0'],
                        ['relative_humidity', 'RH_P0_L103_GLC0'],
                        ['wind_speed', ['UGRD_P0_L103_GLC0', 'VGRD_P0_L103_GLC0']],
                        ['direct_radiation_cc', ['DSWRF_P0_L1_GLC0', 'TCDC_P0_L10_
↪GLC0']],
                        ['diffusive_radiation_cc', ['DSWRF_P0_L1_GLC0', 'TCDC_P0_
↪L10_GLC0']],
                        ['temperature', 'TMP_P0_L1_GLC0'],
                        ['cloud_cover_pc', 'TCDC_P0_L10_GLC0'],
                       ]
```

lsm_data_to_arc_ascii(*data_var_map_array*, *main_output_folder=''*)
Writes extracted data to Arc ASCII file format into folder to be read in by GSSHA. Also generates the HMET_ASCII card file for GSSHA in the folder named 'hmet_file_list.txt'.

> **Warning:** For GSSHA 6 Versions, for GSSHA 7 or greater, use lsm_data_to_subset_netcdf.

Note:

GSSHA CARDS:

- HMET_ASCII pointing to the hmet_file_list.txt
- LONG_TERM (see: http://www.gsshawiki.com/Long-term_Simulations:Global_parameters)

Parameters

- **data_var_map_array** (*list*) – Array to map the variables in the LSM file to the matching required GSSHA data.
- **main_output_folder** (*Optional[str]*) – This is the path to place the generated ASCII files. If not included, it defaults to os.path.join(self.gssha_project_folder, "hmet_ascii_data").

GRIDtoGSSHA Example:

```
from gsshapy.grid import GRIDtoGSSHA

#STEP 1: Initialize class
```

```
g2g = GRIDtoGSSHA(gssha_project_folder='/path/to/gssha_project',
                  gssha_project_file_name='gssha_project.prj',
                  lsm_input_folder_path='/path/to/wrf-data',
                  lsm_search_card='*.nc',
                  lsm_lat_var='XLAT',
                  lsm_lon_var='XLONG',
                  lsm_time_var='Times',
                  lsm_lat_dim='south_north',
                  lsm_lon_dim='west_east',
                  lsm_time_dim='Time',
                  )

#STEP 2: Generate ASCII DATA

#SEE: http://www.meteo.unican.es/wiki/cordexwrf/OutputVariables

#EXAMPLE DATA ARRAY 1: WRF GRID DATA BASED
data_var_map_array = [
                      ['precipitation_acc', ['RAINC', 'RAINNC']],
                      ['pressure', 'PSFC'],
                      ['relative_humidity', ['Q2', 'PSFC', 'T2']], #MUST BE␣
↪IN ORDER: ['SPECIFIC HUMIDITY', 'PRESSURE', 'TEMPERATURE']
                      ['wind_speed', ['U10', 'V10']], #['U_VELOCITY', 'V_
↪VELOCITY']
                      ['direct_radiation', ['SWDOWN', 'DIFFUSE_FRAC']], #MUST␣
↪BE IN ORDER: ['GLOBAL RADIATION', 'DIFFUSIVE FRACTION']
                      ['diffusive_radiation', ['SWDOWN', 'DIFFUSE_FRAC']],
↪#MUST BE IN ORDER: ['GLOBAL RADIATION', 'DIFFUSIVE FRACTION']
                      ['temperature', 'T2'],
                      ['cloud_cover' , 'CLDFRA'], #'CLOUD_FRACTION'
                     ]

g2g.lsm_data_to_arc_ascii(data_var_map_array)
```

HRRRtoGSSHA Example:

```
from gsshapy.grid import HRRRtoGSSHA

#STEP 1: Initialize class
h2g = HRRRtoGSSHA(
                  #YOUR INIT PARAMETERS HERE
                 )

#STEP 2: Generate ASCII DATA

#EXAMPLE DATA ARRAY 1: HRRR GRID DATA BASED
data_var_map_array = [
                      ['precipitation_rate', 'prate'],
                      ['pressure', 'sp'],
                      ['relative_humidity', '2r'],
                      ['wind_speed', ['10u', '10v']],
                      ['direct_radiation_cc', ['dswrf', 'tcc']],
                      ['diffusive_radiation_cc', ['dswrf', 'tcc']],
                      ['temperature', 't'],
                      ['cloud_cover_pc' , 'tcc'],
                     ]

h2g.lsm_data_to_arc_ascii(data_var_map_array)
```

lsm_data_to_subset_netcdf(*netcdf_file_path*, *data_var_map_array*, *resample_method=None*)
Writes extracted data to the NetCDF file format

> **Warning:** The NetCDF GSSHA file is only supported in GSSHA 7 or greater.

Note:

GSSHA CARDS:

- HMET_NETCDF pointing to the netcdf_file_path
- LONG_TERM (see: http://www.gsshawiki.com/Long-term_Simulations:Global_parameters)

Parameters

- **netcdf_file_path** (*string*) – Path to output the NetCDF file for GSSHA.
- **data_var_map_array** (*list*) – Array to map the variables in the LSM file to the matching required GSSHA data.
- **resample_method** (*Optional[gdalconst]*) – Resample input method to match hmet data to GSSHA grid for NetCDF output. Default is None.

GRIDtoGSSHA Example:

```
from gsshapy.grid import GRIDtoGSSHA

#STEP 1: Initialize class
g2g = GRIDtoGSSHA(gssha_project_folder='/path/to/gssha_project',
                  gssha_project_file_name='gssha_project.prj',
                  lsm_input_folder_path='/path/to/wrf-data',
                  lsm_search_card='*.nc',
                  lsm_lat_var='XLAT',
                  lsm_lon_var='XLONG',
                  lsm_time_var='Times',
                  lsm_lat_dim='south_north',
                  lsm_lon_dim='west_east',
                  lsm_time_dim='Time',
                  )

#STEP 2: Generate NetCDF DATA

#EXAMPLE DATA ARRAY 1: WRF GRID DATA BASED
#SEE: http://www.meteo.unican.es/wiki/cordexwrf/OutputVariables

data_var_map_array = [
                      ['precipitation_acc', ['RAINC', 'RAINNC']],
                      ['pressure', 'PSFC'],
                      ['relative_humidity', ['Q2', 'PSFC', 'T2']], #MUST BE␣
↪IN ORDER: ['SPECIFIC HUMIDITY', 'PRESSURE', 'TEMPERATURE']
                      ['wind_speed', ['U10', 'V10']], #['U_VELOCITY', 'V_
↪VELOCITY']
                      ['direct_radiation', ['SWDOWN', 'DIFFUSE_FRAC']], #MUST␣
↪BE IN ORDER: ['GLOBAL RADIATION', 'DIFFUSIVE FRACTION']
                      ['diffusive_radiation', ['SWDOWN', 'DIFFUSE_FRAC']],
↪#MUST BE IN ORDER: ['GLOBAL RADIATION', 'DIFFUSIVE FRACTION']
```

```
                        ['temperature', 'T2'],
                        ['cloud_cover' , 'CLDFRA'], #'CLOUD_FRACTION'
                       ]

g2g.lsm_data_to_subset_netcdf("E/GSSHA/gssha_wrf_data.nc",
                              data_var_map_array)
```

HRRRtoGSSHA Example:

```
from gsshapy.grid import HRRRtoGSSHA

#STEP 1: Initialize class
h2g = HRRRtoGSSHA(
                   #YOUR INIT PARAMETERS HERE
                  )

#STEP 2: Generate NetCDF DATA

#EXAMPLE DATA ARRAY 2: HRRR GRID DATA BASED
data_var_map_array = [
                        ['precipitation_rate', 'prate'],
                        ['pressure', 'sp'],
                        ['relative_humidity', '2r'],
                        ['wind_speed', ['10u', '10v']],
                        ['direct_radiation_cc', ['dswrf', 'tcc']],
                        ['diffusive_radiation_cc', ['dswrf', 'tcc']],
                        ['temperature', 't'],
                        ['cloud_cover_pc' , 'tcc'],
                       ]

h2g.lsm_data_to_subset_netcdf("E:/GSSHA/gssha_wrf_data.nc",
                              data_var_map_array)
```

lsm_precip_to_gssha_precip_gage(*out_gage_file*, *lsm_data_var*, *precip_type='RADAR'*)

This function takes array data and writes out a GSSHA precip gage file. See: http://www.gsshawiki.com/Precipitation:Spatially_and_Temporally_Varied_Precipitation

---

Note:

GSSHA CARDS:

- PRECIP_FILE card with path to gage file
- RAIN_INV_DISTANCE or RAIN_THIESSEN

Parameters

- **out_gage_file** (*str*) – Location of gage file to generate.
- **lsm_data_var** (*str or list*) – This is the variable name for precipitation in the LSM files. If there is a string, it assumes a single variable. If it is a list, then it assumes the first element is the variable name for RAINC and the second is for RAINNC (see: http://www.meteo.unican.es/wiki/cordexwrf/OutputVariables).
- **precip_type** (*Optional[str]*) – This tells if the data is the ACCUM, RADAR, or GAGES data type. Default is 'RADAR'.

GRIDtoGSSHA Example:

```
from gsshapy.grid import GRIDtoGSSHA

#STEP 1: Initialize class
g2g = GRIDtoGSSHA(gssha_project_folder='/path/to/gssha_project',
                  gssha_project_file_name='gssha_project.prj',
                  lsm_input_folder_path='/path/to/wrf-data',
                  lsm_search_card='*.nc',
                  lsm_lat_var='XLAT',
                  lsm_lon_var='XLONG',
                  lsm_time_var='Times',
                  lsm_lat_dim='south_north',
                  lsm_lon_dim='west_east',
                  lsm_time_dim='Time',
                  )

#STEP 2: Generate GAGE data (from WRF)
g2g.lsm_precip_to_gssha_precip_gage(out_gage_file="E:/GSSHA/wrf_gage_1.gag",
                                    lsm_data_var=['RAINC', 'RAINNC'],
                                    precip_type='ACCUM')
```

HRRRtoGSSHA Example:

```
from gsshapy.grid import HRRRtoGSSHA

#STEP 1: Initialize class
h2g = HRRRtoGSSHA(
                  #YOUR INIT PARAMETERS HERE
                 )

#STEP 2: Generate GAGE data
g2g.lsm_precip_to_gssha_precip_gage(out_gage_file="E:/GSSHA/hrrr_gage_1.gag",
                                    lsm_data_var='prate',
                                    precip_type='RADAR')
```

Download HRRR

gsshapy.grid.hrrr_to_gssha.**download_hrrr_for_gssha**(*main_directory*, *forecast_start_date_string*, *forecast_start_hour_string*, *leftlon=-180*, *rightlon=180*, *toplat=90*, *bottomlat=-90*)

Function to download HRRR data for GSSHA

URL: http://nomads.ncep.noaa.gov/cgi-bin/filter_hrrr_2d.pl

Parameters

- **main_directory** (*str*) – Location of the output for the forecast data.
- **forecast_start_date_string** (*str*) – String for day of forecast. Ex. '20160913'
- **forecast_start_hour_string** (*str*) – String for hour of forecast start. Ex. '02'
- **leftlon** (*Optional[double,int]*) – Left bound for longitude. Default is -180.
- **rightlon** (*Optional[double,int]*) – Right bound for longitude. Default is 180.

- **toplat** (*Optional[double,int]*) – Top bound for latitude. Default is 90.
- **bottomlat** (*Optional[double,int]*) – Bottom bound for latitude. Default is -90.

Returns List of paths to downloaded files.

Return type downloaded_file_list(list)

Example:

```
from gsshapy.grid.hrrr_to_gssha import download_hrrr_for_gssha

hrrr_folder = '/HRRR'
leftlon = -95
rightlon = -75
toplat = 35
bottomlat = 30
downloaded_file_list = download_hrrr_for_gssha(hrrr_folder,'20160914','01',
                                               leftlon,rightlon,toplat,bottomlat)
```

Enjoy!

1.5 Supported Files

Last Updated: July 30, 2014

Not all files are supported by GsshaPy. A summary of the files that are supported and the *file* class handler that reads the file is provided in the following table.

1.5.1 Input Files Supported

The following table lists the input files that are supported by the current version of GsshaPy. The files are listed with the appropriate project file card and *file* class handler. Some files do not have a specified file extension. These are indicated with an ellipses (. . .).

Project File Card	File Extension	Handler
#PROJECTION_FILE	pro	*ProjectionFile*
MAPPING_TABLE	cmt	*MapTableFile*
PRECIP_FILE	gag	*PrecipitationFile*
CHANNEL_INPUT	cif	*ChannelInputFile*
STREAM_CELL	gst	*GridStreamFile*
IN_THETA_LOCATION	ith	*OutputLocationFile*
IN_HYD_LOCATION	ihl	*OutputLocationFile*
IN_SED_LOC	isl	*OutputLocationFile*
IN_GWFLUX_LOCATION	igf	*OutputLocationFile*
HMET_WES	. . .	*HmetFile*
NWSRFS_ELEV_SNOW	. . .	*NwsrfsFile*
HMET_OROG_GAGES	. . .	*OrthographicGageFile*
STORM_SEWER	spn	*StormPipeNetworkFile*
GRID_PIPE	gpi	*GridPipeFile*
OVERLAND_DEPTH_LOCATION	odi	*OutputLocationFile*
OVERLAND_WSE_LOCATION	owi	*OutputLocationFile*

Continued on next page

Table 1.1 – continued from previous page

Project File Card	File Extension	Handler
OUT_WELL_LOCATION	igw	*OutputLocationFile*
REPLACE_PARAMS	...	*ReplaceParamFile*
REPLACE_VALS	...	*ReplaceValFile*
ELEVATION	ele	*RasterMapFile*
WATERSHED_MASK	msk	*RasterMapFile*
ROUGHNESS	ovn	*RasterMapFile*
RETEN_DEPTH	...	*RasterMapFile*
READ_OV_HOTSTART	...	*RasterMapFile*
STORAGE_CAPACITY	...	*RasterMapFile*
INTERCEPTION_COEFF	...	*RasterMapFile*
CONDUCTIVITY	...	*RasterMapFile*
CAPILLARY	...	*RasterMapFile*
POROSITY	...	*RasterMapFile*
MOISTURE	...	*RasterMapFile*
PORE_INDEX	...	*RasterMapFile*
RESIDUAL_SAT	...	*RasterMapFile*
FIELD_CAPACITY	...	*RasterMapFile*
SOIL_TYPE_MAP	...	*RasterMapFile*
WATER_TABLE	wte	*RasterMapFile*
READ_SM_HOTSTART	...	*RasterMapFile*
ALBEDO	alb	*RasterMapFile*
WILTING_POINT	wtp	*RasterMapFile*
TCOEFF	tcf	*RasterMapFile*
VHEIGHT	vht	*RasterMapFile*
CANOPY	cpy	*RasterMapFile*
INIT_SWE_DEPTH	...	*RasterMapFile*
AQUIFER_BOTTOM	aqe	*RasterMapFile*
GW_BOUNDFILE	bnd	*RasterMapFile*
GW_POROSITY_MAP	por	*RasterMapFile*
GW_HYCOND_MAP	hyd	*RasterMapFile*
EMBANKMENT	dik	*RasterMapFile*
DIKE_MASK	dik	*RasterMapFile*
CONTAM_MAP	...	*RasterMapFile*
INDEX_MAP*	idx	*IndexMapFile*

Note: *Index maps are listed in the mapping table file with the INDEX_MAP card.

1.5.2 Output Files Supported

The following table lists the output files that are supported by the current version of GsshaPy. The files are listed with the appropriate project file card and *file* class handler. Some files do not have a specified file extension. These are indicated with an ellipses (...).

Project File Card	File Extension	Handler
OUTLET_HYDRO	otl	*TimeSeriesFile*
OUT_THETA_LOCATION	oth	*TimeSeriesFile*

Continued on next page

Table 1.2 – continued from previous page

Project File Card	File Extension	Handler
OUT_HYD_LOCATION	ohl	*TimeSeriesFile*
OUT_DEP_LOCATION	odl	*TimeSeriesFile*
OUT_SED_LOC	osl	*TimeSeriesFile*
CHAN_DEPTH	cdp	*LinkNodeDatasetFile*
CHAN_STAGE	cds	*LinkNodeDatasetFile*
CHAN_DISCHARGE	vdq	*LinkNodeDatasetFile*
CHAN_VELOCITY	cdv	*LinkNodeDatasetFile*
OUT_GWFULX_LOCATION	ogf	*TimeSeriesFile*
OUTLET_SED_FLUX	osf	*TimeSeriesFile*
OUTLET_SED_TSS	oss	*TimeSeriesFile*
OUT_TSS_LOC	tss	*TimeSeriesFile*
MAX_SED_FLUX	...	*LinkNodeDatasetFile*
OUT_CON_LOCATION	ocl	*TimeSeriesFile*
OUT_MASS_LOCATION	oml	*TimeSeriesFile*
SUPERLINK_JUNC_FLOW	...	*TimeSeriesFile*
SUPERLINK_NODE_FLOW	...	*TimeSeriesFile*
OVERLAND_DEPTHS	odo	*TimeSeriesFile*
OVERLAND_WSE	owo	*TimeSeriesFile*
GW_OUTPUT	...	*WMSDatasetFile*
DISCHARGE	...	*WMSDatasetFile*
INF_DEPTH	...	*WMSDatasetFile*
SURF_MOIS	...	*WMSDatasetFile*
RATE_OF_INFIL	...	*WMSDatasetFile*
DIS_RAIN	...	*WMSDatasetFile*
GW_OUTPUT	...	*WMSDatasetFile*
GW_RECHARGE_CUM	...	*WMSDatasetFile*
GW_RECHARGE_INC	...	*WMSDatasetFile*
DEPTH	dep	*WMSDatasetFile*
SNOW_SWE_FILE	swe	*WMSDatasetFile*

Note: WMS Dataset Files can only be read in if the map type was set to 1 during the model run.

1.5.3 Partial Support

Many files are not fully supported by GsshaPy, meaning that they are not abstracted into several objects to make working with them easy. However, many of these files are partially supported via the `gsshapy.orm.GenericFile` object. This file works by reading the entire contents of the file into a single text field in the database. The files that are supported by this class are listed in the following table. Some files do not have a specified file extension. These are indicated with an ellipses (...).

Project Card	File Extension
ST_MAPPING_TABLE	smt
SECTION_TABLE	…
SOIL_LAYER_INPUT_FILE	…
EXPLIC_HOTSTART	…
READ_CHAN_HOTSTART	…
CHAN_POINT_INPUT	…
HMET_SURFAWAYS	…
HMET_SAMSON	…
HMET_ASCII	…
GW_FLUXBOUNDTABLE	flx
SUPERLINK_JUNC_LOCATION	…
SUPERLINK_NODE_LOCATION	…
SUMMARY	sum
EXPLIC_BACKWATER	…
WRITE_CHAN_HOTSTART	…
LAKE_OUTPUT	lel
GW_WELL_LEVEL	owl
ADJUST_ELEV	ele
NET_SED_VOLUME	…
VOL_SED_SUSP	…
OPTIMIZE	opt

1.6 API Documentation

API documentation is provided in this section.

1.6.1 Input File Objects

Project File

File extension: **prj**

This file object supports spatial objects.

File Object

class gsshapy.orm.**ProjectFile**(*name=None*, *map_type=None*, *project_directory=None*)

Bases: sqlalchemy.ext.declarative.api.Base, gsshapy.base.file_base.GsshaPyFileObjectBase

Object interface for the Project File.

The project file is the main configuration file for GSSHA models. As such, the project file object is different than most of the other file objects. In addition to providing read and write methods for the project file, a project file instance also provides methods for reading and writing the GSSHA project as a whole. These methods should be the primary interface for working with GSSHA models.

The project file is composed of a series of cards and values. Each card in the project file is read into a supporting object: *ProjectCard*.

See: http://www.gsshawiki.com/Project_File:Project_File

tableName = u'prj_project_files'
Database tablename

id
PK

precipFileID
FK

mapTableFileID
FK

channelInputFileID
FK

stormPipeNetworkFileID
FK

hmetFileID
FK

nwsrfsFileID
FK

orographicGageFileID
FK

gridPipeFileID
FK

gridStreamFileID
FK

projectionFileID
FK

replaceParamFileID
FK

replaceValFileID
FK

srid
SRID

projectCards
RELATIONSHIP

mapTableFile
RELATIONSHIP

channelInputFile
RELATIONSHIP

precipFile
RELATIONSHIP

stormPipeNetworkFile
RELATIONSHIP

hmetFile
RELATIONSHIP

nwsrfsFile
RELATIONSHIP

orographicGageFile
RELATIONSHIP

gridPipeFile
RELATIONSHIP

gridStreamFile
RELATIONSHIP

projectionFile
RELATIONSHIP

replaceParamFile
RELATIONSHIP

replaceValFile
RELATIONSHIP

timeSeriesFiles
RELATIONSHIP

outputLocationFiles
RELATIONSHIP

maps
RELATIONSHIP

linkNodeDatasets
RELATIONSHIP

genericFiles
RELATIONSHIP

wmsDatasets
RELATIONSHIP

projectFileEventManager
RELATIONSHIP

fileExtension
STRING

name
STRING

mapType
INTEGER

appendDirectory(*directory*, *projectFilePath*)
Append directory to relative paths in project file. By default, the project file paths are read and written as relative paths. Use this method to prepend a directory to all the paths in the project file.

Parameters

- **directory** (*str*) – Directory path to prepend to file paths in project file.
- **projectFilePath** (*str*) – Path to project file that will be modified.

readProject(*directory*, *projectFileName*, *session*, *spatial=False*, *spatialReferenceID=None*)
Read all files for a GSSHA project into the database.

This method will read all the files, both input and output files, that are supported by GsshaPy into a database. To use GsshaPy more efficiently, it is recommended that you use the readInput method when performing pre-processing tasks and readOutput when performing post-processing tasks.

Parameters

- **directory** (*str*) – Directory containing all GSSHA model files. This method assumes that all files are located in the same directory.
- **projectFileName** (*str*) – Name of the project file for the GSSHA model which will be read (e.g.: 'example.prj').
- **session** (sqlalchemy.orm.session.Session) – SQLAlchemy session object bound to PostGIS enabled database
- **spatial** (*bool, optional*) – If True, spatially enabled objects will be read in as PostGIS spatial objects. Defaults to False.
- **spatialReferenceID** (*int, optional*) – Integer id of spatial reference system for the model. If no id is provided GsshaPy will attempt to automatically lookup the spatial reference ID. If this process fails, default srid will be used (4326 for WGS 84).

readInput (*directory*, *projectFileName*, *session*, *spatial=False*, *spatialReferenceID=None*)

Read only input files for a GSSHA project into the database.

Use this method to read a project when only pre-processing tasks need to be performed.

Parameters

- **directory** (*str*) – Directory containing all GSSHA model files. This method assumes that all files are located in the same directory.
- **projectFileName** (*str*) – Name of the project file for the GSSHA model which will be read (e.g.: 'example.prj').
- **session** (sqlalchemy.orm.session.Session) – SQLAlchemy session object bound to PostGIS enabled database
- **spatial** (*bool, optional*) – If True, spatially enabled objects will be read in as PostGIS spatial objects. Defaults to False.
- **spatialReferenceID** (*int, optional*) – Integer id of spatial reference system for the model. If no id is provided GsshaPy will attempt to automatically lookup the spatial reference ID. If this process fails, default srid will be used (4326 for WGS 84).

readOutput (*directory*, *projectFileName*, *session*, *spatial=False*, *spatialReferenceID=None*)

Read only output files for a GSSHA project to the database.

Use this method to read a project when only post-processing tasks need to be performed.

Parameters

- **directory** (*str*) – Directory containing all GSSHA model files. This method assumes that all files are located in the same directory.
- **projectFileName** (*str*) – Name of the project file for the GSSHA model which will be read (e.g.: 'example.prj').
- **session** (sqlalchemy.orm.session.Session) – SQLAlchemy session object bound to PostGIS enabled database
- **spatial** (*bool, optional*) – If True, spatially enabled objects will be read in as PostGIS spatial objects. Defaults to False.

- **spatialReferenceID** (*int, optional*) – Integer id of spatial reference system for the model. If no id is provided GsshaPy will attempt to automatically lookup the spatial reference ID. If this process fails, default srid will be used (4326 for WGS 84).

readInputFile(*card_name*, *directory*, *session*, *spatial=False*, *spatialReferenceID=None*, ***kwargs*)
Read specific input file for a GSSHA project to the database.

Parameters

- **card_name** (*str*) – Name of GSSHA project card.
- **directory** (*str*) – Directory containing all GSSHA model files. This method assumes that all files are located in the same directory.
- **session** (sqlalchemy.orm.session.Session) – SQLAlchemy session object bound to PostGIS enabled database
- **spatial** (*bool, optional*) – If True, spatially enabled objects will be read in as PostGIS spatial objects. Defaults to False.
- **spatialReferenceID** (*int, optional*) – Integer id of spatial reference system for the model. If no id is provided GsshaPy will attempt to automatically lookup the spatial reference ID. If this process fails, default srid will be used (4326 for WGS 84).

Returns file object

readOutputFile(*card_name*, *directory*, *session*, *spatial=False*, *spatialReferenceID=None*, ***kwargs*)
Read specific input file for a GSSHA project to the database.

Parameters

- **card_name** (*str*) – Name of GSSHA project card.
- **directory** (*str*) – Directory containing all GSSHA model files. This method assumes that all files are located in the same directory.
- **session** (sqlalchemy.orm.session.Session) – SQLAlchemy session object bound to PostGIS enabled database
- **spatial** (*bool, optional*) – If True, spatially enabled objects will be read in as PostGIS spatial objects. Defaults to False.
- **spatialReferenceID** (*int, optional*) – Integer id of spatial reference system for the model. If no id is provided GsshaPy will attempt to automatically lookup the spatial reference ID. If this process fails, default srid will be used (4326 for WGS 84).

Returns file object

writeProject(*session*, *directory*, *name*)
Write all files for a project from the database to file.

Use this method to write all GsshaPy supported files back into their native file formats. If writing to execute the model, increase efficiency by using the writeInput method to write only the file needed to run the model.

Parameters

- **session** (sqlalchemy.orm.session.Session) – SQLAlchemy session object bound to PostGIS enabled database
- **directory** (*str*) – Directory where the files will be written.
- **name** (*str*) – Name that will be given to project when written (e.g.: 'example'). Files that follow the project naming convention will be given this name with the appropriate

extension (e.g.: 'example.prj', 'example.cmt', and 'example.gag'). Files that do not follow this convention will retain their original file names.

writeInput(*session*, *directory*, *name*)
Write only input files for a GSSHA project from the database to file.

Parameters

- **session** (sqlalchemy.orm.session.Session) – SQLAlchemy session object bound to PostGIS enabled database
- **directory** (*str*) – Directory where the files will be written.
- **name** (*str*) – Name that will be given to project when written (e.g.: 'example'). Files that follow the project naming convention will be given this name with the appropriate extension (e.g.: 'example.prj', 'example.cmt', and 'example.gag'). Files that do not follow this convention will retain their original file names.

writeOutput(*session*, *directory*, *name*)
Write only output files for a GSSHA project from the database to file.

Parameters

- **session** (sqlalchemy.orm.session.Session) – SQLAlchemy session object bound to PostGIS enabled database
- **directory** (*str*) – Directory where the files will be written.
- **name** (*str*) – Name that will be given to project when written (e.g.: 'example'). Files that follow the project naming convention will be given this name with the appropriate extension (e.g.: 'example.prj', 'example.cmt', and 'example.gag'). Files that do not follow this convention will retain their original file names.

getFileKeys()
Retrieve a list of file keys that have been read into the database.

This is a utility method that can be used to programmatically access the GsshaPy file objects. Use these keys in conjunction with the dictionary returned by the getFileObjects method.

Returns List of keys representing file objects that have been read into the database.

Return type list

getFileObjects()
Retrieve a dictionary of file objects.

This is a utility method that can be used to programmatically access the GsshaPy file objects. Use this method in conjunction with the getFileKeys method to access only files that have been read into the database.

Returns Dictionary with human readable keys and values of GsshaPy file object instances. Files that have not been read into the database will have a value of None.

Return type dict

getCard(*name*)
Retrieve card object for given card name.

Parameters **name** (*str*) – Name of card to be retrieved.

Returns Project card object. Will return None if the card is not available.

Return type *ProjectCard* or None

setCard(*name*, *value*, *add_quotes=False*)
Adds/updates card for gssha project file

Parameters

- **name** (*str*) – Name of card to be updated/added.
- **value** (*str*) – Value to attach to the card.
- **add_quotes** (*Optional[bool]*) – If True, will add quotes around string. Default is False.

deleteCard(*card_name*, *db_session*)
Removes card from gssha project file

getModelSummaryAsKml(*session*, *path=None*, *documentName=None*, *withStreamNetwork=True*, *withNodes=False*, *styles={}*)
Retrieve a KML representation of the model. Includes polygonized mask map and vector stream network.

Parameters

- **session** (sqlalchemy.orm.session.Session) – SQLAlchemy session object bound to PostGIS enabled database
- **path** (*str, optional*) – Path to file where KML file will be written. Defaults to None.
- **documentName** (*str, optional*) – Name of the KML document. This will be the name that appears in the legend. Defaults to 'Stream Network'.
- **withStreamNetwork** (*bool, optional*) – Include stream network. Defaults to True.
- **withNodes** (*bool, optional*) – Include nodes. Defaults to False.
- **styles** (*dict, optional*) – Custom styles to apply to KML geometry. Defaults to empty dictionary.

  Valid keys (styles) include:

  - streamLineColor: tuple/list of RGBA integers (0-255) e.g.: (255, 0, 0, 128)
  - streamLineWidth: float line width in pixels
  - nodeIconHref: link to icon image (PNG format) to represent nodes (see: http://kml4earth.appspot.com/icons.html)
  - nodeIconScale: scale of the icon image
  - maskLineColor: tuple/list of RGBA integers (0-255) e.g.: (255, 0, 0, 128)
  - maskLineWidth: float line width in pixels
  - maskFillColor: tuple/list of RGBA integers (0-255) e.g.: (255, 0, 0, 128)

Returns KML string

Return type str

getModelSummaryAsWkt(*session*, *withStreamNetwork=True*, *withNodes=False*)
Retrieve a Well Known Text representation of the model. Includes polygonized mask map and vector stream network.

Parameters

- **session** (sqlalchemy.orm.session.Session) – SQLAlchemy session object bound to PostGIS enabled database

- **withStreamNetwork** (*bool, optional*) – Include stream network. Defaults to True.
- **withNodes** (*bool, optional*) – Include nodes. Defaults to False.

Returns Well Known Text string

Return type str

getModelSummaryAsGeoJson(*session*, *withStreamNetwork=True*, *withNodes=False*)
Retrieve a GeoJSON representation of the model. Includes vectorized mask map and stream network.

Parameters

- **session** (sqlalchemy.orm.session.Session) – SQLAlchemy session object bound to PostGIS enabled database
- **withStreamNetwork** (*bool, optional*) – Include stream network. Defaults to True.
- **withNodes** (*bool, optional*) – Include nodes. Defaults to False.

Returns GeoJSON string

Return type str

getGridByCard(*gssha_card_name*)
Returns GDALGrid object of GSSHA grid

Paramters: gssha_card_name(str): Name of GSSHA project card for grid.

Returns GDALGrid

getGrid(*use_mask=True*)
Returns GDALGrid object of GSSHA model bounds

Paramters: use_mask(bool): If True, uses watershed mask. Otherwise, it uses the elevaiton grid.

Returns GDALGrid

getIndexGrid(*name*)
Returns GDALGrid object of index map

Paramters: name(str): Name of index map in 'cmt' file.

Returns GDALGrid

getWkt()
Returns GSSHA projection WKT string

getOutlet()
Gets the outlet latitude and longitude.

Returns Latitude of grid cell center. longitude(float): Longitude of grid cell center.

Return type latitude(float)

setOutlet(*col*, *row*, *outslope=None*)
Sets the outlet grid cell information in the project file.

Parameters

- **col** (*float*) – 1-based column index.

- **row** (*float*) – 1-based row index.
- **outslope** (*Optional[float]*) – River slope at outlet.

findOutlet(*shapefile_path*)
Calculate outlet location

calculateOutletSlope()
Attempt to determine the slope at the OUTLET

timezone
timezone of GSSHA model

centerLatLon()
Get the center lat/lon of model

Supporting Objects

class gsshapy.orm.**ProjectCard**(*name*, *value*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing data for a single card in the project file.

tableName = u'prj_project_cards'
Database tablename

id
PK

projectFileID
FK

projectFile
RELATIONSHIP

name
STRING

value
STRING

write(*originalPrefix*, *newPrefix=None*)
Write project card to string.

Parameters

- **originalPrefix** (*str*) – Original name to give to files that follow the project naming convention (e.g: prefix.gag).
- **newPrefix** (*str, optional*) – If new prefix is desired, pass in this parameter. Defaults to None.

Returns Card and value as they would be written to the project file.

Return type str

Channel Input File

File extension: **cif**

This file object supports spatial objects.

File Object

class gsshapy.orm.**ChannelInputFile**(*alpha=None*, *beta=None*, *theta=None*, *links=None*, *maxNodes=None*)

Bases: sqlalchemy.ext.declarative.api.Base, gsshapy.base.file_base.GsshaPyFileObjectBase

Object interface for the Channel Input File.

The contents of the channel input file is abstracted into several objects including: *StreamLink*, *UpstreamLink*, *StreamNode*, *Weir*, *Culvert*, *Reservoir*, *ReservoirPoint*, *BreakpointCS*, *Breakpoint*, and *TrapezoidalCS*. See the documentation provided for each object for a more details.

See: http://www.gsshawiki.com/Surface_Water_Routing:Channel_Routing

tableName = u'cif_channel_input_files'
Database tablename

id
PK

fileExtension
STRING

projectFile
RELATIONSHIP

streamLinks
RELATIONSHIP

linkNodeDatasets
RELATIONSHIP

alpha
FLOAT

beta
FLOAT

theta
FLOAT

links
INTEGER

maxNodes
INTEGER

getFluvialLinks()
Retrieve only the links that represent fluvial portions of the stream. Returns a list of StreamLink instances.

Returns A list of fluvial *StreamLink* objects.

Return type list

getOrderedLinks(*session*)
Retrieve the links in the order of the link number.

Parameters **session** (sqlalchemy.orm.session.Session) – SQLAlchemy session object bound to PostGIS enabled database.

Returns A list of *StreamLink* objects.

Return type list

getStreamNetworkAsKml(*session*, *path=None*, *documentName=u'Stream Network'*, *withNodes=False*, *styles={}*)

Retrieve the stream network visualization in KML format.

Parameters

- **session** (sqlalchemy.orm.session.Session) – SQLAlchemy session object bound to PostGIS enabled database
- **path** (*str, optional*) – Path to file where KML will be written. Defaults to None.
- **documentName** (*str, optional*) – Name of the KML document. This will be the name that appears in the legend. Defaults to 'Stream Network'.
- **withNodes** (*bool, optional*) – Include nodes. Defaults to False.
- **styles** (*dict, optional*) – Custom styles to apply to KML geometry. Defaults to empty dictionary.

  Valid keys (styles) include:

  - lineColor: tuple/list of RGBA integers (0-255) e.g.: (255, 0, 0, 128)
  - lineWidth: float line width in pixels
  - nodeIconHref: link to icon image (PNG format) to represent nodes (see: http://kml4earth.appspot.com/icons.html)
  - nodeIconScale: scale of the icon image

Returns KML string

Return type str

getStreamNetworkAsWkt(*session*, *withNodes=True*)

Retrieve the stream network geometry in Well Known Text format.

Parameters

- **session** (sqlalchemy.orm.session.Session) – SQLAlchemy session object bound to PostGIS enabled database
- **withNodes** (*bool, optional*) – Include nodes. Defaults to False.

Returns Well Known Text string.

Return type str

getStreamNetworkAsGeoJson(*session*, *withNodes=True*)

Retrieve the stream network geometry in GeoJSON format.

Parameters

- **session** (sqlalchemy.orm.session.Session) – SQLAlchemy session object bound to PostGIS enabled database
- **withNodes** (*bool, optional*) – Include nodes. Defaults to False.

Returns GeoJSON string.

Return type str

Supporting Objects

class gsshapy.orm.**StreamLink**(*linkNumber*, *type*, *numElements*, *dx=None*, *erode=False*, *subsurface=False*)

Bases: sqlalchemy.ext.declarative.api.Base, gsshapy.base.geom.GeometricObjectBase

Object containing generic stream link or reach data.

GSSHA stream networks are composed of a series of stream links and nodes. A stream link is composed of two or more nodes. A basic fluvial stream link contains the cross section. Stream links can also be used to describe structures on a stream such as culverts, weirs, or reservoirs.

This object inherits several methods from the gsshapy.orm.GeometricObjectBase base class for generating geometric visualizations.

See: http://www.gsshawiki.com/Surface_Water_Routing:Channel_Routing#5.1.4.1.4_-_Link_.28Reach.29_information

tableName = u'cif_links'
Database tablename

id
PK

channelInputFileID
FK

downstreamLinkID
INTEGER

numUpstreamLinks
INTEGER

geometry
GEOMETRY

channelInputFile
RELATIONSHIP

upstreamLinks
RELATIONSHIP

nodes
RELATIONSHIP

weirs
RELATIONSHIP

culverts
RELATIONSHIP

reservoir
RELATIONSHIP

breakpointCS
RELATIONSHIP

trapezoidalCS
RELATIONSHIP

datasets
RELATIONSHIP

linkNumber
INTEGER

type
STRING

numElements
INTEGER

dx
FLOAT

erode
BOOLEAN

subsurface
BOOLEAN

class gsshapy.orm.**StreamNode**(*nodeNumber*, *x*, *y*, *elevation*)
Bases: sqlalchemy.ext.declarative.api.Base, gsshapy.base.geom.GeometricObjectBase

Object containing the stream node data in the channel network.

Stream nodes represent the computational unit of GSSHA stream networks. Each stream link must consist of two or more stream nodes.

This object inherits several methods from the gsshapy.orm.GeometricObjectBase base class for generating geometric visualizations.

See: http://www.gsshawiki.com/Surface_Water_Routing:Channel_Routing#5.1.4.1.4.2.1.4_Node_information

tableName = u'cif_nodes'
Database tablename

id
PK

linkID
FK

geometry
GEOMETRY

streamLink
RELATIONSHIP

datasets
RELATIONSHIP

nodeNumber
INTEGER

x
FLOAT

y
FLOAT

elevation
FLOAT

class gsshapy.orm.**UpstreamLink**(*upstreamLinkID*)
Bases: sqlalchemy.ext.declarative.api.Base

Object used to map stream links with their upstream link counterparts.

See: http://www.gsshawiki.com/Surface_Water_Routing:Channel_Routing#5.1.4.1.3_.E2.80.93_Channel_network_connectivity

tableName = u'cif_upstream_links'
Database tablename

id
PK

linkID
INTEGER

streamLink
RELATIONSHIP

upstreamLinkID
INTEGER

class gsshapy.orm.**Weir**(*type*, *crestLength*, *crestLowElevation*, *dischargeCoeffForward*, *dischargeCoeffReverse*, *crestLowLocation*, *steepSlope*, *shallowSlope*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing data that defines a weir structure for a stream link.

See: http://www.gsshawiki.com/Surface_Water_Routing:Channel_Routing#5.1.4.1.4.2_-_Structure_channel_links

tableName = u'cif_weirs'
Database tablename

id
PK

linkID
FK

streamLink
RELATIONSHIP

type
STRING

crestLength
FLOAT

crestLowElevation
FLOAT

dischargeCoeffForward
FLOAT

dischargeCoeffReverse
FLOAT

crestLowLocation
FLOAT

steepSlope
FLOAT

shallowSlope
FLOAT

class gsshapy.orm.**Culvert**(*type*, *upstreamInvert*, *downstreamInvert*, *inletDischargeCoeff*, *reverseFlowDischargeCoeff*, *slope*, *length*, *roughness*, *diameter*, *width*, *height*)

Bases: sqlalchemy.ext.declarative.api.Base

Object containing a culvert structure data for a stream link.

See: http://www.gsshawiki.com/Surface_Water_Routing:Channel_Routing#5.1.4.1.4.2_-_Structure_channel_links

tableName = u'cif_culverts'
Database tablename

id
PK

linkID
FK

streamLink
RELATIONSHIP

type
STRING

upstreamInvert
FLOAT

downstreamInvert
FLOAT

inletDischargeCoeff
FLOAT

reverseFlowDischargeCoeff
FLOAT

slope
FLOAT

length
FLOAT

roughness
FLOAT

diameter
FLOAT

width
FLOAT

height
FLOAT

class gsshapy.orm.**Reservoir**(*initWSE*, *minWSE*, *maxWSE*)

Bases: sqlalchemy.ext.declarative.api.Base

Object containing a data that defines a reservoir for a stream link.

See: http://www.gsshawiki.com/Surface_Water_Routing:Channel_Routing#5.1.4.1.4.3_-_Reservoir_channel_links

tableName = u'cif_reservoirs'
Database tablename

id
PK

linkID
FK

streamLink
RELATIONSHIP

reservoirPoints
RELATIONSHIP

initWSE
FLOAT

minWSE
FLOAT

maxWSE
FLOAT

class gsshapy.orm.**ReservoirPoint**(*i*, *j*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing the cells/points that define the maximum inundation area of a reservoir.

See: http://www.gsshawiki.com/Surface_Water_Routing:Channel_Routing#

tableName = u'cif_reservoir_points'
Database tablename

id
PK

reservoirID
FK

reservoir
RELATIONSHIP

i
INTEGER

j
INTEGER

class gsshapy.orm.**BreakpointCS**(*mannings_n*, *numPairs*, *numInterp*, *mRiver*, *kRiver*, *erode*, *subsurface*, *maxErosion*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing breakpoint type cross section data for fluvial stream links.

See: http://www.gsshawiki.com/Surface_Water_Routing:Channel_Routing#5.1.4.1.4.2.1.2_Natural_cross-section

tableName = u'cif_breakpoint'
Database tablename

id
PK

linkID
FK

streamLink
RELATIONSHIP

breakpoints
RELATIONSHIP

mannings_n
FLOAT

numPairs
INTEGER

numInterp
INTEGER

mRiver
FLOAT

kRiver
FLOAT

erode
BOOLEAN

subsurface
BOOLEAN

maxErosion
FLOAT

class gsshapy.orm.**Breakpoint**(*x*, *y*)
Bases: sqlalchemy.ext.declarative.api.Base

Object used to define points in a *BreakpointCS* object.

See: http://www.gsshawiki.com/Surface_Water_Routing:Channel_Routing#5.1.4.1.4.2.1.2_Natural_cross-section

tableName = u'cif_bcs_points'
Database tablename

id
PK

crossSectionID
FK

crossSection
RELATIONSHIP

x
FLOAT

y
FLOAT

class gsshapy.orm.**TrapezoidalCS**(*mannings_n*, *bottomWidth*, *bankfullDepth*, *sideSlope*, *mRiver*, *kRiver*, *erode*, *subsurface*, *maxErosion*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing trapezoidal type cross section data for fluvial stream links.

See: http://www.gsshawiki.com/Surface_Water_Routing:Channel_Routing#5.1.4.1.4.2.1.1_Trapezoidal_cross-section

tableName = u'cif_trapezoid'
Database tablename

id
PK

linkID
FK

streamLink
RELATIONSHIP

mannings_n
FLOAT

bottomWidth
FLOAT

bankfullDepth
FLOAT

sideSlope
FLOAT

mRiver
FLOAT

kRiver
FLOAT

erode
BOOLEAN

subsurface
BOOLEAN

maxErosion
BOOLEAN

Grid Pipe File

File extension: **gpi**

File Object

class gsshapy.orm.**GridPipeFile**
Bases: sqlalchemy.ext.declarative.api.Base, gsshapy.base.file_base.GsshaPyFileObjectBase

Object interface for the Grid Pipe File.

The grid pipe file is used to map the grid pipe network for subsurface drainage to the model grid. The contents of the grid pipe file is abstracted into two types of objects including: *GridPipeCell* and *GridPipeNode*. Each cell lists the pipe nodes that are contained in the cell and each pipe node defines the percentage of a pipe that is contained inside a cell. See the documentation provided for each object for a more details.

See: http://www.gsshawiki.com/Subsurface_Drainage:Subsurface_Drainage http://www.gsshawiki.com/images/d/d6/SUPERLINK_TN.pdf

tableName = u'gpi_grid_pipe_files'
Database tablename

id
PK

gridPipeCells
RELATIONSHIP

projectFile
RELATIONSHIP

pipeCells
INTEGER

fileExtension
STRING

Supporting Objects

class gsshapy.orm.**GridPipeCell**(*cellI*, *cellJ*, *numPipes*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing the pipe data for a single grid cell. A cell can contain several pipe nodes.

tableName = u'gpi_grid_pipe_cells'
Database tablename

id
PK

gridPipeFileID
FK

gridPipeFile
RELATIONSHIP

gridPipeNodes
RELATIONSHIP

cellI
INTEGER

cellJ
INTEGER

numPipes
INTEGER

class gsshapy.orm.**GridPipeNode**(*linkNumber*, *nodeNumber*, *fractPipeLength*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing data for a single pipe.

tableName = u'gpi_grid_pipe_nodes'
Database tablename

id
PK

gridPipeCellID
FK

gridPipeCell
RELATIONSHIP

linkNumber
INTEGER

nodeNumber
INTEGER

fractPipeLength
FLOAT

Grid Stream File

File extension: **gst**

File Object

class gsshapy.orm.**GridStreamFile**
Bases: sqlalchemy.ext.declarative.api.Base, gsshapy.base.file_base.GsshaPyFileObjectBase

Object interface for the Grid Stream File.

The grid stream file is used to map the stream network to the model grid. The contents of the grid stream file is abstracted into two types of objects including: *GridStreamCell* and *GridStreamNode*. Each cell lists the stream nodes that are contained in it and each stream node defines the percentage of that stream that is contained inside a cell. See the documentation provided for each object for a more details.

See: http://www.gsshawiki.com/Surface_Water_Routing:Channel_Routing#5.1.4.2.1_-_STREAM_CELL_file_identifier

tableName = u'gst_grid_stream_files'
Database tablename

id
PK

streamCells
INTEGER

fileExtension
STRING

gridStreamCells
RELATIONSHIP

projectFile
RELATIONSHIP

Supporting Objects

class gsshapy.orm.**GridStreamCell**(*cellI*, *cellJ*, *numNodes*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing the stream data for a single grid cell. A cell can contain several stream nodes.

tableName = u'gst_grid_stream_cells'
Database tablename

id
PK

gridStreamFileID
FK

gridStreamFile
RELATIONSHIP

gridStreamNodes
RELATIONSHIP

cellI
INTEGER

cellJ
INTEGER

numNodes
INTEGER

class gsshapy.orm.**GridStreamNode**(*linkNumber*, *nodeNumber*, *nodePercentGrid*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing data for a single stream.

tableName = u'gst_grid_stream_nodes'
Database tablename

id
PK

gridStreamCellID
FK

gridStreamCell
RELATIONSHIP

linkNumber
INTEGER

nodeNumber
INTEGER

nodePercentGrid
FLOAT

Hydrometeorological Files

File extensions: **None**

File Object

class gsshapy.orm.**HmetFile**
Bases: sqlalchemy.ext.declarative.api.Base, gsshapy.base.file_base.GsshaPyFileObjectBase

Object interface for the Hydrometeorological Input Files (HMET Files).

An HMET file contains time series hydrometeorological parameters that are required to perform long term simulations. GSSHAPY currently only supports the HMET WES file format.

See: http://www.gsshawiki.com/Continuous:Hydrometeorological_Data

tableName = u'hmet_files'
Database tablename

id
PK

fileExtension
STRING

hmetRecords
RELATIONSHIP

projectFile
RELATIONSHIP

Supporting Objects

class gsshapy.orm.**HmetRecord**(*hmetDateTime*, *barometricPress*, *relHumidity*, *totalSkyCover*, *windSpeed*, *dryBulbTemp*, *directRad*, *globalRad*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing data for a single record in an HMET file.

tableName = u'hmet_records'
Database tablename

id
PK

hmetConfigID
INTEGER

hmetFile
RELATIONSHIP

hmetDateTime
DATETIME

barometricPress
FLOAT

relHumidity
INTEGER

totalSkyCover
INTEGER

windSpeed
INTEGER

dryBulbTemp
INTEGER

directRad
FLOAT

globalRad
FLOAT

Map Files

Although index maps and other raster maps are both GRASS ASCII maps, a special table was created for index maps for easier implementation.

Index Map File Object

File extension: **idx**

This file object supports spatial objects.

class gsshapy.orm.**IndexMap**(*name=None*)

Bases: sqlalchemy.ext.declarative.api.Base, gsshapy.base.file_base.GsshaPyFileObjectBase, gsshapy.base.rast.RasterObjectBase

Object interface for Index Map Files.

GSSHA uses GRASS ASCII rasters to store spatially distributed parameters. Index maps are stored using a different object than other raster maps, because they are closely tied to the mapping table file objects and they are stored with different metadata than the other raster maps. Index maps are declared in the mapping table file.

The values for each cell in an index map are integer indices that correspond with the indexes of the mapping tables that reference the index map. Many different hydrological parameters are distributed spatially in this manner. The result is that far fewer maps are needed to parametrize a GSSHA model.

If the spatial option is enabled when the rasters are read in, the rasters will be read in as PostGIS raster objects. There are no supporting objects for index map file objects.

This object inherits several methods from the gsshapy.orm.RasterObjectBase base class for generating raster visualizations.

See: http://www.gsshawiki.com/Mapping_Table:Index_Maps

tableName = u'idx_index_maps'
Database tablename

rasterColumnName = u'raster'
Raster column name

defaultNoDataValue = -1
Default no data value

discreet = True
Index maps should be discreet

id
PK

mapTableFileID
FK

north
FLOAT

south
FLOAT

east
FLOAT

west
FLOAT

rows
INTEGER

columns
INTEGER

srid
SRID

filename
STRING

raster
RASTER

fileExtension
STRING

mapTableFile
RELATIONSHIP

mapTables
RELATIONSHIP

indices
RELATIONSHIP

contaminants
RELATIONSHIP

name
STRING

rasterText
STRING

write(*directory*, *name=None*, *session=None*, *replaceParamFile=None*)
Index Map Write to File Method

Raster Map File Object

File extensions: **Variable** (e.g.: ele, msk, aqe)

This file object supports spatial objects.

class gsshapy.orm.**RasterMapFile**
Bases: sqlalchemy.ext.declarative.api.Base, gsshapy.base.file_base.GsshaPyFileObjectBase, gsshapy.base.rast.RasterObjectBase

Object interface for Raster Map type files.

GSSHA uses GRASS ASCII rasters to store spatially distributed parameters. Rasters that are not index maps are stored using this object. Index maps are stored separately, because they are closely tied to the mapping table file objects and they are stored with different metadata than the other raster maps.

Raster maps are declared in the project file. Examples of cards that require raster maps are ELEVATION, ROUGHNESS WATERSHED_MASK, WATER_TABLE, and MOISTURE. Many of these map inputs are mutually exclusive with the mapping tables for the same variable.

If the spatial option is enabled when the rasters are read in, the rasters will be read in as PostGIS raster objects. There are no supporting objects for raster map file objects.

This object inherits several methods from the `gsshapy.orm.RasterObjectBase` base class for generating raster visualizations.

See: http://www.gsshawiki.com/Project_File:Project_File

tableName = u'raster_maps'
Database tablename

rasterColumnName = u'raster'
Raster column name

defaultNoDataValue = 0
Default no data value

id
PK

projectFileID
FK

north
FLOAT

south
FLOAT

east
FLOAT

west
FLOAT

rows
INTEGER

columns
INTEGER

fileExtension
STRING

rasterText
STRING

raster
RASTER

filename
STRING

projectFile
RELATIONSHIP

write(*session*, *directory*, *name*, *replaceParamFile=None*, ***kwargs*)
Wrapper for GsshaPyFileObjectBase write method

Mapping Table File

File extension: **cmt**

This file object supports spatial objects (when the mapping table is read into the database, the index maps are read in as well).

File Object

class gsshapy.orm.**MapTableFile**(*project_file=None*)

Bases: sqlalchemy.ext.declarative.api.Base, gsshapy.base.file_base.GsshaPyFileObjectBase

Object interface for the Mapping Table File.

Hydrological parameters are distributed spatially in GSSHA through mapping tables and index maps. Index maps are raster maps of integers. The mapping tables define the hydrological values for each unique index on a map. Most of the mapping tables are abstracted into three objects representing three different parts of the table. *MapTable* contains the data for the mapping table header, *MTIndex* contains the data for the indexes defined by the mapping table, and *MTValue* contains the actual value of the hydrological parameters defined by the mapping table.

In addition, there are two special mapping tables that break the common format: Contaminant/Constituent Transport and Sediment Transport. The data for these mapping tables is contained in the *MTContaminant* and Sediment objects, respectively.

The GSSHA documentation used to design this object can be found by following these links: http://www.gsshawiki.com/Mapping_Table:Mapping_Table_File

tableName = u'cmt_map_table_files'
Database tablename

id
PK

indexMaps
RELATIONSHIP

mapTables
RELATIONSHIP

projectFile
RELATIONSHIP

fileExtension
STRING

getOrderedMapTables(*session*)
Retrieve the map tables ordered by name

deleteMapTable(*name*, *session*)
Remove duplicate map table if it exists

addRoughnessMapFromLandUse(*name*, *session*, *land_use_grid*, *land_use_to_roughness_table=None*, *land_use_grid_id=None*)
Adds a roughness map from land use file

Example:

```
from gsshapy.orm import ProjectFile
from gsshapy.lib import db_tools as dbt

from os import path, chdir

gssha_directory = '/gsshapy/tests/grid_standard/gssha_project'
land_use_grid = 'LC_5min_global_2012.tif'
land_use_to_roughness_table = ''/gsshapy/gridtogssha/land_cover/land_cover_
↪glcf_modis.txt'
```

```
# Create Test DB
sqlalchemy_url, sql_engine = dbt.init_sqlite_memory()

# Create DB Sessions
db_session = dbt.create_session(sqlalchemy_url, sql_engine)

# Instantiate GSSHAPY object for reading to database
project_manager = ProjectFile()

# Call read method
project_manager.readInput(directory=gssha_directory,
                          projectFileName='grid_standard.prj',
                          session=db_session)

project_manager.mapTableFile.addRoughnessMapFromLandUse("roughness",
                                                        db_session,
                                                        land_use_to_
↪roughness_table,
                                                        land_use_grid,
                                                        )
# WRITE OUT UPDATED GSSHA PROJECT FILE
project_manager.writeInput(session=db_session,
                           directory=gssha_directory,
                           name='grid_standard')
```

Supporting Objects

class gsshapy.orm.**MapTable**(*name*, *numIDs=None*, *maxNumCells=None*, *numSed=None*, *numContam=None*, *maxSoilID=None*)

Bases: sqlalchemy.ext.declarative.api.Base

Object containing header data for a mapping table.

See: **http://www.gsshawiki.com/Mapping_Table:Mapping_Tables** http://www.gsshawiki.com/Mapping_Table:Index_Maps

tableName = u'cmt_map_tables'
Database tablename

id
PK

idxMapID
FK

mapTableFileID
FK

mapTableFile
RELATIONSHIP

indexMap
RELATIONSHIP

values
RELATIONSHIP

sediments
RELATIONSHIP

name
STRING

numIDs
INTEGER

maxNumCells
INTEGER

numSed
INTEGER

numContam
INTEGER

class gsshapy.orm.**MTIndex**(*index*, *description1=u”*, *description2=u”*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing mapping table index data. Mapping table index objects link the mapping table values to index maps.

See: http://www.gsshawiki.com/Mapping_Table:Mapping_Tables

tableName = u'cmt_indexes'
Database tablename

id
PK

idxMapID
FK

values
RELATIONSHIP

indexMap
RELATIONSHIP

index
INTEGER

description1
STRING

description2
STRING

class gsshapy.orm.**MTValue**(*variable*, *value=None*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing the hydrological variable and value data for mapping tables.

See: http://www.gsshawiki.com/Mapping_Table:Mapping_Tables

tableName = u'cmt_map_table_values'
Database tablename

id
PK

mapTableID
FK

mapTableIndexID
FK

contaminantID
FK

mapTable
RELATIONSHIP

index
RELATIONSHIP

contaminant
RELATIONSHIP

variable
STRING

value
FLOAT

class gsshapy.orm.**MTContaminant**(*name*, *outputFilename*, *precipConc*, *partition*, *numIDs*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing data in contaminant transport type mapping tables.

See: http://www.gsshawiki.com/Mapping_Table:Constituent_Mapping_Tables

tableName = u'cmt_contaminants'
Database tablename

id
PK

idxMapID
FK

indexMap
RELATIONSHIP

values
RELATIONSHIP

name
STRING

outputFilename
STRING

precipConc
FLOAT

partition
FLOAT

numIDs
INTEGER

class gsshapy.orm.**MTSediment**(*description*, *specificGravity*, *particleDiameter*, *outputFilename*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing data in sediment transport type mapping tables.

See: http://www.gsshawiki.com/Mapping_Table:Sediment_Erosion_Mapping_Tables

tableName = u'cmt_sediments'
Database tablename

id
PK

mapTableID
FK

mapTable
RELATIONSHIP

description
STRING

specificGravity
FLOAT

particleDiameter
FLOAT

outputFilename
STRING

Output Location Files

File extensions: **Variable** (e.g.: ihl, isl, igf)

File Object

class gsshapy.orm.**OutputLocationFile**
Bases: sqlalchemy.ext.declarative.api.Base, gsshapy.base.file_base.GsshaPyFileObjectBase

Object interface for the output location type files.

There are several files that are used to specify output at internal locations in the model. These files are specified by the following cards in the project file: IN_HYD_LOCATION, IN_THETA_LOCATION, IN_GWFLUX_LOCATION, IN_SED_LOC, OVERLAND_DEPTH_LOCATION, OVERLAND_WSE_LOCATION, and OUT_WELL_LOCATION.

Output location files contain either a list of cell addresses (i and j) for output from the grid **or** a list of link node addresses (link number and node number) for output requested from the stream network. The output is generated as timeseries at each location. The contents of this file is abstracted to one other object: *OutputLocation*.

See: http://www.gsshawiki.com/Project_File:Output_Files_%E2%80%93_Required

tableName = u'loc_output_location_files'
Database tablename

id
PK

projectFileID
FK

fileExtension
STRING

numLocations
INTEGER

projectFile
RELATIONSHIP

outputLocations
RELATIONSHIP

Supporting Objects

class gsshapy.orm.**OutputLocation**(*linkOrCellI*, *nodeOrCellJ*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing the data for a single output location coordinate pair. Depending on whether the file requests output on the grid or on the stream network, the coordinate pair will represent either cell i j or link node coordinates, respectively.

tableName = u'loc_output_locations'
Database tablename

id
PK

outputLocationFileID
FK

outputLocationFile
RELATIONSHIP

linkOrCellI
INTEGER

nodeOrCellJ
INTEGER

Precipitation File

File extension: **gag**

File Object

class gsshapy.orm.**PrecipFile**
Bases: sqlalchemy.ext.declarative.api.Base, gsshapy.base.file_base.GsshaPyFileObjectBase

Object interface for the Precipitation Input File.

The contents of the precipitation file are abstracted into three types of objects including: *PrecipEvent*, *PrecipValue*, and *PrecipGage*. One precipitation file can consist of multiple events and each event can have several gages and a time series of values for each gage.

See: http://www.gsshawiki.com/Precipitation:Spatially_and_Temporally_Varied_Precipitation

tableName = u'gag_precipitation_files'
Database tablename

id
PK

fileExtension
STRING

precipEvents
RELATIONSHIP

projectFile
RELATIONSHIP

Supporting Objects

class gsshapy.orm.**PrecipEvent**(*description*, *nrGag*, *nrPds*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing data for a single precipitation event.

tableName = u'gag_events'
Database tablename

id
PK

precipFileID
FK

values
RELATIONSHIP

gages
RELATIONSHIP

precipFile
RELATIONSHIP

description
STRING

nrGag
INTEGER

nrPds
INTEGER

class gsshapy.orm.**PrecipValue**(*valueType*, *dateTime*, *value*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing data for a single time stamped precipitation value.

tableName = u'gag_values'
Database tablename

id
PK

eventID
FK

coordID
FK

event
RELATIONSHIP

gage
RELATIONSHIP

valueType
STRING

dateTime
DATETIME

value
FLOAT

class gsshapy.orm.**PrecipGage**(*description*, *x*, *y*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing data for a single precipitation gage or location.

tableName = u'gag_coord'
Database tablename

id
PK

values
RELATIONSHIP

event
RELATIONSHIP

description
STRING

x
FLOAT

y
FLOAT

Projection File

File extension: **pro**

File Object

class gsshapy.orm.**ProjectionFile**
Bases: sqlalchemy.ext.declarative.api.Base, gsshapy.base.file_base.GsshaPyFileObjectBase

Object interface for the Projection File.

The projection file contains the Well Known Text version of the spatial reference system for the GSSHA model. This file contains a single line, so the file contents is stored in the file object. No supporting objects are needed.

See: **http://www.geoapi.org/3.0/javadoc/org/opengis/referencing/doc-files/WKT.html** http://spatialreference.org/

tableName = 'pro_projection_files'
Database tablename

id
PK

projection
STRING

fileExtension
STRING

projectFile
RELATIONSHIP

classmethod lookupSpatialReferenceID(*directory*, *filename*)
Look up spatial reference system using the projection file.

Parameters

- **directory** (*str*) –
- **filename** (*str*) –

Returns Spatial Reference ID

Return type int

Replacement File

Two files are used to accomplish the automated parameter replacement functionality in GSSHA models.

Replace Parameter File

File extension: **None**

class gsshapy.orm.**ReplaceParamFile**
Bases: sqlalchemy.ext.declarative.api.Base, gsshapy.base.file_base.GsshaPyFileObjectBase

Object interface for the Replacement Parameters File.

The contents of this file are abstracted to one other supporting object: *TargetParameter*. Use this object in conjunction with the *ReplaceValFile*.

See: **http://www.gsshawiki.com/Alternate_Run_Modes:Simulation_Setup_for_Alternate_Run_Modes**
http://www.gsshawiki.com/File_Formats:Project_File_Format#Replacement_cards

tableName = u'rep_replace_param_files'
Database tablename

id
PK

numParameters
INTEGER

fileExtension
STRING

projectFile
RELATIONSHIP

targetParameters
RELATIONSHIP

Supporting Objects

class gsshapy.orm.**TargetParameter**(*targetVariable*, *varFormat*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing data for a single target value as defined in the Replacement Parameters File.

tableName = u'rep_target_parameter'
Database tablename

id
PK

replaceParamFileID
FK

replaceParamFile
RELATIONSHIP

targetVariable
STRING

varFormat
STRING

Replace Value File

File extension: **None**

class gsshapy.orm.**ReplaceValFile**
Bases: sqlalchemy.ext.declarative.api.Base, gsshapy.base.file_base.GsshaPyFileObjectBase

Object interface for the Replacement Values File.

Use this object in conjunction with the *ReplaceParamFile*.

See: **http://www.gsshawiki.com/Alternate_Run_Modes:Simulation_Setup_for_Alternate_Run_Modes**
http://www.gsshawiki.com/File_Formats:Project_File_Format#Replacement_cards

tableName = u'rep_replace_val_files'
Database tablename

id
PK

values
STRING

fileExtension
STRING

projectFile
RELATIONSHIP

lines
RELATIONSHIP

Snow Simulation Related Files

There are two files that are used for snow simulations.

NWSRFS File

File extension: **None**

class gsshapy.orm.**NwsrfsFile**
Bases: sqlalchemy.ext.declarative.api.Base, gsshapy.base.file_base.GsshaPyFileObjectBase

Object interface for the NWSRFS Snow File.

The contents of this file is abstracted into one supporting object: *NwsrfsRecord*.

See: GSSHAWIKI

tableName = u'snw_nwsrfs_files'
Database tablename

id
PK

numBands
INTEGER

fileExtension
STRING

nwsrfsRecords
RELATIONSHIP

projectFile
RELATIONSHIP

Supporting Objects

class gsshapy.orm.**NwsrfsRecord**(*lowerElev*, *upperElev*, *mfMin*, *mfMax*, *scf*, *frUse*, *tipm*, *nmf*, *fua*, *plwhc*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing data for a single NWSRFS record from the NWSRFS snow file.

tableName = u'snw_nwsrfs_records'
Database tablename

id
PK

nwsrfsFileID
FK

nwsrfsFile
RELATIONSHIP

lowerElev
INTEGER

upperElev
INTEGER

mfMin
FLOAT

mfMax
FLOAT

scf
FLOAT

frUse
FLOAT

tipm
FLOAT

nmf
FLOAT

fua
FLOAT

plwhc
FLOAT

Orographic Gage File

File extension: **None**

class gsshapy.orm.**OrographicGageFile**
Bases: sqlalchemy.ext.declarative.api.Base, gsshapy.base.file_base.GsshaPyFileObjectBase

Object interface for the Orographic Gage File.

The contents of this file is abstracted into one supporting object: *OrographicMeasurement*.

See: GSSHAWIKI

tableName = u'snw_orographic_gage_files'
Database tablename

id
PK

numSites
INTEGER

elevBase
FLOAT

elev2
FLOAT

fileExtension
STRING

orographicMeasurements
RELATIONSHIP

projectFile
RELATIONSHIP

Supporting Objects

class gsshapy.orm.**OrographicMeasurement**(*dateTime*, *temp2*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing data for a single orographic gage as defined in the orographic gage file.

tableName = u'snw_orographic_measurements'
Database tablename

id
PK

orthoGageID
FK

orographicGageFile
RELATIONSHIP

dateTime
DATETIME

temp2
FLOAT

Storm Pipe Network File

File extension: **spn**

File Object

class gsshapy.orm.**StormPipeNetworkFile**
Bases: sqlalchemy.ext.declarative.api.Base, gsshapy.base.file_base.GsshaPyFileObjectBase

Object interface for the Storm Pipe Network File.

This file is similar in structure as the channel input file. The contents of this file is abstracted to several supporting objects: *SuperLink*, *SuperNode*, *Pipe*, *SuperJunction*, and *Connection*.

See: http://www.gsshawiki.com/Subsurface_Drainage:Subsurface_Drainage http://www.gsshawiki.com/images/d/d6/SUPERLINK_TN.pdf

tableName = u'spn_storm_pipe_network_files'
Database tablename

id
PK

fileExtension
STRING

connections
RELATIONSHIP

superLinks
RELATIONSHIP

superJunctions
RELATIONSHIP

projectFile
RELATIONSHIP

Supporting Objects

class gsshapy.orm.**SuperLink**(*slinkNumber*, *numPipes*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing data for a single super link in the subsurface drainage network. A super link consists of several pipes and super nodes.

tableName = u'spn_super_links'
Database tablename

id
PK

stormPipeNetworkFileID
FK

stormPipeNetworkFile
RELATIONSHIP

superNodes
RELATIONSHIP

pipes
RELATIONSHIP

slinkNumber
INTEGER

numPipes
INTEGER

class gsshapy.orm.**SuperNode**(*nodeNumber*, *groundSurfaceElev*, *invertElev*, *manholeSA*, *nodeInletCode*, *cellI*, *cellJ*, *weirSideLength*, *orificeDiameter*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing data for a single super node in the subsurface drainage network. Super nodes belong to one super link.

tableName = u'spn_super_nodes'
Database tablename

id
PK

superLinkID
FK

superLink
RELATIONSHIP

nodeNumber
INTEGER

groundSurfaceElev
FLOAT

invertElev
FLOAT

manholeSA
FLOAT

nodeInletCode
INTEGER

cellI
INTEGER

cellJ
INTEGER

weirSideLength
FLOAT

orificeDiameter
FLOAT

class gsshapy.orm.**Pipe**(*pipeNumber*, *xSecType*, *diameterOrHeight*, *width*, *slope*, *roughness*, *length*, *conductance*, *drainSpacing*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing data for a single pipe in the subsurface drainage network. Pipes belong to one super link.

tableName = u'spn_pipes'
Database tablename

id
PK

superLinkID
FK

superLink
RELATIONSHIP

pipeNumber
INTEGER

xSecType
INTEGER

diameterOrHeight
FLOAT

width
FLOAT

slope
FLOAT

roughness
FLOAT

length
FLOAT

conductance
FLOAT

drainSpacing
FLOAT

class gsshapy.orm.**SuperJunction**(*sjuncNumber*, *groundSurfaceElev*, *invertElev*, *manholeSA*, *inletCode*, *linkOrCellI*, *nodeOrCellJ*, *weirSideLength*, *orificeDiameter*)

Bases: sqlalchemy.ext.declarative.api.Base

Object containing data for a single super junction in the subsurface drainage network. Super junctions are where two or more super links join or the unconnected end of a super link.

tableName = u'spn_super_junctions'
Database tablename

id
PK

stormPipeNetworkFileID
FK

stormPipeNetworkFile
RELATIONSHIP

sjuncNumber
INTEGER

groundSurfaceElev
FLOAT

invertElev
FLOAT

manholeSA
FLOAT

inletCode
INTEGER

linkOrCellI
INTEGER

nodeOrCellJ
INTEGER

weirSideLength
FLOAT

orificeDiameter
FLOAT

class gsshapy.orm.**Connection**(*slinkNumber*, *upSjuncNumber*, *downSjuncNumber*)

Bases: sqlalchemy.ext.declarative.api.Base

Object containing data for a single connection in the subsurface drainage network. Connections between super links and super junctions are mapped via these records.

tableName = u'spn_connections'
Database tablename

id
PK

stormPipeNetworkFileID
FK

stormPipeNetworkFile
RELATIONSHIP

slinkNumber
INTEGER

upSjuncNumber
INTEGER

downSjuncNumber
INTEGER

1.6.2 Output File Objects

Generic Files

File extension: **None**

File Object

class gsshapy.orm.**GenericFile**
Bases: sqlalchemy.ext.declarative.api.Base, gsshapy.base.file_base.GsshaPyFileObjectBase

Object interface for Generic Files.

This object is used to store files that are not fully supported in GsshaPy. The files must be non-binary text files to be stored as a GenericFile object. The object simply reads the contents of the file into a text field during reading and dumps it again during writing. This allows these files to be carried through the entire GsshaPy cycle.

tableName = 'gen_generic_files'
Database tablename

id
PK

projectFileID
FK

text
STRING

binary
BINARY

name
STRING

fileExtension
STRING

projectFile
RELATIONSHIP

Link Node Dataset Files

File extensions: **Variable** (e.g.: cdp, cdq, cds)

This file object supports spatial objects.

File Object

class gsshapy.orm.**LinkNodeDatasetFile**

Bases: sqlalchemy.ext.declarative.api.Base, gsshapy.base.file_base.GsshaPyFileObjectBase

Object interface for Link Node Dataset files.

As the name implies, link node datasets store output data for link node networks. In the case of GSSHA, this type of file is used to write output for the stream network nodes. The contents of this file is abstracted to several supporting objects including: *LinkNodeTimeStep*, *LinkDataset*, and *NodeDataset*.

Note: The link node dataset must be linked with the channel input file to generate spatial visualizations.

tableName = u'lnd_link_node_dataset_files'
Database tablename

id
PK

projectFileID
FK

channelInputFileID
FK

fileExtension
STRING

name
STRING

numLinks
INTEGER

timeStepInterval
INTEGER

numTimeSteps
INTEGER

startTime
STRING

projectFile
RELATIONSHIP

timeSteps
RELATIONSHIP

channelInputFile
RELATIONSHIP

linkDatasets
RELATIONSHIP

nodeDatasets
RELATIONSHIP

linkToChannelInputFile(*session*, *channelInputFile*, *force=False*)
Create database relationships between the link node dataset and the channel input file.

The link node dataset only stores references to the links and nodes–not the geometry. The link and node geometries are stored in the channel input file. The two files must be linked with database relationships to allow the creation of link node dataset visualizations.

This process is not performed automatically during reading, because it can be very costly in terms of read time. This operation can only be performed after both files have been read into the database.

Parameters

- **session** (`sqlalchemy.orm.session.Session`) – SQLAlchemy session object bound to PostGIS enabled database
- **channelInputFile** (*gsshapy.orm.ChannelInputFile*) – Channel input file object to be associated with this link node dataset file.
- **force** (*bool, optional*) – Force channel input file reassignment. When false (default), channel input file assignment is skipped if it has already been performed.

getAsKmlAnimation(*session*, *channelInputFile*, *path=None*, *documentName=None*, *styles={}*)

Generate a KML visualization of the the link node dataset file.

Link node dataset files are time stamped link node value datasets. This will yield a value for each stream node at each time step that output is written. The resulting KML visualization will be an animation.

The stream nodes are represented by cylinders where the z dimension/elevation represents the values. A color ramp is applied to make different values stand out even more. The method attempts to identify an appropriate scale factor for the z dimension, but it can be set manually using the styles dictionary.

Parameters

- **session** (`sqlalchemy.orm.session.Session`) – SQLAlchemy session object bound to PostGIS enabled database
- **channelInputFile** (*gsshapy.orm.ChannelInputFile*) – Channel input file object to be associated with this link node dataset file.
- **path** (*str, optional*) – Path to file where KML will be written. Defaults to None.
- **documentName** (*str, optional*) – Name of the KML document. This will be the name that appears in the legend. Defaults to the name of the link node dataset file.
- **styles** (*dict, optional*) – Custom styles to apply to KML geometry. Defaults to empty dictionary.

  Valid keys (styles) include:

  - zScale (float): multiplier to apply to the values (z dimension)
  - radius (float): radius in meters of the node cylinder
  - colorRampEnum (`mapkit.ColorRampGenerator.ColorRampEnum` or dict): Use ColorRampEnum to select a default color ramp or a dictionary with keys 'colors' and 'interpolatedPoints' to specify a custom color ramp. The 'colors' key must be a list of RGB integer tuples (e.g.: (255, 0, 0)) and the 'interpolatedPoints' must be an integer representing the number of points to interpolate between each color given in the colors list.

Returns KML string

Return type str

Supporting Objects

class gsshapy.orm.**LinkNodeTimeStep**(*timeStep*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing data for a single time step of a link node dataset file. Each link node time step will have a link dataset for each stream link in the channel input file.

tableName = u'lnd_time_steps'
Database tablename

id
PK

linkNodeDatasetFileID
FK

linkNodeDataset
RELATIONSHIP

linkDatasets
RELATIONSHIP

timeStep
INTEGER

class gsshapy.orm.**LinkDataset**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing data for a single link dataset in a link node dataset file. A link dataset will have a node dataset for each of the stream nodes belonging to the stream link that it is associated with.

tableName = u'lnd_link_datasets'
Database tablename

id
PK

timeStepID
FK

streamLinkID
FK

linkNodeDatasetFileID
FK

numNodeDatasets
INTEGER

linkNodeDatasetFile
RELATIONSHIP

timeStep
RELATIONSHIP

nodeDatasets
RELATIONSHIP

link
RELATIONSHIP

class gsshapy.orm.**NodeDataset**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing data for a single node dataset in a link node dataset file. The values stored in a link node dataset file are found in the node datasets.

tableName = u'lnd_node_datasets'
Database tablename

id
PK

linkDatasetID
FK

streamNodeID
FK

linkNodeDatasetFileID
FK

status
INTEGER

value
FLOAT

linkNodeDatasetFile
RELATIONSHIP

linkDataset
RELATIONSHIP

node
RELATIONSHIP

Time Series Files

File extensions: **Variable** (e.g.: ohl, ocl, otl)

File Object

class gsshapy.orm.**TimeSeriesFile**
Bases: sqlalchemy.ext.declarative.api.Base, gsshapy.base.file_base.GsshaPyFileObjectBase

Object interface for Time Series Files.

This object stores information from several time series output files. There are two supporting objects that are used to store the contents of this file: *TimeSeries* and *TimeSeriesValue*.

See:

tableName = u'tim_time_series_files'
Database tablename

id
PK

projectFileID
FK

fileExtension
STRING

projectFile
RELATIONSHIP

timeSeries
RELATIONSHIP

as_dataframe()
Return time series as pandas dataframe

Supporting Objects

class gsshapy.orm.**TimeSeries**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base

Object that stores data for a single time series in a time series file.

Time series files can contain several time series datasets. The values for the times series are stored in *TimeSeriesValue* objects.

tableName = u'tim_time_series'
Database tablename

id
PK

timeSeriesFileID
FK

timeSeriesFile
RELATIONSHIP

values
RELATIONSHIP

class gsshapy.orm.**TimeSeriesValue**(*simTime*, *value*)
Bases: sqlalchemy.ext.declarative.api.Base

Object containing the data for a single time series value. Includes the time stamp and value.

tableName = u'tim_time_series_values'
Database tablename

id
PK

timeSeriesID
FK

timeSeries
RELATIONSHIP

simTime
FLOAT

value
FLOAT

WMS Dataset Files

File extensions: **Variable** (e.g.: dep and swe)

This file object supports spatial objects.

File Object

class gsshapy.orm.**WMSDatasetFile**

Bases: sqlalchemy.ext.declarative.api.Base, gsshapy.base.file_base.GsshaPyFileObjectBase

Object interface for WMS Dataset Files.

The WMS dataset file format is used to store gridded timeseries output data for GSSHA. The file contents are abstracted into one other object: *WMSDatasetRaster*. The WMS dataset contains a raster for each time step that output is written.

Note: only the scalar form of the WMS dataset file is supported.

See: http://www.xmswiki.com/xms/WMS:ASCII_Dataset_Files

tableName = u'wms_dataset_files'
Database tablename

id
PK

projectFileID
FK

type
INTEGER

fileExtension
STRING

objectType
STRING

vectorType
STRING

objectID
INTEGER

numberData
INTEGER

numberCells
INTEGER

name
STRING

projectFile
RELATIONSHIP

rasters
RELATIONSHIP

read(*directory*, *filename*, *session*, *maskMap*, *spatial=False*, *spatialReferenceID=4236*)
Read file into the database.

write(*session*, *directory*, *name*, *maskMap*)
Write from database to file.

session = SQLAlchemy session object

directory = to which directory will the files be written (e.g.: '/example/path')

name = name of file that will be written (e.g.: 'my_project.ext')

getAsKmlGridAnimation(*session*, *projectFile=None*, *path=None*, *documentName=None*, *colorRamp=None*, *alpha=1.0*, *noDataValue=0.0*)
Retrieve the WMS dataset as a gridded time stamped KML string.

Parameters

- **session** (sqlalchemy.orm.session.Session) – SQLAlchemy session object bound to PostGIS enabled database.
- **projectFile** (*gsshapy.orm.ProjectFile*) – Project file object for the GSSHA project to which the WMS dataset belongs.
- **path** (*str, optional*) – Path to file where KML file will be written. Defaults to None.
- **documentName** (*str, optional*) – Name of the KML document. This will be the name that appears in the legend. Defaults to 'Stream Network'.
- **colorRamp** (mapkit.ColorRampGenerator.ColorRampEnum or dict, optional) – Use ColorRampEnum to select a default color ramp or a dictionary with keys 'colors' and 'interpolatedPoints' to specify a custom color ramp. The 'colors' key must be a list of RGB integer tuples (e.g.: (255, 0, 0)) and the 'interpolatedPoints' must be an integer representing the number of points to interpolate between each color given in the colors list.
- **alpha** (*float, optional*) – Set transparency of visualization. Value between 0.0 and 1.0 where 1.0 is 100% opaque and 0.0 is 100% transparent. Defaults to 1.0.
- **noDataValue** (*float, optional*) – The value to treat as no data when generating visualizations of rasters. Defaults to 0.0.

Returns KML string

Return type str

getAsKmlPngAnimation(*session*, *projectFile=None*, *path=None*, *documentName=None*, *colorRamp=None*, *alpha=1.0*, *noDataValue=0*, *drawOrder=0*, *cellSize=None*, *resampleMethod=u'NearestNeighbour'*)
Retrieve the WMS dataset as a PNG time stamped KMZ

Parameters

- **session** (sqlalchemy.orm.session.Session) – SQLAlchemy session object bound to PostGIS enabled database.
- **projectFile** (*gsshapy.orm.ProjectFile*) – Project file object for the GSSHA project to which the WMS dataset belongs.
- **path** (*str, optional*) – Path to file where KML file will be written. Defaults to None.
- **documentName** (*str, optional*) – Name of the KML document. This will be the name that appears in the legend. Defaults to 'Stream Network'.

- **colorRamp** (mapkit.ColorRampGenerator.ColorRampEnum or dict, optional) – Use ColorRampEnum to select a default color ramp or a dictionary with keys 'colors' and 'interpolatedPoints' to specify a custom color ramp. The 'colors' key must be a list of RGB integer tuples (e.g.: (255, 0, 0)) and the 'interpolatedPoints' must be an integer representing the number of points to interpolate between each color given in the colors list.
- **alpha** (*float, optional*) – Set transparency of visualization. Value between 0.0 and 1.0 where 1.0 is 100% opaque and 0.0 is 100% transparent. Defaults to 1.0.
- **noDataValue** (*float, optional*) – The value to treat as no data when generating visualizations of rasters. Defaults to 0.0.
- **drawOrder** (*int, optional*) – Set the draw order of the images. Defaults to 0.
- **cellSize** (*float, optional*) – Define the cell size in the units of the project projection at which to resample the raster to generate the PNG. Defaults to None which will cause the PNG to be generated with the original raster cell size. It is generally better to set this to a size smaller than the original cell size to obtain a higher resolution image. However, computation time increases exponentially as the cell size is decreased.
- **resampleMethod** (*str, optional*) – If cellSize is set, this method will be used to resample the raster. Valid values include: NearestNeighbour, Bilinear, Cubic, CubicSpline, and Lanczos. Defaults to NearestNeighbour.

Returns Returns a KML string and a list of binary strings that are the PNG images.

Return type (str, list)

Supporting Objects

class gsshapy.orm.**WMSDatasetRaster**

Bases: sqlalchemy.ext.declarative.api.Base, gsshapy.base.rast.RasterObjectBase

Object storing a single raster dataset for a WMS dataset file.

This object inherits several methods from the gsshapy.orm.RasterObjectBase base class for generating raster visualizations. These methods can be used to generate individual raster visualizations for specific time steps.

tableName = u'wms_dataset_rasters'
Database tablename

id
PK

timeStep
INTEGER

timestamp
FLOAT

iStatus
INTEGER

rasterText
STRING

raster
RASTER

getAsWmsDatasetString(*session*)
Retrieve the WMS Raster as a string in the WMS Dataset format

1.6.3 Base Classes

Gssha File Object Base

This class is inherited by all other GsshaPy file classes. It defines interface used by file objects to read and write files including the `read()` and `write()` methods.

class gsshapy.base.**GsshaPyFileObjectBase**
Abstract base class for all file objects in the GsshaPy ORM.

This base class provides two methods for reading and writing files: `read()` and `write()`. These methods in turn call the private `_read()` and `_write()` methods which must be implemented by child classes.

read(*directory*, *filename*, *session*, *spatial=False*, *spatialReferenceID=4236*, *replaceParamFile=None*, ***kwargs*)
Generic read file into database method.

Parameters

- **directory** (*str*) – Directory containing the file to be read.
- **filename** (*str*) – Name of the file which will be read (e.g.: 'example.prj').
- **session** (sqlalchemy.orm.session.Session) – SQLAlchemy session object bound to PostGIS enabled database.
- **spatial** (*bool, optional*) – If True, spatially enabled objects will be read in as PostGIS spatial objects. Defaults to False.
- **spatialReferenceID** (*int, optional*) – Integer id of spatial reference system for the model. Required if spatial is True. Defaults to srid 4236.
- **replaceParamFile** (*gsshapy.orm.ReplaceParamFile*, optional) – ReplaceParamFile instance. Use this if the file you are reading contains replacement parameters.

write(*session*, *directory*, *name*, *replaceParamFile=None*, ***kwargs*)
Write from database back to file.

Parameters

- **session** (sqlalchemy.orm.session.Session) – SQLAlchemy session object bound to PostGIS enabled database.
- **directory** (*str*) – Directory where the file will be written.
- **name** (*str*) – The name of the file that will be created (including the file extension is optional).
- **replaceParamFile** (*gsshapy.orm.ReplaceParamFile*, optional) – ReplaceParamFile instance. Use this if the file you are writing contains replacement parameters.

_read(*directory*, *filename*, *session*, *path*, *name*, *extension*, *spatial*, *spatialReferenceID*, *replaceParamFile*)
Private file object read method. Classes that inherit from this base class must implement this method.

The `read()` method that each file object inherits from this base class performs the processes common to all file read methods, after which it calls the file object's `_read()` (the preceding underscore denotes that the method is a private method).

The purpose of the `_read()` method is to perform the file read operations that are specific to the file that the file object represents. This method should add any supporting SQLAlchemy objects to the session without committing. The common `read()` method handles the database commit for all file objects.

The `read()` method processes the user input and passes on the information through the many parameters of the `_read()` method. As the `_read()` method should never be called by the user directly, the arguments will be defined in terms of what they offer for the developer of a new file object needing to implement this method.

Parameters

- **directory** (*str*) – Directory containing the file to be read. Same as given by user in `read()`.
- **filename** (*str*) – Name of the file which will be read (e.g.: 'example.prj'). Same as given by user. Same as given by user in `read()`.
- **session** (`sqlalchemy.orm.session.Session`) – SQLAlchemy session object bound to PostGIS enabled database. Same as given by user in `read()`.
- **path** (*str*) – Directory and filename combined into the path to the file. This is a convenience parameter.
- **name** (*str*) – Name of the file without extension. This is a convenience parameter.
- **extension** (*str*) – Extension of the file without the name. This is a convenience parameter.
- **spatial** (*bool, optional*) – If True, spatially enabled objects will be read in as PostGIS spatial objects. Defaults to False. Same as given by user in `read()`.
- **spatialReferenceID** (*int, optional*) – Integer id of spatial reference system for the model. Required if spatial is True. Same as given by user in `read()`.
- **replaceParamFile** (*gsshapy.orm.ReplaceParamFile*, optional) – Handle the case when replacement parameters are used in place of normal variables. If this is not None, then the user expects there to be replacement variables in the file. Use the gsshapy.lib.parsetools.valueReadPreprocessor() to handle these.

_write (*session*, *openFile*, *replaceParamFile*)

Private file object write method. Classes that inherit from this base class must implement this method.

The `write()` method that each file object inherits from this base class performs the processes common to all file write methods, after which it calls the file object's `_write()` (the preceding underscore denotes that the method is a private method).

The purpose of the `_write()` method is to perform the write operations that are specific to the file that the file object represents. This method is passed Python file object that has been "opened". Thus, the developer implementing this method does not need to worry about paths, but simply writes to the opened file object.

Some files have special naming conventions. In these cases, override the write() method and write a custom method.

The `write()` method processes the user input and passes on the information to the `_write()` method. As the `_write()` method should never be called by the user directly, the arguments will be defined in terms of what they offer for the developer of a new file object needing to implement this method.

Parameters

- **session** (sqlalchemy.orm.session.Session) – SQLAlchemy session object bound to PostGIS enabled database. Use this object to query the database during file writing. Same as given by user in `write()`.
- **openFile** (file) – File object that has been instantiated and "opened" for writing by the `write()` method. Write lines of the file directly to this object. (e.g.: `openFile.write('foo')`)
- **replaceParamFile** (*gsshapy.orm.ReplaceParamFile*, optional) – Handle the case when replacement parameters are used in place of normal variables. If this is not None, then the user expects there to be replacement variables in the file. Use the gsshapy.lib.parsetools.valueWritePreprocessor() to handle these.

_namePreprocessor(*name*)
Override this method to preprocess the filename during writing.

Parameters **name** (*str*) – The name of the file without the extension. The file extension will be appended after preprocessing

Returns Filename that will given to the file being written.

Return type str

Geometric Object Base

The `GeometricObject` provides common methods for generating visualizations for geometry type objects. All objects that contain geometry fields inherit from this base class.

class gsshapy.base.**GeometricObjectBase**
Abstract base class for geometric objects.

tableName = None
Name of the table that the geometry column belongs to

id = None
ID of the record with the geometry column in the table that will be retrieved

geometryColumnName = None
Name of the geometry column

getAsKml(*session*)
Retrieve the geometry in KML format.

This method is a veneer for an SQL query that calls the `ST_AsKml()` function on the geometry column.

Parameters **session** (sqlalchemy.orm.session.Session) – SQLAlchemy session object bound to PostGIS enabled database.

Returns KML string representation of geometry.

Return type str

getAsWkt(*session*)
Retrieve the geometry in Well Known Text format.

This method is a veneer for an SQL query that calls the `ST_AsText()` function on the geometry column.

Parameters **session** (sqlalchemy.orm.session.Session) – SQLAlchemy session object bound to PostGIS enabled database.

Returns Well Known Text string representation of geometry.

Return type str

getAsGeoJson(*session*)
Retrieve the geometry in GeoJSON format.

This method is a veneer for an SQL query that calls the `ST_AsGeoJSON()` function on the geometry column.

Parameters **session** (`sqlalchemy.orm.session.Session`) – SQLAlchemy session object bound to PostGIS enabled database.

Returns GeoJSON string representation of geometry.

Return type str

getSpatialReferenceId(*session*)
Retrieve the spatial reference id by which the geometry column is registered.

This method is a veneer for an SQL query that calls the `ST_SRID()` function on the geometry column.

Parameters **session** (`sqlalchemy.orm.session.Session`) – SQLAlchemy session object bound to PostGIS enabled database.

Returns PostGIS spatial reference ID.

Return type str

Raster Object Base

The `RasterObject` provides common methods for generating visualizations for raster type files. All objects that contain rasters inherit from this base class.

class gsshapy.base.**RasterObjectBase**
Abstract base class for raster objects.

getAsKmlGrid(*session*, *path=None*, *documentName=None*, *colorRamp=0*, *alpha=1.0*, *noDataValue=None*)
Retrieve the raster as a KML document with each cell of the raster represented as a vector polygon. The result is a vector grid of raster cells. Cells with the no data value are excluded.

Parameters

- **session** (`sqlalchemy.orm.session.Session`) – SQLAlchemy session object bound to PostGIS enabled database.
- **path** (*str, optional*) – Path to file where KML file will be written. Defaults to None.
- **documentName** (*str, optional*) – Name of the KML document. This will be the name that appears in the legend. Defaults to 'Stream Network'.
- **colorRamp** (`mapkit.ColorRampGenerator.ColorRampEnum` or dict, optional) – Use ColorRampEnum to select a default color ramp or a dictionary with keys 'colors' and 'interpolatedPoints' to specify a custom color ramp. The 'colors' key must be a list of RGB integer tuples (e.g.: (255, 0, 0)) and the 'interpolatedPoints' must be an integer representing the number of points to interpolate between each color given in the colors list.
- **alpha** (*float, optional*) – Set transparency of visualization. Value between 0.0 and 1.0 where 1.0 is 100% opaque and 0.0 is 100% transparent. Defaults to 1.0.
- **noDataValue** (*float, optional*) – The value to treat as no data when generating visualizations of rasters. Defaults to 0.0.

Returns KML string

Return type str

getAsKmlClusters(*session*, *path=None*, *documentName=None*, *colorRamp=0*, *alpha=1.0*, *noDataValue=None*)

Retrieve the raster as a KML document with adjacent cells with the same value aggregated into vector polygons. The result is a vector representation cells clustered together. Cells with the no data value are excluded.

Parameters

- **session** (sqlalchemy.orm.session.Session) – SQLAlchemy session object bound to PostGIS enabled database.
- **path** (*str, optional*) – Path to file where KML file will be written. Defaults to None.
- **documentName** (*str, optional*) – Name of the KML document. This will be the name that appears in the legend. Defaults to 'Stream Network'.
- **colorRamp** (mapkit.ColorRampGenerator.ColorRampEnum or dict, optional) – Use ColorRampEnum to select a default color ramp or a dictionary with keys 'colors' and 'interpolatedPoints' to specify a custom color ramp. The 'colors' key must be a list of RGB integer tuples (e.g.: (255, 0, 0)) and the 'interpolatedPoints' must be an integer representing the number of points to interpolate between each color given in the colors list.
- **alpha** (*float, optional*) – Set transparency of visualization. Value between 0.0 and 1.0 where 1.0 is 100% opaque and 0.0 is 100% transparent. Defaults to 1.0.
- **noDataValue** (*float, optional*) – The value to treat as no data when generating visualizations of rasters. Defaults to 0.0.

Returns KML string

Return type str

getAsKmlPng(*session*, *path=None*, *documentName=None*, *colorRamp=0*, *alpha=1.0*, *noDataValue=None*, *drawOrder=0*, *cellSize=None*, *resampleMethod='NearestNeighbour'*)

Retrieve the raster as a PNG image ground overlay KML format. Coarse grid resolutions must be resampled to smaller cell/pixel sizes to avoid a "fuzzy" look. Cells with the no data value are excluded.

Parameters

- **session** (sqlalchemy.orm.session.Session) – SQLAlchemy session object bound to PostGIS enabled database.
- **path** (*str, optional*) – Path to file where KML file will be written. Defaults to None.
- **documentName** (*str, optional*) – Name of the KML document. This will be the name that appears in the legend. Defaults to 'Stream Network'.
- **colorRamp** (mapkit.ColorRampGenerator.ColorRampEnum or dict, optional) – Use ColorRampEnum to select a default color ramp or a dictionary with keys 'colors' and 'interpolatedPoints' to specify a custom color ramp. The 'colors' key must be a list of RGB integer tuples (e.g.: (255, 0, 0)) and the 'interpolatedPoints' must be an integer representing the number of points to interpolate between each color given in the colors list.
- **alpha** (*float, optional*) – Set transparency of visualization. Value between 0.0 and 1.0 where 1.0 is 100% opaque and 0.0 is 100% transparent. Defaults to 1.0.

- **noDataValue** (*float, optional*) – The value to treat as no data when generating visualizations of rasters. Defaults to 0.0.
- **drawOrder** (*int, optional*) – Set the draw order of the images. Defaults to 0.
- **cellSize** (*float, optional*) – Define the cell size in the units of the project projection at which to resample the raster to generate the PNG. Defaults to None which will cause the PNG to be generated with the original raster cell size. It is generally better to set this to a size smaller than the original cell size to obtain a higher resolution image. However, computation time increases exponentially as the cell size is decreased.
- **resampleMethod** (*str, optional*) – If cellSize is set, this method will be used to resample the raster. Valid values include: NearestNeighbour, Bilinear, Cubic, CubicSpline, and Lanczos. Defaults to NearestNeighbour.

Returns Returns a KML string and a list of binary strings that are the PNG images.

Return type (str, list)

getAsGrassAsciiGrid(*session*)

Retrieve the raster in the GRASS ASCII Grid format.

Parameters **session** (sqlalchemy.orm.session.Session) – SQLAlchemy session object bound to PostGIS enabled database.

Returns GRASS ASCII string.

Return type str

1.6.4 GsshaPy Utilities API

Database Tools

These tools will initialize the database and provide connections to the database for querying the database.

SQLite Database

gsshapy.lib.db_tools.**init_sqlite_db**(*path*, *initTime=False*)

Initialize SQLite Database

Parameters

- **path** (*str*) – Path to database (Ex. '/home/username/my_sqlite.db').
- **initTime** (*Optional[bool]*) – If True, it will print the amount of time to generate database.

Example:

```
from gsshapy.lib.db_tools import init_sqlite_db, create_session

sqlite_db_path = '/home/username/my_sqlite.db'

init_postgresql_db(path=sqlite_db_path)

sqlalchemy_url = init_sqlite_db(path=sqlite_db_path)

db_work_sessionmaker = get_sessionmaker(sqlalchemy_url)
```

```
db_work_session = db_work_sessionmaker()

##DO WORK

db_work_session.close()
```

gsshapy.lib.db_tools.**init_sqlite_memory**(*initTime=False*)

Initialize SQLite in Memory Only Database

Parameters **initTime** (*Optional[bool]*) – If True, it will print the amount of time to generate database.

Returns The tuple contains sqlalchemy_url(str), which is the path to use when creating a session as well as engine(str), which is the path to use when creating a session.

Return type tuple

Example:

```
from gsshapy.lib.db_tools import init_sqlite_memory, create_session

sqlalchemy_url, engine = init_sqlite_memory()

db_work_sessionmaker = get_sessionmaker(sqlalchemy_url, engine)

db_work_session = db_work_sessionmaker()
##DO WORK

db_work_session.close()
```

PostgreSQL Database

gsshapy.lib.db_tools.**init_postgresql_db**(*username*, *host*, *database*, *port=''*, *password=''*, *initTime=False*)

Initialize PostgreSQL Database

Note: psycopg2 or similar driver required

Parameters

- **username** (*str*) – Database username.
- **host** (*str*) – Database host URL.
- **database** (*str*) – Database name.
- **port** (*Optional[int,str]*) – Database port.
- **password** (*Optional[str]*) – Database password.
- **initTime** (*Optional[bool]*) – If True, it will print the amount of time to generate database.

Example:

```
from gsshapy.lib.db_tools import init_postgresql_db, create_session

sqlalchemy_url = init_postgresql_db(username='gsshapy',
                                    host='localhost',
                                    database='gsshapy_mysql_tutorial',
                                    port='5432',
                                    password='pass')

db_work_sessionmaker = get_sessionmaker(sqlalchemy_url)

db_work_session = db_work_sessionmaker()

##DO WORK

db_work_session.close()
```

MySQL Database

gsshapy.lib.db_tools.**init_mysql_db**(*username*, *host*, *database*, *port=''*, *password=''*, *initTime=False*)

Initialize MySQL Database

Note: mysql-python or similar driver required

Parameters

- **username** (*str*) – Database username.
- **host** (*str*) – Database host URL.
- **database** (*str*) – Database name.
- **port** (*Optional[int,str]*) – Database port.
- **password** (*Optional[str]*) – Database password.
- **initTime** (*Optional[bool]*) – If True, it will print the amount of time to generate database.

Example:

```
from gsshapy.lib.db_tools import init_mysql_db, create_session

sqlalchemy_url = init_mysql_db(username='gsshapy',
                               host='localhost',
                               database='gsshapy_mysql_tutorial',
                               port='5432',
                               password='pass')

db_work_sessionmaker = get_sessionmaker(sqlalchemy_url)

db_work_session = db_work_sessionmaker()
##DO WORK

db_work_session.close()
```

1.6.5 GRID API

ERA output to GSSHA input (ERAtoGSSHA)

- https://software.ecmwf.int/wiki/display/CKB/What+is+ERA5
- http://www.ecmwf.int/en/research/climate-reanalysis/era-interim

ERAtoGSSHA

class gsshapy.grid.**ERAtoGSSHA**(*gssha_project_folder*, *gssha_project_file_name*, *lsm_input_folder_path*, *lsm_search_card='*.nc'*, *lsm_lat_var='latitude'*, *lsm_lon_var='longitude'*, *lsm_time_var='time'*, *lsm_lat_dim='latitude'*, *lsm_lon_dim='longitude'*, *lsm_time_dim='time'*, *output_timezone=None*, *download_start_datetime=None*, *download_end_datetime=None*, *era_download_data='era5'*)

Bases: gsshapy.grid.grid_to_gssha.GRIDtoGSSHA

This class converts the ERA5 or ERA Interim output data to GSSHA formatted input. This class inherits from class:*GRIDtoGSSHA*.

Note: https://software.ecmwf.int/wiki/display/CKB/How+to+download+ERA5+test+data+via+the+ECMWF+Web+API

gssha_project_folder
str – Path to the GSSHA project folder

gssha_project_file_name
str – Name of the GSSHA elevation grid file.

lsm_input_folder_path
str – Path to the input folder for the LSM files.

lsm_search_card
str – Glob search pattern for LSM files. Ex. "*.grib2".

lsm_lat_var
Optional[str] – Name of the latitude variable in the LSM netCDF files. Defaults to 'lat'.

lsm_lon_var
Optional[str] – Name of the longitude variable in the LSM netCDF files. Defaults to 'lon'.

lsm_time_var
Optional[str] – Name of the time variable in the LSM netCDF files. Defaults to 'time'.

lsm_lat_dim
Optional[str] – Name of the latitude dimension in the LSM netCDF files. Defaults to 'lat'.

lsm_lon_dim
Optional[str] – Name of the longitude dimension in the LSM netCDF files. Defaults to 'lon'.

lsm_time_dim
Optional[str] – Name of the time dimension in the LSM netCDF files. Defaults to 'time'.

output_timezone
Optional[tzinfo] – This is the timezone to output the dates for the data. Default is he GSSHA model timezone. This option does NOT currently work for NetCDF output.

download_start_datetime
Optional[datetime.datetime] – Datetime to start download.

download_end_datetime
Optional[datetime.datetime] – Datetime to end download.

era_download_data
Optional[str] – You can choose 'era5' or 'interim'. Defaults to 'era5'.

Example:

```
from datetime import datetime
from gsshapy.grid import ERA5toGSSHA

e2g = ERA5toGSSHA(gssha_project_folder='E:\GSSHA',
                  gssha_project_file_name='gssha.prj',
                  lsm_input_folder_path='E:\GSSHA\era5-data',
                  lsm_search_card="*.grib",
                  #download_start_datetime=datetime(2016,1,2),
                  #download_end_datetime=datetime(2016,1,4),
                  )


out_gage_file = 'E:\GSSHA\era5_rain1.gag
e2g.lsm_precip_to_gssha_precip_gage(out_gage_file,
                                    lsm_data_var="tp",
                                    precip_type="GAGES")

data_var_map_array = [
                        ['precipitation_inc', 'tp'],
                        ['pressure', 'sp'],
                        ['relative_humidity_dew', ['d2m','t2m']],
                        ['wind_speed', ['u10', 'v10']],
                        ['direct_radiation', 'aluvp'],
                        ['diffusive_radiation', 'aluvd'],
                        ['temperature', 't2m'],
                        ['cloud_cover', 'tcc'],
                       ]

e2g.lsm_data_to_arc_ascii(data_var_map_array)
```

Download ERA

gsshapy.grid.era_to_gssha.**download_era5_for_gssha**(*main_directory*, *start_datetime*, *end_datetime*, *leftlon=-180*, *rightlon=180*, *toplat=90*, *bottomlat=-90*, *precip_only=False*)

Function to download ERA5 data for GSSHA

Note: https://software.ecmwf.int/wiki/display/WEBAPI/Access+ECMWF+Public+Datasets

Parameters

- **main_directory** (str) – Location of the output for the forecast data.
- **start_datetime** (str) – Datetime for download start.

- **end_datetime** (str) – Datetime for download end.
- **leftlon** (Optional[float]) – Left bound for longitude. Default is -180.
- **rightlon** (Optional[float]) – Right bound for longitude. Default is 180.
- **toplat** (Optional[float]) – Top bound for latitude. Default is 90.
- **bottomlat** (Optional[float]) – Bottom bound for latitude. Default is -90.
- **precip_only** (*Optional[bool]*) – If True, will only download precipitation.

Example:

```
from gsshapy.grid.era_to_gssha import download_era5_for_gssha

era5_folder = '/era5'
leftlon = -95
rightlon = -75
toplat = 35
bottomlat = 30
download_era5_for_gssha(era5_folder, leftlon, rightlon, toplat, bottomlat)
```

gsshapy.grid.era_to_gssha.**download_interim_for_gssha**(*main_directory*, *start_datetime*, *end_datetime*, *leftlon=-180*, *rightlon=180*, *toplat=90*, *bottomlat=-90*, *precip_only=False*)

Function to download ERA5 data for GSSHA

Note: https://software.ecmwf.int/wiki/display/WEBAPI/Access+ECMWF+Public+Datasets

Parameters

- **main_directory** (str) – Location of the output for the forecast data.
- **start_datetime** (str) – Datetime for download start.
- **end_datetime** (str) – Datetime for download end.
- **leftlon** (Optional[float]) – Left bound for longitude. Default is -180.
- **rightlon** (Optional[float]) – Right bound for longitude. Default is 180.
- **toplat** (Optional[float]) – Top bound for latitude. Default is 90.
- **bottomlat** (Optional[float]) – Bottom bound for latitude. Default is -90.
- **precip_only** (*Optional[bool]*) – If True, will only download precipitation.

Example:

```
from gsshapy.grid.era_to_gssha import download_era_interim_for_gssha

era_interim_folder = '/era_interim'
leftlon = -95
rightlon = -75
toplat = 35
bottomlat = 30
download_era_interim_for_gssha(era5_folder, leftlon, rightlon, toplat, bottomlat)
```

National Water Model output to GSSHA input (NWMtoGSSHA)

http://water.noaa.gov/about/nwm

NWMtoGSSHA

class gsshapy.grid.**NWMtoGSSHA**(*gssha_project_folder*, *gssha_project_file_name*, *lsm_input_folder_path*, *lsm_search_card='*.nc'*, *lsm_lat_var='y'*, *lsm_lon_var='x'*, *lsm_time_var='time'*, *lsm_lat_dim='y'*, *lsm_lon_dim='x'*, *lsm_time_dim='time'*, *output_timezone=None*)

Bases: gsshapy.grid.grid_to_gssha.GRIDtoGSSHA

This class converts the National Water Model output data to GSSHA formatted input. This class inherits from class:*GRIDtoGSSHA*.

gssha_project_folder
str – Path to the GSSHA project folder

gssha_project_file_name
str – Name of the GSSHA elevation grid file.

lsm_input_folder_path
str – Path to the input folder for the LSM files.

lsm_search_card
str – Glob search pattern for LSM files. Ex. "*.grib2".

lsm_lat_var
Optional[str] – Name of the latitude variable in the LSM netCDF files. Defaults to 'lat'.

lsm_lon_var
Optional[str] – Name of the longitude variable in the LSM netCDF files. Defaults to 'lon'.

lsm_time_var
Optional[str] – Name of the time variable in the LSM netCDF files. Defaults to 'time'.

lsm_lat_dim
Optional[str] – Name of the latitude dimension in the LSM netCDF files. Defaults to 'lat'.

lsm_lon_dim
Optional[str] – Name of the longitude dimension in the LSM netCDF files. Defaults to 'lon'.

lsm_time_dim
Optional[str] – Name of the time dimension in the LSM netCDF files. Defaults to 'time'.

output_timezone
Optional[tzinfo] – This is the timezone to output the dates for the data. Default is he GSSHA model timezone. This option does NOT currently work for NetCDF output.

Example:

```
from datetime import datetime
from gsshapy.grid import NWMtoGSSHA

n2g = NWMtoGSSHA(gssha_project_folder='E:\GSSHA',
                 gssha_project_file_name='gssha.prj',
                 lsm_input_folder_path='E:\GSSHA\nwm-data',
                 lsm_search_card="*.grib")
```

```
# example rain gage
out_gage_file = 'E:\GSSHA\nwm_rain1.gag'
n2g.lsm_precip_to_gssha_precip_gage(out_gage_file,
                                    lsm_data_var="RAINRATE",
                                    precip_type="RADAR")

# example data var map array
# WARNING: This is not complete
data_var_map_array = [
    ['precipitation_rate', 'RAINRATE'],
    ['pressure', 'PSFC'],
    ['relative_humidity', ['Q2D','T2D', 'PSFC']],
    ['wind_speed', ['U2D', 'V2D']],
    ['direct_radiation', 'SWDOWN'],  # ???
    ['diffusive_radiation', 'SWDOWN'],  # ???
    ['temperature', 'T2D'],
    ['cloud_cover', '????'],
]
e2g.lsm_data_to_arc_ascii(data_var_map_array)
```

1.6.6 Modeling API

GSSHA Framework

(Optional) Install spt_dataset_manager

Part of the code depends on spt_dataset_manager. This can be installed by following the instructions here: https://github.com/erdc-cm/spt_dataset_manager.

Warning: Make sure you have your Miniconda gssha environment activated during installation.

GSSHAFramework

class gsshapy.modeling.**GSSHAFramework**(*gssha_executable*, *gssha_directory*, *project_filename*, *gssha_simulation_start=None*, *gssha_simulation_end=None*, *gssha_simulation_duration=None*, *load_simulation_datetime=False*, *spt_watershed_name=None*, *spt_subbasin_name=None*, *spt_forecast_date_string=None*, *ckan_engine_url=None*, *ckan_api_key=None*, *ckan_owner_organization=None*, *path_to_rapid_qout=None*, *connection_list_file=None*, *lsm_folder=None*, *lsm_data_var_map_array=None*, *lsm_precip_data_var=None*, *lsm_precip_type=None*, *lsm_lat_var=None*, *lsm_lon_var=None*, *lsm_time_var='time'*, *lsm_lat_dim=None*, *lsm_lon_dim=None*, *lsm_time_dim='time'*, *lsm_search_card='*.nc'*, *precip_interpolation_type=None*, *event_min_q=None*, *et_calc_mode=None*, *soil_moisture_depth=None*, *output_netcdf=False*, *write_hotstart=False*, *read_hotstart=False*, *hotstart_minimal_mode=False*, *grid_module='grid'*)

This class is for automating the connection between RAPID to GSSHA and LSM to GSSHA. There are several different configurations depending upon what you choose.

There are three options for RAPID to GSSHA:

1. Download and run using forecast from the Streamflow Prediction Tool (See: https://streamflow-prediction-tool.readthedocs.io)
2. Run from RAPID Qout file
3. Don't run using RAPID to GSSHA

There are two options for LSM to GSSHA:

1. Run from LSM to GSSHA
2. Don't run using LSM to GSSHA

Parameters

- **gssha_executable** (*str*) – Path to GSSHA executable.
- **gssha_directory** (*str*) – Path to directory for GSSHA project.
- **project_filename** (*str*) – Name of GSSHA project file.
- **gssha_simulation_start** (*Optional[datetime]*) – Datetime object with date of start of GSSHA simulation.
- **gssha_simulation_end** (*Optional[datetime]*) – Datetime object with date of end of GSSHA simulation.
- **gssha_simulation_duration** (*Optional[timedelta]*) – Datetime timedelta object with duration of GSSHA simulation.

- **load_simulation_datetime** (*Optional[bool]*) – If True, this will load in datetime information from the project file. Default is False.
- **spt_watershed_name** (*Optional[str]*) – Streamflow Prediction Tool watershed name.
- **spt_subbasin_name** (*Optional[str]*) – Streamflow Prediction Tool subbasin name.
- **spt_forecast_date_string** (*Optional[str]*) – Streamflow Prediction Tool forecast date string.
- **ckan_engine_url** (*Optional[str]*) – CKAN engine API url.
- **ckan_api_key** (*Optional[str]*) – CKAN api key.
- **ckan_owner_organization** (*Optional[str]*) – CKAN owner organization.
- **path_to_rapid_qout** (*Optional[str]*) – Path to the RAPID Qout file. Use this if you do NOT want to download the forecast and you want to use RAPID streamflows.
- **connection_list_file** (*Optional[str]*) – CSV file with list connecting GSSHA rivers to RAPID river network. See: http://rapidpy.readthedocs.io/en/latest/rapid_to_gssha.html
- **lsm_folder** (*Optional[str]*) – Path to folder with land surface model data. See: *lsm_input_folder_path* variable at *GRIDtoGSSHA()*.
- **lsm_data_var_map_array** (*Optional[str]*) – Array with connections for LSM output and GSSHA input. See: ()
- **lsm_precip_data_var** (*Optional[list or str]*) – String of name for precipitation variable name or list of precip variable names. See: *lsm_precip_to_gssha_precip_gage()*.
- **lsm_precip_type** (*Optional[str]*) – Type of precipitation. See: *lsm_precip_to_gssha_precip_gage()*.
- **lsm_lat_var** (*Optional[str]*) – Name of the latitude variable in the LSM netCDF files. See: *GRIDtoGSSHA()*.
- **lsm_lon_var** (*Optional[str]*) – Name of the longitude variable in the LSM netCDF files. See: *GRIDtoGSSHA()*.
- **lsm_time_var** (*Optional[str]*) – Name of the time variable in the LSM netCDF files. See: *GRIDtoGSSHA()*.
- **lsm_lat_dim** (*Optional[str]*) – Name of the latitude dimension in the LSM netCDF files. See: *GRIDtoGSSHA()*.
- **lsm_lon_dim** (*Optional[str]*) – Name of the longitude dimension in the LSM netCDF files. See: *GRIDtoGSSHA()*.
- **lsm_time_dim** (*Optional[str]*) – Name of the time dimension in the LSM netCDF files. See: *GRIDtoGSSHA()*.
- **lsm_search_card** (*Optional[str]*) – Glob search pattern for LSM files. See: GRIDtoGSSHA().
- **precip_interpolation_type** (*Optional[str]*) – Type of interpolation for LSM precipitation. Can be "INV_DISTANCE" or "THIESSEN". Default is "THIESSEN".

- **event_min_q** (*Optional[double]*) – Threshold discharge for continuing runoff events in m3/s. Default is 60.0.
- **et_calc_mode** (*Optional[str]*) – Type of evapo-transpitation calculation for GSSHA. Can be "PENMAN" or "DEARDORFF". Default is "PENMAN".
- **soil_moisture_depth** (*Optional[double]*) – Depth of the active soil moisture layer from which ET occurs (m). Default is 0.0.
- **output_netcdf** (*Optional[bool]*) – If you want the HMET data output as a NetCDF4 file for input to GSSHA. Default is False.
- **write_hotstart** (*Optional[bool]*) – If you want to automatically generate all hotstart files, set to True. Default is False.
- **read_hotstart** (*Optional[bool]*) – If you want to automatically search for and read in hotstart files, set to True. Default is False.
- **hotstart_minimal_mode** (*Optional[bool]*) – If you want to turn off all outputs to only generate the hotstart file, set to True. Default is False.
- **grid_module** (str) – The name of the LSM tool needed. Options are 'grid', 'hrrr', or 'era'. Default is 'grid'.

Example modifying parameters during class initialization:

```
from gsshapy.modeling import GSSHAFramework

gssha_executable = 'C:/Program Files/WMS 10.1 64-bit/gssha/gssha.exe'
gssha_directory = "C:/Users/{username}/Documents/GSSHA"
project_filename = "gssha_project.prj"

#WRF INPUTS
lsm_folder = '"C:/Users/{username}/Documents/GSSHA/wrf-sample-data-v1.0'
lsm_lat_var = 'XLAT'
lsm_lon_var = 'XLONG'
search_card = '*.nc'
precip_data_var = ['RAINC', 'RAINNC']
precip_type = 'ACCUM'

data_var_map_array = [
                      ['precipitation_acc', ['RAINC', 'RAINNC']],
                      ['pressure', 'PSFC'],
                      ['relative_humidity', ['Q2', 'PSFC', 'T2']],
                      ['wind_speed', ['U10', 'V10']],
                      ['direct_radiation', ['SWDOWN', 'DIFFUSE_FRAC']],
                      ['diffusive_radiation', ['SWDOWN', 'DIFFUSE_FRAC']],
                      ['temperature', 'T2'],
                      ['cloud_cover' , 'CLDFRA'],
                     ]

#INITIALIZE CLASS AND RUN
gr = GSSHAFramework(gssha_executable,
                    gssha_directory,
                    project_filename,
                    lsm_folder=lsm_folder,
                    lsm_data_var_map_array=data_var_map_array,
                    lsm_precip_data_var=precip_data_var,
                    lsm_precip_type=precip_type,
                    lsm_lat_var=lsm_lat_var,
```

```
                    lsm_lon_var=lsm_lon_var,
                    )

gr.run_forecast()
```

GSSHA_WRF_Framework

class gsshapy.modeling.**GSSHAWRFFramework**(*gssha_executable, gssha_directory, project_filename, gssha_simulation_start=None, gssha_simulation_end=None, gssha_simulation_duration=None, load_simulation_datetime=False, spt_watershed_name=None, spt_subbasin_name=None, spt_forecast_date_string=None, ckan_engine_url=None, ckan_api_key=None, ckan_owner_organization=None, path_to_rapid_qout=None, connection_list_file=None, lsm_folder=None, lsm_data_var_map_array=None, lsm_precip_data_var=['RAINC', 'RAINNC'], lsm_precip_type='ACCUM', lsm_lat_var='XLAT', lsm_lon_var='XLONG', lsm_time_var='Times', lsm_lat_dim='south_north', lsm_lon_dim='west_east', lsm_time_dim='Time', lsm_search_card='*.nc', precip_interpolation_type=None, event_min_q=None, et_calc_mode=None, soil_moisture_depth=None, output_netcdf=False, write_hotstart=False, read_hotstart=False, hotstart_minimal_mode=False*)

Bases: gsshapy.modeling.framework.GSSHAFramework

This class is for automating the connection between RAPID to GSSHA and WRF to GSSHA. There are several different configurations depending upon what you choose.

There are three options for RAPID to GSSHA:

1. Download and run using forecast from the Streamflow Prediction Tool (See: https://streamflow-prediction-tool.readthedocs.io)
2. Run from RAPID Qout file
3. Don't run using RAPID to GSSHA

There are two options for WRF to GSSHA:

1. Run from WRF to GSSHA
2. Don't run using WRF to GSSHA

Parameters

- **gssha_executable** (*str*) – Path to GSSHA executable.

- **gssha_directory** (*str*) – Path to directory for GSSHA project.
- **project_filename** (*str*) – Name of GSSHA project file.
- **gssha_simulation_start** (*Optional[datetime]*) – Datetime object with date of start of GSSHA simulation.
- **gssha_simulation_end** (*Optional[datetime]*) – Datetime object with date of end of GSSHA simulation.
- **gssha_simulation_duration** (*Optional[timedelta]*) – Datetime timedelta object with duration of GSSHA simulation.
- **load_simulation_datetime** (*Optional[bool]*) – If True, this will load in datetime information from the project file. Default is False.
- **spt_watershed_name** (*Optional[str]*) – Streamflow Prediction Tool watershed name.
- **spt_subbasin_name** (*Optional[str]*) – Streamflow Prediction Tool subbasin name.
- **spt_forecast_date_string** (*Optional[str]*) – Streamflow Prediction Tool forecast date string.
- **ckan_engine_url** (*Optional[str]*) – CKAN engine API url.
- **ckan_api_key** (*Optional[str]*) – CKAN api key.
- **ckan_owner_organization** (*Optional[str]*) – CKAN owner organization.
- **path_to_rapid_qout** (*Optional[str]*) – Path to the RAPID Qout file. Use this if you do NOT want to download the forecast and you want to use RAPID streamflows.
- **connection_list_file** (*Optional[str]*) – CSV file with list connecting GSSHA rivers to RAPID river network. See: http://rapidpy.readthedocs.io/en/latest/rapid_to_gssha.html
- **lsm_folder** (*Optional[str]*) – Path to folder with land surface model data. See: *lsm_input_folder_path* variable at *GRIDtoGSSHA()*.
- **lsm_data_var_map_array** (*Optional[str]*) – Array with connections for WRF output and GSSHA input. See: ()
- **lsm_precip_data_var** (*Optional[list or str]*) – String of name for precipitation variable name or list of precip variable names. See: *lsm_precip_to_gssha_precip_gage()*.
- **lsm_precip_type** (*Optional[str]*) – Type of precipitation. See: *lsm_precip_to_gssha_precip_gage()*.
- **lsm_lat_var** (*Optional[str]*) – Name of the latitude variable in the WRF netCDF files. See: *GRIDtoGSSHA()*.
- **lsm_lon_var** (*Optional[str]*) – Name of the longitude variable in the WRF netCDF files. See: *GRIDtoGSSHA()*.
- **lsm_time_var** (*Optional[str]*) – Name of the time variable in the WRF netCDF files. See: *GRIDtoGSSHA()*.
- **lsm_lat_dim** (*Optional[str]*) – Name of the latitude dimension in the LSM netCDF files. See: *GRIDtoGSSHA()*.
- **lsm_lon_dim** (*Optional[str]*) – Name of the longitude dimension in the LSM netCDF files. See: *GRIDtoGSSHA()*.

- **lsm_time_dim** (*Optional[str]*) – Name of the time dimension in the LSM netCDF files. See: *GRIDtoGSSHA()*.
- **lsm_search_card** (*Optional[str]*) – Glob search pattern for WRF files. See: *GRIDtoGSSHA()*.
- **precip_interpolation_type** (*Optional[str]*) – Type of interpolation for WRF precipitation. Can be "INV_DISTANCE" or "THIESSEN". Default is "THIESSEN".
- **event_min_q** (*Optional[double]*) – Threshold discharge for continuing runoff events in m3/s. Default is 60.0.
- **et_calc_mode** (*Optional[str]*) – Type of evapo-transpitation calculation for GSSHA. Can be "PENMAN" or "DEARDORFF". Default is "PENMAN".
- **soil_moisture_depth** (*Optional[double]*) – Depth of the active soil moisture layer from which ET occurs (m). Default is 0.0.
- **output_netcdf** (*Optional[bool]*) – If you want the HMET data output as a NetCDF4 file for input to GSSHA. Default is False.
- **write_hotstart** (*Optional[bool]*) – If you want to automatically generate all hotstart files, set to True. Default is False.
- **read_hotstart** (*Optional[bool]*) – If you want to automatically search for and read in hotstart files, set to True. Default is False.
- **hotstart_minimal_mode** (*Optional[bool]*) – If you want to turn off all outputs to only generate the hotstart file, set to True. Default is False.

Example running full framework with RAPID and LSM locally stored:

```
from gsshapy.modeling import GSSHAWRFFramework

gssha_executable = 'C:/Program Files/WMS 10.1 64-bit/gssha/gssha.exe'
gssha_directory = "C:/Users/{username}/Documents/GSSHA"
project_filename = "gssha_project.prj"

#LSM TO GSSHA
lsm_folder = '"C:/Users/{username}/Documents/GSSHA/wrf-sample-data-v1.0'

#RAPID TO GSSHA
path_to_rapid_qout = "C:/Users/{username}/Documents/GSSHA/Qout.nc"
connection_list_file = "C:/Users/{username}/Documents/GSSHA/rapid_to_gssha_
↪connect.csv"

#INITIALIZE CLASS AND RUN
gr = GSSHAWRFFramework(gssha_executable,
                       gssha_directory,
                       project_filename,
                       lsm_folder=lsm_folder,
                       path_to_rapid_qout=path_to_rapid_qout,
                       connection_list_file=connection_list_file,
                       )

gr.run_forecast()
```

Example connecting SPT to GSSHA:

```
from gsshapy.modeling import GSSHAWRFFramework

gssha_executable = 'C:/Program Files/WMS 10.1 64-bit/gssha/gssha.exe'
gssha_directory = "C:/Users/{username}/Documents/GSSHA"
project_filename = "gssha_project.prj"

#LSM TO GSSHA
lsm_folder = '"C:/Users/{username}/Documents/GSSHA/wrf-sample-data-v1.0'

#RAPID TO GSSHA
connection_list_file = "C:/Users/{username}/Documents/GSSHA/rapid_to_gssha_
↪connect.csv"

#SPT TO GSSHA
ckan_engine_url='http://ckan/api/3/action'
ckan_api_key='your-api-key'
ckan_owner_organization='your_organization'
spt_watershed_name='watershed_name'
spt_subbasin_name='subbasin_name'
spt_forecast_date_string='20160721.1200'

#INITIALIZE CLASS AND RUN
gr = GSSHAWRFFramework(gssha_executable,
                       gssha_directory,
                       project_filename,
                       lsm_folder=lsm_folder,
                       connection_list_file=connection_list_file,
                       ckan_engine_url=ckan_engine_url,
                       ckan_api_key=ckan_api_key,
                       ckan_owner_organization=ckan_owner_organization,
                       spt_watershed_name=spt_watershed_name,
                       spt_subbasin_name=spt_subbasin_name,
                       spt_forecast_date_string=spt_forecast_date_string,
                       )

gr.run_forecast()
```

Example with Hotstart:

```
from datetime import datetime, timedelta
from gsshapy.modeling import GSSHAWRFFramework

gssha_executable = 'C:/Program Files/WMS 10.1 64-bit/gssha/gssha.exe'
gssha_directory = "C:/Users/{username}/Documents/GSSHA"
project_filename = "gssha_project.prj"
full_gssha_simulation_duration = timedelta(days=5, seconds=0)
gssha_hotstart_offset_duration = timedelta(days=1, seconds=0)

#LSM
lsm_folder = '"C:/Users/{username}/Documents/GSSHA/wrf-sample-data-v1.0'

#RAPID
path_to_rapid_qout = "C:/Users/{username}/Documents/GSSHA/Qout.nc"
connection_list_file = "C:/Users/{username}/Documents/GSSHA/rapid_to_gssha_
↪connect.csv"

#-----------------------------------------------------------------------
# MAIN RUN
```

```
#------------------------------------------------------------------------
mr = GSSHAWRFFramework(gssha_executable,
                       gssha_directory,
                       project_filename,
                       lsm_folder=lsm_folder,
                       path_to_rapid_qout=path_to_rapid_qout,
                       connection_list_file=connection_list_file,
                       gssha_simulation_duration=full_gssha_simulation_duration,
                       read_hotstart=True,
                       )

mr.run_forecast()
#------------------------------------------------------------------------
# GENERATE HOTSTART FOR NEXT RUN
#------------------------------------------------------------------------
hr = GSSHAWRFFramework(gssha_executable,
                       gssha_directory,
                       project_filename,
                       lsm_folder=lsm_folder,
                       path_to_rapid_qout=path_to_rapid_qout,
                       connection_list_file=connection_list_file,
                       gssha_simulation_duration=gssha_hotstart_offset_duration,
                       write_hotstart=True,
                       read_hotstart=True,
                       hotstart_minimal_mode=True,
                       )
hr.run_forecast()
```

GSSHA Model

GSSHAModel

class gsshapy.modeling.**GSSHAModel**(*project_directory*, *project_name=None*, *mask_shapefile=None*, *auto_clean_mask_shapefile=False*, *grid_cell_size=None*, *elevation_grid_path=None*, *simulation_timestep=30*, *out_hydrograph_write_frequency=10*, *roughness=None*, *land_use_grid=None*, *land_use_grid_id=None*, *land_use_to_roughness_table=None*, *load_rasters_to_db=True*, *db_session=None*, *project_manager=None*)

This class manages the generation and modification of models for GSSHA.

Parameters

- **project_directory** (*str*) – Directory to write GSSHA project files to.
- **project_name** (*Optional[str]*) – Name of GSSHA project. Required for new model.
- **mask_shapefile** (*Optional[str]*) – Path to watershed boundary shapefile. Required for new model.
- **auto_clean_mask_shapefile** (*Optional[bool]*) – Chooses the largest region if the input is a multipolygon. Default is False.

- **grid_cell_size** (*Optional[str]*) – Cell size of model (meters). Required for new model.
- **elevation_grid_path** (*Optional[str]*) – Path to elevation raster used for GSSHA grid. Required for new model.
- **simulation_timestep** (*Optional[float]*) – Overall model timestep (seconds). Sets TIMESTEP card. Required for new model.
- **out_hydrograph_write_frequency** (*Optional[str]*) – Frequency of writing to hydrograph (minutes). Sets HYD_FREQ card. Required for new model.
- **roughness** (*Optional[float]*) – Value of uniform manning's n roughness for grid. Mutually exlusive with land use roughness. Required for new model.
- **land_use_grid** (*Optional[str]*) – Path to land use grid to use for roughness. Mutually exlusive with roughness. Required for new model.
- **land_use_grid_id** (*Optional[str]*) – ID of default grid supported in GSSHApy. Mutually exlusive with roughness. Required for new model.
- **land_use_to_roughness_table** (*Optional[str]*) – Path to land use to roughness table. Use if not using land_use_grid_id. Mutually exlusive with roughness. Required for new model.
- **load_rasters_to_db** (*Optional[bool]*) – If True, it will load the created rasters into the database. IF you are generating a large model, it is recommended to set this to False. Default is True.
- **db_session** (*Optional[database session]*) – Active database session object. Required for existing model.
- **project_manager** (*Optional[ProjectFile]*) – Initialized ProjectFile object. Required for existing model.

Model Generation Example:

```
from datetime import datetime, timedelta
from gsshapy.modeling import GSSHAModel

model = GSSHAModel(project_name="gssha_project",
                   project_directory="/path/to/gssha_project",
                   mask_shapefile="/path/to/watershed_boundary.shp",
                   auto_clean_mask_shapefile=True,
                   grid_cell_size=1000,
                   elevation_grid_path="/path/to/elevation.tif",
                   simulation_timestep=10,
                   out_hydrograph_write_frequency=15,
                   land_use_grid='/path/to/land_use.tif',
                   land_use_grid_id='glcf',
                   load_rasters_to_db=False,
                   )
model.set_event(simulation_start=datetime(2017, 2, 28, 14, 33),
                simulation_duration=timedelta(seconds=180*60),
                rain_intensity=2.4,
                rain_duration=timedelta(seconds=30*60),
                )
model.write()
```

CHAPTER 2

Indices and Tables

- genindex
- search

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y