[NOTE: 2](#_Toc346021930)

[Software requirements 2](#_Toc346021931)

[Database: 2](#_Toc346021932)

[ PostgreSQL 2](#_Toc346021933)

[ PostGIS 2](#_Toc346021934)

[ pgAdmin 2](#_Toc346021935)

[ PostGIS Shapefile and DBF Loader - 2](#_Toc346021936)

[Custom Database: 2](#_Toc346021937)

[Python: 2](#_Toc346021938)

[Custom Python: 2](#_Toc346021939)

[Misc: 3](#_Toc346021940)

[Building a national-scale rail network, from Ordnance Survey Meridian 2 Railway Line data 4](#_Toc346021941)

[Loading the Ordnance Survey Meridian 2 Railway Line data 4](#_Toc346021942)

[Create a table of end points (using ESRI ArcGIS) 5](#_Toc346021943)

[Create a table of end points (using PostgreSQL / PostGIS) 5](#_Toc346021944)

[SQL Code to create table of intersection counts between rail\_ln\_polyline data and end points generated in previous step 6](#_Toc346021945)

[Creating a table of junctions 7](#_Toc346021946)

[Creating a table of unique junctions based on geometry 7](#_Toc346021947)

[Adding a type column to the unique junction table 8](#_Toc346021948)

[Combining junctions and original input stations 8](#_Toc346021949)

[Configuring db\_settings 9](#_Toc346021950)

[Configuring layer\_settings.py 10](#_Toc346021951)

[Defining layers in layer settings 10](#_Toc346021952)

[Checking ws2\_database\_network\_handlers.py matches layer\_settings definitions 11](#_Toc346021953)

[Executing the build procedure (Python): 11](#_Toc346021954)

## NOTE:

Please ensure that you have read the accompanying document ***‘Tutorial – Installing the database schema.docx’***, which details the various methods available to install the database to either an existing spatial database with data already defined, or to an empty database.

## Software requirements

### Database:

* PostgreSQL – open source relational database software (9.0.3, 32 bit)
* PostGIS – open source spatial extension for PostgreSQL (1.5.2)
  + - Check PostGIS version with SQL “SELECT PostGIS\_full\_version();”
    - E.g. "POSTGIS="1.5.2" GEOS="3.2.2-CAPI-1.6.2" PROJ="Rel. 4.6.1, 21 August 2008" LIBXML="2.7.6" USE\_STATS"
* pgAdmin – open source database viewer / administration tool (ships with PostgreSQL, currently used 1.12.2)
* PostGIS Shapefile and DBF Loader - http://postgis.refractions.net/download/windows/pg90/postgis-pg90-setup-2.0.1-1.exe

### Custom Database:

* a copy of the network\_interdependency.backup database schema is required. This can be used when an empty database has been created (using the template template\_postgis) and a user wishes to transform this to be able to use the custom schema. This is available from the pg\_schema/backup folder of the database folder structure

OR

* a copy of the ni\_setup\_database.bat script, with knowledge of the required values for the following mandatory parameters:
* PostgreSQL service host name (default usually “localhost”)
* PostgreSQL postgres user name (default is “postgres”)
* Path to PostgreSQL bin folder e.g. C:\Program Files (x86)\PostgreSQL\9.0\bin\
* Name of a database, and can be either an existing database or name of an empty new database just created.

This option can be used to install the database schema in to a database (one created with template template\_postgis) that may already exist, and already has data within it that a user does not wish to overwrite.

### Python:

* Python version: 2.6.5 (32 bit) or later (but not version 3).
* Spyder 2.1.7 or later
* Network X (1.6)
* OSGeo (GDAL & OGR) 1.8.0 or later
* Pscycopg2 (2-2.0.14) - <http://www.stickpeople.com/projects/python/win-psycopg/> - **BE SURE TO PICK THE CORRECT VERSION FOR YOUR COMBINATION OF POSTGRES AND PYTHON**

### Custom Python:

* nx\_pg.py – module to read PostGIS / shapefiles and convert to networkx graph instance
* nx\_pgnet.py – module to read/write networkx graph instances to and from the custom network\_interdependency database schema
* WS2DatabaseHandler.py – basic database handler
* db\_settings.py – custom settings files for each database (this can be replaced with an individual set of database settings, as defined by the user, but using the format described in this file
* layer\_settings.py – custom settings files for each set of layers to be used as inputs for network creation i.e. defines a set of edges and nodes tables in PostGIS to be used as inputs for creating the networkx graph instance
* ws2\_database\_network\_handlers.py – set of network handlers and methods linking between the nx\_pg/nx\_pgnet modules and the db\_settings/layer\_settings modules. The classes within this module allow specific creation of specific networks, based on specific data.

### Misc:

* Quantum GIS (1.7.0+) - <http://hub.qgis.org/projects/quantum-gis/wiki/Download>
* OSGeo4W - <http://hub.qgis.org/projects/quantum-gis/wiki/Download#12-OSGeo4W-Installer>

**NOTE: The following steps assume that you have a working knowledge of PostgreSQL, PostGIS, and the administrative interface, pgAdmin. Furthermore, these steps assume that all the afore-mentioned pieces of software have been installed and configured, in particular:**

* Installation and configuration of PostgreSQL and PostGIS
* Installation of pgAdmin (if using a different version than that which ships with PostgreSQL)
* Installation of the PostGIS Shapefile and DBF Loader
* Installation and configuration of Python, networkx, GDAL/OGR via OSGeo4W installer, nx\_pg and nx\_pgnet modules

## Building a national-scale rail network, from Ordnance Survey Meridian 2 Railway Line data

### Loading the Ordnance Survey Meridian 2 Railway Line data

1. Open pgAdmin
2. Connect to the transport database
3. From the “plugins” menu at the top of pgAdmin, select “PostGIS Shapefile and DBF Loader”. You should now see the plugin dialog window:

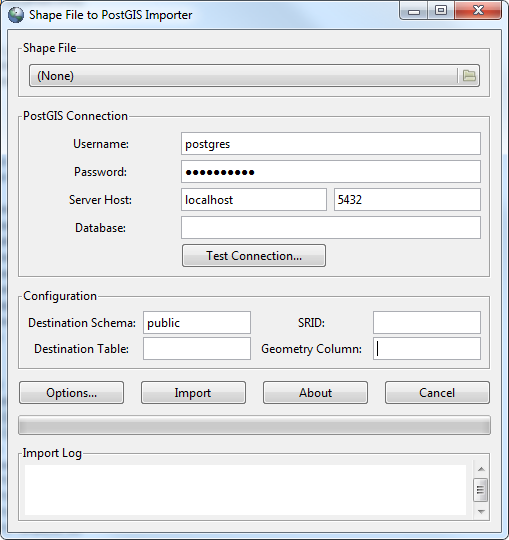


Figure - PostGIS Shapefile and DBF Loader dialog

1. Click the browse button  next to the “Shape File” option at the top of dialog. This should launch a file browser
2. Navigate to the folder where you saved the OS Meridian 2 rail\_ln\_polyline.shp, and select that file. This data is available within the /doc/tutorial/data/OS Meridian 2 Rail Data/ folder of the database schema. This should populate the “Destination Table” value with the same name i.e. rail\_ln\_polyline.
3. Type the name of the database that you wish the data be loaded to, in to the “Database” box. Click “Test Connection” to test that that username, password, server host and port parameters are configured correctly. The words “Connection Succeeded” should appear if this step has been successful.
4. Type the EPSG code (SRID) of the data i.e. 27700 in to the SRID box
5. Type “geom” in to the Geometry Column box. This defines the name of the column that will contain the geometry in the final table in PostGIS, and also the name of the column that is subsequently added to the geometry\_columns table in PostGIS.
6. Click “Import”. The data should begin to import, and upon completion there should be 8495 features.

Repeat steps 4 – 9, but this time selecting the station\_point.shp file, to upload the OS Meridian 2 railway stations.

10. Once the station\_point data has been loaded in to PostGIS, add a field of type varchar, length 50, called “type” e.g.

ALTER TABLE station\_point ADD COLUMN “type” varchar(50);

11. Once the “type” column has been added, values of ‘Station’ need to be set so that when the junctions are generated in the next steps and combined with the stations, it is possible to distinguish which points are stations and which are junctions.

UPDATE station\_point SET “type” = ‘Station’

### Create a table of end points (using ESRI ArcGIS)

This step creates a table of start and end points for every edge in the input rail\_ln\_polyline data. This table can then be used to create a table of junctions that exist on the railway i.e. places where it is feasible for a train to transfer between rail lines. Please note that simply extracting the intersection points from the input rail\_ln\_polyline data is not sufficient as this includes points where it is not possible for a train to transfer between lines.

1) Open ESRI ArcGIS

2) Load the original rail\_ln\_polyline shapefile

3) Open ArcToolbox and select the “Feature Vertices to Points” tool

4) Select the rail\_ln\_polyline file as the Input Features

5) Define an appropriate name for the Output Feature Class e.g. rail\_ln\_polyline\_end\_point\_vertices.shp

6) From the available options, select the available Point Type option “End Points”. This tool will create a file of end points for each line in the input rail\_ln\_polyline.

7) Click OK to execute the tool. The result should take a few moments to process, but should result in 16990 features.

8) Using the PostGIS Shapefile and DBF Loader, load the resultant end point vertices shapefile in to PostGIS. Be sure to keep record of the name of the table created, if different to the name of the input shapefile.

### Create a table of end points (using PostgreSQL / PostGIS)

This following set of steps, replicates the procedure defined above for extracting end points from the railway line data, but instead uses functions available from the database rather than ArcGIS.

1) Open pgAdmin

2) Open a SQL query editor window, by clicking on .

3) Type the following SQL line in to the editor window (changing the name of the railway line data if you have given this a custom name when it was uploaded to the database)

--create a table of start points for all lines in railway line data

CREATE TABLE rail\_ln\_polyline\_start\_point\_vertices AS SELECT gid, code, identifier, “name”, ST\_StartPoint(geom) as geom FROM rail\_ln\_polyline;

--create a table of end points for all lines in railway line data

CREATE TABLE rail\_ln\_polyline\_end\_point\_vertices AS SELECT gid, code, identifier, “name”, ST\_EndPoint(geom) as geom. FROM rail\_ln\_polyline;

--create a table of all start and end points

CREATE TABLE rail\_ln\_polyline\_end\_vertices AS SELECT \* FROM rail\_ln\_polyline\_start\_point\_vertices UNION ALL SELECT \* FROM rail\_ln\_polyline\_end\_point\_vertices;

NOTE: it is possible to create a unique set of points by executing the following steps, prior to the section on ***“SQL Code to create table of intersections counts between rail\_ln\_polyline data and end points generated in previous step”***

--create a table of only unique points, based on geometry

CREATE TABLE rail\_ln\_polyline\_end\_vertices AS SELECT min(gid) as gid, min(code) as code, min(identifier) as identifier, min(“name”) as “name” FROM rail\_ln\_polyline\_end\_vertices\_all GROUP BY geom ORDER BY gid;

--add primary key constraint

ALTER TABLE rail\_ln\_polyline\_end\_vertices ADD CONSTRAINT rail\_ln\_polyline\_end\_vertices\_pkey PRIMARY KEY (gid);

--add dimension constraint

ALTER TABLE rail\_ln\_polyline\_end\_verticese ADD CONSTRAINT enforce\_dims\_geom CHECK (st\_ndims(geom) = 2);

--add geometry type constraint

ALTER TABLE rail\_ln\_polyline\_end\_vertices ADD CONSTRAINT enforce\_geotype\_geom CHECK (geometrytype(geom) = 'POINT'::text OR geom IS NULL);

--add srid constraint

ALTER TABLE rail\_ln\_polyline\_end\_vertices ADD CONSTRAINT enforce\_srid\_geom CHECK (st\_srid(geom) = 27700);

--drop and create spatial index

DROP INDEX IF EXISTS rail\_ln\_polyline\_end\_vertices \_gist;

CREATE INDEX rail\_ln\_polyline\_end\_vertices \_gist

ON rail\_ln\_polyline\_end\_vertices\_unique

USING gist

(geom);

### SQL Code to create table of intersection counts between rail\_ln\_polyline data and end points generated in previous step

DROP TABLE IF EXISTS rail\_ln\_end\_point\_vertices\_intersect\_counts;

--create a table of counts of intersections

CREATE TABLE rail\_ln\_end\_point\_vertices\_intersect\_counts AS SELECT rail\_ln\_polyline\_end\_points\_vertices.gid, COUNT(\*) as intersect\_count FROM rail\_ln\_polyline\_end\_points\_vertices, rail\_ln\_polyline WHERE ST\_Intersects(rail\_ln\_polyline.geom, rail\_ln\_polyline\_end\_points\_vertices.geom) GROUP BY (rail\_ln\_polyline\_end\_points\_vertices.gid) ORDER BY gid ASC;

DELETE FROM rail\_ln\_end\_point\_vertices\_intersect\_counts WHERE intersect\_count = 2;

**NOTE: This final step of removing all points where the intersection count against the original line data is only 2, works well for identifying junctions in the OS Meridian data because the original input line data has been digitised such that the line end points coincide with where either stations or junctions on the network would be located. This may not always be the case with other datasets.**

### Creating a table of junctions

DROP TABLE IF EXISTS rail\_ln\_junctions;

CREATE TABLE rail\_ln\_junctions AS SELECT rail\_ln\_polyline\_end\_points\_vertices.\*, rail\_ln\_end\_point\_vertices\_intersect\_counts.intersect\_count FROM rail\_ln\_polyline\_end\_points\_vertices, rail\_ln\_end\_point\_vertices\_intersect\_counts WHERE rail\_ln\_polyline\_end\_points\_vertices.gid = rail\_ln\_end\_point\_vertices\_intersect\_counts.gid;

DELETE FROM rail\_ln\_junctions USING station\_point WHERE ST\_Intersects(rail\_ln\_junctions.geom, station\_point.geom);

The final result should be a file of 3843 junctions.

NOTE: There may be duplicate junctions generated within this table where end points of lines are coincident with each other. If a unique set of line end points was not originally generated, then it is likely that there are duplicate end points in the resultant data and a unique set of junctions needs to be created from this data.

### Creating a table of unique junctions based on geometry

drop table if exists rail\_ln\_junctions\_unique;

create table rail\_ln\_junctions\_unique as select geom, min(gid) as gid, min(code) as code, min(identifier) as identifier, min("name") as "name", min(orig\_fid) as orig\_fid, min(intersect\_count) as intersect\_count from rail\_ln\_junctions GROUP BY geom ORDER BY gid;

ALTER TABLE rail\_ln\_junctions\_unique ADD CONSTRAINT rail\_ln\_junctions\_unique\_pkey PRIMARY KEY (gid);

ALTER TABLE rail\_ln\_junctions\_unique ADD CONSTRAINT enforce\_dims\_geom CHECK (st\_ndims(geom) = 2);

ALTER TABLE rail\_ln\_junctions\_unique ADD CONSTRAINT enforce\_geotype\_geom CHECK (geometrytype(geom) = 'POINT'::text OR geom IS NULL);

ALTER TABLE rail\_ln\_junctions\_unique ADD CONSTRAINT enforce\_srid\_geom CHECK (st\_srid(geom) = 27700);

DROP INDEX IF EXISTS rail\_ln\_junctions\_unique\_gist;

CREATE INDEX rail\_ln\_junctions\_unique\_gist

ON rail\_ln\_junctions\_unique

USING gist

(geom);

### Adding a type column to the unique junction table

ALTER TABLE rail\_ln\_junctions\_unique ADD COLUMN “type” varchar(50);

Once the “type” column has been added, values of ‘Junction’ need to be set so that when junctions are combined with the original input stations, it is possible to tell them apart.

UPDATE rail\_ln\_junctions\_unique SET “type” = ‘Junction’

### Combining junctions and original input stations

Drop table if exists rail\_ln\_junctions\_stations;

Create table rail\_ln\_junctions\_stations as select gid, code, identifier, “name”, “type”, geom from rail\_ln\_junctions

UNION ALL

Select gid, code, identifier, “name”, “type”, geom from station\_point;

ALTER TABLE rail\_ln\_junctions\_stations ADD CONSTRAINT rail\_ln\_junctions\_stations \_pkey PRIMARY KEY (gid);

ALTER TABLE rail\_ln\_junctions\_stations ADD CONSTRAINT enforce\_dims\_geom CHECK (st\_ndims(geom) = 2);

ALTER TABLE rail\_ln\_junctions\_stations ADD CONSTRAINT enforce\_geotype\_geom CHECK (geometrytype(geom) = 'POINT'::text OR geom IS NULL);

ALTER TABLE rail\_ln\_junctions\_stations ADD CONSTRAINT enforce\_srid\_geom CHECK (st\_srid(geom) = 27700);

DROP INDEX IF EXISTS rail\_ln\_junctions\_stations\_gist;

CREATE INDEX rail\_ln\_junctions\_stations\_gist

ON rail\_ln\_junctions\_stations

USING gist

(geom);

At this point a table of combined railway stations and junctions should be defined in rail\_ln\_junctions\_stations, and the original railway line defined in rail\_ln\_polyline. It is now possible to use these two layers to create a topological network of the rail network.

### Configuring db\_settings

Within the db\_settings.py file there are a number of database connection parameters defined, that help to maintain database connection strings used to connect to PostGIS database instances. A user will need to either create their own version of this file, or simply define their own parameters within this file, to denote the specific connection parameters for their database e.g. connecting to a local database

DB\_HOST = 'localhost'

DB\_PORT = 5432

DB\_USER = 'postgres'

DB\_PASSWORD = <INSERT PASSWORD>

DB\_NAME = ‘a\_database’

DB\_CONNECTION\_STRING = "host='"+DB\_HOST+"' dbname='"+ DB\_NAME +"' user='"+DB\_USER+"' password='"+DB\_PASSWORD+"'"

Furthermore, there is a dictionary of database connections defined within the default db\_settings.py file. The dictionary contains two main elements:

**‘connection\_settings’** – these define the basic connection parameters e.g. host, port, user, password

**‘database\_settings’** – these store the names and specific connection strings to specific databases i.e. contains lists of elements like TRANSPORT\_DB\_NAME and TRANSPORT\_DB\_CONNECTION\_STRING

The db\_settings.py file is utilised as part of the WS2DatabaseHandler.py file, which is used by the ws2\_database\_network\_handlers.py file to define database connection parameters for use when reading and writing network data.

### Configuring layer\_settings.py

Within the layer\_settings.py file there are dictionaries of layer names that reference layers stored in PostGIS and can be used as inputs to nx\_pg/nx\_pgnet for network creation. In general the dictionaries are split by the sector to which they relate e.g. transport, energy, and they are then further split by data source e.g. meridian, strategi etc, and then finally split by whether a layer denotes nodes or edges. It is also possible to define your own layer\_settings.py, or supply the edges and nodes layers as parameters to some of the build methods available in the ws2\_database\_network\_handlers.py files. Each network building class within the ws2\_database\_network\_handlers.py file utilises one of the specific dictionaries defined within the layer\_settings files. The usage is seen in Table.1:

|  |  |
| --- | --- |
| **Ws2\_database\_network\_handlers: object** | **Layer\_settings dictionary used** |
| WS2RoadNetwork | Transport\_layers |
| WS2RailNetwork |
| WS2AirNetwork |
|  |  |
| WS2GasNationalTransmissionNetwork | Energy\_layers |
| WS2GasRegionalDistributionNetwork |
| WS2ElectricityNationalTransmissionNetwork |
| WS2ElectricityRegionalDistributionNetwork |
|  |  |
| WS2SolidWasteNetwork | SolidWaste\_layers |
|  |  |
| WS2WaterNetwork | Water\_Layers  WasteWater\_Layers |
| WS2WasteWaterNetwork |

Table - layer setting usage by network handling objects

NOTE: If you wish to define your own dictionaries of input data within the layer\_settings.py file, then please do so, but note that you may need to alter the ws2\_database\_network\_handlers.py file to reflect those changes, or simply supply your own layer names as arguments to the build functions.

### Defining layers in layer settings

1. Define a Python variable with the same name as the chosen layer name in PostGIS for edges e.g. rail\_ln\_polyline, and assign it the same name as a string i.e. rail\_ln\_polyline = ‘rail\_ln\_polyline’

2. Define a Python variable with the same name as the chosen layer name in PostGIS for nodes e.g. rail\_ln\_junctions\_stations, and assign it the same name as a string i.e. rail\_ln\_junctions\_stations = ‘rail\_ln\_junctions\_stations’

3. Alter the transport\_layers->rail\_layers->meridian->edges dictionary to include the rail\_ln\_polyline variable e.g. defined in 1.

4. Alter the transport\_layers->rail\_layers->meridian->nodes dictionary to include the rail\_ln\_junctions\_stations variable e.g. defined in 2.

### Checking ws2\_database\_network\_handlers.py matches layer\_settings definitions

In order to build the national-scale rail network using the meridian data layers that you have created so far, the WS2RailNetwork class will be used. Checks need to be made that the layers defined previously within the layer\_settings files are being used within the respective build methods of the WS2RailNetwork class. Effectively this requires only one change to the “build\_meridian\_rail\_network\_nx\_pg” method:

self.meridian\_rail\_layers\_edges[‘'] change to self.meridian\_rail\_layers\_edges[rail\_ln\_polyline ']

self.meridian\_rail\_layers\_nodes[‘'] change to self.meridian\_rail\_layers\_nodes[‘rail\_ln\_junctions\_stations ']

This has told the method which layers to use from the PostGIS database as inputs to build the network.

### Executing the build procedure (Python):

from ws2\_database\_network\_handlers import WS2RailNetwork

import networkx as nx

ws2\_os\_meridian\_rail\_network = WS2RailNetwork('public', 'itrc\_test\_network\_interdependency', 'itrc\_test\_network\_interdependency')

network\_name = 'OSMeridian2\_Rail'

ws2\_os\_meridian\_rail\_network\_information = ws2\_os\_meridian\_rail\_network.build\_meridian\_rail\_network\_nx\_pg(network\_name, with\_node\_table=True)

ws2\_os\_meridian\_rail\_network.store\_meridian\_rail\_network('nx\_pgnet', ws2\_os\_meridian\_rail\_network.get\_meridian\_rail\_network\_nx\_pg(), network\_name, epsg=27700, overwrite=True)