OSM, QGis, PostGIS et SFCGAL en pratique

Comment, avec quatre outil Open Source, résoudre une question d'aménagement apparemment anodine… mais pas autant que ça !

L'offre *Open Source* en géomatique, qu'elle concerne les données (*Open data* ou *Open Street Map*) et les logiciels (*PostGIS*, *QGIS*…), a atteint un niveau de maturité qui permet désormais de résoudre des problèmes assez délicats en quelques manipulations, pourvu que l'on maîtrise correctement chacun des outils (ce qui n'est pas évident). Dans ce « *mini-tutoriel* », nous allons détailler comment résoudre une problématique simple, calculer la pente moyenne d'une voie ferrée, kilomètre par kilomètre, en n'utilisant que des ressources libres, à savoir la *BD Alti®* de l'IGN au pas de 75 m (disponible en téléchargement sur le site de l'Institut), le tracé issu de la base *Open Street Map*, et les logiciels *QGis 2.2* (dernière version publiée) ainsi que PostGIS 2.1 accompagné de l'extension *SFCGAL*[1].

Création de la base de données

La question que nous allons résoudre consiste à évaluer la pente moyenne, kilomètre par kilomètre, du tracé de l'extension de la ligne TGV est-européenne, longue d'une trentaine de kilomètres, et réalisée partiellement en tunnel sous le massif vosgien. Nous considérerons les parties sous tunnel comme étant horizontales (ce qui constitue une hypothèse à peu près raisonnable, du moins en première approximation).

La première étape consiste à constituer une base de données (nommons la, par exemple « *LGV* ») dans laquelle nous allons stocker nos deux couches initiales : la *BD Alti®* 75 m et le tracé de la ligne issue des données OSM.

La *BD Alti®* 75 m, comme précédemment signalé, est librement téléchargeable sur le site de l'IGN dans sa partie professionnelle. Elle est fournie sous la forme de dalles .ASC compressées, chacune d'environ 1 Mio. Le format est facilement interprétable : chaque nombre correspond à l'altitude d'une des cases de la grille dont les caractéristiques (taille du maillage et position d'une des cases d'extrémité) sont données dans l'en-tête :

```
ncols         1000
nrows         1000
xllcorner     974962.500000000000
yllcorner     6750037.500000000000
cellsize      75.000000000000
NODATA_value  -99999
```

Nous savons ainsi que le fichier élémentaire (ici, celui qui nous intéresse, baptisé **BDALTIr_2-0_MNT_EXT_0975_6900_LAMB93_IGN69_20110929.asc**) est composé d'une grille de mille points par mille points, que les coordonnés précisées sont celles du point sud-ouest (*ll : lower left*) et que la maille mesure, on le savait déjà, soixante-quinze mètres. Enfin la valeur prétexte, -99 999 (utilisée pour les points hors territoire), n'a pas d'importance ici puisque l'intégralité de la dalle est située en France.

La deuxième étape consiste à récupérer les données *Open*

[1]. *SFCGAL est une bibliothèque d'interface entre PostGIS et la bibliothèque de « géométrie computationnelle » CGAL développée en grande partie par l'INRIA. Cette bibliothèque enrichit PostGIS de nouvelles fonctions inédites, particulièrement dans le domaine de la 3D, mais pas seulement.*

Street Map. Théoriquement, ceci est directement faisable à l'intérieur de QGis en utilisant, dans le menu Vecteur, l'entrée *OpenStreetMap* ⇨ *Télécharger* des données. Malheureusement, la plupart du temps, cette option ne fonctionne pas (on obtient un message d'erreur). La façon la plus efficace, et la plus fiable, de télécharger ces données reste donc d'utiliser la partie cartographique du site *openstreetmap.fr*, qui renvoie elle-même vers *openstreetmap.org*. Là, sélectionner l'emprise d'intérêt sur le fond cartographique (ici, grossièrement, un rectangle englobant les villes de Metz, Nancy et Strasbourg) puis utiliser le lien sur l' « *API passerelle* », qui permet de rapatrier des quantités de données plus importantes que le site standard.

Le processus d'importation, depuis l'intégration de la fonctionnalité *OSM* dans le cœur de *QGis*, est assez complexe. Il faut utiliser d'abord la fonction *Vecteur* ⇨ *OpenStreetMap* ⇨ *Importer la topologie depuis un XML* pour créer, à partir des données *OSM* brutes (XML), un fichier compatible avec la base de données *SpatiaLite* qu'utilise *QGis* pour ses besoins propres. L'opération prend un certain temps, variable selon l'importance des données et la puissance de la machine qui exécute le script.

Après quelques minutes, on obtient un fichier **.db** en sortie qu'il va falloir maintenant exploiter pour obtenir des couches affichables. Pour cela, la troisième fonction *Exporter la topologie en SpatiaLite* permet de préciser le type d'objet que l'on souhaite extraire (points, lignes, polygones), et de sélectionner les attributs (*tags*) que l'on souhaite conserver (figure 3). Dans le cas qui nous intéresse, ce sont les polylignes (tracés routiers et ferrés) que nous allons extraire, en sélectionnant les attributs **name**, **railway** et **tunnel**. Le résultat est un maillage assez dense de polylignes (car même les tronçons sans attributs sont extraits) visible figure 4.

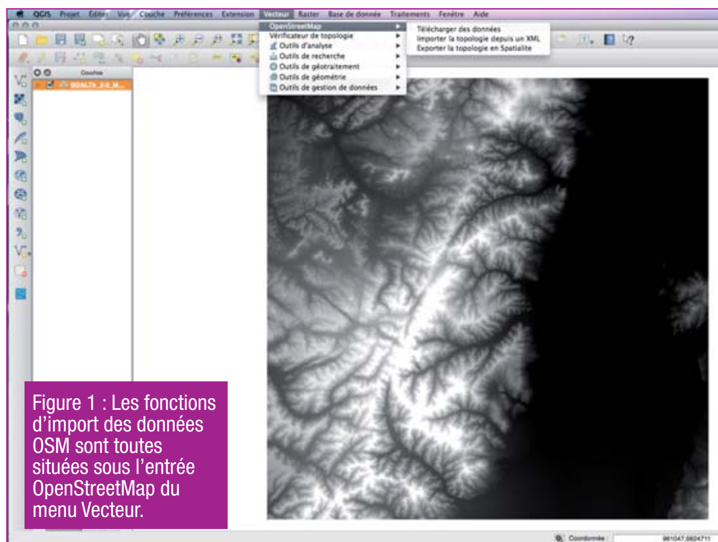


Figure 1 : Les fonctions d'import des données OSM sont toutes situées sous l'entrée OpenStreetMap du menu Vecteur.

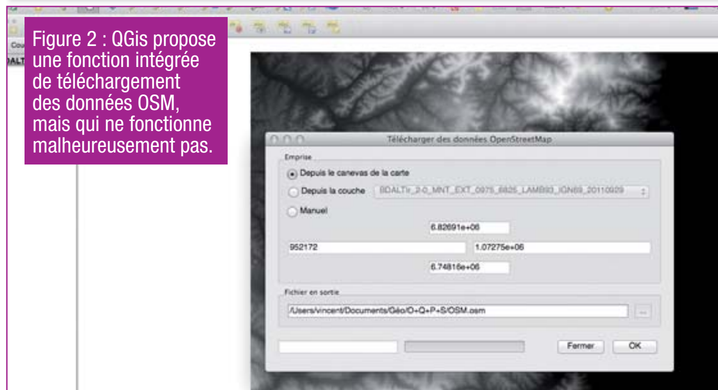


Figure 2 : QGis propose une fonction intégrée de téléchargement des données OSM, mais qui ne fonctionne malheureusement pas.

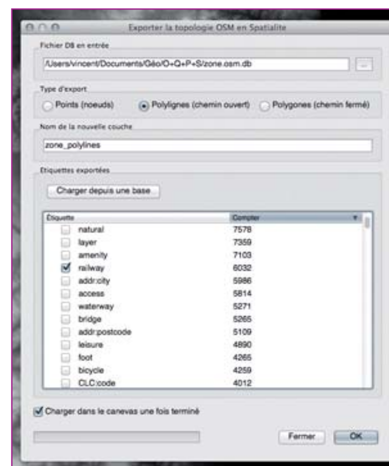


Figure 3 : Après la transformation des données XML en tables SpatiaLite, la troisième fonction Exporter la topologie en SpatiaLite permet de créer une couche géométrique et de sélectionner les champs à conserver.

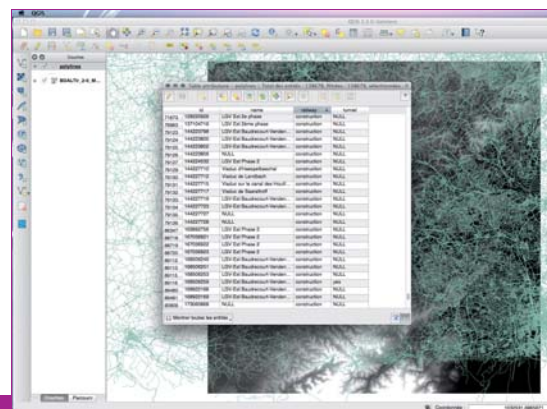
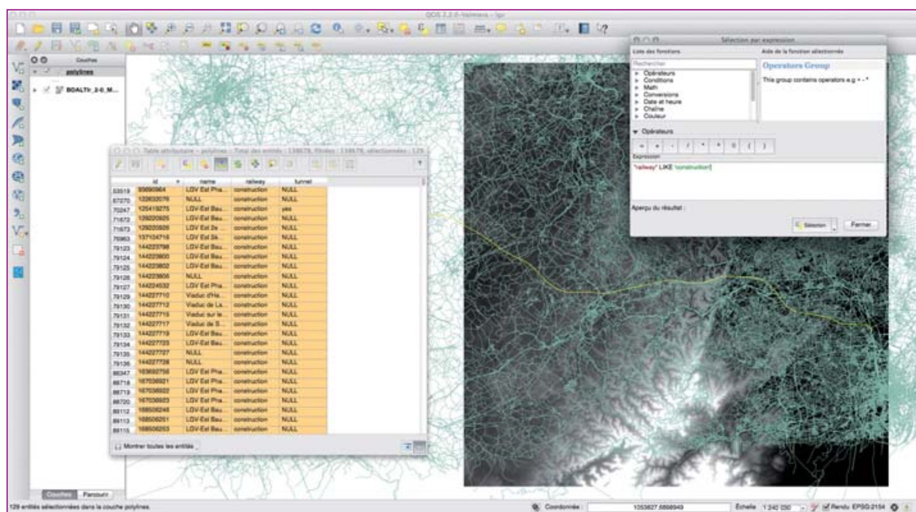
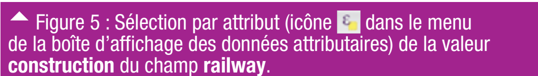


Figure 4 : Après importation, on se retrouve avec l'ensemble des données filaires. La consultation de la table des attributs permet de trouver la valeur des tags que l'on souhaite extraire.

Il nous faut maintenant extraire les données utiles dans ce réseau extrêmement dense. Comme on peut observer dans la table attributaire affichée figure 4, il semble que la ligne TGV soit, dans le champ **railway**, affublée de la valeur « *construction* ». Nous

allons donc générer une nouvelle couche en sélectionnant par attribut les tronçons portant le tag **railway:construction**. Le résultat correspond bien au futur tracé de la ligne nouvelle (figure 5).





▲ Figure 5 : Sélection par attribut (icône dans le menu de la boîte d'affichage des données attributaires) de la valeur **construction** du champ **railway**.

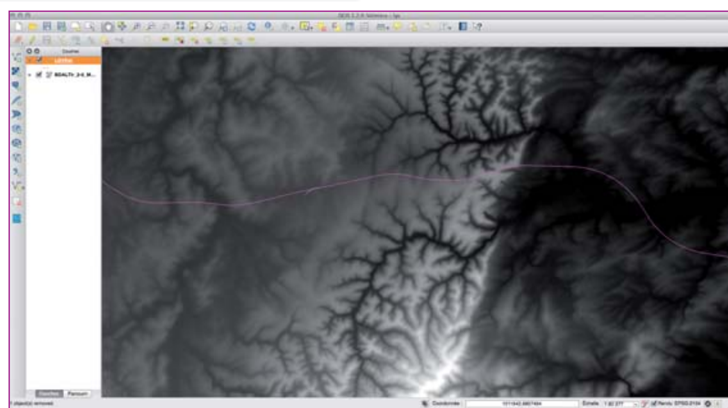
Nous créons donc une nouvelle couche sous forme de fichier *Shapefile* en utilisant l'option *Sauvegarder la sélection sous*. Nous la rechargeons, puis nous utilisons l'outil de sélection rectangulaire pour limiter le tracé à la seule portion où nous disposons des données d'altimétrie, puis nous sauvegardons de nouveau la sélection sous un nouveau fichier *Shapefile* (il est également possible de rassembler les deux sélections en une seule pour aller plus vite).

Quand on examine le tracé de près, on s'aperçoit qu'il est en fait constitué des deux voies dessinées l'une à côté de l'autre. Ceci va nous poser un souci, car il va falloir gérer une géométrie de type *multi-* alors que nous souhaiterions obtenir un tracé « *simple* ». En outre, la couche comprend également deux raccordements d'accès que nous ne souhaitons pas prendre en compte. Il va donc falloir les éliminer.

À ce stade, il est expédient de constituer un base de données *PostGIS* et d'y injecter le tracé et le MNT (grâce aux fonctions *raster*). On ne peut apparemment pas créer directement une nouvelle base de données *PostGIS* à l'intérieur de *QGis*, même en utilisant le gestionnaire de bases de données. Il faut donc le faire, soit au travers de l'utilitaire d'administration *PGAdmin III*, soit manuellement en ligne de commande en tapant **createdb <nom_de_la_base>**, puis en injectant les différentes bases nécessaires au bon fonctionnement de la cartouche spatiale (ne pas oublier, dans ce cas, ni le module **rtpostgis** de gestion des *raster*, ni le module **sfcgal**.) Ici, nous choisirons *LGV* comme nom de base. Pour pouvoir accéder à celle-ci à l'intérieur du gestionnaire de bases de données *QGis*, il faut auparavant avoir défini les paramètres de connexion en cliquant sur l'icône de chargement d'un couche *PostGIS*, ce qui ouvre le dialogue approprié.

Une fois la connexion créée et vérifiée, on peut donc retourner dans le gestionnaire de base de données, sélectionner la nouvelle base et injecter directement le tracé grâce à la fonction « *importer une couche ou un fichier* ». En revanche, le *raster* ne peut pas être stocké dans *PostGIS* depuis *QGis*, il faut donc utiliser l'utilitaire **raster2pgsql**, fourni avec *PostGIS*. Par exemple, sous *Unix/Linux/OS X* :

▼ Figure 6 : Après sélection par attribut puis par rectangle englobant, on obtient l'extrait du tracé de la ligne LGV dans la zone d'intérêt.



```
raster2pgsql -s 2154 -b 1 -c
-f geom -I -C BDALTIr_2-0_
MNT_EXT_0975_6900_LAMB93_
IGN69_20110929.asc mnt | psql
-d LGV
```

Il est conseillé d'utiliser un *alias* dans la commande **raster2pgsql** (le « *mnt* » juste avant le |) de façon à éviter de créer une table portant le même nom que le fichier, ce qui ne serait pas du tout commode par la suite.

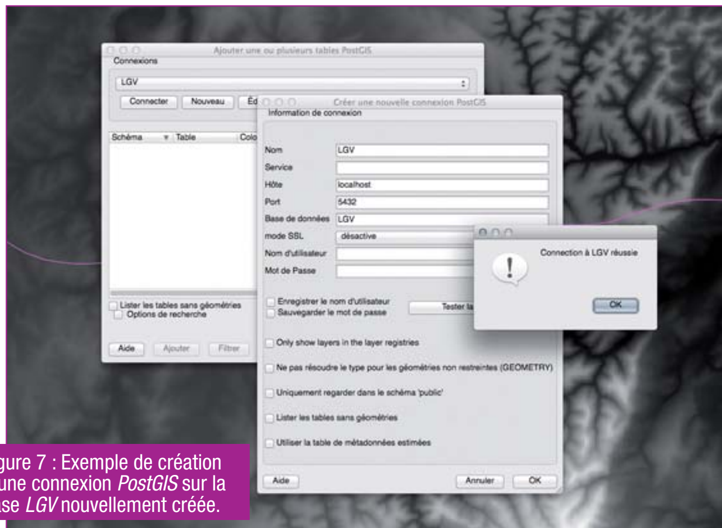


Figure 7 : Exemple de création d'une connexion *PostGIS* sur la base *LGV* nouvellement créée.

Traitement de la donnée de voie

La donnée issue d'*OSM* nécessite encore quelques pré-traitements avant d'être utilisable. Tout d'abord, il faut éliminer les tronçons partiellement hors-MNT, puis les tronçons de raccordement qui ne nous intéressent pas. Pour cela, le plus aisé est d'utiliser QGis pour éditer la couche, de sélectionner manuellement les tronçons à ôter, puis de sauvegarder le résultat (figure 8).

Il faut ensuite « *fusionner* » les deux voies parallèles pour ne conserver que l'axe du tracé. Cette opération est un peu délicate et nécessite du doigté. Pour commencer, il faut convertir les deux voies en une sorte de « *boudin* » polygonal, en créant des buffers autour de chacun des axes, puis en les fusionnant. Cette opération est réalisable à l'aide des outils *Python* fournis en standard avec *QGIS* (menu *Vecteur* ⇨ *Outils de géotraitement*), mais il vaut mieux la conduire directement dans *PostGIS*.

```
CREATE TABLE boudin AS (SELECT
ST_Union(ST_Buffer (geom, 200))
AS geom FROM tracé);
```

Le choix de deux cents mètres comme largeur du buffer est arbitraire, l'idée étant de retenir une valeur suffisante pour couvrir tous les écartements rencontrés. Le résultat est visible figure 9.

Il faut maintenant trouver le moyen de recréer, à partir de ce corridor, un axe central. Pour cela, la nouvelle fonction **ST_StraightSkeleton** issue du module *SFCGAL* va nous aider. Son rôle est, en partant d'un polygone, d'en calculer la « *squelettisation* », c'est-à-dire les axes rigides d'orientation. Le calcul met en jeu la bibliothèque *CGAL* et prend un certain temps (quelques minutes). Le résultat se présente sous la forme de l'axe central du corridor, et de multiples lignes qui connectent perpendiculairement cet axe aux limites du corridor aux endroits où celui-ci change de direction (figure 10).

```
CREATE TABLE squelette AS
(SELECT ST_StraightSkeleton
(geom) AS geom FROM boudin);
```

Il faut donc se débarrasser de ces segments perpendiculaires. Pour ce faire, nous allons dessiner un *buffer* autour de chaque segment. Seul celui qui entoure l'axe central sera entièrement contenu dans le boudin que nous avons créé auparavant. Il suffit donc d'éliminer les segments dont le *buffer* n'est

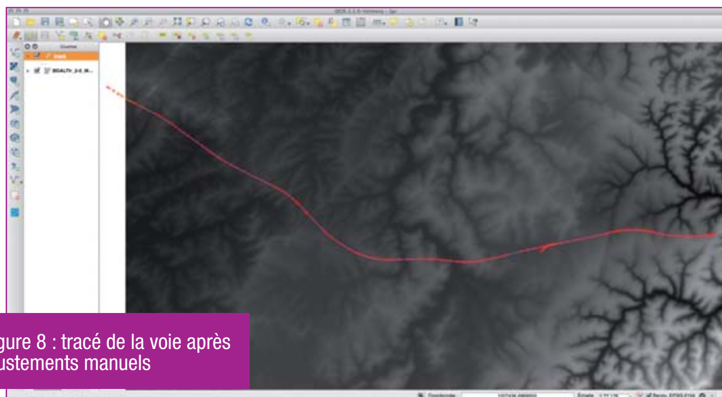


Figure 8 : tracé de la voie après ajustements manuels

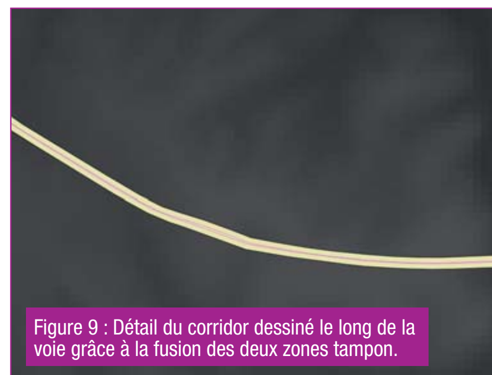


Figure 9 : Détail du corridor dessiné le long de la voie grâce à la fusion des deux zones tampon.

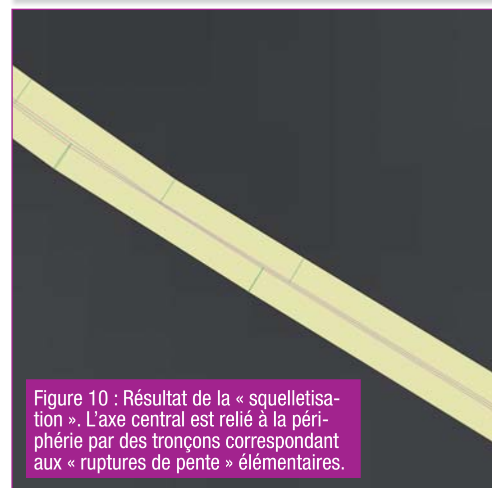


Figure 10 : Résultat de la « squelletisation ». L'axe central est relié à la périphérie par des tronçons correspondant aux « ruptures de pente » élémentaires.

pas entièrement contenu dans le boudin, de regrouper tous les segments restants grâce à la fonction **ST_Union**. Bémol préliminaire toutefois : le résultat de la requête précédente, donc la table **squelette**, est fourni sous forme d'une géométrie *Multi-*. Il faut donc préalablement la découper en tronçons élémentaires, ce que fait la fonction **ST_Dump ()**.

```
CREATE TABLE axe AS (WITH sqlt
AS (SELECT (ST_Dump (geom)).
geom AS geom FROM squelette)
SELECT s.geom AS geom FROM
sqlt AS s, boudin AS b WHERE
ST_Contains (b.geom, ST_Buffer
(s.geom, 30)));
```

La clause **WITH** qui précède la vraie requête de sélection réalise le découpage en tronçons, tronçons qui sont ensuite triés individuellement selon la méthode précédemment décrite. La valeur du *buffer*, 30, est arbitraire : suffisante pour éviter les éventuelles ambiguïtés géométriques, mais évidemment inférieure à la largeur du corridor (ici 200). Après cette requête, on se retrouve dans la situation de la figure 11.

Figure 11 : Après élimination des segments « latéraux », il ne reste plus que l'axe du corridor.

L'extrémité ouest doit faire l'objet d'un traitement particulier. On se trouve avec un phénomène « *de fourche* » dû au fait que les deux voies ne se terminent pas strictement au même niveau. On élimine cette fourche à la main avec les outils d'édition de *PostGIS* déjà évoqués.

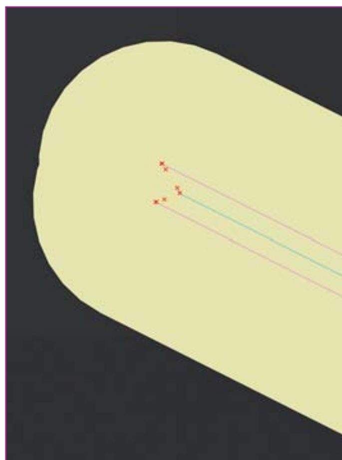


Figure 12 : Élimination de la « fourche » en bout de ligne.

Une fois ce petit nettoyage accompli, il faut regrouper tous les segments en une ligne unique. Pour cela, on va les réunir dans une multi-géométrie grâce à la fonction **ST_Union**, puis on va « *recoudre* » tous les segments de cette multi-géométrie en une seule ligne grâce à la fonction **ST_LineMerge**.

```
CREATE TABLE ligne AS (SELECT
ST_LineMerge (ST_Union (geom))
AS geom FROM axe);
```

Il ne reste maintenant qu'une opération à réaliser, mais de taille. Il s'agit de découper ce nouveau tracé unique et « *médian* » en tronçons d'un kilomètre de longueur (PK). Il n'existe pas, à l'heure actuelle, de fonction *PostGIS* qui permette de réaliser cette opération (**ST_Cut** ? **ST_Carve** ?) En revanche, il est possible de réaliser une « extraction » entre tel et tel PK en utilisant la fonction **ST_Line_Substring (pk1, pk2)** où les PK doivent être exprimés en fraction de la longueur totale (0,5 correspond, par exemple, au point situé à mi-chemin des extrémités).

Il n'y a pas donc d'autre solution que d'écrire un script en langage **pl/pgsql** pour réaliser ce découpage. Ce dernier, détaillé dans le cadre ci-après, prend comme paramètre la table contenant la ligne à segmenter, et rend en sortie une nouvelle table baptisée du même nom affublé du suffixe **_tronçons**, constitué d'une série de tronçons élémentaires, chacun d'un kilomètre de long. Moyennant des améliorations cosmétiques, il est possible de transmettre la longueur du tronçon élémentaire en paramètre et de se constituer ainsi une fonction très générique de segmentation linéaire.

```
CREATE OR REPLACE FUNCTION découpe
(voie VARCHAR) RETURNS VOID AS $$

DECLARE
  pk INTEGER := 0;
  pkf DOUBLE PRECISION;
  long DOUBLE PRECISION;
  voie_t VARCHAR := voie ||
  '_tronçons';
  tmp RECORD;

BEGIN
  EXECUTE 'SELECT CAST ((ST_Length
  (geom) + 1) / 1000 AS INTEGER)
  FROM '|| voie INTO long;
  RAISE NOTICE 'Longueur du tracé :
  %', long;

  EXECUTE 'SELECT true FROM pg_
  tables WHERE tablename = ''' ||
  voie_t || '''' INTO tmp;

  IF (tmp.bool) THEN
    EXECUTE 'DROP TABLE "'
    || voie_t || '"';
  END IF;

  EXECUTE 'CREATE TABLE "'
  || voie_t || '" (' ||
    'pk       INTEGER,' ||
    'pente    DOUBLE PRECISION,' ||
    'geom     GEOMETRY)';

  FOR pk IN 0..long-1 LOOP
    RAISE NOTICE 'Calcul du
    tronçon %', pk;
    pkf := CAST (pk AS DOUBLE
    PRECISION);
    EXECUTE 'INSERT INTO "' ||
    voie_t || '" SELECT '
    || pk || ', 0, ' ||
```

```
      '(SELECT ST_LineSubstring
      (geom, ' || pkf / long
      || ', ' ||
      (pkf + 1) / long || ')
      FROM "' || voie || '")';

   END LOOP;

   PERFORM Populate_geometry_
   columns ();

END;
$$ LANGUAGE plpgsql;
```

Cette fonction n'est pas aisée à comprendre, en raison de l'utilisation systématique de clauses **EXECUTE** qui permettent d'exécuter un texte comme s'il s'agissait d'une requête SQL directe. Cette syntaxe peu explicite est néanmoins nécessaire à partir du moment où l'on souhaite réaliser une fonction flexible, qui puisse travailler sur une table au nom quelconque.

Le bloc **DECLARE** sert à déclarer les variables qui seront utilisées par la suite.

Les premières lignes du bloc **BEGIN** calculent et affichent la longueur de la ligne stockée dans la table dont le nom est passé en paramètre, et l'affectent à la variable **long**.

La requête suivante regarde si la table suffixée par **_tronçons** existe déjà[2]. Si oui (dans ce cas, la variable **tmp.bool** vaut **TRUE**), on l'efface. Puis, dans tous les cas, on la crée (ou on la recrée), en la structurant en trois champs : le PK, la pente et la géométrie du tronçon kilométrique.

Enfin, la boucle **FOR** réalise l'opération de découpage. La variable **pk** prend toutes les valeurs entières entre 0 et la longueur de la ligne moins un (**long-1**), et pour chacune d'entre-elles on insère dans la table **_tronçons** la valeur courante du **pk**, **0** – valeur conventionnelle pour la pente – et enfin la géométrie correspondant à la section de ligne comprise entre les PK **pk** et **pk+1**.

Association finale

Il ne reste plus qu'une opération à réaliser : associer à chaque tronçon la valeur moyenne de la pente correspondante. Pour cela, il faut en premier… calculer la pente ! Pour cela, nous disposons de la fonction **ST_Slope** qui fait partie des fonctions offertes par *PGRaster*.

```
CREATE TABLE pentes AS (SELECT
ST_Slope (geom, 1, '32BF',
'PERCENT', 1.0, FALSE) AS pente
FROM mnt);
```

Les différents paramètres passés à la fonction **ST_Slope**, à part la colonne géométrique, correspondent à la bande (il n'y en a qu'une dans notre cas), le type de codage utilisé pour représenter l'altitude (ici : réels simple précision), le type d'unité souhaité (degrés, radians ou ici pourcentage), un facteur d'échelle et enfin un booléen qui indique s'il faut interpoler là où les données manquent.

Il serait souhaitable de pouvoir réaliser le calcul de la pente moyenne en récupérant directement la valeur des cases situées « *en-dessous* » de la voie. Cela n'est malheureusement pas encore possible dans cette version de *PostGIS*. Il faut donc « *contourner* » cette impossibilité en « *vectorisant* » la couche *raster* grâce à la fonction **ST_DumpAsPolygon**.

```
WITH moyennes AS (WITH zone
AS (SELECT ST_DumpAsPolygons
(pente, 1) AS mixed FROM pentes)
SELECT avg ((zone.mixed).val),
pk FROM zone, ligne_tronçons
WHERE ST_Intersects ((zone.
mixed).geom, ligne_tronçons.
geom) GROUP BY pk) UPDATE
ligne_tronçons SET pente =
moyennes.avg FROM moyennes
WHERE moyennes.pk = ligne_tron-
çons.pk;
```

Cette requête, passablement complexe, est composée en réalité de deux sous-requêtes de type **WITH** et d'une requête principale de type **UPDATE**. La plus imbriquée (**WITH zone AS (SELECT… FROM pentes)**) crée une table appelée **zone** contenant la vectorisation de la couche *raster* **pentes**. Celle-ci se compose d'une colonne de type mixte (structure) composée d'un champ **geom** contenant la géométrie et d'un champ **val** contenant la valeur du pixel (donc ici la pente). Autrement dit, dans la table **zone**, on trouve, pour chaque valeur de la pente possible, un (multi-)polygone correspondant à la réunion de tous les pavés de la grille ayant cette valeur.

Cette table est utilisée dans la seconde prérequête **WITH**. Celle-ci isole toutes les intersections entre les différents tronçons et les (multi-)polygones d'isopente, et, pour chaque intersection, note la valeur de pente et le PK correspondants, avant de moyenner les valeurs de pente par PK (**avg ((zone. mixed). val) … GROUP BY pk;**), puis de stocker les résultats dans la table **moyennes**.

Juste avant la requête finale, on se trouve donc avec une table **moyenne** contenant deux

2. La suite de quatre apostrophes '''', un peu surprenante, correspond à un caractère ' lui-même entre apostrophes. La norme SQL précise que si l'on souhaite placer une apostrophe dans une chaîne de caractères, qui doit être elle-même délimitée par des apostrophes, alors il faut redoubler l'apostrophe médiane. '''' équivaut donc à ' " '.

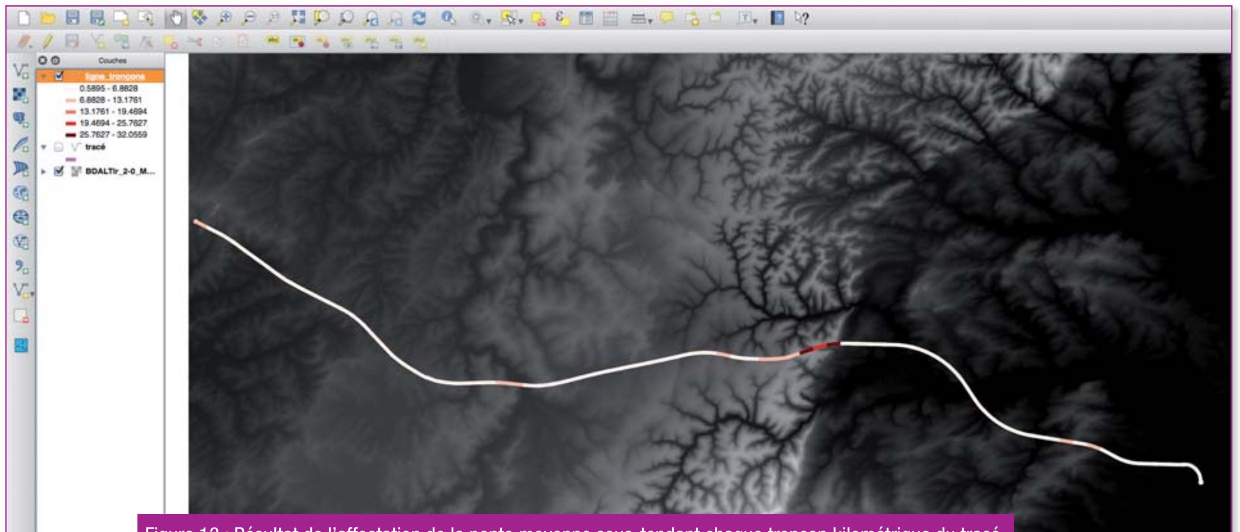


Figure 13 : Résultat de l'affectation de la pente moyenne sous-tendant chaque tronçon kilométrique du tracé.

champs, l'un donnant le PK, et l'autre la pente moyenne à ce PK. Il ne reste plus qu'à mettre à jour l'information de pente stockée dans la table **ligne_tronçons** précédemment créée en effectuant une jointure par l'intermédiaire du PK. Le résultat apparaît figure 13.

Nouveau bémol toutefois : notre couche ne prend pas en compte les passages sous les tunnels. Cette information est contenue dans la table originale (tag *tunnel* à la valeur *yes*). Pour effectuer la correction et forcer la pente à zéro sous les tunnels, il suffit de repérer les tronçons qui intersectent les parties désignées comme tunnel, moyennant le fait que notre axe se trouve légèrement décalé par rapport au tracé original (bi-voie).

```
UPDATE ligne_tronçons SET pente
= 0 FROM “LGV” WHERE ST_
Intersects (ST_Buffer (ligne_
tronçons.geom, 30), “LGV”.geom)
AND “LGV”.tunnel ILIKE ‘yes’;
```

La figure 14 montre le résultat final après reclassification du tronçon.

Conclusion

Nous avons donc parcouru rapidement quelques unes des possibilités offertes par l'utilisation conjointe des logiciels libres que sont *QGis* et *PostGIS* avec les différentes bibliothèques sur lesquelles ils s'appuient : *GDAL*, *SFCGAL* et *CGAL*. Dans un prochain opus, nous nous intéresserons aux problématiques 3D et à l'installation du *plug-in QGis* baptisé *Horao*, également développé par *OSLandia*. ❙

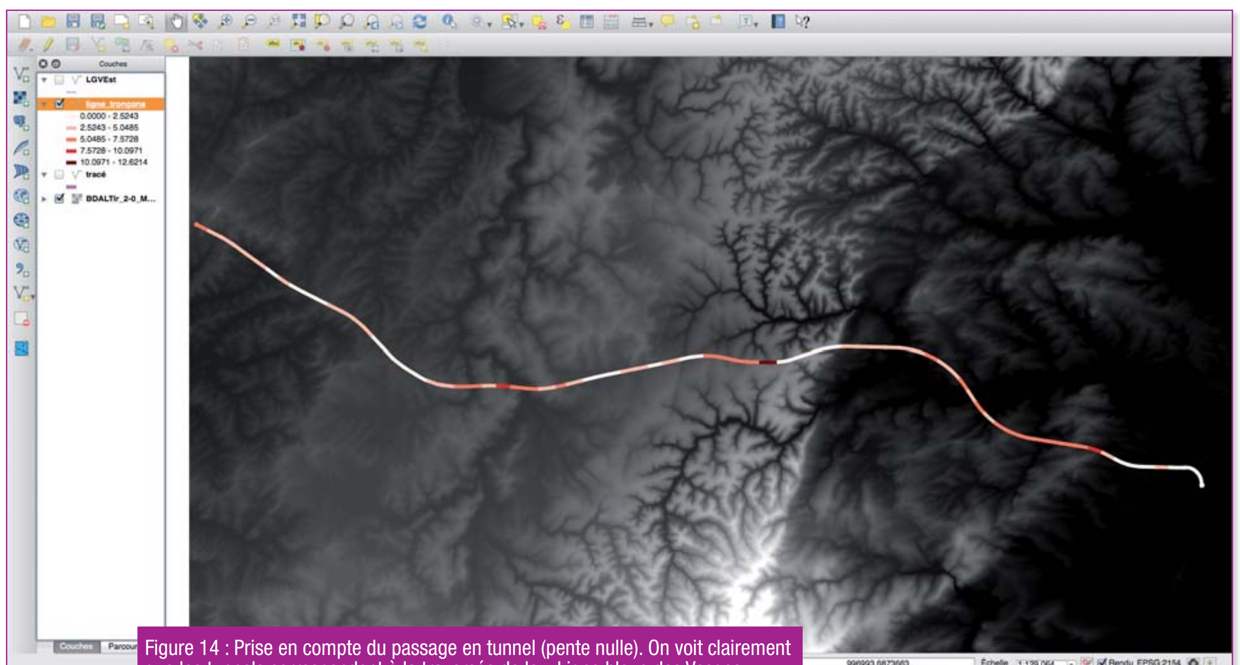


Figure 14 : Prise en compte du passage en tunnel (pente nulle). On voit clairement que les tunnels correspondent à la traversée de la « Ligne bleue des Vosges ».