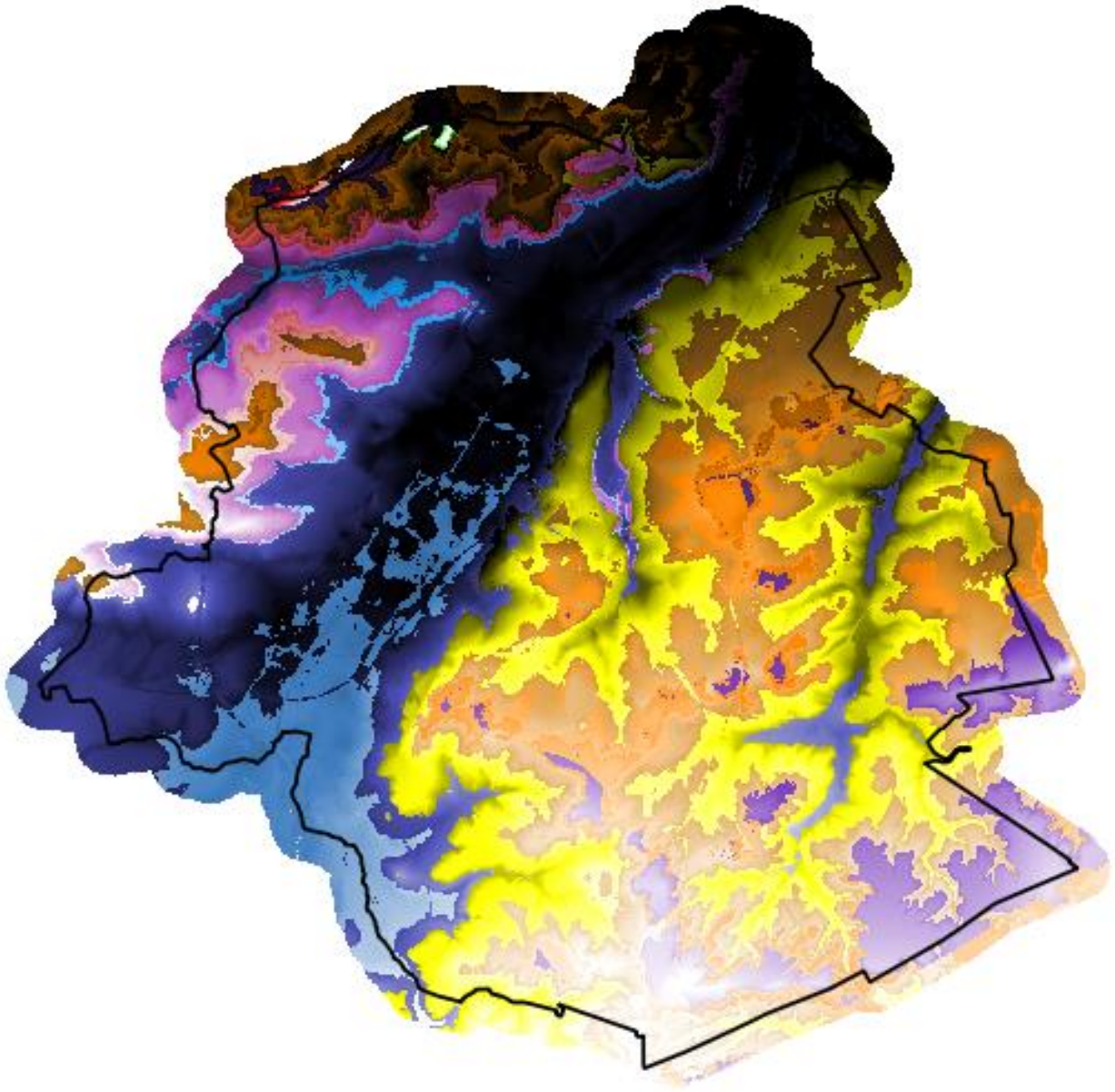
BRUSTRATI3D VERSION 1.1 : HARMONISATION ET CORRECTIONS APPORTEES AUX RASTERS DES TOITS DES UNITES STRATIGRAPHIQUES DU MODELE BRUSTRATI3D VERSION 1.0 ET CALCUL DES RASTERS D'EPAISSEURS

ADDENDUM AU RAPPORT BRUSTRATI3D VERSION 1.0 (SERVICE GEOLOGIQUE DE BELGIQUE, X. DEVLEESCHOUWER, C. GOFFIN, J. VANDAELE & B. MEYVIS, 2017)



OCTOBRE 2018

BRUSTRATI3D VERSION 1.1 : HARMONISATION ET CORRECTIONS APPORTEES AUX RASTERS DES TOITS DES UNITES STRATIGRAPHIQUES DU MODELE BRUSTRATI3D VERSION 1.0 ET CALCUL DES RASTERS D'EPAISSEURS

ADDENDUM AU RAPPORT BRUSTRATI3D VERSION 1.0 (SERVICE GEOLOGIQUE DE BELGIQUE, X. DEVLEESCHOUWER, C. GOFFIN, J. VANDAELE & B. MEYVIS, 2017)

SOMMAIRE

ILLUSTRATIONS

TABLEAUX

ABREVIATIONS ET ACRONYMES

BE : Bruxelles Environnement

MNT : Modèle Numérique de Terrain

RBC : Région Bruxelles Capitale

RBC+500m : Contour de la RBC agrandi d'une zone tampon de 500m

SCR : Système de coordonnées de référence

SGB : Service Géologique Belge

SIG : Système d'Informations Géographiques

UH/RBC : Unité Hydrogéologique de la Région Bruxelles-Capitale

US/RBC : Unité Stratigraphique de la Région de Bruxelles-Capitale

WFS : Web Feature Service

INTRODUCTION

Le modèle BruStrati3D version (v) 1.0 est un modèle géologique couvrant la RBC construit en 2017 par le SGB. Celui-ci est composé de rasters d'élévation des toits des unités stratigraphiques de la région de Bruxelles-Capitale (US/RBC) et de la base de l'UH/RBC_01 Système aquifère du Quaternaire.

Dans la perspective de sa diffusion via téléchargement libre, webgis ou webservices, BE a apporté quelques modifications et corrections à ce modèle. Ce document traite des modifications apportées. Il s'agit notamment :

1. D'une redéfinition de la nomenclature[1] utilisée pour désigner les US/RBC ;

2. D'une harmonisation des rasters du modèle (formatage, reprojection, « snapping », harmonisation de l'extension spatiale des données) ;

3. D'une correction apportée à certains pixels incohérents (générant localement des épaisseurs de couches négatives) sur les rasters des toits des US/RBC ;

4. Du calcul des rasters d'épaisseur des US/RBC et de l'UH/RBC_01 Système aquifère du Quaternaire ;

5. De la redéfinition de l'extension spatiale des US/RBC sur la base des épaisseurs calculées (suppression des pixels correspondant à une épaisseur nulle dans les rasters de toits).

Ces modifications ont principalement été effectuées à l'aide de scripts codés en Python, qui présentent l'avantage d'automatiser la procédure de traitement des données et de pouvoir être réutilisés en cas d'évolution du modèle à la faveur de l'enrichissement de la base de données des forages/sondages utilisée pour l'interpolation des couches.

L'ensemble de ces modifications a été effectué uniquement sur les rasters des toits des couches et sur le raster de la base du Quaternaire. Les rasters des bases des unités n'ont pas été traités dans le cadre de ce travail. Les rasters issus de ces modifications forment le modèle BruStrati3D v1.1.

[1] Pour les Unités Stratigraphiques de la Région de Bruxelles-Capitale (US/RBC), la modification de la nomenclature a uniquement impacté les codes utilisés pour les désigner. Ainsi US/RBC_91 Argiles de Hannut (Membre de Lincent) est devenue **US/RBC_82 Argiles de Hannut (Membre de Lincent)**, US/RBC_101 Craies de Gulpen est devenue **US/RBC_91 Craies de Gulpen** et US/RBC_102 Socle Paléozoïque est devenue **US/RBC_92 Socle Paléozoïque**. La nomenclature utilisée pour désigner les unités stratigraphiques peut être consultée dans le « Tableau des Unités Stratigraphiques de la Région de Bruxelles-Capitale (US/RBC) » téléchargeable depuis la page suivante :

https://environnement.brussels/thematiques/geologie-et-hydrogeologie/geologie

A noter que ces modifications se sont inscrites dans une redéfinition plus globale de la nomenclature et de la discrétisation des Unités Hydrogéologiques de la Région de Bruxelles-Capitale (UH/RBC). Celles-ci ne font pas l'objet de ce rapport. Plus d'information à ce sujet est disponible à la page suivante :

https://environnement.brussels/thematiques/geologie-et-hydrogeologie/eaux-souterraines/hydrogeologie/unites-hydrogeologiques-de-la

I. DONNEES DE DEPART

I.1. RASTERS DE TOITS DU MODELE BRUSTRATI3D V1.0

Le modèle géologique BruStrati3D v1.0 (Figure 1) est constitué de 18 rasters d'élévation des toits des différentes unités stratigraphiques du sous-sol bruxellois ainsi que du raster d'élévation de la base de l'UH/RBC_01 Système aquifère du Quaternaire (Tableau 1).

Tableau 1 : unités stratigraphiques et hydrogéologiques pour lesquelles un raster de toit et/ou de base est modélisé dans BruStrati3D v1.0 (d'après Devleeschouwer et al, 2017)

Ere	Formation	Unité
IV (Cénozoïque)	-	UH/RBC_01 Système aquifère du Quaternaire *(base uniquement)*
III (Cénozoïque)	Diest	US/RBC_21 Sables de Diest *(toit uniquement)*
	Bolderberg	US/RBC_22 Sables de Bolderberg
	Sint-Huilbrechts-Hern	US/RBC_23 Sables et argiles de Sint-Huilbrechts-Hern
	Maldegem (Onderdale)	US/RBC_25 Sables de Maldegem (membre de Onderdale)
	Maldegem (Ursel et Asse)	US/RBC_31 Argiles de Maldegem (membre de Ursel et Asse)
	Maldegem (Wemmel)	US/RBC_41 Sables de Maldegem (membre de Wemmel)
	Lede	US/RBC_42 Sables de Lede
	Bruxelles	US/RBC_43 Sables de Bruxelles
	Gent (Vlierzele)	US/RBC_44 Sables de Gent (membre de Vlierzele)
	Gent (Merelbeke)	US/RBC_51 Argiles de Gent (membre de Merelbeke)
	Tielt	US/RBC_61 Sables et argiles de Tielt
	Kortrijk (Aalbeke)	US/RBC_71 Argiles de Kortrijk (membre d'Aalbeke)
	Kortrijk (Moen)	US/RBC_72 Sables et argiles de Kortrijk (membre de Moen)
	Kortrijk (Saint-Maur)	US/RBC_73 Argiles de Kortrijk (membre de Saint Maur)
	Hannut (Grandglise)	US/RBC_81 Sables de Hannut (Membre de Grandglise)
	Hannut (Lincent)	US/RBC_82 Argiles de Hannut (Membre de Lincent)
II (Mesozoïque)	Gulpen	US/RBC_91 Craies de Gulpen
I (Paléozoïque)	Tubize	US/RBC_92 Socle Paléozoïque *(uniquement le toit)*

Ces rasters ont été obtenus via l'interpolation de données de 9266 ouvrages verticaux situés dans ou à proximité de la RBC. Le détail de la construction du modèle BruStrati3D v1.0 est fourni dans le rapport Devleeschouwer et al (2017).

Dans Brustrati3D v1.0, les bases des unités stratigraphiques ont aussi été modélisées, mais ne sont pas reprises dans la version 1.1. En effet, l'ensemble des rasters de toits suffisent à définir la stratigraphie du sous-sol bruxellois et ces derniers ont de plus grandes probabilités d'être précis, ayant été modélisés à partir de plus de points que les bases (Devleeschouwer et al 2017).

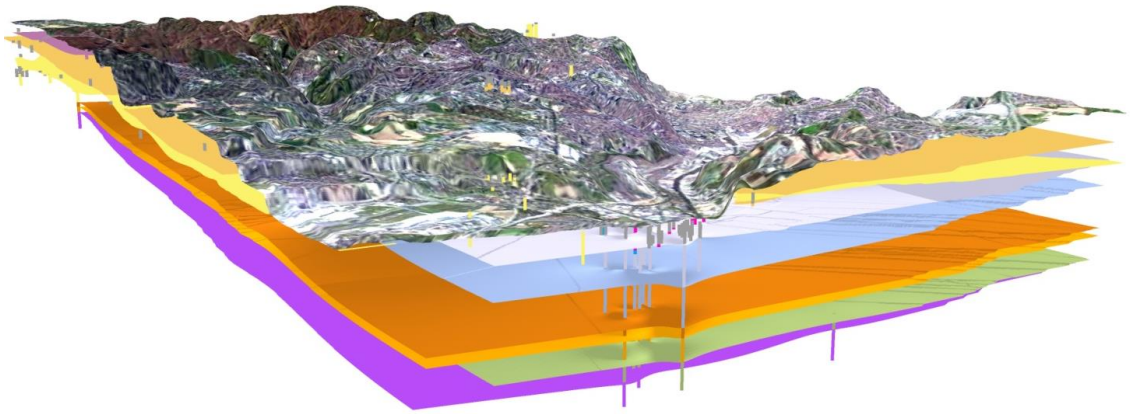


Figure 1 : visualisation sur ArcScene du modèle Brustrati3D (Auteur : SGB). Les couches de différentes couleurs représentent les toits des couches géologiques, modélisés par des rasters (Devleeschouwer et al, 2017).

I.2. TOPOGRAPHIE

Le MNT utilisé dans le cadre de ce travail est celui qui a été utilisé pour la construction du modèle BruStrati3D v1.0 (**Lidar10_SRTM80.adf**). Il s'agit d'un MNT de résolution 10x10m issu de l'agrégation de :

- un MNT issu de données lidar d'Urbis (CIRB, 2015), initialement à une résolution de 1x1m, mais ayant subi un rééchantillonnage afin d'être à une résolution de 10x10m ;

- un MNT issu des données SRTM (Shuttle Radar Topography Mission), initialement d'une résolution de 80x80m et rééchantillonné à une résolution de 10x10m.

Pour plus de détails sur la construction de ce MNT, consulter Devleeschouwer et al (2017) ainsi que les spécifications techniques des données Urbis (CIRB, 2015) :

https://cirb.brussels/fr/nos-solutions/urbis-solutions/fichiers/specifications-techniques-urbis-dtm.pdf

L'utilisation du même raster de topographie que celui ayant servi à construire le modèle géologique était indispensable dans le cadre de ce travail, car l'utilisation d'un autre MNT, même très proche en termes de valeurs d'élévations topographiques aurait pu générer localement des incohérences, telles que des zones où les toits de couches géologiques se seraient situés au-dessus du niveau topographique.

I.3. GRILLE DE REFERENCE

Par ailleurs, afin d'aligner l'ensemble des rasters sur une grille commune, le raster de topographie **UrbDTM_GRID_Y12_RBC_1m.tif** fourni par Urbis a aussi été utilisé. Celui-ci n'a pas été utilisé pour ses valeurs d'élévation topographique mais uniquement pour servir de « grille de référence ». Il présente l'avantage d'être un raster couvrant l'ensemble de la RBC, en projection Belgian Lambert 72, de dimensions carrées et dont les positions des angles donnent des chiffres ronds. De plus, ce fichier est en libre téléchargement en ligne et est fourni par un service officiel de la région bruxelloise, ce qui en fait un bon candidat pour servir de référence. Son extension spatiale (BoundingBox) est celle qui a été utilisée pour harmoniser l'ensemble des rasters du modèle. En revanche, la résolution utilisée pour la « grille de référence » est de 10x10m (contre une résolution de 1x1m pour le raster UrbDTM_GRID_Y12_RBC_1m.tif fourni par Urbis). En résumé, la grille de référence est obtenue en effectuant un changement de résolution de 1x1m à 10x10m du raster de topographie fourni par Urbis UrbDTM_GRID_Y12_RBC_1m.tif. Les caractéristiques de la grille de référence sont recensées dans le Tableau 2.

Le téléchargement du raster UrbDTM_GRID_Y12_RBC_1m.tif et de la documentation associée est disponible sur le site d'Urbis solutions : https://cirb.brussels/fr/nos-solutions/urbis-solutions/urbis-data/urbis-data

Tableau 2 : caractéristiques de la grille de référence utilisée pour harmoniser les rasters.

Paramètre	Valeur
Format	GTiff
Nombre de canaux	1
Dimensions (en pixels)	1900 x 1900
Système de coordonnées	EPSG 31370 : Belge 1972 / Belgian Lambert 72
Cordonnées de l'angle Nord-Ouest	140000, 179000
Emprise (BoundingBox)	140000.0000000000000000,160000.0000000000000000 ; 159000.0000000000000000,179000.0000000000000000
Résolution	10x10 m
Valeur des pixels sans données	-3.40282e+38

I.4. TABLE DE SYNTHESE DES DONNEES

Afin d'indiquer aux scripts pythons la structure du modèle géologique (l'ordre des unités stratigraphiques), un fichier CSV a été construit. Celui-ci indique ligne par ligne, dans l'ordre stratigraphique, quels sont les noms des fichiers rasters de toits et d'épaisseurs à utiliser pour chacune des unités aux différentes phases de traitement des rasters. Ce fichier contient aussi le nombre de points qui ont servi à construire chacun des rasters de toit (cf. Devleeschouwer et al, 2017). Cela sert à définir l'ordre de priorité des rasters lors de la phase de correction (cf. chapitre III.2). Le fichier utilisé est présenté en annexe VI.1.

I.1. CONTOURS DE LA RBC

Afin de découper les données et de les limiter à la RBC, le contour de cette dernière a été utilisé. La couche utilisée dans le cadre de ce travail est la couche « Re » du service Urbis WFS, avec le système de coordonnées {EPSG 31370 : Belge 1972 / Belgian Lambert 72}.

Les Web services d'Urbis sont disponibles à l'adresse suivante :

https://cirb.brussels/fr/nos-solutions/urbis-solutions/urbis-applications

Dans le cadre de ce travail, les données rasters ont été limitées à la RBC agrandie d'une bande tampon de 500 m.

II. OUTILS UTILISES

Le traitement des rasters de BruStrati3D a été réalisé à l'aide de deux outils principaux :

- Python, qui a été l'outil central de ce travail via la programmation d'algorithmes ;

- QGIS, qui a servi à visualiser les données produites par les algorithmes, à vérifier leur cohérence et à effectuer quelques traitements ponctuels pour lesquels des algorithmes auraient été plus lourds.

L'utilisation de Python pour le traitement des rasters de BruStrati3D a eu plusieurs avantages par rapport à un logiciel de SIG classique comme QGIS :

- Cas des opérations répétitives : le traitement du modèle géologique demande de répéter de très nombreuses fois de multiples opérations. Par exemple, pour vérifier qu'aucune incohérence n'existe, cela demande de soustraire deux à deux les 19 rasters de toits, soit 171 opérations de soustraction de rasters, puis de vérifier qu'il n'existe pas de pixel à valeur négative dans les 171 rasters résultants. Ceci est facilement géré par des boucles dans les algorithmes ;

- Automatisation : le modèle BruStrati3D peut être amené à évoluer avec l'acquisition de nouvelles données enrichissant la base de données (forage, sondage,..). Les algorithmes pourront être réutilisés dans ce cas ;

- Souplesse : certaines fonctions existent sur Python avec des options qui n'apparaissent pas dans QGIS. Par ailleurs, quand une fonction n'existe pas, il est possible de la programmer. Ceci-dit QGIS à l'avantage de disposer de tout un catalogue d'outils avancés déjà programmés qui n'existent pas toujours sur Python.

- Transparence de la méthode : l'algorithme utilisé peut être consulté à tout moment.

Cependant, l'utilisation de Python présente trois inconvénients majeurs, compensés par l'utilisation de QGIS :

- La visualisation cartographique est possible (avec des bibliothèques comme MathPlotLib par exemple) mais demande du code, alors qu'elle est automatique sur QGIS ;

- Effectuer des opérations simples en petit nombre (ex : reprojeter une seule couche) demande forcément quelques lignes de code sur Python, alors que c'est accessible en quelques clics sur QGIS ;

- Les SIG comme QGIS gèrent automatiquement les données ayant des caractéristiques différentes (systèmes de coordonnées, extensions variables…). Avec Python, le traitement de données aux caractéristiques hétérogènes requière une attention particulière et complexifie la programmation.

II.1. PYTHON

Python est un langage de programmation orienté objet placé sous licence libre. L'ensemble de la documentation et des liens de téléchargement pour son installation sont disponibles à l'adresse suivante : https://www.python.org/

La version de Python utilisée dans le cadre de ce travail est **Python 3.6.**

Outre la bibliothèque standard, Python possède une multitude de bibliothèques complémentaires permettant d'étendre les fonctionnalités de base du langage. Dans le cadre de ce travail, deux bibliothèques ont été centrales dans le traitement des données rasters : **rasterio** pour la manipulation des fichiers rasters et **numpy** pour le calcul matriciel. D'autres bibliothèques ont été utilisées ponctuellement pour lire des données vecteurs (**fiona**), effectuer des opérations mathématiques avancées (**math**) ou gérer plus facilement les fichiers (**Path**, **os**, **shutil et sys**).

La documentation relative à ces librairies et les versions utilisées sont résumées dans le Tableau 3.

Dans le cadre de ce travail, les scripts ont été codés à l'aide de l'environnement de développement **Spyder (version 3.2.7)**, accessible depuis le logiciel **Anaconda Navigator**. Dans Anaconda, un environnement (nommé « gis ») a été créé en important, entre autres, les packages suivants : psycopg2, IO tools, pandas, geopandas, geojson, geopy, fiona, rasterio, numpy. La liste détaillée des packages de l'environnement « gis » est donnée en annexe VI.5.

Tableau 3 : versions et documentations des bibliothèques Python utilisées pour le traitement des données stratigraphiques.

Bibliothèque	Version	Documentation
Rasterio	0.36.0	https://rasterio.readthedocs.io/en/latest/index.html
Numpy	1.14.1	http://www.numpy.org/
Fiona	1.7.10	https://github.com/Toblerity/Fiona
Math	Libraire standard	https://docs.python.org/3.7/library/math.html
Os	Libraire standard	https://docs.python.org/3.7/library/os.html
Path	11.0	https://github.com/jaraco/path.py
Shutil	Libraire standard	https://docs.python.org/3/library/shutil.html
Sys	Libraire standard	https://docs.python.org/3/library/sys.html

II.2. QGIS

QGIS est un SIG libre d'accès, permettant entre autre de traiter de l'information géographique et de construire des cartes. La documentation et les liens de téléchargement du logiciel sont disponibles au lien suivant :

https://qgis.org/en/site/

La version de QGIS utilisée dans le cadre de ce travail est **QGIS 2.14.3.**

III. METHODOLOGIE DE TRAITEMENT DES DONNEES

Le traitement des rasters de BruStrati3D v1.0 est passé par quatre grandes phases (Figure 2) :

(1) Harmoniser l'ensemble des rasters afin qu'ils aient tous les même caractéristiques que la grille de référence. Cette étape se divise en une première « sous-phase » de traitements manuels et en une seconde « sous-phase » automatisée ;

(2) Appliquer des corrections aux rasters des unités stratigraphiques du modèle BruStrati3D, pour éviter d'avoir localement des épaisseurs de couches négatives ;

(3) Construire les rasters d'épaisseurs des unités stratigraphiques ;

(4) Supprimer de l'extension des toits des couches les pixels où l'épaisseur calculée est nulle.

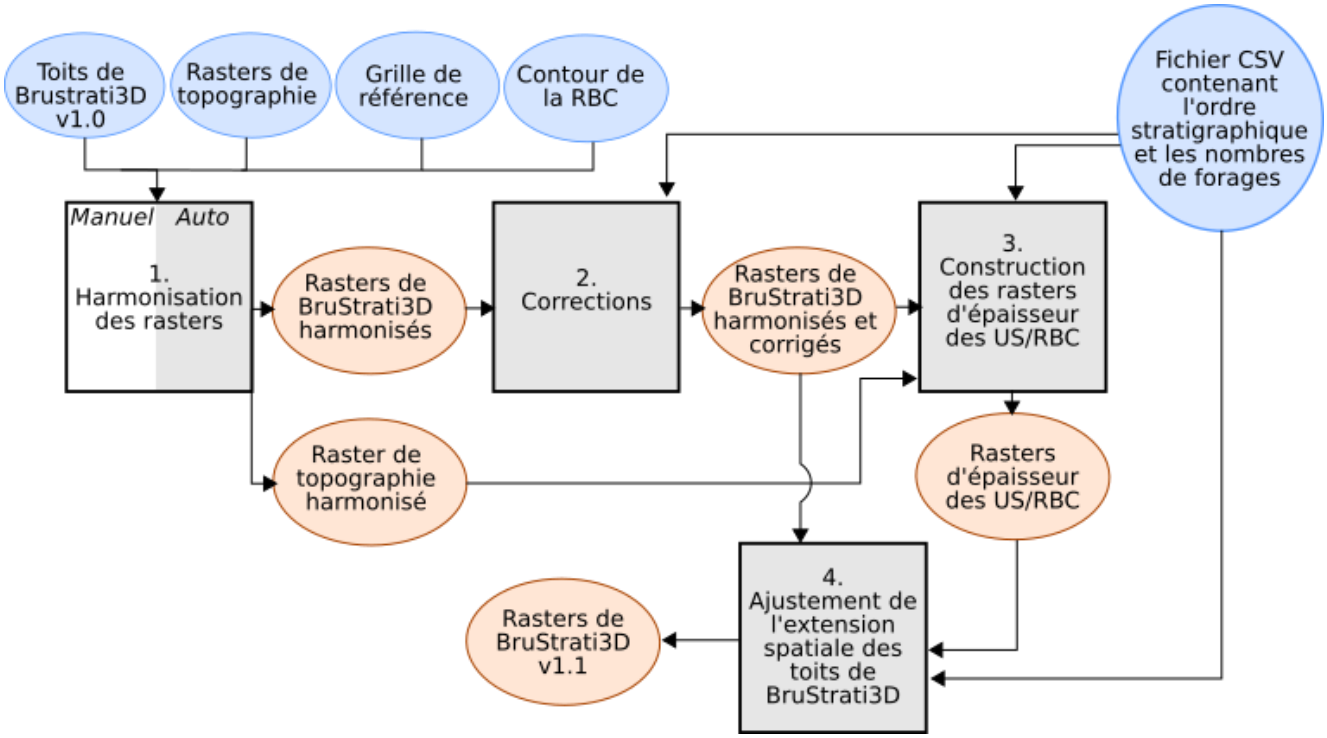


Figure 2 : bloc diagramme schématisant les 4 grandes phases (rectangles gris) de traitement des rasters de BruStrati3D v1.0 pour aboutir à la version 1.1.

Les traitements manuels de la phase 1 ont été réalisés sur QGIS. Le détail de la procédure employée est donné en annexe VI.2.

Pour le traitement automatique des données, un package Python appelé **Brugepy** a été construit (Figure 3). Celui-ci contient plusieurs modules dans lesquels sont codées les fonctions et objets auxquels quatre scripts font appel. Le traitement des données est réalisé en utilisant successivement ces quatre scripts, qui correspondent respectivement aux quatre grandes étapes de la Figure 2. L'annexe VI.3 donne des recommandations sur l'utilisation de ces scripts. Les codes pythons des scripts et des modules de Brugepy sont donnés en annexe VI.4.

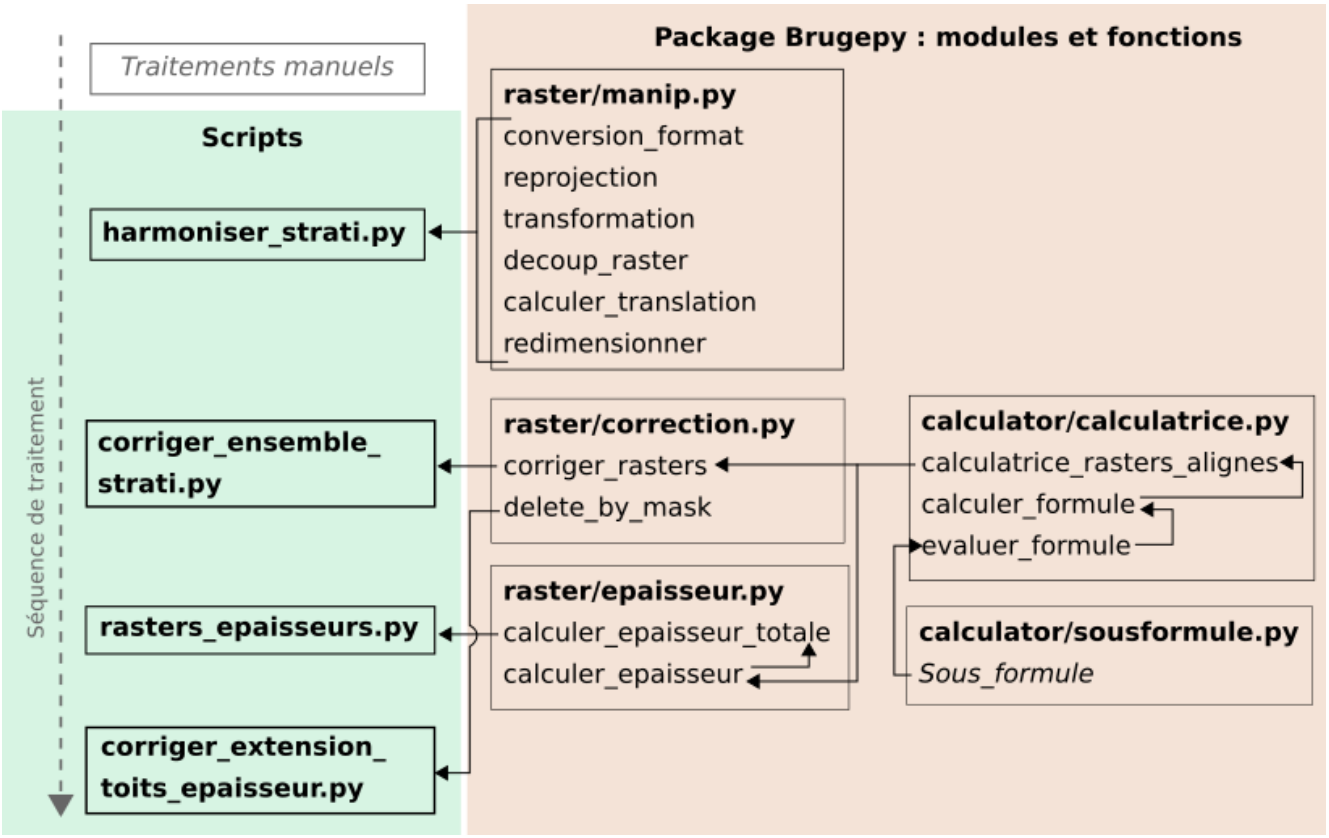


Figure 3 : structure du package Brugepy. Interactions entre ses différentes fonctions et les scripts de traitement des données de BruStrati3D.

III.1. HARMONISATION DES RASTERS

L'objectif de la première phase est d'harmoniser les rasters utilisés pour qu'ils aient tous les mêmes caractéristiques (format, projection, extension spatiale, résolution…).

Pour harmoniser les rasters sur une base commune, la couche utilisée comme référence est le raster de topographie fourni par Urbis UrbDTM_GRID_Y12_RBC_1m.tif rééchantillonné à une résolution de 10x10 m (chapitre I.3). Les caractéristiques à harmoniser sont recensées dans le Tableau 4.

Lors de cette phase, les données des rasters des unités stratigraphiques de BruStrati3D ont aussi été découpées pour être limitées à une zone correspondant aux contours de la RBC étendus d'une zone tampon de 500 m.

L'harmonisation des rasters a été divisée en deux « sous-phases » :

- Une première « sous-phase » de traitements manuels effectués sur QGIS (cf. annexe VI.2), afin de :

  - Construire le raster servant de « grille de référence » ;
  - Harmoniser le raster de topographie avec la grille de référence ;
  - Calculer la translation à appliquer à l'ensemble du modèle stratigraphique pour que celui-ci « suive » l'harmonisation de la topographie ;
  - Construire un Shapefile contenant une zone tampon de 500 m autour des limites de la RBC.

- Une seconde « sous-phase » de traitements automatisés (via le script Python harmoniser_strati.py donné en annexe VI.4.1.1) pour harmoniser l'ensemble des rasters des unités stratigraphiques de BruStrati3D. Les traitements effectués par ce script sont :

- Une conversion au format GeoTiff ;

- Une reprojection en RGF1993, Lambert Belge 72 (EPSG : 313770) ;

- Un ajustement de la résolution à 10 x 10 m² ;

- Une translation des couches, nécessaire pour « suivre » l'alignement de la topographie avec la grille de référence. En effet, afin de garder la cohérence du modèle géologique BruStrati3D, on souhaite qu'à l'état final, les pixels des unités stratigraphiques soient alignés avec les pixels de la topographie dont ils étaient le plus proche initialement. Pour cela, il est nécessaire d'appliquer aux rasters du modèle géologique le même vecteur de translation que celui qu'a subit le raster de topographie au moment de son harmonisation avec la grille de référence. La Figure 4 illustre en quoi il est important de translater l'ensemble du modèle géologique de la même manière qu'a été translaté le raster de topographie ;

- Un alignement de chacune des couches avec la grille de référence ;

- Le découpage des couches avec le contour de la RBC+500m ;

- Le redimensionnement de l'emprise spatiale des rasters, pour correspondre à l'emprise spatiale de la grille de référence.

Tableau 4 : caractéristiques initiales des différents rasters. L'objectif est d'harmoniser les caractéristiques de tous les rasters pour qu'elles correspondent à celles de la « grille de référence ».

Caractéristiques	BruStrati 3D v1.0	Topographie	Grille de référence
Format	.adf Pilote GDAL provider AIG	.adf Pilote GDAL provider AIG	.tif Pilote GTiff
SCR	Personnalisé	Personnalisé	Lambert 72
Résolution	10 x 10 m	10 x 10 m	10 x 10 m
Grille	Les grilles ne sont pas alignées d'un raster à l'autre	La grille n'est pas alignée avec la grille du raster Urbis de référence	-
Décalage de la grille par rapport à la grille de référence	Variable d'un raster à l'autre	Axe X (E-O) : 0,0178016429999843m Axe Y (N-S) : 2,49018521499238m	-
Emprise spatiale (dans le SCR initial)	Variable d'un raster à l'autre	Xmin : 20949.9817584999836981 Ymin : 21032.4899865002371371 Xmax : 295249.9817584999836981 Ymax : 250022.4899865002371371	Xmin : 140000.00000 Ymin : 160000.000000 Xmax : 159000.000000 Ymax : 179000.000000

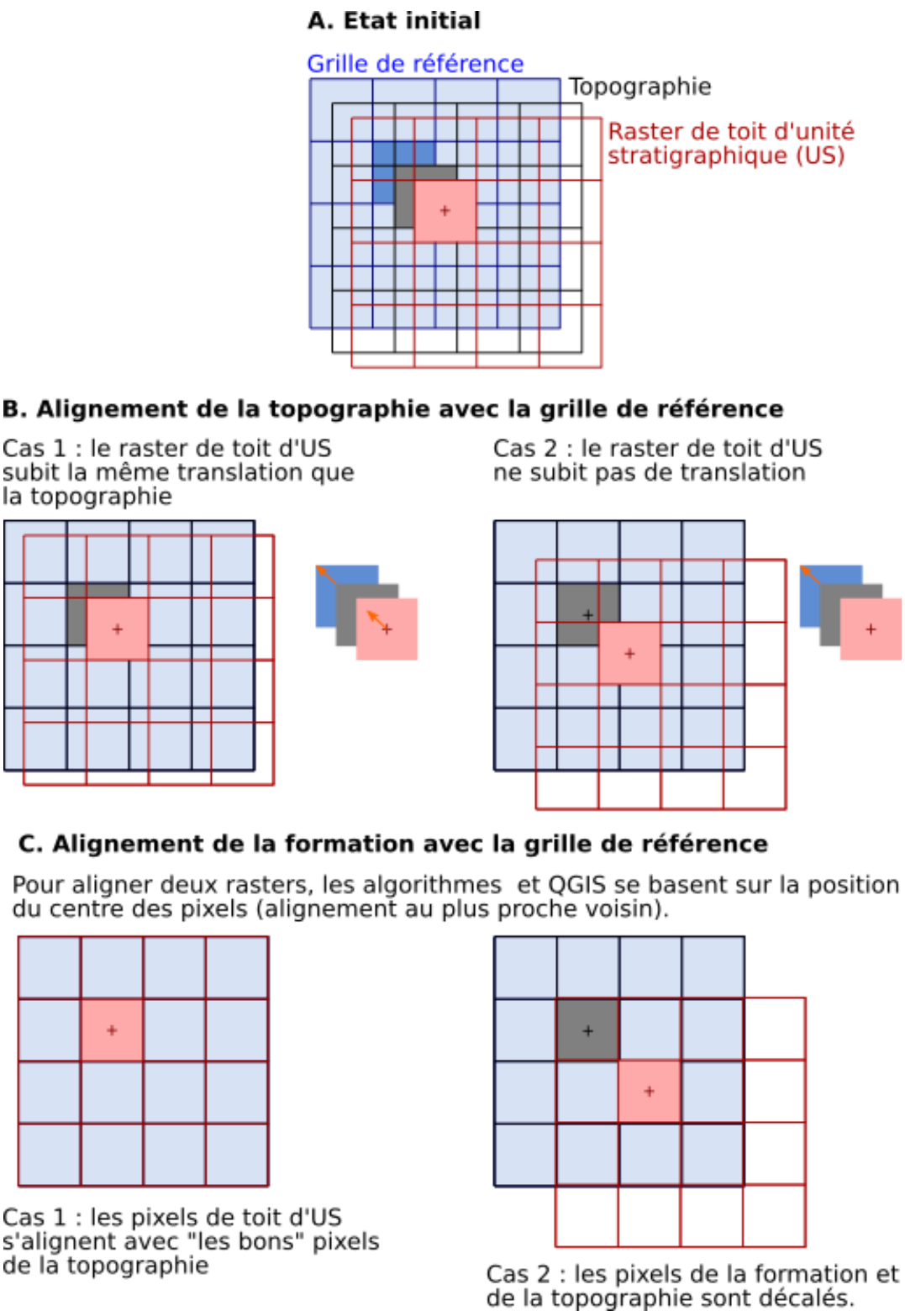


Figure 4 : schéma illustrant les résultats successifs de l'alignement du raster de topographie et d'un raster de toit d'unité stratigraphique avec la grille de référence. Sont illustrés deux cas : (1) si on applique au raster de toit d'unité stratigraphique la même translation qu'appliquée au raster de topographie, avant de l'aligner lui-même avec la grille de référence ; (2) si on applique pas cette translation au raster de toit d'unité stratigraphique avant de l'aligner avec la grille de référence. Comme l'alignement se fait « au plus proche voisin », on voit que le cas (2) risque de créer des incohérences en décallant les pixels qui auraient dû se correspondre initialement entre la topographie et l'unité stratigraphique. Le cas (1) assure cette cohérence, même si le raster de toit d'unité stratigraphique subit une translation totale plus importante que dans le cas (2).

III.2. CORRECTION DES INTERSECTIONS DE COUCHES

Le modèle géologique BruStrati3D v1.0 contient très localement des zones où les toits de certaines couches « se recoupent ». Autrement dit, des pixels du toit de certaines couches ont une valeur d'altitude plus élevée que les pixels correspondants de couches normalement au-dessus dans la stratigraphie. Cela est impossible d'un point de vue géologique et génèrerait des zones d'épaisseurs négatives (Figure 5).

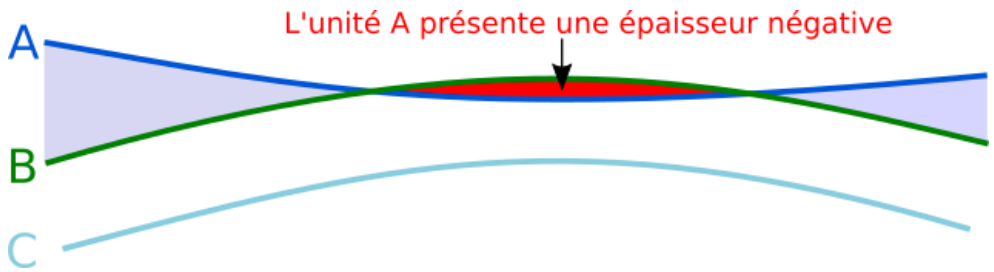


Figure 5 : coupe schématique illustrant un problème d'intersection entre deux toits de couches. L'unité A est normalement au-dessus de l'unité B (zones normales d'épaisseurs positives en bleu). Sur cette coupe, la partie en rouge présente une zone où ceci n'est pas respecté et où le toit de B à des altitudes supérieures au toit de A. Dans le modèle, ces erreurs sont généralement limitées à quelques pixels par unité.

L'étape de correction va donc faire en sorte d'éviter les zones d'épaisseurs négatives en redéfinissant l'altitude des pixels incohérents. Plus précisément, lorsque deux couches se recoupent, les pixels incohérents d'une des deux couches vont prendre les valeurs des pixels correspondant de l'autre couche. Cela créera une zone d'épaisseur nulle pour la couche supérieure et non plus négative (Figure 6). Cette étape de correction est réalisée par le script corriger_ensemble_strati.py donné en annexe VI.4.1.2.

Afin de déterminer quelle couche est corrigée et quelle couche fournie ses valeurs à l'autre, le script va prendre en considération le nombre de données de forages qui ont servies à construire les différentes couches. Cela part du principe que les couches issues de l'interpolation d'un plus grand nombre de forages sont certainement modélisées plus précisément que les autres. On va donc chercher à corriger les couches modélisées « potentiellement moins précisément », c'est-à-dire celles qui ont le moins de données de forages (Figure 6).

Le script va comparer deux à deux chacune des couches du modèle et vérifier s'il y a des incohérences. Si des incohérences sont repérées, c'est la couche ayant été construite avec le moins de données de forage qui est modifiée. Cette couche corrigée viendra remplacer la couche initialement incohérente pour être comparée avec les autres couches et être recorrigée si nécessaire.

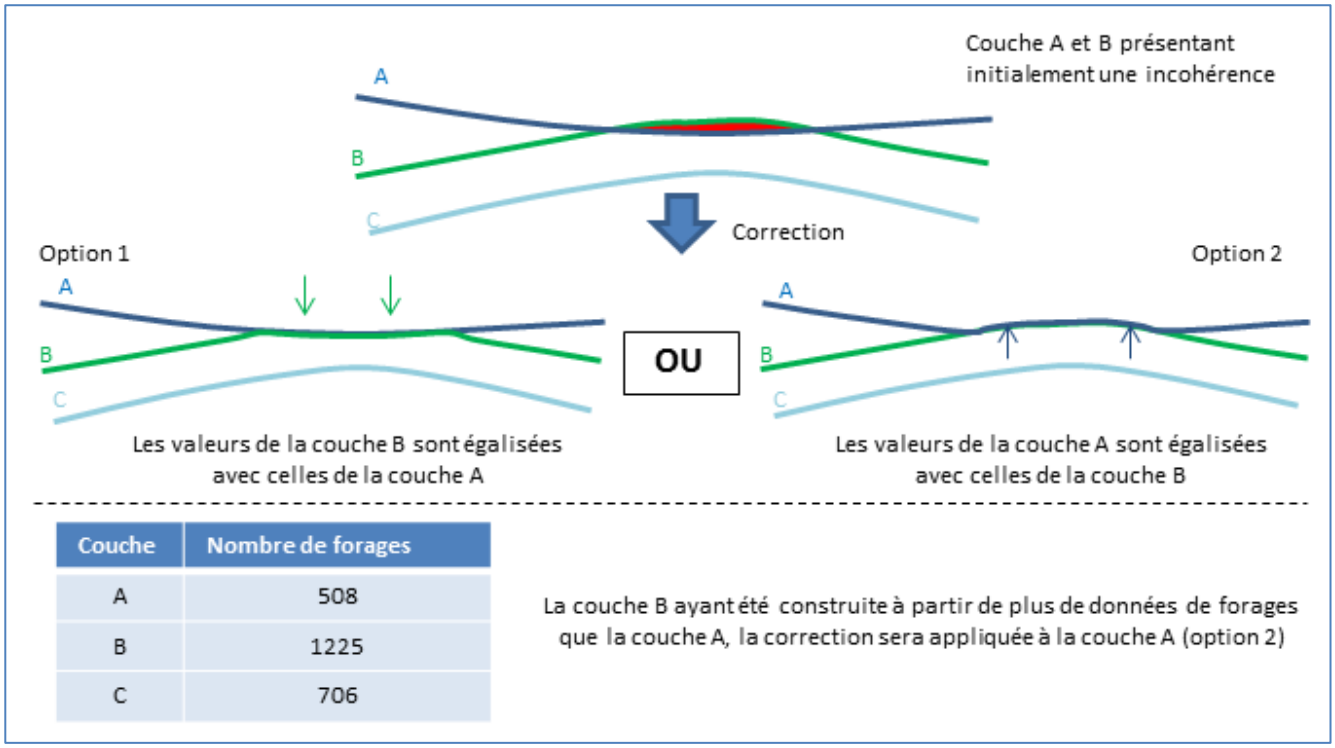


Figure 6 : schéma illustrant les corrections appliquées aux couches. Les couches A et B présentent localement une incohérence. Pour les corriger, deux options sont possibles : soit l'on corrige B pour que ses pixels incohérents prennent les valeurs de ceux de A (option 1), soit l'inverse (option 2). Dans le cas de cet exemple fictif, c'est l'option 2 qui sera choisie car la couche B a été construite avec plus de données que A.

III.3. CONSTRUCTION DES RASTERS D'EPAISSEURS

Cette phase vise à produire les rasters d'épaisseurs de chacune des unités modélisées, (excepté le paléozoïque pour lequel il est impossible de calculer une épaisseur, ne disposant que du toit). Elle est réalisée par le script rasters_epaisseurs.py donné en annexe VI.4.1.3.

Etant donné que les toits des différentes couches ont des extensions variables, l'épaisseur d'une couche donnée n'est pas uniquement définie par la différence d'altitude entre son toit et celui de la couche directement inférieure. Il faut prendre en compte le fait que certaines couches peuvent ne pas être présentes à certains endroits (discontinuité). Ainsi, l'épaisseur d'une unité peut potentiellement être définie par la différence d'altitude entre son toit et le toit de toute unité sous-jacente, selon les endroits (Figure 7).

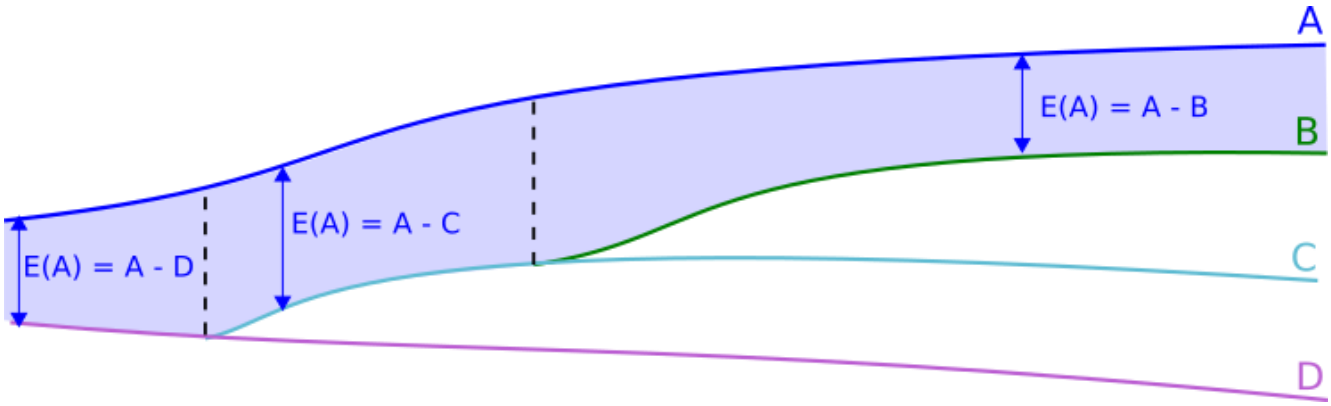


Figure 7: coupe schématique illustrant l'évolution latérale du calcul de l'épaisseur d'une couche donnée (couche A) en fonction de l'extension des couches sous-jacentes. La partie colorée en bleu représente l'épaisseur calculée pour la couche A. Dans les formules E(A) signifie « épaisseur » de A et chaque lettre représentent l'altitude du toit auquel elle fait référence.

Pour prendre en compte cela, l'algorithme va, pour chacune des unités, construire un raster d'épaisseurs en « recollant » étape par étape chaque zone caractérisée par un calcul des épaisseurs issu de la considération des toits de l'unité sous-jacente directement présente. Pour une unité donnée (nommée A), l'algorithme va commencer par calculer les épaisseurs où l'unité directement sous-jacente est présente (unité B). Puis, aux zones où l'unité B n'existe pas, il va faire de même avec l'unité sous-jacente à B (unité C) et ainsi de suite.

Cela est réalisé à l'aide d'un masque, qui évolue au fur et à mesure de la construction du raster d'épaisseur de l'unité. Au départ, le masque indique que l'épaisseur n'a été calculée en aucun endroit de l'extension unité A. Une fois l'épaisseur calculée à l'aide de l'unité B, l'extension de l'unité B est retirée du masque. L'algorithme sait alors que l'épaisseur de l'unité A ne doit plus être calculée qu'en dehors de l'extension de l'unité B. Il effectue la même procédure avec l'unité C et ainsi de suite.

Cas de l'épaisseur du Quaternaire : problème des « zones creuses »

Dans le modèle BruStrati3D v 1.0, un raster de la base du Quaternaire a été construit (Devleeschouwer et al, 2017). Cette base du Quaternaire, construite à l'aide des données d'épaisseur du Quaternaire, ne correspond pas toujours au toit de l'unité tertiaire la plus haute. Autrement dit, en certains endroits il y a une différence entre l'altitude de la base du quaternaire et l'altitude du premier toit de couche situé sous cette base. Cela crée des « zones creuses » dans le modèle (Figure 8).

Pour éviter ces « zones creuses », l'épaisseur du Quaternaire a été calculée à l'aide de la différence entre la topographie (considérée comme le toit du quaternaire) et l'élévation des toits des couches sous-jacentes (et non à l'aide de l'élévation de la base du Quaternaire). Cela a pour effet de « remplir les zones creuses » avec du Quaternaire.

Si cette méthode permet d'éviter des « zones creuses » dans le modèle, et de rendre les épaisseurs et les altitudes de toits cohérentes entre elles, elle présente tout de même deux inconvénients :

- Le quaternaire a tendance à être épaissi au niveau des sommets (qui ont plus tendance à présenter des zones creuses que les fonds de vallées, Figure 8 « ZC^1 »). D'un point de vue géologique, le quaternaire devrait plutôt être aminci sur les sommets ;

- Cela prend en compte dans l'épaisseur du Quaternaire des zones qui visuellement (en coupe) semblent en continuité d'autres couches (Figure 8 « ZC^2 »).

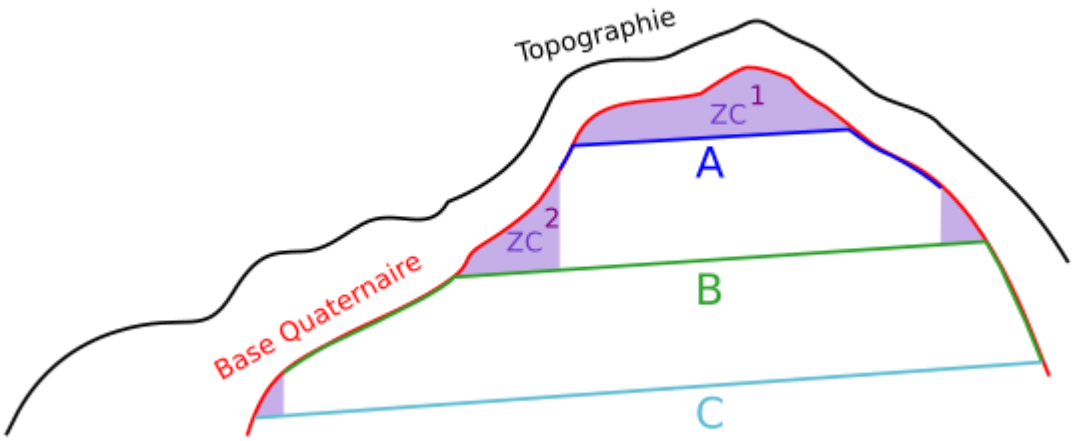


Figure 8 : coupe schématique illustrant la base du Quaternaire plus élevée que le toit de la couche directement sous-jacente. Calculer l'épaisseur du Quaternaire à l'aide de la topographie et du raster de la base du Quaternaire génèrerait des « zones creuses » (ZC, en violet), qui ne seraient prises en compte dans aucune épaisseur de couche. Pour éviter cela, l'épaisseur du Quaternaire est calculée comme la différence entre l'élévation topographique et l'altitude du toit de la couche sous-jacente la plus haute.

III.4. CORRECTION DE L'EXTENSION DES TOITS DES FOMATIONS

Suite aux corrections effectuées précédemment, le modèle présente localement des pixels où les rasters de toits indiquent la présence d'une unité stratigraphique (une altitude est indiquée) alors que l'épaisseur calculée en ces pixels est nulle. La dernière étape consiste donc à supprimer des rasters de toits ces pixels où l'épaisseur de la couche est en fait nulle (et donc où l'unité n'existe en fait pas).

L'algorithme va donc parcourir les rasters d'épaisseurs et les rasters de toits des unités stratigraphiques et sélectionne deux à deux les rasters de toits et d'épaisseurs qui se correspondent. Pour chaque unité stratigraphique, une matrice de masque indiquant les pixels où l'épaisseur est égale à 0 est extraite du raster d'épaisseurs. Ce masque est ensuite appliqué à la matrice du toit de la couche correspondante, puis les pixels masqués vont être remplis avec la valeur de « nodata ». La matrice est ensuite écrite dans un nouveau raster.

Ce processus n'est cependant pas appliqué au toit du Quaternaire qui est représenté par la topographie.

Cette étape est effectuée par l'algorithme corriger_extension_toits_epaisseurs.py donné en annexe VI.4.1.4.

IV. DONNEES EN SORTIE : SYNTHESE

Le modèle stratigraphique BruStrati3D v1.1, est donc constitué des éléments suivants :

- Un raster de topographie, qui correspond au raster de topographie de BruStrati3D v1.0 ayant été reprojeté, légèrement translaté et dont l'emprise spatiale a été harmonisée avec la grille de référence ;

- Les rasters d'élévations des toits des mêmes unités stratigraphiques que BruStrati3D v1.0 dont les caractéristiques ont été harmonisées avec la grille de référence et dont la valeur de certains pixels ont été modifiées pour éviter des épaisseurs de couches négatives ou nulles ;

- Les rasters d'épaisseurs des mêmes unités stratigraphiques que BruStrati3D v1.0, excepté le Paléozoïque.

Tous ces rasters présentent les caractéristiques suivantes :

Paramètre	Valeur
Format	GTiff
Nombre de canaux	1
Dimensions en pixels	1900 x 1900
Système de coordonnées	EPSG 31370 : Belge 1972 / Belgian Lambert 72
Cordonnées de l'angle Nord-Ouest	140000, 179000
Emprise (BoundingBox)	140000.0000000000000000,160000.0000000000000000 ; 159000.0000000000000000,179000.0000000000000000
Résolution	10x10m
Valeur des pixels sans données	-3.40282e+38

V. REFERENCES

V.1. RAPPORTS

CIRB (2015) Spécifications techniques des données Urbis-DTM. Disponible à l'adresse :

https://cirb.brussels/fr/nos-solutions/urbis-solutions/fichiers/specifications-techniques-urbis-dtm.pdf

Devleeschouwer X., Goffin C, Vandaele J. & Meyvis B. (2017) « Modélisation stratigraphique en 2D et 3D du sous-sol de la Région de Bruxelles-Capitale ». Projet n°2016B0512 de l'IBGE. Rapport final, 97 pages. Téléchargeable à l'adresse : https://environnement.brussels/thematiques/geologie-et-hydrogeologie/geologie

V.2. SITES INTERNET

QGIS: https://qgis.org/en/site/

Python : https://www.python.org/

Urbis : https://cirb.brussels/fr/nos-solutions/urbis-solutions/

V.3. DOCUMENTATION EN LIGNE DES LIBRAIRIES PYTHON UTILISEES

Fiona : https://github.com/Toblerity/Fiona

Math : https://docs.python.org/2/library/math.html

Os : https://docs.python.org/2/library/os.html

Path: https://github.com/jaraco/path.py

Rasterio : https://rasterio.readthedocs.io/en/latest/index.html

Shutil : https://docs.python.org/3/library/shutil.html

V.4. DONNEES EN LIGNE

Raster de topographie Urbis : https://cirb.brussels/fr/nos-solutions/urbis-solutions/urbis-data/urbis-data

Service WFS Urbis : https://cirb.brussels/fr/nos-solutions/urbis-solutions/urbis-applications

Rédaction: Louis GAUDARE

Comité de lecture: Mathieu AGNIEL

Ed. Resp. : F. Fontaine et B. Dewulf – Av du Port 86C/3000- 1000 Bruxelles

VI. ANNEXES

VI.1. FICHIER CSV DE STRUCTURE DES DONNEES

Numero	Lettre	Nom	Forages	Fichier_toit_harmonise	Fichier_toit_corrige	Fichier_epaisseur
1	A	Topographie	6981	B01_Base_Quaternaire	A01_Quaternaire	A01_Quaternaire_epa
2	B	Diest	7	A02_Diest	A02_Diest	A02_Diest_epa
3	C	Bolderberg	37	A03_Bolderberg	A03_Bolderberg	A03_Bolderberg_epa
4	D	Saint_Huibrechts_Hern	232	A04_StHuibrechts_Hern	A04_StHuibrechts_Hern	A04_StHuibrechts_Hern_epa
5	E	Onderdale	9	A05_Onderdale	A05_Onderdale	A05_Onderdale_epa
6	F	Ursel_et_Asse	368	A06_UrselAsse	A06_UrselAsse	A06_UrselAsse_epa
7	G	Wemmel	654	A07_Wemmel	A07_Wemmel	A07_Wemmel_epa
8	H	Lede	1458	A08_Lede	A08_Lede	A08_Lede_epa
9	I	Bruxelles	2551	A09_Bruxelles	A09_Bruxelles	A09_Bruxelles_epa
10	J	Vlierzel	130	A10_Vlierzele	A10_Vlierzele	A10_Vlierzele_epa
11	K	Merelbeke	206	A11_Merelbeke	A11_Merelbeke	A11_Merelbeke_epa
12	L	Tielt	1003	A12_Tielt	A12_Tielt	A12_Tielt_epa
13	M	Aalbeke	566	A13_Aalbeke	A13_Aalbeke	A13_Aalbeke_epa
14	N	Moen	2557	A14_Moen	A14_Moen	A14_Moen_epa
15	O	Saint-Maur	1254	A15_StMaur	A15_StMaur	A15_StMaur_epa
16	P	Grandglise	567	A16_Grandglise	A16_Grandglise	A16_Grandglise_epa
17	Q	Lincent	271	A17_Lincent	A17_Lincent	A17_Lincent_epa
18	R	Cretace	116	A18_Cretace	A18_Cretace	A18_Cretace_epa
19	S	Paleozoique	250	A19_Paleozoique	A19_Paleozoique	

VI.2. MANIPULATIONS MANUELLES

En amont des traitements automatisés, des traitements manuels sur QGIS sont nécessaires. Les numéros des étapes se réfèrent à ceux indiqués sur le schéma logique suivant :

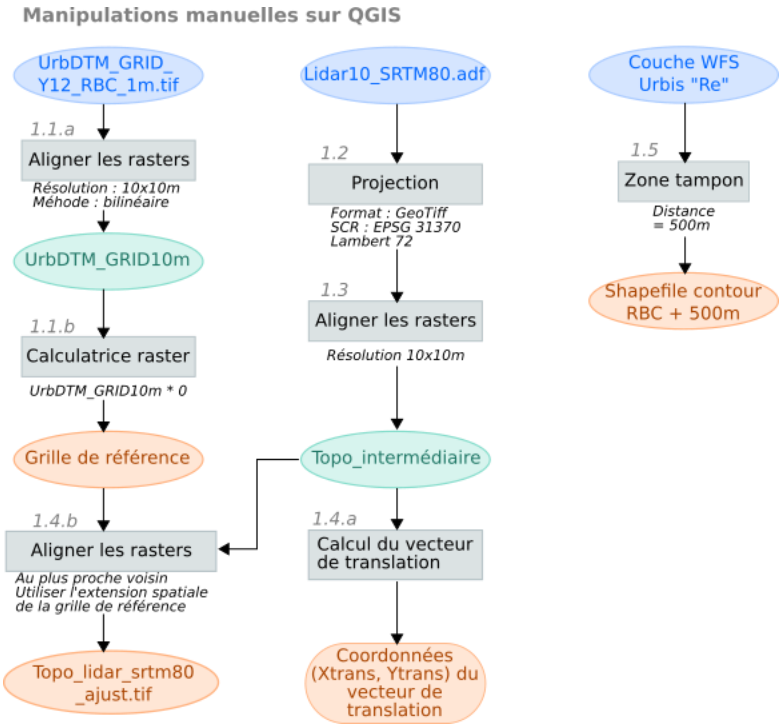


Schéma logique des traitements manuels effectués sur QGIS, en amont du traitement automatisé des données. Les ellipses représentent les données, les rectangles représentent les outils QGIS. Bleu : donnée initiale ; Vert : donnée intermédiaire ; Orange : donnée finale.

1.1.a / 1.1.b Construction de la « grille de référence »

La première étape consiste à construire la « grille de référence » à partir du raster de topographie Urbis UrbDTM_GRID_Y12_RBC_1m.tif (cf. chapitre I.3). Cette étape va nécessiter deux manipulations sur QGIS :

- 1.1.a : ré-échantillonner le raster UrbDTM_GRID_Y12_RBC_1m.tif à une résolution de 10x10 m.

Raster → Aligner les rasters…

Couche raster à aligner : UrbDTM_GRID_Y12_RBC_1m.tif / Nom du fichier en sortie : UrbDTM_GRID10m / Méthode de rééchantillonnage : bilinéaire (noyau 2x2) / SCR : {EPSG 31370 : Belge 1972 / Belgian Lambert 72} / Taille de cellule : 10,0000 ; 10,0000 / Laisser les autres paramètres par défaut, ne pas ajouter d'autres rasters à aligner.

- 1.1.b : utiliser la calculatrice raster sur le raster obtenu précédemment afin de supprimer les valeurs des pixels en les remplaçant par des zéros.

Raster → Calculatrice rasters…

*Bandes raster : UrbDTM_GRID10m@1 / Expression de la calculatrice raster : ''UrbDTM_GRID10m@1"*0 / Format en sortie : GeoTIFF / Emprise actuelle de la couche / SCR sélectionné {EPSG 31370 : Belge 1972 / Belgian Lambert 72} / Couche_en_sortie : grille_de_reference.tif*

Le raster ainsi obtenu (nommé **grille_de_reference.tif**) fera office de « grille de référence » pour la suite des traitements.

1.2 Conversion de format et reprojection du raster initial de topographie

Le raster initial de topographie « Lidar10_SRTM80.adr » (issu du modèle BruStrati3D v1.0) doit, pour être harmonisé avec la grille de référence, être converti en format GeoTiff, et être reprojeté. En effet, son système de coordonnées n'est pas correctement reconnu (il est défini comme « personnalisé »). Ces deux modifications peuvent se faire en une étape à l'aide de l'outil « projection » de QGIS.

Raster → Projections → Projection…

Raster en entrée : « hdr.adf » / Fichier en sorite : topo_reproj.tif (bien choisir GeoTiff) / Pas de SCR source / SCR cible : {EPSG 31370 : Belge 1972 / Belgian Lambert 72} / Laisser les autres paramètres par défaut.

1.3 Ajustement de la résolution du raster de topographie

L'extension spatiale du raster produit à l'étape précédente (1.2), montre que la résolution du raster a légèrement changée lors de la reprojection, et n'est plus exactement de 10x10m (la variation est infinitésimale, mais pour s'assurer du bon fonctionnement des algorithmes par la suite, celle-ci ne peut pas être négligée). Pour cela un rééchantillonnage doit être effectué pour ajuster la résolution à 10x10m.

Raster → Aligner les rasters…

Couche raster à aligner : topo_reproj.tif / Nom du fichier en sortie : topo_intermediaire.tif / Méthode de rééchantillonnage : Au plus proche voisin / Taille de cellule : 10,0000 ; 10,0000 / Laisser les autres paramètres par défaut, ne pas ajouter d'autres rasters à aligner.

1.4.a / 1.4.b Translation de la topographie et découpage de son emprise spatiale

Cette étape consiste à aligner le MNT obtenu à l'étape précédente avec la grille de référence.

Hormis le découpage nécessaire pour que les emprises spatiales soient identiques, il faut aussi aligner (« snapper ») les grilles des deux rasters. Cette étape d'alignement correspond en fait à une translation appliquée aux pixels du MNT pour qu'ils s'alignent avec ceux de la grille de référence. Afin de garder la cohérence du modèle géologique, il faudra appliquer cette même translation à l'ensemble des rasters des unités stratigraphiques du modèle avant de les aligner avec la grille de référence (cf. chapitre III.1, Figure 4). Il est donc important de connaitre les coordonnées du vecteur de translation qui sera appliqué à la topographie pour l'aligner avec la grille de référence.

- 1.4.a : avant même d'aligner la topographie avec la grille de référence, il faut récupérer les coordonnées du vecteur de translation qu'elle va subir lors de l'alignement (pour les utiliser à l'étape 1.9). Pour cela, plusieurs méthodes sont possibles :

    - Cela est faisable à l'aide de la fonction Python « calculer_translation » codée dans le fichier « manip.py » du package Brugepy (cf. annexe VI.4.2.1).

    - Il est aussi possible de calculer ce vecteur soit même à l'aide des coordonnées respectives des angles Nord-Ouest des deux vecteurs. Pour cela il faut tout d'abord déterminer le sens en X et en Y de la translation, visuellement, en repérant deux pixels qui se correspondent entre la topographie et la topographie de référence (UrbDTM_GRID10m.tif). La coordonnée X du vecteur de translation correspondra au reste de la division entière de la différence entre la coordonnée X de l'angle Nord-Ouest de la grille de référence et de la coordonnée X de l'angle Nord-Ouest du raster de topographie, par la valeur en X de la résolution (idem pour Y). Si la valeur obtenue est inférieure à la moitié de la résolution, il s'agit de la bonne valeur. Sinon, il faut la soustraire à la valeur de la résolution. Le signe du vecteur dépend du sens que l'on veut donner à la translation (X positif vers l'Est, négatif vers l'Ouest, Y positif vers le Nord, négatif vers le Sud).

Valeur lue de l'angle Nord-Ouest de la grille UrbDTM_GRID10m.tif :

XminR = 140000.00000 ; YmaxR = 179000.000000

Valeur lue de l'emprise spatiale de la topographie intermédiaire obtenue à l'étape 1.3 :

XminT = 20949.9821984663722105 ; YmaxT = 250022.4901855140924454

Calcul du vecteur de translation à appliquer pour aligner les grilles (le symbole % permet de renvoyer le reste par la division entière) :

XminR – XminT = 119050.01780153363 / YmaxR – YmaxT = -71022.49018551409

DX = (XminR – XminT) % 10 = 0.017801533627789468

DY = (YmaxR – YmaxT) % 10 = - 7.509814485907555

DY étant supérieur à 5 (résolution divisée par 2), on veut en fait :

DY = 10 - |DY| = 10 - 7.509814485907555 = 2.4901855140924454

En observant les pixels sur QGIS, on détermine que la translation doit être vers le Sud-Est

(donc la coordonnée en X doit être positive et en Y doit être négative).

On obtient les coordonnées de la translation qui sera appliquée à la topographie

Xtrans = 0.017801533627789468 ; Ytrans = - 2.4901855140924454

- 1.4.b : Afin d'aligner le raster de topographie intermédiaire avec la grille de référence, on utilise l'outil « Aligner les rasters » de QGIS, en utilisant comme couche de référence la grille_de_reference.tif.

Raster → Aligner les rasters…

Couche raster à aligner : (1) grille_de_reference.tif / Nom du fichier en sortie : peu importe (à supprimer ensuite) / Méthode de rééchantillonnage : Au plus proche voisin // Couche raster à aligner : (2) topo_intermediaire.tif / Nom du fichier en sortie : Topo_lidar_SRTM80_ajust.tif / Méthode de rééchantillonnage : Au plus proche voisin // Couche de référence : grille_de_reference.tif / Cocher Découpe à étendre / Sélectionner Emprise de la couche / Laisser les autres paramètres par défaut.

Le raster ainsi obtenu **Topo_lidar_SRTM80_ajust.tif** est le raster de topographie qui sera utilisé pour la suite des calculs. Il contient les valeurs de topographie du MNT de départ, mais alignées et découpées à l'emprise de la grille de référence.

1.5 Construction d'un Shapefile contenant une zone tampon de 500 m autour de la RBC

L'étape 1.5 consiste à extraire le contour de la RBC de la couche WFS « Re » d'Urbis, de lui appliquer une zone tampon de 500m et d'enregistrer la géométrie ainsi obtenue dans un nouveau shapefile. Cette géométrie servira par la suite à limiter l'extension des données. La zone tampon créée a été construite avec une segmentation de 30 pour l'approximation.

Vecteur → Outils de géotraitement → Tampon(s)…

Couche vectorielle de saisie : « Re » / Segments pour l'approximation : 30 / Distance tampon : 500 / Fichier en sortie : RBD_tampon500_30.shp / Laisser les autres paramètres par défaut.

VI.3. UTILISATION DES SCRIPTS

L'utilisation des scripts se fait via un environnement de développement (Spyder, IDLE Python ou autre) afin de pouvoir définir manuellement les données en entrée. En effet, chaque fichier de script est décomposé en plusieurs sections :

- Un entête contenant le nom du script, un descriptif, son auteur et la date de création ;
- Un code de connexion au package Brugepy ;
- L'import des modules et des fonctions ;
- Un descriptif des actions effectuées par le Script ;
- Les paramètres à définir ;
- Le script en lui-même.

Lors de l'utilisation des scripts, il est donc possible de définir les données à traiter dans la section « PARAMETRES A DEFINIR ». C'est là que l'utilisateur peut indiquer le chemin des dossiers ou des rasters à traiter, la valeur de certaines variables ou encore indiquer si certaines actions doivent être effectuées ou non. Les autres sections ne doivent pas être modifiées.

Une fois ces variables définies, l'utilisateur peut lancer le script.

Recommandations :

- Pour l'utilisation des scripts (excepté « harmonisation_strati.py »), il est important que les jeux de rasters utilisés en entrée soient harmonisés (même format, même emprise spatiale, même SCR, même résolution, même valeur de 'nodata'). Si ça n'est pas le cas, cela risque de générer des erreurs ;
- Le fichier CSV (Annexe VI.1) permet de communiquer à Python de nombreuses informations telles que l'ordre stratigraphique, la correspondance entre les rasters de toit et rasters d'épaisseurs ou encore le nombre de forage ayant servi à construire les rasters de toits. Pour remplir ce rôle, sa structure est primordiale. La valeur de chaque cellule peut être modifiée, mais l'ajout, la suppression ou la modification de l'ordre des colonnes nécessitera en revanche d'adapter les scripts Pythons qui font appel à ce fichier ;
- Lors de l'utilisation des scripts faisant appel au fichier CSV, bien vérifier la cohérence entre les données du fichier CSV et les noms des fichiers. Entre chacune des 4 grandes phases, il peut être parfois utile de renommer manuellement les rasters ;
- Pour un ensemble de rasters donné (ex : les rasters des toits de couches corrigés) toujours créer un dossier dédié contenant uniquement ces rasters. En effet, les scripts Python parcourent les dossiers dans leur entièreté. Si d'autres fichiers sont présents dans le dossier traité, ils risquent de parasiter les résultats.

VI.4. SCRIPTS ET MODULES COMMENTES

VI.4.1. Scripts

VI.4.1.1. harmoniser_strati.py

```
# -*- coding: utf-8 -*-

# harmoniser_strati.py

# Script permettant d'harmoniser automatiquement l'ensemble des rasters d'un
# dossier, en leur redéfinissant un format et un SRC commun et en alignant
# leur extension et leur résolution sur une grille de référence.
# Utilise notamment les fonctions du module manip.py de Brugepy.
# Auteur : LOUIS GAUDARE - Bruxelles Environnement
# Date : avril 2018

# ---------------CONNECTION AU PACKAGE BRUGEPY (si nécessaire)-----------------
import sys
import os
PACKAGE_PARENT = '..'
SCRIPT_DIR = os.path.dirname(os.path.realpath(os.path.join(os.getcwd(),
                             os.path.expanduser(__file__))))
PACKAGE_DIR = os.path.normpath(os.path.join(SCRIPT_DIR, PACKAGE_PARENT))
PACKAGE_DIR = os.path.normpath(os.path.join(PACKAGE_DIR, PACKAGE_PARENT))
if PACKAGE_DIR not in sys.path:
    sys.path.append(PACKAGE_DIR)

# ---------------IMPORT DES MODULES / FONCTIONS------------------

from path import Path
from brugepy.raster.manip import conversion_format
from brugepy.raster.manip import reprojection, transformation
from brugepy.raster.manip import calculer_translation
from brugepy.raster.manip import decoup_raster
from brugepy.raster.manip import redimensionner

# ---------------------PARAMETRES A DEFINIR---------------------

# Chemin du dossier contenant les rasters à traiter
dossier_travail = 'D:\\Travail\\Traitements\\Entrees\\Model_strati'

# Chemin du dossier vers lequel sont envoyés les rasters intermédiaires
# (ceux-ci sont automatiquement supprimés au fur et à mesure du script)
dossier_temporaire = 'D:\\Travail\\Traitements\\Temp'

# Chemin du dossier vers lequel sont envoyés les fichiers finaux
dossier_sortie = 'D:\\Travail\\Traitements\\Sorties\\Strati_uniform'

# Format voulu pour les fichiers finaux
nouveau_format = 'GTiff'

# SCR voulu pour les rasters finaux
nouveau_SCR = 'EPSG:31370'

# Chemin du raster servant de grille de référence pour l'harmonisation
# Les rasters à harmoniser auront la même extension spatiale que cette grille.
canevas =
'D:\\Travail\\Traitements\\Entrees\\Canevas\\Canevas_Topo_lidar_srtm801_ajust.tif'

# Vecteur de translation à appliquer à l'ensemble du modèle pour faire
# correspondre la topographie du modèle à la topographie de référence
# (calculé à l'étape 1.4.a des traitements manuels)
# Si aucune translation ne doit être effectuée, mettre appliquer_translat=False
# Xtrans est la coordonnée en X (Ouest-Est) du vecteur
```

```
# Ytrans est la coordonnée en Y (Sud-Nord) du vecteur
appliquer_translat = True
Xtrans = -(152849.9821983573201578-152850.000000000000000000000)
Ytrans = -(171242.4901852159528062-171240.000000000000000000000)

# Résolution souhaitée pour les rasters.
# Si l'on veut garder la résolution initiale des rasters,
# mettre  forcer_resolution = False.
# rexX : résolution en X
# resY : résolution en Y
forcer_resolution = True
resX = 10.0
resY = 10.0

# Canal contenant les données dans les rasters en entrée (doit être le même
# pour tous les rasters)
canal = 1

# Polygone de découpage des rasters (RBC + tampon 500m)
# (Construit manuellement à l'étape 1.5)
# En dehors de ce polygone, les pixels prendront la valeur de 'nodata'
shp_decoup = 'D:\\Travail\\Traitements\\Entrees\\SHP_RBC\\RBC_tampon500_30.shp'

# --------------------DEBUT DU SCRIPT-------------------------

# Construction d'une liste des chemins des rasters à traiter
liste_fichiers_init = []

dossier_parent = Path(dossier_travail)

# Dans le cas de BruStrati3D on cherche tous les fichiers nommés hdr.adf
# qui correspondent aux rasters à traiter
for i in dossier_parent.walkfiles("hdr.adf"):

    directory, name = os.path.split(i)

    nom_fichier = os.path.basename(directory)

    print(nom_fichier)

    nom_fichier_ext = nom_fichier + format(name)

    print(nom_fichier_ext)

    chemin_fichier = os.path.join(directory, nom_fichier_ext)

    liste_fichiers_init.append(chemin_fichier)

print(liste_fichiers_init)

for fichier_entree in liste_fichiers_init:

    # Construction des noms des rasters intermédiaires et du raster final
    name_fichier = os.path.basename(os.path.dirname(fichier_entree))

    name_fichier_temp01 = name_fichier + '_temp01' + format(".tif")

    name_fichier_temp02 = name_fichier + '_temp02' + format(".tif")

    name_fichier_temp03 = name_fichier + '_temp03' + format(".tif")

    name_fichier_temp04 = name_fichier + '_temp04' + format(".tif")
```

```
    name_fichier_temp05 = name_fichier + '_temp05' + format(".tif")

    name_fichier_sortie = name_fichier + '_lamb72_final' + format(".tif")

    fichier_temp01 = os.path.join(dossier_temporaire, name_fichier_temp01)

    fichier_temp02 = os.path.join(dossier_temporaire, name_fichier_temp02)

    fichier_temp03 = os.path.join(dossier_temporaire, name_fichier_temp03)

    fichier_temp04 = os.path.join(dossier_temporaire, name_fichier_temp04)

    fichier_temp05 = os.path.join(dossier_temporaire, name_fichier_temp05)

    fichier_sortie = os.path.join(dossier_sortie, name_fichier_sortie)

    # Conversion du format des rasters
    conversion_format(fichier_entree, fichier_temp01, nouveau_format)

    # Reprojection des rasters pour harmoniser le SCR
    reprojection(fichier_temp01, fichier_temp02, nouveau_SCR)

    # Suppression du fichier temporaire 1
    os.remove(fichier_temp01)

    # Première translation des rasters avec la même translation que celle
    # appliquée à la topographie (la topographie a été translatée
    # 'manuellement' sur QGIS hors de ce script – cf. étapes 1.4.a et 1.4.b)
    transformation(fichier_temp02, fichier_temp03,
                   appliquer_translat, Xtrans, Ytrans,
                   forcer_resolution, resX, resY)

    # Suppression du fichier temporaire 2
    os.remove(fichier_temp02)

    # Calcul de la transformation à appliquer pour aligner les rasters à la
    # grille de référence
    (transX, transY) = calculer_translation(fichier_temp03, canevas,
                                            resX, resY)

    # Aligne le raster à la grille de référence à l'aide de la transformation
    # calculée précédemment
    transformation(fichier_temp03, fichier_temp04, True, transX, transY)

    # Suppression du fichier temporaire 3
    os.remove(fichier_temp03)

    # Découpe le raster à l'aide des contours de la RBC + 500m
    decoup_raster(fichier_temp04, canal, shp_decoup, fichier_temp05)

    # Suppression du fichier temporaire 4
    os.remove(fichier_temp04)

    # Redimensionne le raster pour que son extension corresponde à la grille de
    # référence
    redimensionner(fichier_temp05, canevas, fichier_sortie)

    # Suppression du fichier temporaire 5
    os.remove(fichier_temp05)

    print(name_fichier_sortie+' créé dans le dossier Sortie')

print('script effectué')
```

VI.4.1.2. corriger_ensemble_strati.py

```
# -*- coding: utf-8 -*-

# corriger_ensemble_strati.py

# Script permettant de corriger automatiquement l'ensemble des raster d'un
# dossier.
# Utilise notamment les fonctions du module coorection.py de Brugepy.
# Auteur : LOUIS GAUDARE - Bruxelles Environnement
# Date : avril 2018

# ---------------CONNECTION AU PACKAGE BRUGEPY (si nécessaire)-----------------
import sys
import os
PACKAGE_PARENT = '..'
SCRIPT_DIR = os.path.dirname(os.path.realpath(os.path.join(os.getcwd(),
                             os.path.expanduser(__file__))))
PACKAGE_DIR = os.path.normpath(os.path.join(SCRIPT_DIR, PACKAGE_PARENT))
PACKAGE_DIR = os.path.normpath(os.path.join(PACKAGE_DIR, PACKAGE_PARENT))
if PACKAGE_DIR not in sys.path:
    sys.path.append(PACKAGE_DIR)

# ---------------IMPORT DES MODULES / FONCTIONS------------------
import shutil
from brugepy.raster.correction import corriger_rasters

# Ce script analyse l'ensemble des rasters d'un dossier, effectue les
# corrections lorsqu'il y a des incohérences et copie les rasters corrigés
# (et non corrigés lorsqu'aucune correction n'a été nécessaire) dans un
# dossier en sortie.

# Incohérence : lorsqu'une couche a des valeurs de pixels supérieures à une
# couche située au-dessus dans la stratigraphie.

# Pour connaitre l'ordre stratigraphique, un fichier CSV recensant les
# couches dans l'ordre stratigraphique est donné en entrée. Pour chaque
# couche, ce fichier CSV contient le nom du raster en 5ème colonne (sans chemin
# ni extension).

# Enfin lorsque deux couches se croisent, ce sera la couche qui a été
# construite avec le plus de données de forages qui servira à corriger l'autre.
# Le nombre de forages est aussi stocké dans le fichier CSV, en 4ème colonne.

# IMPORTANT : l'ensemble des rasters utilisés dans ce script doivent
# préalablement avoir été harmonisés (même SCR, même résolution, même extension
# spatiale, même valeur de 'nodata').

# --------------------PARAMETRES A DEFINIR--------------------

# Chemin du dossier contenant les rasters à corriger
dossier_entree = 'D:\\Travail\\Traitements\\Entrees\\Corrections\\Layers'

# Chemin du dossier où seront envoyés les fichiers temporaires (non supprimés
# automatiquement en fin d'exécution du script)
dossier_temp = 'D:\\Travail\\Traitements\\Temp'

# Dossier où seront envoyés les rasters finaux corrigés de chaque unité
stratigraphique
dossier_sortie = 'D:\\Travail\\Traitements\\Sorties\\Strati_corrigee_2'

# Chemin du fichier CSV recensant les couches géologiques dans l'ordre
# stratigraphique et donnant pour chacune le nom du raster de toit (sans chemin
```

```
# sans extension) en 5ème colonne et le nombre de forages en 4ème colonne.
classeur_couches = 'D:\\Travail\\Traitements\\Entrees\\Classeur_formations_topo.csv'

# ----------- SCRIPT----------------------------------------

# Dictionnaire qui à chaque nom de raster associe une liste d'éléments
# concernant ce raster. Cette liste contient :
# [numéro de la formation, lettre de la formation, nom de la formation,
# nombre de forages, booléen indiquant si la formation a été corrigée ou non]
dico_formations = {}

# Liste des noms des rasters dans l'ordre du fichier CSV
liste_formations = []

# Dictionnaire donnant pour chaque nom de raster la dernière version de
# modification (située dans le dossier temporaire)
dico_formations_modifiees = {}

# Lecture du fichier CSV et construction de la liste des formations
# et du dico_formations
with open(classeur_couches, 'r') as fichier:

    fichier.readline()

    for ligne in fichier:

        ligne = ligne[:-1]

        ligne_liste = ligne.split(';')

        num = ligne_liste[0]
        lettre = ligne_liste[1]
        nom = ligne_liste[2]
        forage = ligne_liste[3]
        nom_fich = ligne_liste[4]

        dico_formations[nom_fich] = [num, lettre, nom, forage, False]

        liste_formations.append(nom_fich)

# Parcours de la liste des formations
for nA in range(len(liste_formations)-1):

    formation_A = liste_formations[nA]

    # Récupération du nombre de forages de la formation A
    num_for_A = int(dico_formations[formation_A][3])

    # Parcours des formations situées après formation_A dans la liste
    for nB in range(nA+1, len(liste_formations)):

        formation_B = liste_formations[nB]

        # Récupération du nombre de forages de la formation B
        num_for_B = int(dico_formations[formation_B][3])

        # Récupération des booléens indiquant si les formations ont déjà été
        # modifiées ou non
        bool_A = dico_formations[formation_A][4]

        bool_B = dico_formations[formation_B][4]

        # Si les formations ont déjà été modifiées, on utilise la dernière
```

```
        # version de la formation pour tester et effectuer les corrections
        if bool_A is True:

            raster_A = dossier_temp+'\\'+dico_formations_modifiees[formation_A]

        # Sinon on utilise le raster initial
        else:

            raster_A = dossier_entree+'\\'+formation_A+'.tif'

        # idem pour la formation B
        if bool_B is True:

            raster_B = dossier_temp+'\\'+dico_formations_modifiees[formation_B]

        # idem pour la formation B
        else:

            raster_B = dossier_entree+'\\'+formation_B+'.tif'

        # Application des corrections au raster ayant le plus petit nombre
        # de forages
        if num_for_A >= num_for_B:

            modif, name_m, new_name = corriger_rasters(raster_A, raster_B,
                                                       dossier_temp,
                                                       formation_A,
                                                       formation_B, 1)

            # Si une correction a été effectuée (True) : mise à jour du
            # booléen de la formation et du dictionnaire des formations
            # modifiées
            if modif is True:

                dico_formations[formation_B][4] = True

                dico_formations_modifiees[name_m] = new_name

        else:

            print('A<B')

            modif, name_m, new_name = corriger_rasters(raster_A, raster_B,
                                                       dossier_temp,
                                                       formation_A,
                                                       formation_B, 2)

            if modif is True:

                dico_formations[formation_A][4] = True

                dico_formations_modifiees[name_m] = new_name

# Copie des rasters dans le dossier de sortie final

# Parcours l'ensemble des formations
for forma in liste_formations:

    bool_forma = dico_formations[forma][4]

    # Si la formation n'a pas été modifiée, copie du fichier d'origine
    if bool_forma is False:
```

```
        anc_rast = dossier_entree+'\\'+forma+'.tif'
        new_rast = dossier_sortie+'\\'+forma+'.tif'

    # Si elle a été modifiée, copie de la dernière version corrigée
    else:

        anc_rast = dossier_temp+'\\'+dico_formations_modifiees[forma]
        new_rast = dossier_sortie+'\\'+dico_formations_modifiees[forma]

    shutil.copy(anc_rast, new_rast)

print('sript_effectué')
```

VI.4.1.3. rasters_epaisseurs.py

```
# -*- coding: utf-8 -*-

# rasters_epaisseurs.py

# Script permettant la construction des rasters d'épaisseurs des formations
# géologiques.
# Utilise notamment les fonctions du module epaisseur.py de Brugepy.
# Auteur : LOUIS GAUDARE - Bruxelles Environnement
# Date : avril 2018

# ---------------CONNECTION AU PACKAGE BRUGEPY (si nécessaire)-----------------
import sys
import os
PACKAGE_PARENT = '..'
SCRIPT_DIR = os.path.dirname(os.path.realpath(os.path.join(os.getcwd(),
                             os.path.expanduser(__file__))))
PACKAGE_DIR = os.path.normpath(os.path.join(SCRIPT_DIR, PACKAGE_PARENT))
PACKAGE_DIR = os.path.normpath(os.path.join(PACKAGE_DIR, PACKAGE_PARENT))
if PACKAGE_DIR not in sys.path:
    sys.path.append(PACKAGE_DIR)

# ---------------IMPORT DES MODULES / FONCTIONS------------------
from brugepy.raster.epaisseur import calculer_epaisseur_totale

# Ce script calcule et retourne sous forme de raster, pour chaque formation,
# l'épaisseur des formations à partir des rasters des toits contenus dans
# un seul et même dossier.

# Pour connaitre l'ordre stratigraphique des couches, un fichier CSV est
# donné en entrée avec le nom des fichiers tif correspondant à chaque couche,
# et où les couches sont indiquées dans l'ordre stratigraphique

# Important : l'ensemble des rasters utilisés en entrée de ce script doivent
# avoir été préalablement harmonisés et corrigés.

# Important : le toit de la dernière formation (paléozoïque) doit
# couvrir l'ensemble de la zone d'étude (sinon, les épaisseurs des
# couches sus-jacentes risquent de ne pas être calculées en certains endroits)

# Pour calculer l'épaisseur du Quaternaire, utiliser comme toit la
# topographie découpée à la zone d'étude

# --------------------PARAMETRES A DEFINIR--------------------

# Chemin du dossier contenant les rasters de toits des formations.
dossier_entree = 'D:\\Travail\\Traitements\\Entrees\\Strati_corrigee'

# Chemin du dossier qui contiendra les rasters d'épaisseurs en sortie
dossier_sortie = 'D:\\Travail\\Traitements\\Sorties\\Epaisseurs'
```

```
# Fichier CSV donnant, dans l'ordre stratigraphique, les noms des fichiers
# rasters de toit corrigés (6ème colonne - sans chemin ni extension).
classeur_couches = 'D:\\Travail\\Traitements\\Entrees\\Classeur_formations_topo.csv'

# completer_zero indique si les 'nodata' sont remplies avec des 0 (True)
# ou non (False)
completer_zero = True

# Indique si les épaisseurs nulles sont définies comme sans données (True)
zero_is_nodata = True

# ----------- SCRIPT----------------------------------------
val_nodat = None
if zero_is_nodata is True:
    val_nodat = 0

# Liste des noms des rasters dans l'ordre du fichier CSV
liste_formations = []

liste_epaisseurs = []

dico_noms_forma = {}

# Lecture du fichier CSV et construction de la liste des formations
# et du dico_formations (étape 3.2.a)
with open(classeur_couches, 'r') as fichier:

    fichier.readline()

    for ligne in fichier:

        ligne = ligne[:-1]

        ligne_list = ligne.split(';')

        nom = ligne_list[2]

        forage = ligne_list[3]

        nom_fich = ligne_list[5]

        chemin_fich = dossier_entree+'\\'+nom_fich+'.tif'

        liste_formations.append(chemin_fich)

        print(liste_formations)

        dico_noms_forma[chemin_fich] = nom_fich

# Parcours de la liste des formations (étape 3.2.b)
for nA in range(len(liste_formations)-1):

    formation_A = liste_formations[nA]

    print(formation_A)

    liste_form_underA = []

    rast_sortie = dossier_sortie+'\\'+dico_noms_forma[formation_A]+'_epa.tif'

    liste_epaisseurs.append(rast_sortie)
```

```
    # Construction de la liste des formations situées après formation_A
    # (étape 3.2.c)
    for nB in range(nA+1, len(liste_formations)):

        liste_form_underA.append(liste_formations[nB])

    # Calcule de l'épaisseur de la formation_A (étape 3.3)
    calculer_epaisseur_totale(formation_A, liste_form_underA, rast_sortie,
                              completer_zero, val_nodat)

print('Script effectué')
```

VI.4.1.4. corriger_extension_toits_epaisseurs.py

```
# -*- coding: utf-8 -*-

# corriger_extension_toits_epaisseur.py

# Script permettant de supprimer des rasters de toits les pixels
# où l'épaisseur est nulle.
# Utilise notamment les fonctions du module correction.py de Brugepy.
# Auteur : LOUIS GAUDARE - Bruxelles Environnement
# Date : juillet 2018

# ---------------CONNECTION AU PACKAGE BRUGEPY (si nécessaire)-----------------
import sys
import os
PACKAGE_PARENT = '..'
SCRIPT_DIR = os.path.dirname(os.path.realpath(os.path.join(os.getcwd(),
                             os.path.expanduser(__file__))))
PACKAGE_DIR = os.path.normpath(os.path.join(SCRIPT_DIR, PACKAGE_PARENT))
PACKAGE_DIR = os.path.normpath(os.path.join(PACKAGE_DIR, PACKAGE_PARENT))
if PACKAGE_DIR not in sys.path:
    sys.path.append(PACKAGE_DIR)

# ---------------IMPORT DES MODULES / FONCTIONS------------------
import shutil
import rasterio
from brugepy.raster.correction import delete_by_mask

# Ce script permet de supprimer les pixels où l'épaisseur est nulle des rasters
# d'altitude des toits du modèle stratigraphique. Cela évite de visualiser des
# extensions erronées des couches géologiques en faisant apparaître des pixels
# avec valeurs alors que l'épaisseur est nulle.

# IMPORTANT : l'ensemble des rasters utilisés dans ce script doivent
# préalablement avoir été harmonisés (même SCR, même résolution, même extension
# spatiale, même valeur de 'nodata').

# --------------------PARAMETRES A DEFINIR--------------------

# Chemin du dossier contenant les rasters à corriger
dossier_toit = 'D:\\Travail\\Traitements\\Entrees\\Corrections\\Toits_corrig'

# Chemin du dossier contenant les rasters d'épaisseur
dossier_epai = 'D:\\Travail\\Traitements\\Entrees\\Corrections\\Epaisseurs'

# Chemin du dossier où seront envoyés les fichiers temporaires (non supprimés
# automatiquement en fin d'exécution du script)
dossier_sort = 'D:\\Travail\\Traitements\\Sorties\\Strati_extensions_corrig'

# Suffixe à ajouter au nom en sortie (string)
suf = ''
```

```
# Valeur d'épaisseur définissant les pixels à supprimer des toits.
val_suppr = 0

# Valeur des nodata, si l'on veut la changer. Laisser sur None pour garder
# la valeur de nodata par défaut des rasters en entrée.
forced_noval = None

# Chemin du fichier CSV recensant les couches géologiques dans l'ordre
# stratigraphique et donnant pour chacune le nom du raster de toit corrige
# en 6ème colonne et le raster d'épaisseur en 7ème colonne.
classeur_couches = 'D:\\Travail\\Traitements\\Entrees\\Classeur_formations_topo.csv'

# ----------- SCRIPT----------------------------------------

# Liste des noms des rasters de toit dans l'ordre du fichier CSV
liste_toits = []

# Liste des noms des rasters d'épaisseur dans l'ordre du fichier CSV
liste_epa = []

# Lecture du fichier CSV et construction de la liste des toits des formations et
# de la liste des épaisseurs des formations, ordonnées de la même manière
with open(classeur_couches, 'r') as fichier:

    fichier.readline()

    for ind, ligne in enumerate(fichier):

        ligne = ligne[:-1]

        ligne_liste = ligne.split(';')

        nom_toit = ligne_liste[5]
        nom_epa = ligne_liste[6]

        # Ceci permet de ne pas corriger la première couche (Quaternaire), qui
        # doit avoir une extension couvrant l'ensemble de la RBC, même si son
        # épaisseur est nulle (pour les cartes de conductivités thermiques)
        if ind == 0:
            nom_epa = ''

        liste_toits.append(nom_toit)

        liste_epa.append(nom_epa)

# Parcours des formations pour appliquer la correction de l'extension
for toit, epai in zip(liste_toits, liste_epa):

    rast_t = dossier_toit + '\\' + toit + '.tif'
    rast_e = dossier_epai + '\\' + epai + '.tif'
    rast_s = dossier_sort + '\\' + toit + suf + '.tif'

    if not epai == '':
        with rasterio.open(rast_e, 'r') as rasterepa:

            mat_epa = rasterepa.read(1)

        mat_mask = mat_epa == val_suppr

        delete_by_mask(rast_t, mat_mask, rast_s, forced_noval)

        print(toit+' a été modifié')
```

```
    else:
        shutil.copy(rast_t, rast_s)

        print(toit+' dupliqué')

print('sript_effectué')
```

VI.4.2. Modules de Brugepy

VI.4.2.1. Brugepy/raster/manip.py

```
# -*- coding: utf-8 -*-

# manip.py

# Fonctions de traitement de rasters qui ont permis l'harmonisation des
# rasters de la stratigraphie de BruStrati3D
# Auteur : LOUIS GAUDARE - Bruxelles Environnement
# Date : avril 2018

import rasterio
from rasterio.warp import calculate_default_transform, reproject, Resampling
import rasterio.transform as rt
import rasterio.mask
import fiona
import math
import rasterio.merge as rm


# Fonction créant une copie d'un raster dans le format souhaité
# raster_in : chemin du raster en entrée
# raster_out : chemin du nouveau raster
# new_format : driver du format souhaité
def conversion_format(raster_in, raster_out, new_format):

    with rasterio.open(raster_in, 'r') as rasterInit:

        # Récupération des métadonnées du raster en entrée
        kwargs = rasterInit.meta.copy()

        # Mise à jour du driver pour le raster en sortie
        kwargs.update({'driver': new_format})

        # Ecriture (canal par canal) du nouveau raster
        with rasterio.open(raster_out, 'w', **kwargs) as rasterFin:

            for i in range(1, rasterInit.count + 1):

                rasterFin.write(rasterInit.read(i), i)


# Fonction créant une copie d'un raster dans un nouveau SCR
# raster_in : chemin du raster en entrée
# raster_out : chemin du nouveau raster
# dst_csr : SCR souhaité
def reprojection(raster_in, raster_out, dst_crs):

    with rasterio.open(raster_in, 'r') as src:

        # calcul de la matrice de transformation (objet Affine) et des
        # dimensions (en pixels) du nouveau raster
        transform, width, height = calculate_default_transform(
                src.crs, dst_crs, src.width, src.height, *src.bounds)
```

```
        # Récupération des métadonnées du raster en entrée
        kwargs = src.meta.copy()

        # Mise à jour des métadonnées pour le raster en sortie
        kwargs.update({
                'crs': dst_crs,
                'transform': transform,
                'width': width,
                'height': height
                })

        # Ecriture (canal par canal) du nouveau raster à l'aide des métadonnées
        # mises à jour
        with rasterio.open(raster_out, 'w', **kwargs) as dst:
            for i in range(1, src.count + 1):
                reproject(
                        source=rasterio.band(src, i),
                        destination=rasterio.band(dst, i),
                        src_transform=src.transform,
                        src_crs=src.crs,
                        dst_transform=transform,
                        dst_crs=dst_crs,
                        resampling=Resampling.nearest)
    return True


# Fonction créant un nouveau raster ayant subi des transformations à partir
# d'un raster source. Les transformations possibles sont : translation et
# agrandissement de la taille des pixels (change donc aussi l'emprise)
# raster_in : chemin du raster en entrée
# raster_out : chemin du nouveau raster
# translat : True si on veut effectuer une translation, False sinon
# translatX : vecteur X de translation
# translatY : vecteur Y de translation
# chg_resol : True si on veut modifier la taille du pixel, False sinon
# resolX : nouvelle taille en X du pixel
# resolY : nouvelle taille en Y du pixel
def transformation(raster_in, raster_out,
                   translat=True, translatX=0, translatY=0,
                   chg_resol=False, resolX=10, resolY=10):

    with rasterio.open(raster_in, 'r') as rasterInit:

        # Obtention des coordonnées du pixel en haut à gauche
        (x0, y0) = rasterInit.affine*(0, 0)

        # Obtention du nombre de pixels en longueur et en largeur
        (xWidthPix, yHeightPix) = (rasterInit.width, rasterInit.height)

        # Obtention des coordonnées du pixel en bas à droite
        (xDR, yDR) = rasterInit.affine*(xWidthPix, yHeightPix)

        # Calcul de la résolution
        (resX, resY) = ((xDR-x0)/xWidthPix, (y0-yDR)/yHeightPix)

        (newResX, newResY) = (resX, resY)

        # Si l'utilisateur veut imposer une résolution
        if chg_resol is True:

            # Prise en compte de la résolution définie par l'utilisateur
            (newResX, newResY) = (resolX, resolY)
```

```
        # Récupération des vecteurs de translation
        transX = translatX
        transY = translatY

        # Vecteurs de translations égaux à 0 si translat est False
        if translat is False:
            transX = 0
            transY = 0

        # Calcul la matrice de transformation (Affine) du nouveau raster
        # à l'aide des coordonnées de l'angle en haut à gauche du raster
        # translaté et de la résolution calculée ou définie.
        transform = rt.from_origin(x0+transX, y0+transY,
                                   newResX, newResY)

        # Ecriture du nouveau raster à l'aide de cette matrice de
        # transformation
        with rasterio.open(raster_out, 'w',
                           driver=rasterInit.driver,
                           height=rasterInit.height,
                           width=rasterInit.width,
                           count=rasterInit.count,
                           dtype=rasterInit.meta['dtype'],
                           crs=rasterInit.crs,
                           transform=transform,
                           nodata=rasterInit.nodata) as rasterFin:

            for i in range(1, rasterInit.count + 1):

                rasterFin.write(rasterInit.read(i), i)


# Fonction créant une copie d'un raster découpé à l'aide d'un polygone contenu
# dans un shapefile.
# raster_in : chemin du raster en entrée
# Canal du raster à découper
# shapefile_zone : chemin du shapefile de découpe
# raster_out : chemin du nouveau raster
def decoup_raster(raster_in, band, shapefile_zone, raster_out, cro=True,
                  novalue=None):

    # Ouverture du raster et du shapefile en entrée en mode lecture
    rasterData = rasterio.open(raster_in,
                               'r')

    parcelle = fiona.open(shapefile_zone,
                          'r')

    # Récupération de la géométrie de la forme de découpe
    formeParcelle = [parcelle[0]["geometry"]]

    # Récupération de la matrice (out_image) et des données de tranformation
    # (out_transforme) du raster extrait par masque. all_touched permet de
    # définir comment les pixels sont gérés en bordure du découpage. Si
    # all_touched=True, tous les pixels touchés par la forme de découpage
    # sont conservés. Si all_touched=false, seuls ceux dont le centre se
    # trouve à l'intérieur de la forme de découpage sont conservés.
    # crop permet de définir si l'extension du raster est ajustée à la
    # forme de découpage (crop=True) ou si elle reste identique à celle
    # du raster initial (crop=False).
    out_image, out_transform = rasterio.mask.mask(rasterData, formeParcelle,
                                                  nodata=novalue,
                                                  all_touched=True, crop=cro)
```

```
    # Récupération des métadonnées du raster initial puis adaptation pour
    # les faire correspondre au raster issu du masque.
    out_meta = rasterData.meta.copy()

    out_meta.update({"driver": "GTiff",
                     "height": out_image.shape[1],
                     "width": out_image.shape[2],
                     "transform": out_transform})

    if novalue is not None:
        out_meta.update({"nodata": novalue})

    # Ecriture du nouveau raster issu de l'extraction par masque
    with rasterio.open(raster_out,
                       "w", **out_meta, filled=False) as sortie:
        sortie.write(out_image)

    # Fermeture des fichiers ouverts
    rasterData.close()
    parcelle.close()

    # La fonction renvoie True si tout s'est bien passé
    return True


# Fonction calculant le plus petit vecteur de translation possible pour que
# les pixels de deux rasters en entrée soient parfaitement alignés.
# Retourne une tuple (X, Y) contenant les coordonnées de ce vecteur.
# A noter : les deux rasters doivent avoir la même résolution et le même SCR.
# raster_in : chemin du raster que l'on voudra translater
# raster_ref : chemin du raster vers lequel on veut translater
# resolX : résolution en X des rasters
# resolY : résolution en Y des rasters
def calculer_translation(raster_in, raster_ref, resolX, resolY):

    with rasterio.open(raster_in, 'r') as rasterInit:

        # récupération des coordonnées de l'angle Nord-Ouest du 1er raster
        (xi0, yi0) = rasterInit.affine*(0, 0)

        with rasterio.open(raster_ref, 'r') as rasterRef:

            # récupération des coordonnées de l'angle Nord-Ouest du raster à
            # déplacer
            (xr0, yr0) = rasterRef.affine*(0, 0)

            # calcul de la distance horizontale et verticale entre les angles
            # Nord-Ouest des deux rasters
            distGauche = math.sqrt((xi0 - xr0)**2)

            distHaut = math.sqrt((yi0 - yr0)**2)

            # calcul de la distance horizontale et verticale entre les angles
            # Nord-Ouest entre les pixels des deux rasters
            dGauche = distGauche % resolX

            dHaut = distHaut % resolY

            # calcul du vecteur de translation en fonction des multiples
            # positions relatives des deux rasters
            demiResX = resolX/2
            demiResY = resolY/2
```

```
            if xi0 == xr0:
                transX = 0
            elif xi0 > xr0:
                if dGauche <= demiResX:
                    transX = -dGauche
                else:
                    transX = resolX-dGauche
            else:
                if dGauche >= demiResX:
                    transX = -(resolX-dGauche)
                else:
                    transX = dGauche
            if yi0 == yr0:
                transY = 0
            elif yi0 < yr0:
                if dHaut <= demiResY:
                    transY = dHaut
                else:
                    transY = -(resolY-dHaut)
            else:
                if dHaut >= demiResY:
                    transY = resolY-dHaut
                else:
                    transY = -dHaut
            return (transX, transY)

# Fonction créant un nouveau raster à partir d'un raster source en adaptant son
# emprise à un raster de référence.
# raster_in : chemin du raster que l'on veut adapter
# raster_ref : chemin du raster servant de grille de référence
# raster_out : chemin du nouveau raster
def redimensionner(raster_in, raster_ref, raster_out):
    with rasterio.open(raster_in, 'r') as rasterInit:
        liste_datasets = [rasterInit]
```

```
with rasterio.open(raster_ref, 'r') as rasterRef:

    # Récupération des limites du raster de référence
    bounds_box = rasterRef.bounds

    # Utilisation détournée de la fonction « merge » pour créer une nouveau
    # raster adapté aux limites du raster de référence
    array_fin, affine_fin = rm.merge(liste_datasets,
                                     bounds=bounds_box)

    # Ecriture du nouveau raster
    with rasterio.open(raster_out, 'w',
                       driver=rasterInit.driver,
                       height=rasterRef.height,
                       width=rasterRef.width,
                       count=rasterInit.count,
                       dtype=rasterInit.meta['dtype'],
                       crs=rasterInit.crs,
                       transform=affine_fin,
                       nodata=rasterInit.nodata) as rasterFin:

        for i in range(0, rasterInit.count):
            rasterFin.write(array_fin[i], i+1)
```

VI.4.2.2. Brugepy/raster/correction.py

```
# -*- coding: utf-8 -*-

# correction.py

# Fonctions de traitement de rasters qui ont permis la correction des
# rasters de la stratigraphie de BruStrati3D
# Auteur : LOUIS GAUDARE - Bruxelles Environnement
# Date : avril 2018

import rasterio
import numpy as np
import numpy.ma as ma
import os
from ..calculator.calculatrice import calculatrice_rasters_alignes


# Fonction qui, à partir de deux rasters en entrée, crée un nouveau raster qui
# correspond à l'un des deux rasters en entrée dont les valeurs des pixels
# incohérents ont été remplacées par les valeurs de l'autre raster.
# Sont considérés comme incohérents les pixels pour lesquels la différence
# entre la valeur du premier raster (raster_up) et du deuxième (raster_down)
# est négative. Cela corrige les problèmes de toits de couches qui se croisent.
# Si priorite=1, c'est le raster down qui est corrigé par le raster_up.
# Si priorité=2 ce sera l'inverse.
#
# -------------- Entrées---------------------
# raster_up : raster de la couche géologique la plus haute
# raster_down : raster de la couche géologique la plus basse
# dossier_out : dossier où sera écrit le raster en sortie
# nom_up : nom remplaçant le nom de la couche raster_up
#          (utile uniquement pour le nom du fichier en sortie).
#           Si aucun nom n'est entré la fonction utilisera le nom de raster_up
# nom_down :nom de la couche raster_down  (idem)
# priorite : (1) raster_up corrige raster_down (les valeurs de raster_up sont
#              envoyées vers raster_down. (2) inverse.
#
# -------------- Sorties---------------------
# - True si une correction a été effectuée, False sinon
# - Nom du raster qui a été modifié
# - Nom du raster en sortie
#
# -------------- IMPORTANT---------------------
# Pour utiliser cette fonction il faut que les rasters en entrée
# aient la même extension spatiale, la même résolution, le même SCR
# et le même format.
def corriger_rasters(raster_up, raster_down, dossier_out,
                     nom_up='', nom_down='', priorite=1):

    # Calcule le raster de différence entre la couche sup. et la couche inf.
    # et récupère la matrice de valeurs de cette différence
    lr = [raster_up, raster_down]

    mat_diff, mat_mask = calculatrice_rasters_alignes(lr, dossier_out, '(A-B)',
                                                      only_matrice=True)

    # Masque les zones sans données de la matrice de différences
    mat_diff_masked = ma.masked_array(mat_diff, mat_mask)

    # Calcule si il y a des valeurs de la couche inférieure (raster_down) qui
    # sont plus élevées que celles de la couche supérieure (raster_up) pour des
    # pixels situés au même endroit
    mini = mat_diff_masked.min()
```

```
test_min_mask = bool(mat_mask.min())

# Si aucune valeur n'est incohérente, pas besoin de correction.
# La fonction retourne False et les chemins de deux rasters en entrée
if mini >= 0:

    return False, raster_up, raster_down

# Si aucun des pixels contenant des valeurs des deux rasters ne se
# superpose, idem, pas besoin de correction.
if test_min_mask is True:

    return False, raster_up, raster_down

# Sinon (des pixels se superposent et il y a une erreur) :
# Récupération d'une matrice de masque représentant les zones où il y a une
# incohérence.
mask_error_no_value = mat_diff_masked < 0

mask_error = mask_error_no_value.filled(False)

# Définition du raster à modifier et de celui qui fournira les valeurs
if priorite == 2:
    rast_prioritaire = raster_down
    rast_a_modifer = raster_up

    name_prio = nom_down
    name_modif = nom_up

else:
    rast_prioritaire = raster_up
    rast_a_modifer = raster_down

    name_prio = nom_up
    name_modif = nom_down

# Définition du nom du raster en sortie
if name_prio == '':

    name_prio, ext_p = os.path.basename(rast_prioritaire).split('.')

if name_modif == '':

    name_modif, ext_m = os.path.basename(rast_a_modifer).split('.')

name_out = name_modif+'_corrige_par_'+name_prio+'.tif'

raster_out = dossier_out+'/'+name_out

# Lecture des deux rasters
with rasterio.open(rast_prioritaire, 'r') as couche_prio:

    with rasterio.open(rast_a_modifer, 'r') as couche_modif:

        mat_prio = couche_prio.read(1)

        mat_modif = couche_modif.read(1)

        # La matrice de valeurs du raster à modifier est masquée par le
        # masque des valeurs incohérentes (qui contient True lorsqu'une
        # valeur est incohérente)
        mat_modif_masked = ma.masked_array(mat_modif, mask_error)
```

```
            # Les valeurs incohérentes sont remplacées par les valeurs
            # du raster prioritaire
            mat_corrige = mat_modif_masked.filled(mat_prio)

            # Ecriture du raster en sortie
            kwargs = couche_modif.meta.copy()

            with rasterio.open(raster_out, 'w', **kwargs) as raster_corrig:

                raster_corrig.write(mat_corrige, 1)

    return True, name_modif, name_out


# Fonction qui à partir d'un raster et d'une matrice mask ayant les mêmes
# proportions que le raster, construit un raster où les données des pixels
# masqués du premier raster sont effacées ('nodata').
#
# -------------- Entrées---------------------
# - rast_toit : chemin du raster duquel on veut effacer des valeurs.
#
# - mask : matrice booléenne servant de masque, contenant True au niveau des
# pixels que l'on veut effacer et False au niveau de ceux que l'on ne veut pas
# effacer. Il est important que les dimensions de cette matrice soient les
# même que la matrice de rast_toit.
#
# - rast_out : chemin du raster qui sera construit.
#
# - noval : Si l'on veut imposer une valeur pour les pixels sans données.
# Si aucune valeur n'est entrée, la valeur de 'nodata' du rast_toit est
# utilisée.
#
# -------------- Sorties---------------------
# - mat_toit : matrice de valeurs du raster avec les pixels supprimés.
#
# - mask : matrice de masque ayant servi à supprimer les valeurs
#
# -------------- IMPORTANT---------------------
# Pour utiliser cette fonction il faut que la matrice mask soit une matrice
# booléenne, ayant les mêmes dimensions que le raster rast_toit, et contenant
# True au niveau des pixels à définir 'sans données'.
def delete_by_mask(rast_toit, mask, rast_out, noval=None):

    # Lecture du raster d'entrée. Récupération de sa matrice et des métadonnées.
    with rasterio.open(rast_toit, 'r') as toit:

        mat_toit = toit.read(1)
        meta = toit.meta.copy()

    # Si une valeur de 'nodata' est imposée : adaptation de la matrice mask
    # pour qu'elle prenne en compte les pixels déjà sans donnée du raster
    # en entrée. Mise à jour de la valeur de 'nodata' des métadonnées.
    if noval is not None:

        mask_noval = mat_toit == meta['nodata']

        mask_noval = mask_noval == 0

        mask_inv = np.invert(mask)

        mask = np.invert(mask_inv * mask_noval)
```

```
        meta['nodata'] = noval

    # Masque la matrice de données
    mat_toit_masked = ma.masked_array(mat_toit, mask)

    # Rempli les pixels masqués avec la valeur de 'nodata'
    mat_toit = mat_toit_masked.filled(meta['nodata'])

    # Ecriture du raster en sortie
    with rasterio.open(rast_out, 'w', **meta) as sortie:

        sortie.write(mat_toit, 1)

    return mat_toit, mask
```

VI.4.2.3. Brugepy/raster/epaisseur.py

```
# -*- coding: utf-8 -*-

# epaisseur.py

# Fonctions permettant de construire les épaisseurs des couches stratigraphiques
# Auteur : LOUIS GAUDARE - Bruxelles Environnement
# Date : avril 2018

import numpy as np
import numpy.ma as ma
import rasterio
import rasterio.mask
from ..calculator.calculatrice import calculatrice_rasters_alignes


# calcul_epaisseur calcul les épaisseurs entre deux couches et met à jours
# une matrice d'épaisseur (mat_Ep) de la première couche (couche_A) sur
# la zone où une différence entre les deux couches a été calculée.
# Elle renvoie la matrice d'épaisseurs mise à jour ainsi qu'une matrice de mask
# booléenne où False représente les pixels où l'épaisseur a déjà été
# calculée et True les pixels où l'épaisseur est encore à calculer.
# A noter : si il s'agit du premier calcul d'épaisseurs pour une couche donnée
# laisser "mat_Ep" ET "mask_EpACalc" vides. Cela permet de
# calculer une première matrice d'épaisseurs et un premier masque.
# Sinon, entrer la matrice d'épaisseur et le masque obtenus précédemment.
# -------------- Entrées---------------------
# - couche_A : chemin du raster du toit de la couche dont on veut calculer
#              l'épaisseur
# - couche_B : chemin du raster du toit de la couche sous-jacente permettant
#              de calculer l'épaisseur
# - mat_Ep : matrice d'épaisseur de la couche A à mettre à jour.
# - mask_EpACalc : matrice masque de la matrice d'épaisseur contenant True au
#                  niveau des pixels où l'épaisseur n'a pas encore été
#                  calculée et False où l'épaisseur a déjà été calculée.
# - nodata_zeo : permet de choisir si les pixels sans données sont remplis
#               de 0 (True), ou si ils gardent la valeur de "nodata" par
#               défaut (False).
# -------------- Sorties---------------------
# new_matrice_Ep : matrice d'épaisseur mise à jour
# new_mask_Ep : masque de la matrice d'épaisseur mis à jour
#
# -------------- IMPORTANT---------------------
# IMPORTANT : pour utiliser cette fonction il faut que les rasters en entrée
# aient tous la même extension spatiale, la même résolution, le même SCR
# et le même format.
def calculer_epaisseur(couche_A, couche_B, mat_Ep=[], mask_EpACalc=[],
                       nodata_zero=False):
```

```
mat_diff, mask_diff = calculatrice_rasters_alignes([couche_A,
                                                    couche_B], '',
                                                   '(A-B)',
                                                   only_matrice=True)

# mask_diff est False lorsque une différence a été calculée
# et True lorsqu'aucune différence n'a été calculée

# mask_diff_inv a True lorsqu'une différence a été calculée
# et True lorsqu'aucune différence n'a été calculée
mask_diff_inv = np.invert(mask_diff)

# Matrice des épaisseurs
matrice_Ep = mat_Ep

# Masque avec False sur les pixels où l'épaisseur a déjà été calculée et
# True au niveau des pixels où l'épaisseur est encore à calculer
mask_Ep = mask_EpACalc

if (matrice_Ep == []) and (mask_Ep == []):

    with rasterio.open(couche_A, 'r') as rastInit:

        maskInit = rastInit.read_masks(1)

        # Le mask_Ep est initialisé avec l'ensemble de l'étendu de la
        # couche à calculer (False) et les zones où la couche n'existe pas
        # (no data) étant considérées comme déjà calculées (True)
        mask_no_value = maskInit < 255

        mask_Ep = maskInit > 0

        matInit = rastInit.read(1)

        matInit_uni = matInit*0-9999

        matInit_uni_masked = ma.masked_array(matInit_uni, mask_no_value)

        no_data = rastInit.meta.copy()['nodata']

        if nodata_zero is True:

            no_data = 0

        # La matrice des épaisseurs est initialisée comme une matrice
        # contenant la valeur 'no_data' lorsque le toit de la couche
        # n'existe pas et -9999 là où la couche existe (et où il faut
        # calculer une épaisseur)
        matrice_Ep = matInit_uni_masked.filled(no_data)

# mask_Ep_a_ajouter : matrice avec True au niveau des zones où l'épaisseur
# est à mettre à jour et False où ça n'est pas le cas
mask_Ep_a_ajouter = mask_Ep * mask_diff_inv

# Masque la matrice des épaisseurs. Masque les pixels "à remplir".
matrice_Ep_masked = ma.masked_array(matrice_Ep, mask_Ep_a_ajouter)

# Rempli les pixels masqués avec les valeurs d'épaisseurs calculées
new_matrice_Ep = matrice_Ep_masked.filled(mat_diff)

# Mise à jour du mask des épaisseurs restant à calculer
# (avec True si pixel à calculer, et False sinon)
```

```
    new_mask_Ep = mask_Ep * mask_diff

    return new_matrice_Ep, new_mask_Ep


# calculer_epaisseur_totale calcule l'épaisseur d'une couche sur l'ensemble
# de son extension en prenant en compte l'ensemble des toits des couches
# sous-jacentes.
# En plus de créer le raster d'épaisseurs, elle retourne la matrice des
# épaisseurs et la matrice de masque où l'épaisseur n'a pas été calculée.
#
# -------------- Entrées---------------------
# - couche_top : chemin du raster du toit de la couche dont on veut calculer
# l'épaisseur.
# - liste_couche_under : liste des toits de couches se situant
# stratigraphiquement sous la couche étudiée, ordonnée de la couche la plus
# haute à la couche la plus profonde.
# - raster_out : chemin du raster à écrire en sortie, qui contiendra
# l'épaisseur.
# - nodata_to_zero : si False, les pixels sans données prendront
# la valeur de no_data de couche_top. Si True, les valeurs sans données
# prendront la valeur 0. False par défaut.
# -------------- Sorties---------------------
# - mat_epaisseur : matrice d'épaisseur finale
# - mat_mask : matrice de masque finale prenant la valeur True au niveau des
# pixels où aucune épaisseur n'a été calculée et False où les pixels ont été
# calculés.
# -------------- IMPORTANT---------------------
# IMPORTANT : pour utiliser cette fonction il faut que les rasters en entrée
# aient tous la même extension spatiale, la même résolution, le même SCR
# et le même format.
def calculer_epaisseur_totale(couche_top, liste_couche_under, raster_out,
                              nodata_to_zero=False, val_nodata=None):

    # Initialisation des matrices d'épaisseurs et de masque
    mat_epaisseur = []
    mat_masque = []

    # Parcoure les couches sous-jacentes
    for couche_down in liste_couche_under:

        # Pour chaque couche, mise à jour de la matrice d'épaisseur et de mask
        # au niveau des pixels où couche_down représente la base de couche_top.
        mat_epaisseur, mat_masque = calculer_epaisseur(couche_top,
                                                       couche_down,
                                                       mat_epaisseur,
                                                       mat_masque,
                                                       nodata_to_zero)

    # Ecriture du raster d'épaisseur
    with rasterio.open(couche_top, 'r') as rasterInit:

        kwargs = rasterInit.meta.copy()

    if val_nodata is not None:

        kwargs['nodata'] = val_nodata

    with rasterio.open(raster_out, 'w', **kwargs) as rasterOut:

        rasterOut.write(mat_epaisseur, 1)

    return mat_epaisseur, mat_masque
```

VI.4.2.4. Brugepy/calculator/calculatrice.py

```
# -*- coding: utf-8 -*-

# calculatrice.py

# Fonctions permettant d'effectuer des opérations entre rasters
# Auteur : LOUIS GAUDARE - Bruxelles Environnement
# Date : avril 2018

import rasterio
import rasterio.mask
from .sousformule import Sous_formule
import numpy as np
import numpy.ma as ma
import os


# Crée un nouveau raster à partir d'un calcul effectué sur une liste de
# rasters, à l'aide d'une formule.
#
# -------------- Entrées---------------------
# - liste_raster : liste des rasters en entrée
#                  (ex: ['\dossier\raster1.tif', '\dossier\raster2.tif', ...])
# - raster_out : chemin du raster en sortie
# - formule de calcul : Chaque raster est représenté par une lettre majuscule
#   en fonction de son ordre dans la liste_raster. Par ailleurs la formule
#   et chaque sous-formule de la formule générale DOIT être encadrée de
#   parenthèses. Il s'agit d'une chaine de caractères.
#   ex : liste_entée : [raster1, raster2, raster3, raster4]
#       On veut faire le calcul (raster3-raster1)/raster2+raster4
#       La formule est '(((C-A)/B)+D)'
#   Une sous-formule est donc toujours uniquement composée des éléments
#   suivants, dans l'ordre suivant, sans espace :
#       --> Une parenthèse ouvrante
#       --> Une premier membre qui peut être une lettre majuscule (pour un
#           raster) ou une autre sous-formule
#       --> Un opérateur +, -, / ou *
#       --> Un second membre qui peut être une lettre majuscule (pour un
#           raster) ou une autre sous-formule
#       --> Une parenthèse fermante
#
# - canal : canal à calculer. Doit-être le même pour tous les rasters.
# - only_matrice : si True, la fonction ne créera pas de raster en sortie
#   mais retournera uniquement la matrice des valeurs calculées et la matrice
#   de mask.
#
# -------------- Sorties---------------------
# matrice_result_final : matrice calculée
# matrice_mask_invert : matrice de mask avec True pour les pixels sans valeur
# et false pour les pixels avec valeur.
#
# -------------- IMPORTANT---------------------
# IMPORTANT : pour utiliser cette fonction il faut que les rasters en entrée
# aient tous la même extension spatiale, la même résolution, le même SCR
# et le même format.
# IMPORTANT : les opérations possibles sont '+', '-', '/' et '*'
# A noter : le raster final aura des valeurs uniquement au niveau des pixels ayant une
valeur dans tous les rasters donnés en entrée.
# (la somme d'un pixel sans donnée et d'un pixel avec donnée donnera un
# pixel sans donnée).
def calculatrice_rasters_alignes(liste_raster, raster_out,
                                 formule, canal=1, only_matrice=False):
```

```
# INITIALISATION
# Dictionnaire qui donnera l'équivalence entre une lettre ('A') et le
# chemin vers un raster
dico_raster = {}

# kwargs prendra les métadonnées du premier raster de la liste. Elles
# seront utilisées pour écrire le raster en sortie.
kwargs = {}

# matrice_mask sera une matrice booléenne avec False pour les zones sans
# sans valeur et True pour les zones avec valeur. Il s'agira d'une
# combinaison des masks de l'ensemble des rasters en entrée.
matrice_mask = []

# Valeur des pixels sans donnée
no_value = 0

# Parcours la liste des rasters en entrée pour construire
# le dictionnaire dico_raster et la matrice_mask
for n in range(0, len(liste_raster)):

    rast = liste_raster[n]

    with rasterio.open(rast) as raster_open:

        # Lecture de la matrice de valeur du raster
        matrice = raster_open.read(canal)

        # Obtient la lettre correspondant (chr(65) = 'A', 66='B'...)
        dico_raster[chr(65+n)] = matrice

        # Mask du raster (avec 0 si sans donnée et 255 si donnée)
        matriceMask = raster_open.read_masks(canal)

        # Conversion en mask booléen
        matriceMaskBool = matriceMask > 0

        # Si on lit le premier raster...
        if n == 0:

            # Récupération des métadonnées
            kwargs = raster_open.meta.copy()

            # Récupération de la valeur de 'nodata'
            no_value = kwargs['nodata']

            # La matrice de mask combinée n'est pour l'instant égale qu'à
            # celle de ce premier raster
            matrice_mask = matriceMaskBool

        # Pour les rasters suivant...
        else:
            # On ajoute le mask de ce raster à la combinaison des masks des
            # précédents rasters
            matrice_mask = matrice_mask*matriceMaskBool

# Effectue le calcul à l'aide de la formule et de dico_raster.
matrice_result = calculer_formule(formule, dico_raster)

# Inverse la matrice de mask (pour obtenir True au niveau des 'nodata'
# False au niveau des pixels avec données)
matrice_mask_invert = np.invert(matrice_mask)
```

```
    # Masque la matrice de résultats avec le masque des zones sans données
    matrice_result_masked = ma.masked_array(matrice_result,
                                            matrice_mask_invert)

    # Rempli les zones sensées être sans données avec la valeur de no_value
    matrice_result_final = matrice_result_masked.filled(no_value)

    # Si only_matrice == False, création d'un fichier raster
    if only_matrice is False:

        with rasterio.open(raster_out, 'w', **kwargs) as rasterFin:

            rasterFin.write(matrice_result_final, 1)

    return matrice_result_final, matrice_mask_invert


# Déchiffre une formule donnée en format texte pour la décomposer en
# sous-formules stockées dans un dictionnaire et ordonnées dans une liste.
# ex : entrée = '(((A+B)/C)-(D*E))
#      sorties :  dico_indice_formule : {0 : sous_formule[True,1,-,True,3],
#                                        1 : sous_formule[True,2,/,False,C],
#                                        2 : sous_formule[False,A,+,False,B],
#                                        3 : sous_formule[False,D,+,False,E]}
#                 liste_indice_ordre : [2,1,3,0] --> les sous-formules devront
#                                      être calculées dans cet ordre
# A noter : dans les sous-formule décomposées, les booléens indiquent si le
# membre correspondant est une autre sous-formule (True) ou un raster (False).
def evaluer_formule(formule):

    # Liste des opérateurs possibles dans une formule
    liste_operateurs = ['+', '-', '*', '/']

    # INITIALISATION
    dico_indice_formule = {}

    liste_indice_ordre = []

    nombre_formules = 0

    formule_en_cours = 0

    formule_prec = 0

    # Parcours un à un les caractères de la formule
    for car in formule:

        # Si le caractère est un opérateur, mise à jour de l'opérateur de la
        # formule en cours
        if car in liste_operateurs:

            dico_indice_formule[formule_en_cours].maj_operateur(car)

        # Si le caractère est '(', on traite une nouvelle sous-formule.
        # Celle-ci est initialisée et ajoutée au dictionnaire des sous-formules
        elif car == '(':

            formule_prec = formule_en_cours

            formule_en_cours = nombre_formules

            nombre_formules += 1
```

```
            sous_formule = Sous_formule()

            dico_indice_formule[formule_en_cours] = sous_formule

            if nombre_formules > 1:

                dico_indice_formule[formule_prec].maj_membre(True,formule_en_cours)

                dico_indice_formule[formule_en_cours].maj_indice_parent(formule_prec)

        # Si le caractère est ')', c'est que la sous-formule en cours est
        # terminée. L'ordre des ')' donne aussi l'ordre dans lequel les
        # sous-formules doivent être traitées.
        elif car == ')':

            liste_indice_ordre.append(formule_en_cours)

            dico_indice_formule[formule_en_cours].maj_position(formule_en_cours)

            formule_en_cours = formule_prec
            if formule_en_cours != 0:
                formule_prec = dico_indice_formule[formule_en_cours].indice_parent

        # Si il y a un espace, ne pas prendre en compte
        elif car == ' ':

            True

        # Tous les autres caractères que ceux précédemment testés sont considérés
        # comme membres de sous-formules. Ils sont injectés dans les
        # sous-formules en cours de traitement
        else:
            dico_indice_formule[formule_en_cours].maj_membre(False, car)

    return dico_indice_formule, liste_indice_ordre


# Calcule une formule donnée sous format texte à l'aide d'un dictionnaire
# donnant l'équivalence entre les lettres majuscules et les valeurs.
def calculer_formule(formule_a_calculer, dictio={}):

    # Déchiffre la formule pour savoir dans quel ordre traiter chacune
    # des sous-formules
    dico_formules, liste_ordre = evaluer_formule(formule_a_calculer)

    dico_resultats = {}

    # Calcule les sous-formules une à une en convertissant leurs membres donnés
    # sous formes de lettres en éléments (chiffres ou matrices)
    # grâce au dictionnaire dictio
    for num_formule in liste_ordre:

        form = dico_formules[num_formule]

        if form.membre_01_is_formule is True:
            form.maj_membre_calc(1, dico_resultats[form.val_membre01])

        if form.membre_02_is_formule is True:
            form.maj_membre_calc(2, dico_resultats[form.val_membre02])

        form.convertir_formule(dictio)
        dico_resultats[num_formule] = form.calculer()
```

```
    # Le résultat final de la formule est le résultat de la sous-formule
    # d'indice 0
    return dico_resultats[0]
```

VI.4.2.5. Brugepy/calculator/sousformule.py

```
# -*- coding: utf-8 -*-

# sousformule.py

# Ce fichier contient la classe Sous_formule.
# Auteur : LOUIS GAUDARE - Bruxelles Environnement
# Date : avril 2018

import numpy


# Sous_formule est un objet représentant une formule avec un premier membre,
# un opérateur, et un second membre (ex : A+B).
# -------------- Entrées---------------------
# - liste_sous_formule : une liste [booléen du premier membre, premier membre,
#                                   opérateur, booléen du second membre,
#                                   second membre]
#                       ex : [False, A, +, False, B]
#   - les booléens des membres prennent la valeur True seulement si le membre
#   correspondant est une sous-formule. Lorsque les deux membres ont comme
#   booléen False, la formule est directement calculable.
#   - les membres peuvent être des entiers, des réels ou des matrices numpy
#     Si le membre est une autre sous-formule, il prend un entier
#     qui représente la position de la sous-formule dans la formule générale
#
#   ex : ((A+B)*C) aura comme sous-formule : [True, 1, *, False, C]
#        avec 1 faisant référence à la sous-formule [False, A, +, False, B]
# - position : la position de la sous-formule dans la formule générale
# - indice_parent : la position de la formule parent dans la formule générale
#
# ex :
# Formule générale : (((A+B)/C)-D)
# sous_formule 0 : [True, 1, -, False, D], position 0, indice parent -1
# sous_formule 1 : [True, 2, /, False, C], position 1, indice parent 0
# sous_formule 2 : [False, A, +, False, B], position 2, indice parent 1
# A noter : dans les sous_formule 0 et 1, les 1 et 2 dans la liste font
# référence aux sous-formules 1 et 2 respectivement.
class Sous_formule:

    def __init__(self, liste_sous_formule=[False, '0', '+', False, '0'],
                 position=0, indice_parent=-1):

        # Booléen du premier membre
        self.membre_01_is_formule = liste_sous_formule[0]

        # Valeur du premier membre (int, float, matrice...)
        self.val_membre01 = liste_sous_formule[1]

        # Valeur de l'opérateur (chaine de texte. ex : '+')
        self.operateur = liste_sous_formule[2]

        # Booléen du second membre (int, float, matrice...)
        self.membre_02_is_formule = liste_sous_formule[3]

        # Valeur du second membre (int, float, matrice...)
        self.val_membre02 = liste_sous_formule[4]
```

```
        # Booléen prenant True si le premier membre a été mis à jour en dernier
        # et False si c'est le second
        self.premier_membre = False

        # Position de la Sous-formule dans la formule générale
        self.position = position

        # Position de la Sous-formule parent
        self.indice_parent = indice_parent

        # Valeur "calculée" du premier membre (si le premier membre est une
        # sous-formule, cette variable prendra la valeur obtenue par le calcul
        # de cette sous-formule)
        self.val_membre01_calc = liste_sous_formule[1]

        # Valeur "calculée" du second membre (idem)
        self.val_membre02_calc = liste_sous_formule[4]

    # Calcule la formule et retourne la valeur calculée.
    # Attention : pour faire appel à cette méthode, il faut que les membres
    # aient été préalablement calculés si il s'agit de sous-formules.
    def calculer(self):

        if self.operateur == '+':

            resultat = self.val_membre01_calc + self.val_membre02_calc

        elif self.operateur == '-':

            resultat = self.val_membre01_calc - self.val_membre02_calc

        elif self.operateur == '*':

            resultat = self.val_membre01_calc * self.val_membre02_calc

        elif self.operateur == '/':

            resultat = self.val_membre01_calc / self.val_membre02_calc

        else:

            return False

        return resultat

    # Modifier un membre : change la valeur et le booléen d'un membre. Si le
    # dernier membre à avoir été modifié était le premier, la méthode
    # modifiera le second et inversement. Si aucun membre n'a déjà été modifié,
    # la méthode modifiera le premier.
    def maj_membre(self, bouleen, valeur):

        if self.premier_membre is False:

            self.membre_01_is_formule = bouleen
            self.val_membre01 = valeur
            self.val_membre01_calc = valeur

            self.premier_membre = True

        else:
            self.membre_02_is_formule = bouleen
            self.val_membre02 = valeur
            self.val_membre02_calc = valeur
```

```
        self.premier_membre = False

# Modifie la valeur "calculée" d'un membre. Si en entrée membre == 1,
# la méthode modifie le premier membre, si == 2 modifie le second
def maj_membre_calc(self, membre, valeur):

    if membre == 1:
        self.val_membre01_calc = valeur

    if membre == 2:
        self.val_membre02_calc = valeur

# Modifie l'opérateur
def maj_operateur(self, oper):

    self.operateur = oper

# Modifie la position
def maj_position(self, posi):

    self.position = posi

# Modifie l'indice parent (Attention : n'apporte aucune modification à
# la sous-formule parent)
def maj_indice_parent(self, indice):

    self.indice_parent = indice

# Modifie les membres "calculés" de la sous-formule à l'aide
# d'un dictionnaire donné en entrée.
# ex : sous-formule (A+B) : (False,'A', '+', False, 'B')
#   'A' et 'B' sont des chaines de texte.
#   On peut alors convertir la formule à l'aide du dictionnaire suivant
# {'A':matriceA, 'B':matriceB} où matriceA et matriceB sont des objets
# matrice numpy. Alors la sous-formule deviendra : (matriceA+matriceB)
# La conversion ne marche pas sur les membres qui sont eux-mêmes des
# sous-formules.
def convertir_formule(self, dico={}):

    if self.membre_01_is_formule is False:

        if self.val_membre01 in dico.keys():

            self.val_membre01_calc = dico[self.val_membre01]

        else:
            conversion = float(self.val_membre01_calc)
            self.val_membre01_calc = conversion

    if self.membre_02_is_formule is False:

        if self.val_membre02 in dico.keys():

            self.val_membre02_calc = dico[self.val_membre02]

        else:
            conversion = float(self.val_membre02_calc)
            self.val_membre02_calc = conversion

# Fonction d'affichage de la sous-formule
def afficher(self):
```

```
print('position : '+str(self.position))
print(self.membre_01_is_formule, self.membre_02_is_formule)
print(str(self.val_membre01)+str(self.operateur)+str(self.val_membre02))
print('')
```

VI.5. PACKAGES INSTALLÉS DANS L'ENVIRONNEMENT « GIS » D'ANACONDA

Obtenu via **Anaconda prompt** à l'aide de la commande suivante : conda list -n gis.

# Name	Version	Build	Channel
affine	2.1.0	pyh128a3a6_1	
alabaster	0.7.10	py36hcd07829_0	
asn1crypto	0.24.0	py36_0	
astroid	1.6.1	py36_0	
attrs	17.4.0	py36_0	
babel	2.5.3	py36_0	
bleach	2.1.3	py36_0	
bokeh	0.12.14	py36_0	
boto3	1.5.32	py36_0	
botocore	1.8.46	py36_0	
ca-certificates	2017.08.26	h94faf87_0	
cartopy	0.16.0	py36hbd42bde_0	
certifi	2018.1.18	py36_0	
cffi	1.11.5	py36h945400d_0	
chardet	3.0.4	py36h420ce6e_1	
click	6.7	py36hec8c647_0	
click-plugins	1.0.3	py36_0	
cligj	0.4.0	py36_0	
cloudpickle	0.5.2	py36_1	
colorama	0.3.9	py36h029ae33_0	
cryptography	2.1.4	py36he1d7878_0	
curl	7.58.0	h7602738_0	
cycler	0.10.0	py36h009560c_0	
decorator	4.2.1	py36_0	
descartes	1.1.0	py36_0	
docutils	0.14	py36h6012d8f_0	
entrypoints	0.2.3	py36hfd66bb0_2	
expat	2.2.5	hcc4222d_0	
fiona	1.7.10	py36h5bf8d1d_0	
freetype	2.8	h51f8f2c_1	
freexl	1.0.4	h342dbcb_5	
gdal	2.2.2	py36hcebd033_1	
geopandas	0.3.0	py36_0	
geos	3.6.2	h9ef7328_2	
hdf4	4.2.13	h712560f_2	
hdf5	1.10.1	h98b8871_1	
html5lib	1.0.1	py36h047fa9f_0	
icc_rt	2017.0.4	h97af966_0	
icu	58.2	ha66f8fd_1	
idna	2.6	py36h148d497_1	
imagesize	1.0.0	py36_0	
intel-openmp	2018.0.0	hd92c6cd_8	
ipykernel	4.8.2	py36_0	
ipython	6.2.1	py36h9cf0123_1	
ipython_genutils	0.2.0	py36h3c5d0ee_0	
isort	4.3.4	py36_0	
jedi	0.11.1	py36_0	
jinja2	2.10	py36h292fed1_0	
jmespath	0.9.3	py36h0745840_0	
jpeg	9b	hb83a4c4_2	
jsonschema	2.6.0	py36h7636477_0	
jupyter_client	5.2.2	py36_0	
jupyter_core	4.4.0	py36h56e9d50_0	
kealib	1.4.7	ha5b336b_5	
kiwisolver	1.0.1	py36h12c3424_0	
krb5	1.14.2	h63dfc2a_6	
lazy-object-proxy	1.3.1	py36hd1c21d2_0	
libboost	1.65.1	he51fdeb_4	
libcurl	7.58.0	h7602738_0	
libgdal	2.2.2	h2727f2b_1	
libiconv	1.15	h1df5818_7	
libkml	1.3.0	hc65d273_3	
libnetcdf	4.4.1.1	h825a56a_8	
libpng	1.6.34	h79bbb47_0	
libpq	9.6.6	hfe3f2bf_0	
libspatialindex	1.8.5	h6538335_2	
libspatialite	4.3.0a	h383548d_18	
libssh2	1.8.0	hd619d38_4	
libtiff	4.0.9	h0f13578_0	
libxml2	2.9.7	h79bbb47_0	
libxslt	1.1.32	hf6f1972_0	
lxml	4.1.1	py36hef2cd61_1	
m2w64-gcc-libgfortran	5.3.0	6	
m2w64-gcc-libs	5.3.0	7	
m2w64-gcc-libs-core	5.3.0	7	
m2w64-gmp	6.1.0	2	
m2w64-libwinpthread-git	5.0.0.4634.697f757	2	
markupsafe	1.0	py36h0e26971_1	
matplotlib	2.2.0	py36h4dabdea_0	
mccabe	0.6.1	py36hb41005a_1	
mistune	0.8.3	py36_0	
mkl	2018.0.1	h2108138_4	
msys2-conda-epoch	20160418	1	
munch	2.2.0	py36_0	
nbconvert	5.3.1	py36h8dc0fde_0	
nbformat	4.4.0	py36h3a5bc1b_0	
networkx	2.1	py36_0	
notebook	5.4.0	py36_0	
numpy	1.14.1	py36hb69e940_2	
numpydoc	0.7.0	py36ha25429e_0	
olefile	0.45.1	py36_0	
openjpeg	2.2.0	h29c51c3_2	
openssl	1.0.2o	h8ea7d77_0	
owslib	0.16.0	py36_0	
packaging	17.1	py36_0	
pandas	0.22.0	py36h6538335_0	
pandoc	1.19.2.1	hb2460c7_1	
pandocfilters	1.4.2	py36h3ef6317_1	
parso	0.1.1	py36hae3edee_0	
path.py	11.0	py36_0	
patsy	0.5.0	py36_0	
pickleshare	0.7.4	py36h9de030f_0	
pillow	5.0.0	py36h0738816_0	
pip	9.0.1	py36_5	
proj4	4.9.3	hcf24537_7	
prompt_toolkit	1.0.15	py36h60b8f86_0	
psutil	5.4.3	py36hfa6e2cd_0	
psycopg2	2.7.4	py36h74b6da3_0	
py4j	0.10.6	py36_0	
pycodestyle	2.3.1	py36h7cc55cd_0	
pycparser	2.18	py36hd053e01_1	
pyepsg	0.3.2	py36h6c744c8_0	
pyflakes	1.6.0	py36h0b975d6_0	
pygments	2.2.0	py36hb010967_0	
pylint	1.8.2	py36_0	
pyopenssl	17.5.0	py36h5b7d817_0	
pyparsing	2.2.0	py36h785a196_1	
pyproj	1.9.5.1	py36_0	

pyqt 5.6.0 py36hb5ed885_5
pysal 1.14.2 py36h583e8e2_1
pyshp 1.2.12 py36h9b5cf0b_0
pysocks 1.6.8 py36_0
pyspark 2.3.0 py36_0
python 3.6.4 h6538335_1
python-dateutil 2.6.1 py36h509ddcb_1
pytz 2018.3 py36_0
pywinpty 0.5.1 py36_0
pyyaml 3.12 py36h1d1928f_1
pyzmq 17.0.0 py36hfa6e2cd_0
qt 5.6.2 vc14h6f8c307_12
qtawesome 0.4.4 py36h5aa48f6_0
qtconsole 4.3.1 py36h99a29a9_0
qtpy 1.3.1 py36hb8717c5_0
rasterio 0.36.0 py36hb8ea33a_1
requests 2.18.4 py36h4371aae_1
rope 0.10.7 py36had63a69_0
rtree 0.8.3 py36_0
s3transfer 0.1.13 py36_0
scikit-learn 0.19.1 py36h53aea1b_0
scipy 1.0.0 py36h1260518_0
send2trash 1.5.0 py36_0
setuptools 38.5.1 py36_0
shapely 1.6.4 py36h2a969d5_0
simplegeneric 0.8.1 py36_2
sip 4.18.1 py36h9c25514_2
six 1.11.0 py36h4db2310_1
snowballstemmer 1.2.1 py36h763602f_0
snuggs 1.4.1 py36hd271b8d_0
sphinx 1.7.1 py36_0
sphinxcontrib 1.0 py36hbbac3d2_1
sphinxcontrib-websupport 1.0.1 py36hb5e5916_1
spyder 3.2.7 py36_0
sqlalchemy 1.2.4 py36hfa6e2cd_0
sqlite 3.22.0 h9d3ae62_0
statsmodels 0.8.0 py36h6189b4c_0
terminado 0.8.1 py36_1
testpath 0.3.1 py36h2698cfe_0
tk 8.6.7 hcb92d03_3
tornado 5.0 py36_0
traitlets 4.3.2 py36h096827d_0
typing 3.6.4 py36_0
urllib3 1.22 py36h276f60a_0
vc 14 h0510ff6_3
vs2015_runtime 14.0.25123 3
wcwidth 0.1.7 py36h3d5aa90_0
webencodings 0.5.1 py36h67c50ae_1
wheel 0.30.0 py36h6c3ec14_1
win_inet_pton 1.0.1 py36he67d7fd_1
wincertstore 0.2 py36h7fe50ca_0
winpty 0.4.3 4
wrapt 1.10.11 py36he5f5981_0
xerces-c 3.2.0 h44c76bb_2
xz 5.2.3 h7c615d8_2
yaml 0.1.7 hc54c509_2
zlib 1.2.11 h8395fce_2