LIÈGE université
Library

https://lib.uliege.be

https://matheo.uliege.be

Mémoire

Auteur : Plancke, Jonas
Promoteur(s) : Kasprzyk, Jean-Paul
Faculté : Faculté des Sciences
Diplôme : Master en sciences géographiques, orientation géomatique, à finalité spécialisée en geodata-expert
Année académique : 2023-2024
URI/URL : http://hdl.handle.net/2268.2/21463

Avertissement à l'attention des usagers :

Tous les documents placés en accès ouvert sur le site le site MatheO sont protégés par le droit d'auteur. Conformément aux principes énoncés par la "Budapest Open Access Initiative"(BOAI, 2002), l'utilisateur du site peut lire, télécharger, copier, transmettre, imprimer, chercher ou faire un lien vers le texte intégral de ces documents, les disséquer pour les indexer, s'en servir de données pour un logiciel, ou s'en servir à toute autre fin légale (ou prévue par la réglementation relative au droit d'auteur). Toute utilisation du document à des fins commerciales est strictement interdite.

Par ailleurs, l'utilisateur s'engage à respecter les droits moraux de l'auteur, principalement le droit à l'intégrité de l'oeuvre et le droit de paternité et ce dans toute utilisation que l'utilisateur entreprend. Ainsi, à titre d'exemple, lorsqu'il reproduira un document par extrait ou dans son intégralité, l'utilisateur citera de manière complète les sources telles que mentionnées ci-dessus. Toute utilisation non explicitement autorisée ci-avant (telle que par exemple, la modification du document ou son résumé) nécessite l'autorisation préalable et expresse des auteurs ou de leurs ayants droit.

Faculté des sciences

Département de géographie

Université de Liège

Amélioration de la précision du découpage des codes postaux par la prise en compte de la structure urbaine avec la bibliothèque Python momepy

Mémoire présenté par : **Jonas PLANCKE**

pour l'obtention du titre de

Master en sciences géographiques, orientation géomatique, à finalité spécialisée en geodata-expert

Année académique :	**2023 - 2024**
Date de défense :	**Septembre 2024**

Président de jury :	Prof. **René WARNANT**
Promoteur :	Prof. **Roland BILLEN**
Co-promoteur :	Assist. **Jean-Paul KASPRZYK**
Jury de lecture :	Prof. **François JONARD**

REMERCIEMENTS

Je tiens à exprimer ma gratitude envers la société GeoPostcodes pour l'opportunité qu'elle m'a offerte de réaliser ce mémoire.

Au sein de l'entreprise, ma reconnaissance va tout particulièrement à Benjamin Deswysen. Sa disponibilité, son aide et ses conseils avisés ont été précieux tout au long de ce travail. Je remercie également Jérôme Urbain pour sa contribution et ses observations pertinentes lors de nos échanges.

Un grand merci à Monsieur Kasprzyk pour sa disponibilité et ses recommandations.

Enfin, je remercie Chloé pour sa relecture et son soutien tout au long de la rédaction de ce mémoire.

RÉSUMÉ

Ce mémoire répond à une demande de l'entreprise GeoPostcodes, gestionnaire d'une base de données de codes postaux pour de nombreux pays. GeoPostcodes dispose de tables de polygones pour les premiers caractères des codes postaux. Mais pour les codes plus précis, certains pays ne fournissent pas de polygone. L'entreprise a donc développé une méthode basée sur des données de codes postaux sous forme de points, en utilisant la tessellation de Voronoï. Cependant, cette approche présente des limitations, notamment le risque de traverser des bâtiments ou de ne pas respecter la structure urbaine. Pour résoudre ce problème, nous avons recours à la bibliothèque Python momepy, capable de prendre en compte les bâtiments et la structure urbaine. Momepy propose trois types de tessellations formées autour des bâtiments plutôt que des points, évitant ainsi les incohérences liées aux bâtiments coupés. Nous avons développé un programme permettant à GeoPostcodes de choisir parmi ces tessellations et d'en ajuster les paramètres. Le programme utilise des données d'OpenStreetMap via OSMnx et l'Overpass API, visant à lier les cellules de tessellation obtenues aux données des codes postaux des points. Les résultats principaux démontrent qu'il est possible d'attribuer correctement des codes postaux aux cellules, moyennant un choix et un paramétrage pertinent des tessellations, ainsi que des potentiels ajustements en post-traitement. Le succès de cette méthode dépend également de la disponibilité et de la cohérence des données, tant pour les bâtiments que pour les points de GeoPostcodes. Pour contrer cette problématique, nous suggérons l'utilisation d'autres sources pour les bâtiments et les barrières, comme les initiatives *Microsoft Global ML Building Footprints* ou *Overture Maps*.

ABSTRACT

This thesis addresses a request from GeoPostcodes, a company managing a postal code database for numerous countries. GeoPostcodes has polygon tables for the first characters of postal codes, but for more precise codes, some countries do not provide polygons. The company has therefore developed a method based on postal code data in point form, by using Voronoi tessellation. However, this approach has limitations, particularly the risk of crossing buildings or not respecting urban structure. To solve this problem, we use the Python library momepy, which can consider buildings and urban structure. Momepy offers three types of tessellations formed around buildings rather than points, thus avoiding inconsistencies related to cut buildings. We have developed a program allowing GeoPostcodes to choose among these tessellations and adjust their parameters. The program uses OpenStreetMap data via OSMnx and the Overpass API, aiming to link the obtained tessellation cells to the postal code data of the points. The main results demonstrate that it is possible to correctly assign postal codes to cells, provided an appropriate choice and parameterization of tessellations, as well as potential post-processing adjustments. The success of this method also depends on the availability and consistency of data, both for buildings and GeoPostcodes points. To counter this issue, we suggest using other sources for buildings and barriers, such as the Microsoft Global ML Building Footprints or Overture Maps initiatives.

TABLE DES MATIÈRES

RÉSUMÉ 3
ABSTRACT 4
TABLE DES MATIÈRES 5
TABLE DES FIGURES 7
1 INTRODUCTION 11
2 ÉTAT DE L'ART 17
2.1 La bibliothèque Momepy 17
2.2 Tessellation de Voronoï 18
2.3 Principe de la tessellation dans momepy 20
2.4 Les différents types de tessellation 21
2.4.1 La tessellation morphologique 21
2.4.2 La tessellation avec barrières/fermée 23
2.4.3 La tessellation basée sur les blocs 25
2.5 La bibliothèque GeoPandas 26
2.6 OpenStreetMap, Overpass API et quickOSM 27
2.7 Base de données PostGIS 28
3 MÉTHODOLOGIE 30
3.1 Les données des codes postaux fournies par GeoPostcodes. 30
3.2 But du mémoire et proposition d'un programme informatique 30
3.3 Décomposition des polygones 31
3.4 Les 3 différents types de tessellations exploitées 32
3.4.1 La tessellation morphologique 32
3.4.2 La tessellation avec barrières/fermée 34
3.4.3 La tessellation basée sur les blocs 48
3.5 Attribution de codes postaux aux cellules de tessellation 49
3.6 Programme principal 54
4 RÉSULTATS 57
4.1 Résultats avec les différentes tessellations 57
4.1.1 Différences entre les 3 types de tessellation 58
4.1.2 Tessellation morphologique 59
4.1.3 Tessellation avec barrières/fermée 60
4.1.4 Tessellation basée sur blocs 67
4.2 Limiter la tessellation 72
4.3 Résumé des 3 types de tessellations 75
4.4 Tessellation associée aux codes postaux 76
4.4.1 Exemple 1 : tessellation fermée 77
4.4.2 Exemple 2 : tessellation basée sur les blocs 79
4.4.3 Exemple 3 : situation idéale 80
4.4.4 Exemple 4 : points confondus 81
4.4.5 Exemple 5 : incohérence ou manque de données 83
5 DISCUSSION ET PERSPECTIVES 85

6 CONCLUSION 88
7 BIBLIOGRAPHIE 90

TABLE DES FIGURES

Figure 1 : Séparation des zones postales au Brésil (GeoPostcodes(2023) & OpenStreetMap (2024)) 12
Figure 2 : Échantillon de la table des codes postaux à un niveau global du Brésil (GeoPostcodes, 2023) 13
Figure 3 : Semi de points associés aux codes postaux au Brésil (GeoPostcodes(2023) & OpenStreetMap(2024)) 14
Figure 4 : Échantillon de la table des codes postaux à un niveau local du Brésil (GeoPostcodes, 2023) 14
Figure 5 : Zone postale « globale » dans la ville de Midland, Canada (GeoPostcodes (2023), OpenStreetMap (2024). 15
Figure 6 : zones postales « locales » dans la ville de Midland, Canada. (GeoPostcodes (2023), momepy (2019), OpenStreetMap (2024)) 15
Figure 7 : Portion d'un diagramme de Voronoi planaire (Okabe, A., Boots, B., & Sugihara, K., 1992) 19
Figure 8 : Diagramme de Voronoï des stations de métro du réseau londonien (DataGenetics, n.d.). 19
Figure 9 : Cellules de tessellations morphologiques à Zurich (Fleischmann et al., 2020) 23
Figure 10 : évolution de la tessellation avec barrières (Fleischmann & Arribas-Bel, 2022) 24
Figure 11 : Création des barrières. Primary_barriers constitue la barrière primaire, souvent un réseau de rues. Additional_barriers utilise d'autres barrières supplémentaires (Fleischmann et al., 2018). 24
Figure 12.1 : Tessellation morphologique et réseau routier. Figure 12.2 : Tessellation-based blocks résultante (Fleischmann et al., 2018) 26
Figure 13 : Query features : détection des barrières (OpenStreetMap, 2024) 36
Figure 14 : Exemple tag Delft, Pays-Bas (OpenStreetMap, 2024) 37
Figure 15 : Utilisation du tag route : railway pour télécharger les voies ferrées dans le quartier de la gare Amsterdam Centraal, aux Pays-Bas. Cette méthode ne renvoie aucun résultat (QGIS, 2024). 38
Figure 16 : Utilisation du tag railway : rail pour télécharger les voies ferrées dans le quartier de la gare Amsterdam Centraal, aux Pays-Bas. Cette méthode fonctionne car le tag correspond à celui de l'Overpass API (QGIS, 2024). 39
Figure 17 : Les tags sont disponibles sur cette page (OpenStreetMap Contributors, 2024). 39
Figure 18 : tessellation débordant sur un lac à Midland, Canada (momepy(2019) & OpenStreetMap(2024)). 40
Figure 19 : Correction de la tessellation grâce à la fonction crop_water() de GeoPostcodes (momepy(2019), GeoPostcodes(2024) & OpenStreetMap(2024)). 40
Figure 20 : Niveau administratif pour Delft, Pays-Bas (Openstreetmap, 2023) 44
Figure 21 : exécution sur QuickOSM via QGIS pour le niveau administratif de Delft, Pays-Bas. Nous voyons que la requête a été exécutée avec succès pour le même niveau administratif (QGIS, 2023). 44
Figure 22 : Niveau administratif pour Midland, Canada (Openstreetmap, 2023) 45

Figure 23 : exécution sur QuickOSM via QGIS pour le niveau administratif de Midland, Canada. Nous voyons que la requête n'a pas été acceptée (QGIS, 2023). 45
Figure 24 : Réseau routier incomplet : based blocks (Fleischmann et al., 2018) 47
Figure 25 : Réseau routier complet : based blocks (Fleischmann et al., 2018) 48
Figure 26 : tessellation morphologique, Midland, Canada (momepy(2019) &OpenStreetMap (2024)). 58
Figure 27 : tessellation fermée, Midland, Canada (momepy(2019) &OpenStreetMap (2024)). 58
Figure 28 : tessellation basée sur les blocs (basée sur une tessellation fermée), Midland, Canada (momepy(2019) &OpenStreetMap (2024)). 59
Figure 29 : Tessellation morphologique à Erechim, Brésil (momepy(2019) &OpenStreetMap (2024)). 59
Figure 30 : Tessellation morphologique à Delft, Pays-Bas (momepy(2019) &OpenStreetMap (2024)). 60
Figure 31 : Tessellation fermée (inclusion des canaux en tant que barrière), Delft, Pays-Bas (OpenStreetMap (2024) & momepy(2019)) 61
Figure 32 : tessellation fermée dans le quartier financier de Manhattan, New York (OpenstreetMap (2024) & momepy (2019)) 62
Figure 33 : tessellation fermée dans le centre-ville de Vancouver, Canada (OpenstreetMap (2024) & momepy (2019)) 62
Figure 34 : tessellation fermée dans le quartier de Bela Vista à Sao Paulo, Brésil (OpenstreetMap (2024) & momepy (2019)) 63
Figure 35 : tessellation fermée dans le quartier financier de Manhattan, New York. Les chemins pour piétons et les pistes cyclables n'y sont pas inclus (OpenstreetMap (2024) & momepy (2019)) 63
Figure 36 : tessellation fermée dans le centre d'Amsterdam, Pays-Bas. L'ellipse rouge met en évidence les barrières créées par le réseau ferroviaire (OpenStreetMap(2024) & momepy(2019)) 64
Figure 37 : Tessellation fermée (sans les canaux comme barrière), Giethoorn, Pays-Bas (OpenStreetMap (2024) & momepy(2019)) 65
Figure 38 : Tessellation fermée (inclusion des canaux en tant que barrière), Giethoorn, Pays-Bas (OpenStreetMap (2024) & momepy(2019)) 65
Figure 39 : Tessellation avec barrières, Kralingen-Crooswijk, Rotterdam (OpenStreetMap (2024) & momepy(2019)) 66
Figure 40 : Tessellation avec barrières (inclusion des parcs urbains en tant que barrière), Kralingen-Crooswijk, Rotterdam (OpenStreetMap (2024) & momepy(2019)) 67
Figure 41 : tessellation basée sur les blocs dans le quartier financier de Manhattan, New York (OpenstreetMap (2024) & momepy (2019)) 68
Figure 42 : tessellation basée sur les blocs (sur base d'une tessellation fermée) dans le quartier financier de Manhattan, New York (OpenStreetMap (2024) & momepy (2019)) 68
Figure 43 : tessellation basée sur les blocs (sur base d'une tessellation fermée sans les chemins pour piétons) dans le quartier financier de Manhattan, New York (OpenstreetMap (2024) & momepy (2019)) 69
Figure 44 : tessellation basée sur les blocs à Erechim, Brésil (OpenStreetMap (2024) & momepy (2019)) 70
Figure 45 : tessellation basée sur les blocs (sur base d'une tessellation fermée) à Erechim, Brésil (OpenStreetMap (2024) & momepy (2019)) 71

Figure 46 : tessellation fermée à Erechim, Brésil (OpenStreetMap (2024) & momepy (2019)) 72
Figure 47 : Limite administrative de niveau 8, Lunenburg, Canada (OpenStreetMap, 2024) 73
Figure 48 : Tessellation fermée utilisant admin_level = 8 comme limite à Lunenburg, Canada (momepy(2019), OpenStreetMap(2024)) 74
Figure 49 : Codes postaux attribués aux cellules de la tessellation fermée à Midland, Canada (GeoPostcodes(2023), momepy(2019) & OpenStreetMap(2024)) 77
Figure 50 : Tessellation fermée et points associés aux codes postaux de bâtiments (GeoPostcodes(2023), momepy(2019), OpenStreetMap(2024)) 78
Figure 51 : Codes postaux attribués aux cellules de la tessellation basée sur les blocs à Midland, Canada (GeoPostcodes(2023), momepy(2019) & OpenStreetMap(2024)) 79
Figure 52 : Codes postaux attribués aux cellules de la tessellation fermée à Delft, Pays-Bas (GeoPostcodes(2023), momepy(2019) & OpenStreetMap(2024)) 80
Figure 53 : Codes postaux attribués aux cellules de la tessellation fermée à Erechim, Brésil (GeoPostcodes(2023), momepy(2019) & OpenStreetMap(2024)) 81
Figure 54 : Zoom sur un code postal avec une « erreur » ? Erechim, Brésil (GeoPostcodes(2023), momepy(2019) & OpenStreetMap(2024)). 82
Figure 55 : Bâtiment avec plusieurs codes postaux (GeoPostcodes, 2023) 82
Figure 56 : Codes postaux attribués aux cellules de la tessellation fermée à Sao Paulo, Brésil (GeoPostcodes(2023), momepy(2019) & OpenStreetMap(2024)) 83

ABRÉVIATIONS

BD : Base de données

SIG : Système d'informations géographiques

SGBD : Système de gestion de base de données

1 INTRODUCTION

L'entreprise GeoPostcodes propose une base de données exhaustive sur les codes postaux, contenant notamment toutes les divisions administratives, les villes et les codes postaux de 247 pays. Celle-ci est constituée d'un ensemble de polygones représentant des codes postaux et des entités administratives. GeoPostcodes dispose également d'une *streets database*, une base de données complémentaire avec des entités administratives et des routes. Il s'agit de la plus vaste base de données de ce type au monde. Toutes les données, géoréférencées et disponibles en formats normalisés ou non, sont exploitables par n'importe quel logiciel SIG (GeoPostcodes, 2024).

Pour obtenir ces polygones, GeoPostcodes utilise l'algorithme de Voronoï, capable de partitionner un plan en régions, chaque région étant appelée une « cellule de Voronoï ». L'algorithme construit des polygones en fonction des différents poids de points représentant les centroïdes des bâtiments associés à un code postal (Lecaché, 2023).

Bien que cette approche soit innovante, elle présente certaines limites qui méritent d'être adressées. Le problème principal est que l'algorithme peut, à certains endroits, construire des polygones qui ne suivent pas la réalité du terrain. Par exemple, les polygones peuvent traverser certains bâtiments ou présenter un côté « haché », peu naturel.

Le but de ce mémoire est de proposer une méthode pour affiner les résultats de l'algorithme de Voronoï afin de mieux délimiter les zones postales. Pour cela, nous utiliserons la bibliothèque Python momepy, qui permet de réaliser des analyses sur la morphologie urbaine. Plus particulièrement, elle offre la possibilité de diviser l'espace urbain selon une tessellation morphologique, une technique de partition de l'espace basée sur les caractéristiques morphologiques du terrain (Fleischmann, 2019). Cette approche, couplée à celle de GeoPostcodes, permettra d'obtenir des limites postales plus précises, qui ne couperont pas les bâtiments et qui suivront plus fidèlement la réalité du terrain.

Il est important de noter que, bien que certains pays fournissent les polygones d'emprises de codes postaux, ce n'est pas une pratique universelle. C'est pour cela que nous avons recours à cette méthode de génération à partir des points. En effet, contrairement à ce que l'on pourrait penser, la distribution de ces données en *open data* par les services publics n'est pas

systématique. De plus, même lorsque ces polygones sont fournis, ils ne sont pas toujours en adéquation avec les derniers jeux de données d'adresses (les « points »), rendant les données de polygones obsolètes (B. Deswysen[1], communication personnelle, 14 août 2024).

Actuellement, GeoPostcodes possède des données sur les codes postaux allant d'un niveau global, sous forme de polygones, à un niveau plus local, sous la forme d'un semi de points associés à différents bâtiments. Dans de nombreux pays, les formats des codes postaux sont composés de chiffres ou sont au format alphanumérique. Les premiers éléments correspondent généralement à une zone géographique globale, tandis que les derniers apportent une plus grande précision de la localisation.

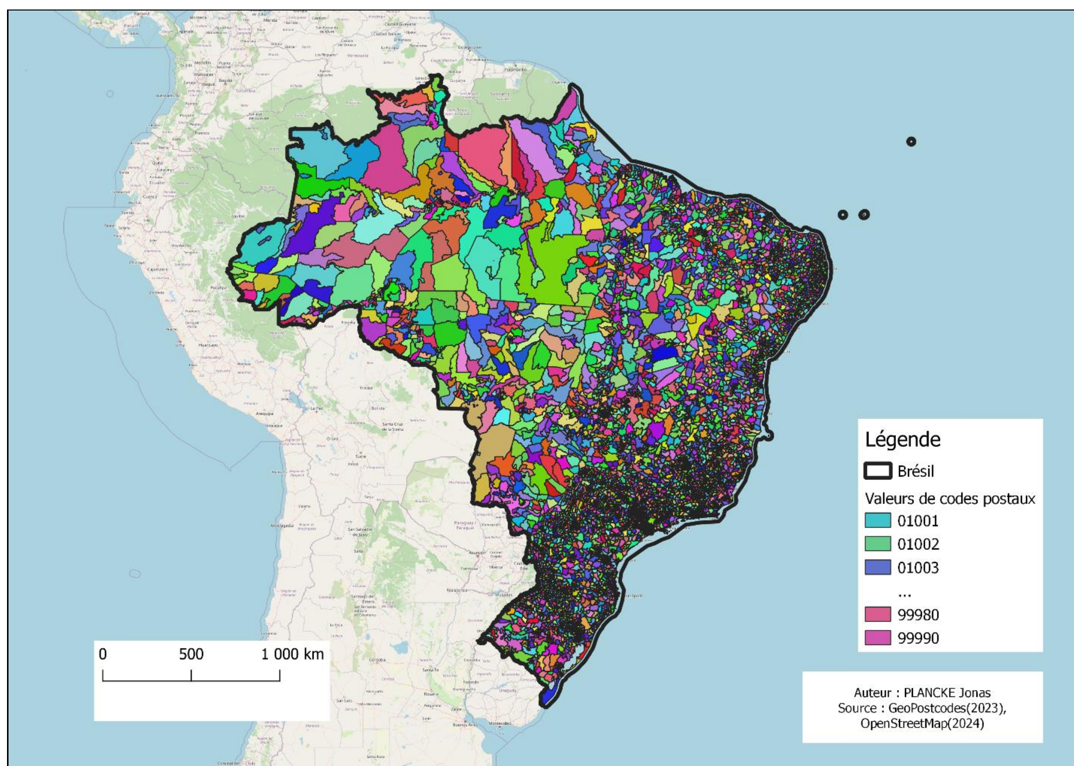

Figure 1 : Séparation des zones postales au Brésil (GeoPostcodes(2023) & OpenStreetMap (2024))

Par exemple, au Canada, le code postal est alphanumérique et comprend 6 caractères au format « A1A 1A1 ». Les 3 premiers éléments indiquent la région de tri d'acheminement qui correspond à une zone géographique, avec le premier caractère désignant un district postal plus large. Les 3 derniers éléments correspondent à une unité de distribution locale (Postal Codes in Canada, n.d.). Un autre exemple est celui du Brésil, où le code postal comprend 8 chiffres : les 5 premiers

[1] Benjamin Deswysen est employé chez GeoPostcodes en tant que spécialiste en SIG (GeoPostcodes, 2024)

chiffres indiquent respectivement la région, la sous-région, le secteur, le sous-secteur et le diviseur de sous-secteur. Les 3 derniers chiffres sont séparés de la première partie par un trait d’union et représentent les identifiants de distribution (codigo-postal.org, 2024).

La Figure 1 illustre la division du Brésil en zones postales fournies par GeoPostcodes. Les tables d'attributs de ces zones postales contiennent les cinq premiers caractères du code postal, sous forme de polygones.

	iso character varying (2)	zip character varying (20)	geom geometry
1	BR	69343	0106000020110F0000010000000103000000010000001E1800003898EBA20BA65A
2	BR	68920	0106000020110F00000100000001030000000100000400C0000EA4C16EAC39F56
3	BR	57025	0106000020110F00001100000001030000000800000200100000C252A3F6A544E
4	BR	57042	0106000020110F00003000000001030000000100000B20200008FEF3A7DA4404E
5	BR	05757	0106000020110F000002000000010300000001000001B0000008DB395DCF6DA53
6	BR	13495	0106000020110F00000B000000010300000003000000B040000A91A544C772D54
7	BR	95895	0106000020110F0000010000000103000000010000008D00000030D022F22A0B56
8	BR	61901	0106000020110F000002000000010300000001000000660000009B61BA7AB6550
9	BR	06448	0106000020110F00000100000001030000000100000099000000D3D4B47234E853
10	BR	99220	0106000020110F00000100000001030000000100000098000000BE4432C8BB0056
11	BR	52291	0106000020110F0000020000000103000000010000000D020000061D631F59A74D
12	BR	14210	0106000020110F0000010000000103000000010000009F060000F3BA08B9365A54
13	BR	84345	0106000020110F000001000000010300000001000000360700003F022B09525255
14	BR	16450	0106000020110F000001000000010300000001000000C50C00005BA2F58D303D55

Figure 2 : Échantillon de la table des codes postaux à un niveau global du Brésil (GeoPostcodes, 2023)

La Figure 2 montre un échantillon des données de GeoPostcodes dans leur base de données PostgreSQL concernant les codes postaux globaux du Brésil, correspondant à 5 chiffres. Chaque géométrie correspond à un polygone (comme vu sur la Figure 1).

La Figure 3 montre le semi de points fourni par GeoPostcodes. Chaque point illustre un bâtiment et est associé au code postal complet de ce bâtiment, c’est-à-dire un code postal contenant huit caractères. La Figure 4 montre un échantillon des données de GeoPostcodes sur les données de codes postaux locaux du Brésil, correspondant à 8 chiffres.

L'objectif de GeoPostcodes est d'affiner les polygones existants pour obtenir une division plus précise, en tenant compte de la structure urbaine et de l’empreinte des bâtiments. En d’autres termes, l’objectif est d’affiner les polygones de codes postaux (contenant les premiers caractères des codes postaux) en se basant sur le semi de points contenant l’entièreté des caractères de ces codes postaux. En utilisant momepy, cela permettrait de créer des polygones qui délimitent précisément les zones bâties, tout en leur attribuant les codes postaux issus des données ponctuelles du semi de points.

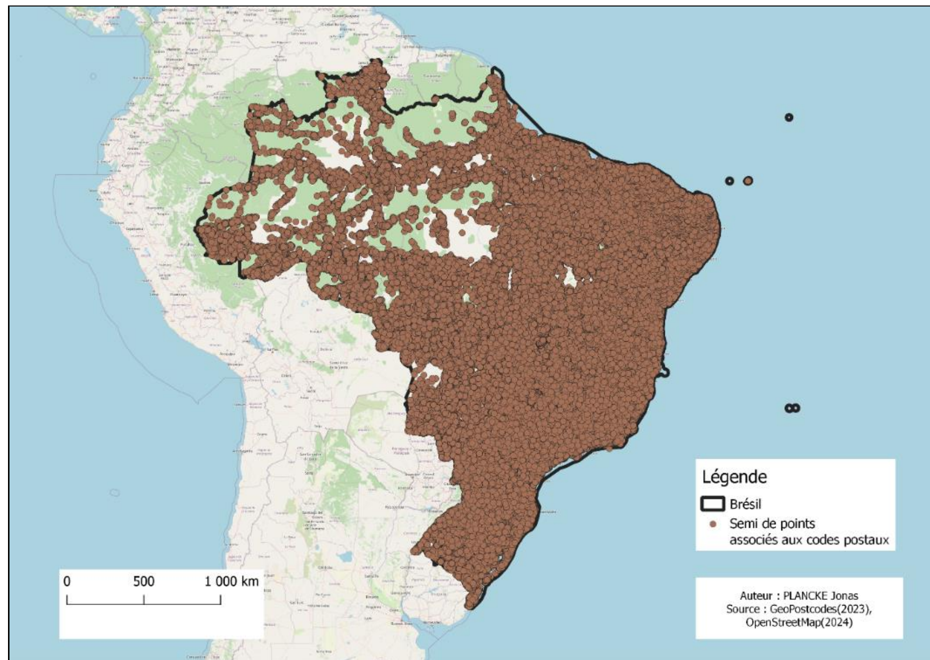

Figure 3 : Semi de points associés aux codes postaux au Brésil (GeoPostcodes(2023) & OpenStreetMap(2024))

	iso character varying (2)	zip character varying	geom geometry	type character varying	id [PK] integer
1	BR	13847-029	0101000020110F0000424A44F7BBEE53C18C94A3F4E37C43...	postcode	34153068
2	BR	13848-114	0101000020110F0000ED523CB8C1ED53C1DA434400167A43...	postcode	34153067
3	BR	13843-269	0101000020110F0000475D29A0F3EE53C1CA22440CD38543...	postcode	34153066
4	BR	13840-028	0101000020110F0000EA400AB338EF53C1C1B28F6BB98043...	postcode	34153065
5	BR	13844-008	0101000020110F00006062702105EF53C1D311A3D0878143C1	postcode	34153064
6	BR	13840-443	0101000020110F00007E31D649F5ED53C1C7845E12A77D43...	postcode	34153063
7	BR	13842-120	0101000020110F000030DEC21D4FED53C1BA4B6FD90F8243...	postcode	34153062
8	BR	13842-252	0101000020110F0000007E94CC23EF53C1FB4FB3FFB88043...	postcode	34153061
9	BR	13842-255	0101000020110F0000007E94CC23EF53C1FB4FB3FFB88043...	postcode	34153060
10	BR	13842-257	0101000020110F0000007E94CC23EF53C1FB4FB3FFB88043...	postcode	34153059
11	BR	13842-259	0101000020110F0000007E94CC23EF53C1FB4FB3FFB88043...	postcode	34153058
12	BR	13842-270	0101000020110F0000007E94CC23EF53C1FB4FB3FFB88043...	postcode	34153057
13	BR	13842-301	0101000020110F00007D308D5912EF53C11B631842FA7F43C1	postcode	34153056
14	BR	13843-218	0101000020110F0000C322842114EF53C1D2F92B9E178343C1	postcode	34153055

Figure 4 : Échantillon de la table des codes postaux à un niveau local du Brésil (GeoPostcodes, 2023)

Pour exemplifier cela et montrer un cas concret, nous allons comparer la situation actuelle à un des résultats obtenus dans le cadre de ce mémoire. Ainsi, sur la Figure 5, nous voyons un des polygones reprenant une zone de code postal global au Canada, dans la ville de Midland, reprenant uniquement les 3 premiers caractères du code postal (L4R). Sur cette carte, chaque point correspond à un bâtiment et possède un code postal unique et complet.

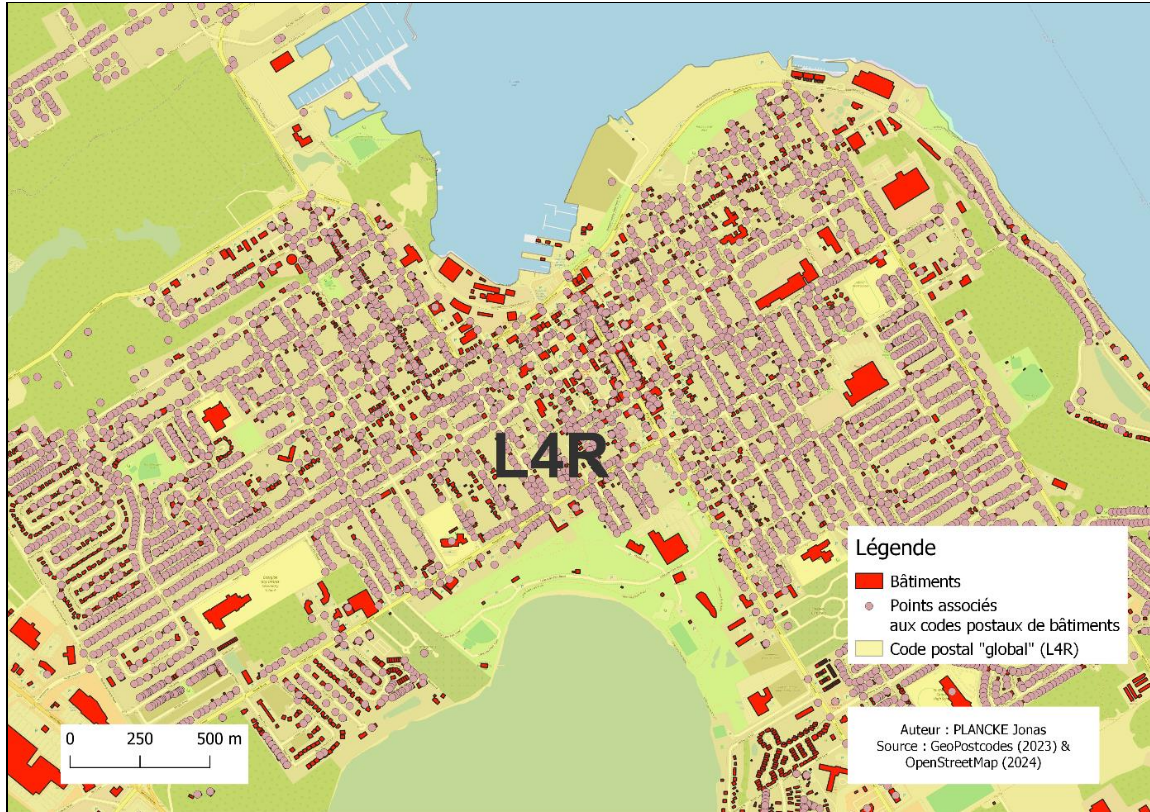

Figure 5 : Zone postale « globale » dans la ville de Midland, Canada (GeoPostcodes (2023), OpenStreetMap (2024).

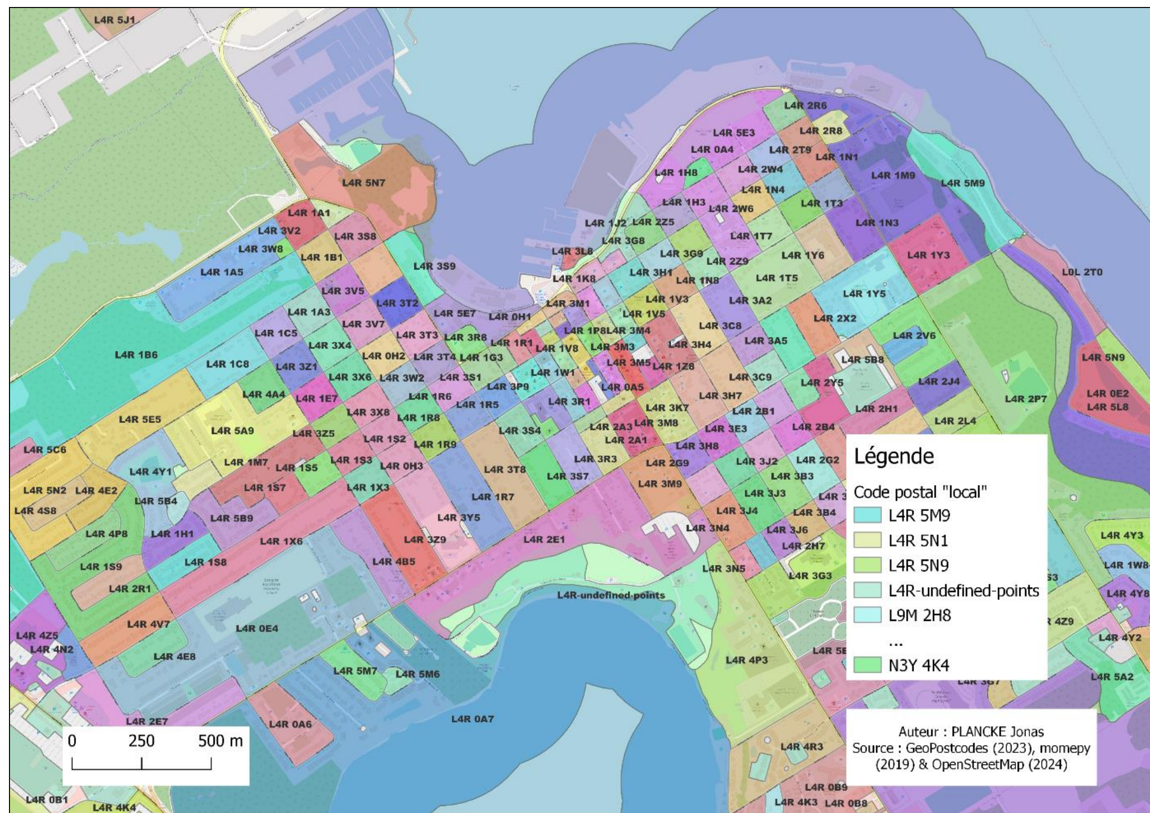

Figure 6 : zones postales « locales » dans la ville de Midland, Canada. (GeoPostcodes (2023), momepy (2019), OpenStreetMap (2024))

L'objectif du mémoire est donc d'obtenir un résultat semblable à la Figure 6, en associant les différents bâtiments en fonction de leurs codes postaux uniques, basés sur les points de GeoPostcodes. Nous voyons sur cette figure que des blocs de rues ont été associés en fonction des codes postaux des bâtiments qui les composent.

GeoPostcodes possède des données sur de nombreux pays. Dans le cadre de ce mémoire, nous nous sommes concentrés sur des villes des pays suivants : le Canada, le Brésil, les Pays-Bas et les États-Unis.

2 ÉTAT DE L'ART

2.1 La bibliothèque Momepy

Momepy (*Morphological Measuring in Python*) est une libraire python pour l'analyse quantitative de la forme urbaine. Elle fait partie de PySAL (Python Spatial Analysis Library) et est construite à partir de GeoPandas, un autre module de PySAL ainsi qu'à partir de networkX[2] (Fleischmann *et al.*, 2018).

Momepy offre différentes fonctionnalités (Fleischmann *et al.*, 2018) telles que :

- Mesurer les dimensions des éléments morphologiques, de leurs parties et des structures agrégées.
- Quantifier les formes de géométries représentant un large éventail de caractéristiques morphologiques.
- Capturer la distribution spatiale des éléments d'un type ainsi que les relations entre différents types.
- Calculer la densité et d'autres types de caractères d'intensité.
- Calculer la diversité des différents aspects de la forme urbaine.
- Capturer la connectivité des réseaux de rues urbains.
- Générer des éléments relationnels de la forme urbaine (par exemple, la tessellation morphologique).

La bibliothèque momepy a été créée par Martin Fleischmann, un chercheur en morphologie urbaine et en science des données spatiales (Fleischmann, 2024). Momepy permet de conduire une analyse à différentes échelles de la forme urbaine, allant du niveau microscopique des bâtiments et des parcelles individuels aux régions macroscopiques métropolitaines. Les six modèles principaux de momepy représentent six catégories de caractères morphologiques urbains : dimensions, forme, distribution spatiale, intensité, connectivité (module graphique), et diversités identifiées dans la littérature et la recherche en cours. Ces six modules ensemble fournissent une large gamme d'algorithmes mesurant différents aspects de la forme urbaine et sont capables de décrire sa complexité avec un degré de précision significatif (Fleischmann, 2019).

[2] NetworkX est un package Python pour la création, la manipulation et l'étude de la structure, de la dynamique et des fonctions des réseaux complexes (NetworkX, 2024).

De plus, momepy offre une fonction générant des tessellations morphologiques de surfaces basées sur la configuration spatiale de l'empreinte de bâtiments en utilisant la tessellation de Voronoï, tel qu'implémentée dans SciPy.

2.2 Tessellation de Voronoï

La tessellation de Voronoï, également appelée la décomposition de Voronoï, ou le diagramme de Voronoï, est un concept mathématique qui permet de partitionner l'espace en régions en fonction d'un ensemble de points donnés. Pour cela, nous partons des différents points dans l'espace et pour chacun de ces points, nous définissons une région qui comprend tous les points de l'espace qui sont plus proches de ce point que tous les autres points de l'ensemble. Ces régions sont appelées cellules de Voronoï. L'ensemble de ces cellules forme une partition complète de l'espace, c'est-à-dire que chaque point de l'espace appartient à exactement une cellule (Du, Q., Faber, V., & Gunzburger, M., 1999).

La tessellation de Voronoï a beaucoup été étudiée par des mathématiciens, particulièrement celles basées sur les processus ponctuels de Poisson (Keeler, 2023).

De manière plus générale, la tessellation de Voronoï peut être utile dans des situations plus pratiques. Par exemple, une carte divisée en cellules de Voronoï, également appelée diagramme de Voronoï peut aider à trouver l'hôpital le plus proche. Comme chaque cellule comprend les points qui sont les plus proches de son centre, le service le plus proche peut être immédiatement localisé par un utilisateur. Les diagrammes de Voronoï sont facilement construits à partir de logiciels informatiques et permettent aisément de partitionner l'espace en fonction de certains paramètres (Lynch, 2017). Par exemple, la Figure 8 montre une carte des stations de métro du réseau londonien. Elle montre quelle est la station de métro la plus proche de n'importe quel emplacement.

De nombreuses autres applications sont possibles, comme l'analyse de réseau, l'astrophysique, l'hydrologie, etc. (Lynch, 2017). Un diagramme de Voronoï peut permettre de planifier des itinéraires sans collision, en maintenant une distance minimale ou maximale par rapport à certains points, et de manière plus générale, un diagramme de Voronoï permet de calculer les plus proches voisins (Erwig, 2000).

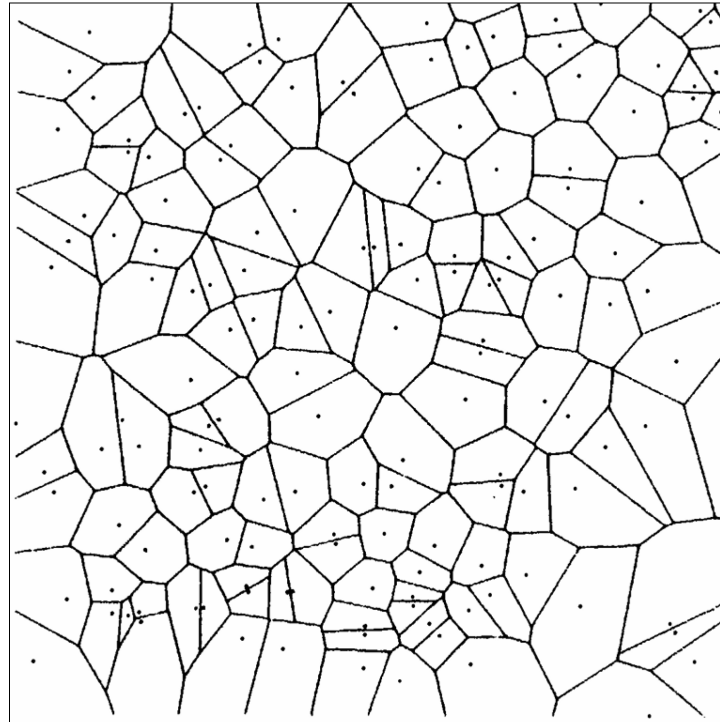

Figure 7 : Portion d'un diagramme de Voronoi planaire (Okabe, A., Boots, B., & Sugihara, K., 1992)

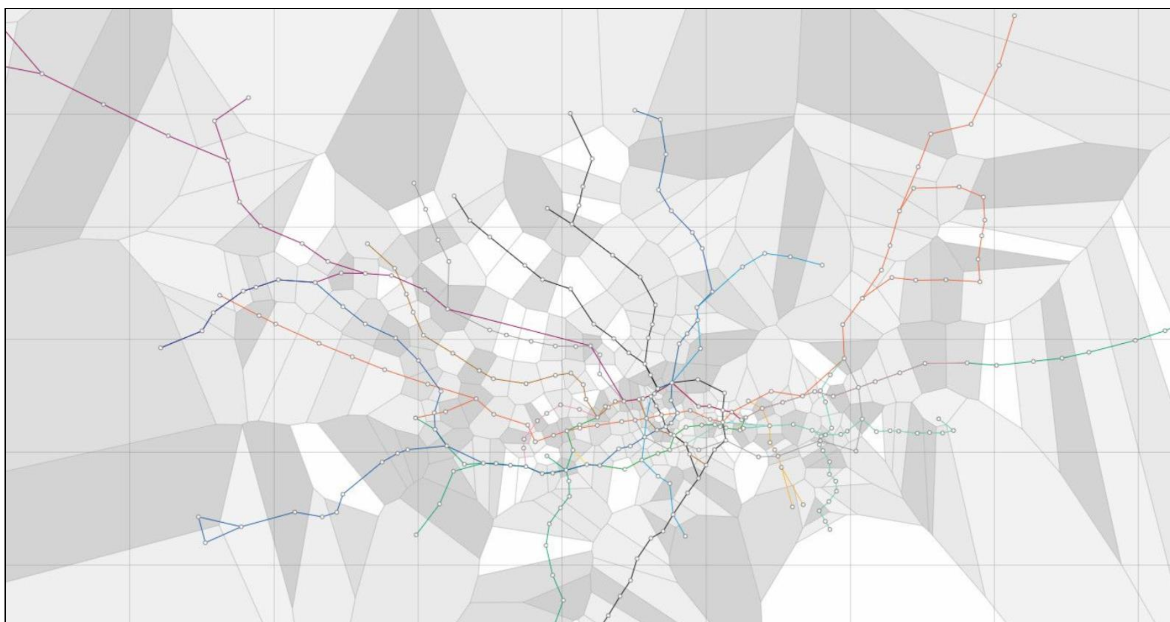

Figure 8 : Diagramme de Voronoï des stations de métro du réseau londonien (DataGenetics, n.d.).

Une littérature complète et passionnante existe sur la tessellation de Voronoï et sur l'utilisation des diagrammes pour effectuer un grand nombre de tâches diverses et variées.

Cette technique est notamment utilisée par GeoPostcodes pour la délimitation de zones géographiques (Lecaché, 2023). Il est intéressant de noter que la bibliothèque momepy propose une approche similaire, mais avec une différence notable : momepy utilise des bâtiments comme

base de la tessellation plutôt que des points (Fleischmann *et al*., 2018). Cette distinction peut avoir un impact sur la forme et la distribution des cellules de Voronoï générées. Cela nous permettra également de générer des tessellations qui ne couperont pas à travers les bâtiments, préoccupation actuelle de GeoPostcodes.

2.3 Principe de la tessellation dans momepy

La tessellation morphologique sert d'unité fiable et universelle pour représenter l'espace urbain. Elle s'appuie sur la tessellation de Voronoï, que nous avons vue auparavant. Chaque cellule de Voronoï contient une portion de l'espace total et représente une zone d'influence, où les points contenus dans chaque cellule de Voronoï correspondent aux points les plus proches du centre de cette cellule de Voronoï par rapport aux autres cellules. La tessellation de Voronoï a déjà été largement utilisée dans l'analyse urbaine bien qu'elle présente une faiblesse potentielle : l'incapacité à distinguer le motif spatial des tissus urbains modernistes, comme ce fut par exemple le cas au Japon après la seconde guerre mondiale. Dans ce type de tissu, nous retrouvons des distances très variables entres les bâtiments et les rues. Le choix de la zone d'analyse est donc crucial (Fleischmann *et al.*, 2020).

Les algorithmes de tessellation suivent une approche intuitive similaire à celle des humains lorsqu'ils observent un espace urbain, en suivant par exemple les centres des rues qui séparent les habitations. Toutefois, la tessellation reste plus pertinente que « l'œil humain » car il n'y a pas de biais de subjectivité (Fleischmann *et al*., 2020).

La tessellation nécessite des données en entrée, c'est-à-dire d'empreintes pour partager l'espace en plusieurs polygones semblables. Certains espaces, comme les routes, les grands espaces verts ou les bâtiments à faible surface peuvent parfois être retirés pour avoir des données plus propres et plus faciles à analyser. Les données en entrée requièrent uniquement une couche de polygone représentant les empreintes de bâtiments (Fleischmann *et al*., 2020). Néanmoins, la plupart des études de morphologie urbaine ont tendance à utiliser des éléments morphologiques comme les segments de rues, les îlots, les bâtiments ou les parcelles comme unité d'analyse. La tessellation morphologique n'a pas de limite, et elle peut s'étendre à l'infini. Pour résoudre ce problème, on introduit un nouveau type de tessellation : *l'enclosed tessellation* en anglais, également appelée tessellation avec barrières ou tessellation fermée dans ce mémoire. Celle-ci est définie comme « la portion d'espace qui résulte de la croissance d'une tessellation morphologique à l'intérieur

d'une enceinte délimitée par une série de barrières naturelles ou construites identifiées à partir de la littérature sur la forme, la fonction et la perception urbaine » (Fleischmann & Arribas-Bel, 2022).

La bibliothèque momepy propose 3 types de tessellation : la tessellation morphologique, la tessellation avec barrières ainsi que la tessellation basée sur les blocs, cette dernière se basant sur la tessellation morphologique pour ensuite créer, comme son nom l'indique, des blocs de tessellation morphologique en fonction d'un réseau de rues. Dans la documentation momepy, des exemples d'applications pour chaque tessellation sont donnés. Généralement, deux exemples sont donnés : le premier utilise un jeu de données inclus directement avec momepy et le deuxième utilise des données originaires d'OpenStreetMap (Fleischmann *et al.*, 2018). Pour le cas de la tessellation basée sur les blocs, un seul exemple est fourni et il s'agit de celui basé sur les données de GeoPandas.

2.4 Les différents types de tessellation

Dans cette partie, nous allons passer en revue les 3 types de tessellations disponibles sur momepy.

2.4.1 La tessellation morphologique

La tessellation morphologique se base sur la tessellation de Voronoï. Contrairement à cette dernière, la tessellation est effectuée autour de polygones représentant des bâtiments et non autour de points. Comme précisé dans la documentation de momepy, cette similarité est volontaire et la base de la tessellation morphologique s'appuie sur l'algorithme de Voronoï de la bibliothèque *scipy.spatial* (Fleischmann *et al*., 2018).

Pour générer une tessellation morphologique, chaque polygone de bâtiment se doit d'avoir un identifiant unique qui permettra ainsi de lier chaque cellule de la tessellation à son bâtiment d'origine.

Comme un diagramme de Voronoï a tendance à s'étendre à l'infini pour les points en bordure, il faut définir une limite spatiale pour la tessellation. Cette limite peut être un polygone représentant la zone d'étude ou être générée automatiquement par une fonction de momepy *momepy.buffered_limit()* en définissant une distance maximale autour des bâtiments. Par défaut, cette limite est fixée à 100 mètres (Fleischmann *et al*., 2018).

La Figure 9 montre une tessellation morphologique autour de bâtiments, dans plusieurs parties de la ville de Zurich. Les bâtiments sont bien séparés les uns par rapport aux autres, chaque cellule représentant la « zone d'influence » du bâtiment, la surface dont chaque point est le plus proche de son bâtiment.

D'autres attributs cruciaux de l'algorithme de tessellation sont *segment* et *shrink*. Les deux sont prédéfinis avec des valeurs équilibrées entre les exigences de calcul et la qualité du résultat. *Segment* définit la distance maximale entre les points générés pour représenter l'empreinte du bâtiment. Plus la valeur de segment est grande et moins il y aura de points générés, ce qui allègera les calculs mais réduira la précision de la tessellation. *Shrink* définit de combien l'empreinte du bâtiment doit être réduite vers l'intérieur pour créer un espace entre les polygones adjacents. Cela évite que les bâtiments se chevauchent (Fleischmann *et al.*, 2018).

La fonction utilisée pour la tessellation morphologique est *momepy.Tessellation().* Elle génère la tessellation à partir d'un GeoDataFrame de bâtiments. Le résultat obtenu est un objet Tessellation dont l'attribut tessellation contient le GeoDataFrame de la tessellation (Fleischmann *et al*., 2018).

La documentation précise que des avertissements peuvent être émis pendant la génération de la tessellation si certains polygones de bâtiments sont mal formés, trop étroits ou se superposent. Il est alors recommandé de corriger manuellement les géométries en amont (Fleischmann *et al*., 2018).

Finalement, une fois la tessellation morphologique générée, chaque cellule est liée à son bâtiment d'origine par un identifiant unique (Fleischmann *et al*., 2018).

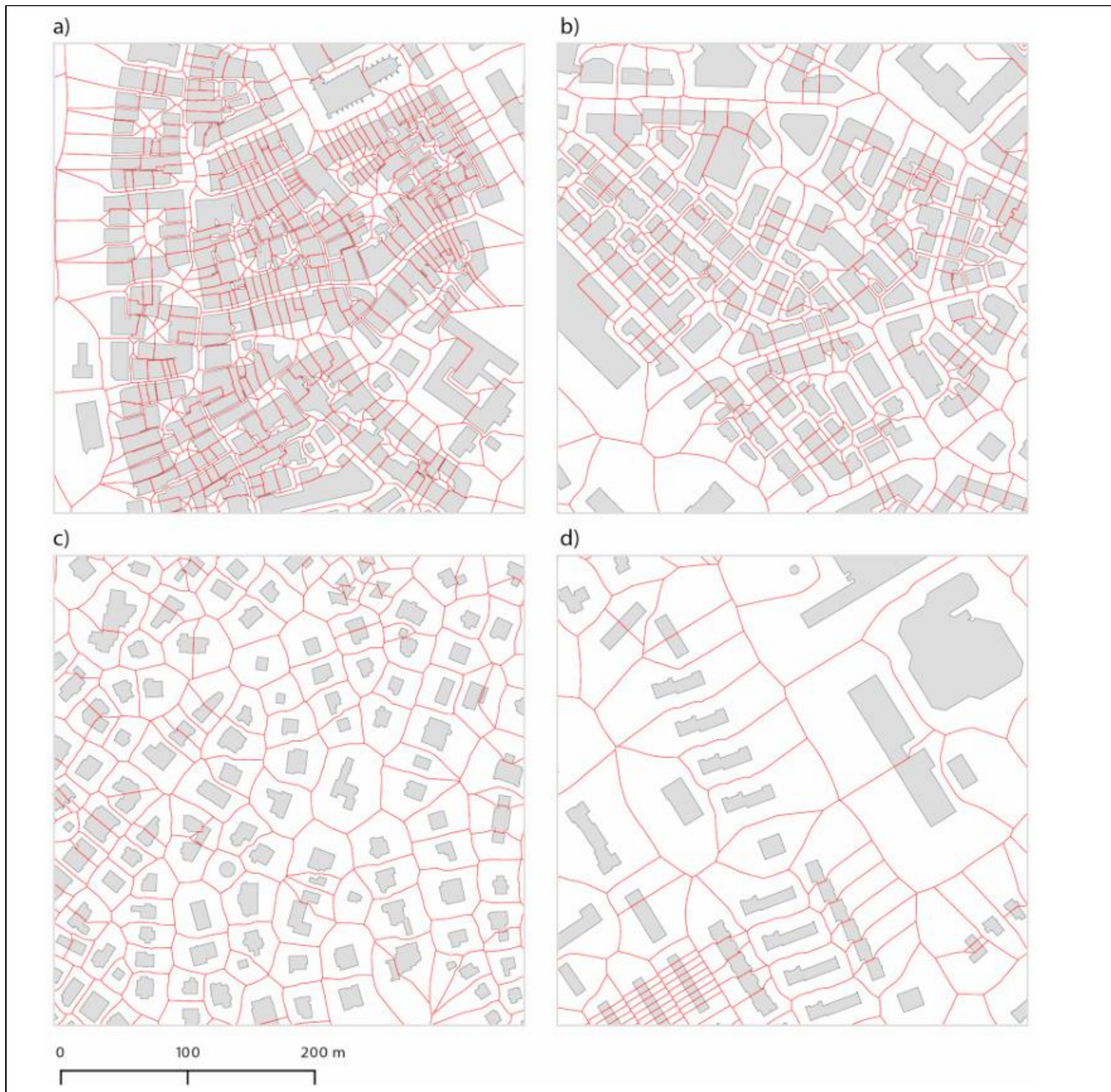

Figure 9 : Cellules de tessellations morphologiques à Zurich (Fleischmann et al., 2020)

2.4.2 La tessellation avec barrières/fermée

La tessellation avec barrières, ou tessellation fermée (*enclosed tessellation* en anglais) est une version plus poussée de la tessellation morphologique, basée sur des barrières et sur l'empreinte de bâtiments.

Les barrières sont des éléments typiques tels que le réseau routier, le réseau ferroviaire ou des éléments naturels comme les rivières, des lignes côtières, des parcs naturels, etc.

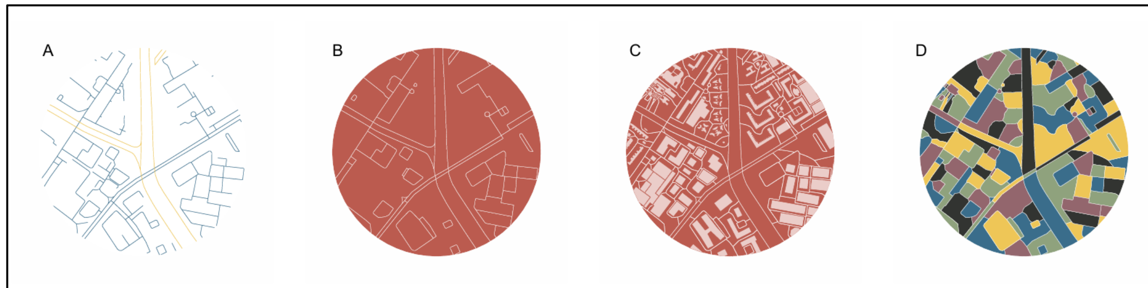

Figure 10 : évolution de la tessellation avec barrières (Fleischmann & Arribas-Bel, 2022)

La Figure 10 nous montre l'évolution de la tessellation fermée : en A, les rues sont en bleu et les rivières en jaune. En B, les rues et les rivières ne forment plus qu'une entité qui sont les barrières. En C, les bâtiments sont incorporés. Et enfin en D, nous obtenons les cellules finales de la tessellation fermée (Fleischmann & Arribas-Bel, 2022).

Il est également conseillé de spécifier les limites externes de la superficie pour laquelle nous voulons générer des barrières. Cette limite est, dans la plupart des situations, tout simplement les limites de notre zone d'étude. Mais dans certains cas, nous pouvons par exemple choisir le réseau routier ou dans le cas d'une île, définir la limite comme étant la ligne de côte (Fleischmann *et al*., 2018).

En pratique, l'utilisation des barrières s'effectue en deux étapes : nous spécifions d'abord une première barrière à momepy, appelée la barrière primaire. Si nous souhaitons inclure des barrières supplémentaires, nous devons alors les regrouper dans une variable qui reprendra toutes les barrières additionnelles, comme nous pouvons le voir dans la documentation de momepy (Figure 11).

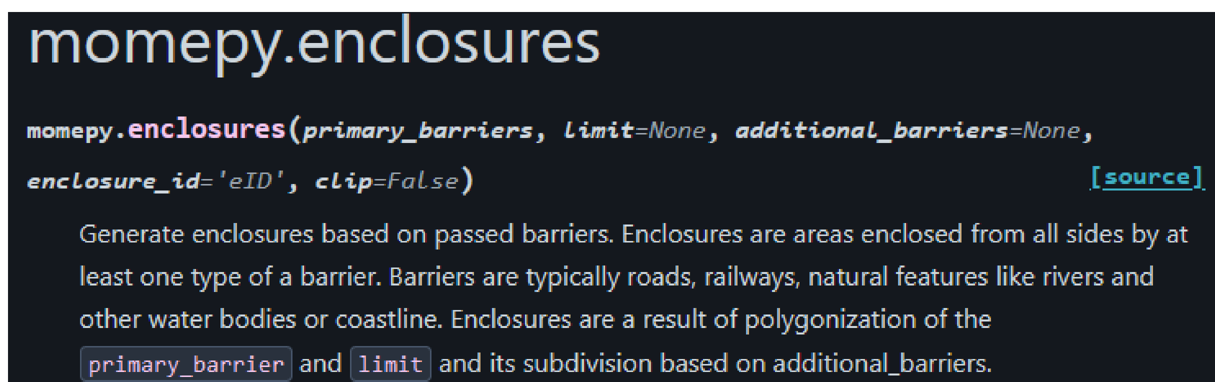

Figure 11 : Création des barrières. Primary_barriers constitue la barrière primaire, souvent un réseau de rues. Additional_barriers utilise d'autres barrières supplémentaires (Fleischmann et al., 2018).

La tessellation avec barrières est généralement plus rapide et moins exigeante en ressources que la tessellation morphologique, car elle utilise la parallélisation (Fleischmann *et al.*, 2018).

Ainsi, la tessellation morphologique et la tessellation avec barrières présentent des différences significatives.

La tessellation morphologique se concentre uniquement sur l'empreinte de chaque bâtiment, sans prendre en compte de barrières externes. Cette méthode s'avère déjà pertinente pour l'objectif de GeoPostcodes d'affiner l'analyse des codes postaux. Elle permet de générer des polygones qui séparent efficacement les habitations, évitant ainsi des chevauchements indésirables. Cependant, ces polygones ne reflètent pas nécessairement les structures urbaines qui peuvent être spécifiques à notre zone.

La tessellation avec barrières offre une solution plus adaptée à ce défi. Cette méthode permet d'intégrer des barrières définies, disponibles via l'Overpass API[3]. Cela permet de personnaliser la tessellation en fonction du contexte urbain de la zone étudiée et permet de dépasser les limitations de la tessellation morphologique. Ainsi, la tessellation avec barrières représente potentiellement une avancée significative pour GeoPostcodes en offrant une analyse plus fine et contextuelle. Ces cellules de tessellations devront bien sûr être liées au semi de points comprenant des valeurs de codes postaux[4].

2.4.3 La tessellation basée sur les blocs

La tessellation basée sur les blocs (*tessellation based-block* en anglais) est une version de la tessellation pour générer des blocs urbains en utilisant une tessellation morphologique. Pour former les blocs, elle se base sur le réseau routier : chaque boucle fermée formera alors un bloc (Fleischmann *et al.*, 2018). Cette approche peut poser un problème si le réseau de rues comporte des trous ou ne forme pas des boucles fermées à certains endroits.

La tessellation basée sur les blocs ne requière ainsi qu'une couche de bâtiments et une couche d'un réseau de rues. La première étape est de générer la tessellation morphologique à partir de

[3] Nous reviendrons sur l'Overpass API dans la partie 2.6.
[4] Pour rappel, les points sont attribués à des bâtiments et contiennent les codes postaux de ces bâtiments. Le but sera donc de fusionner les cellules de tessellation de chaque bâtiment avec le code postal attribué à chacun de ces bâtiments par GeoPostcodes.

ces couches puis de former des blocs en fusionnant les cellules de tessellation se trouvant dans une boucle fermée par un réseau routier.

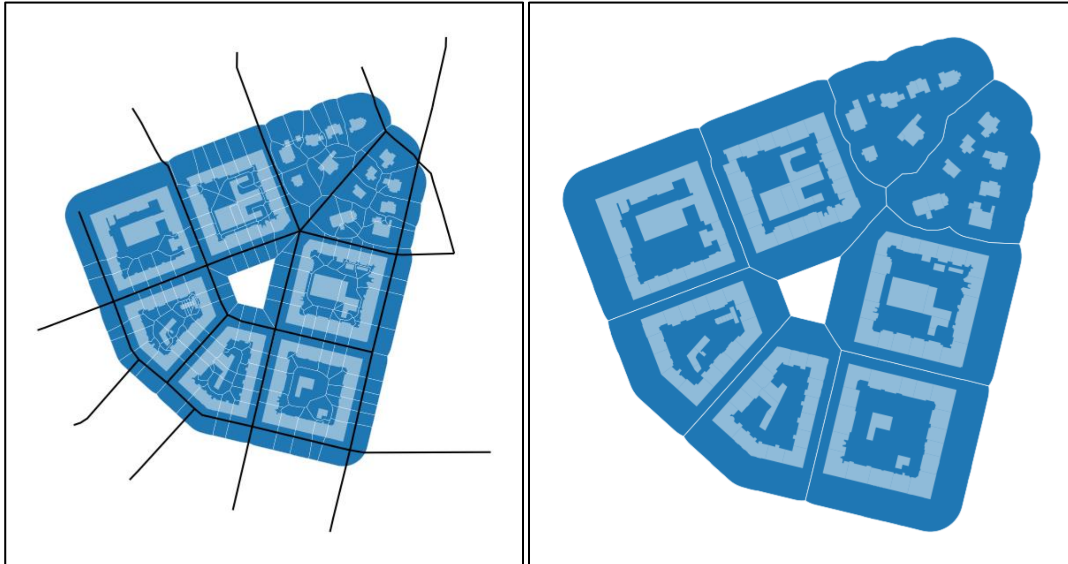

Figure 12.1 : Tessellation morphologique et réseau routier. Figure 12.2 : Tessellation-based blocks résultante (Fleischmann et al., 2018)

La Figure 12.1 montre un réseau de rue à Bubeneč (le dataset de la ville est inclus via momepy), ainsi qu'une tessellation morphologique autour des bâtiments de la zone. La Figure 12.2 montre la tessellation basée sur les blocs à partir de la tessellation morphologique initiale. On peut noter qu'au centre, bien que le réseau routier forme une boucle, il n'y a pas de cellule de tessellation. C'est parce que l'intérieur de la boucle ne contient de pas de bâtiments, et la tessellation morphologique ne se forme qu'autour de bâtiments existants.

2.5 La bibliothèque GeoPandas

GeoPandas est un projet open source qui propose un support pour les données géographiques des objets pandas. Actuellement, il implémente les types *GeoSeries* et *GeoDataFrame*, qui sont respectivement des sous-classes de *pandas.Series* et *pandas.DataFrame*. Les objets GeoPandas peuvent également agir sur des objets géométriques *shapely* et effectuer des opérations géométriques (GeoPandas developers, 2013).

GeoPandas utilise une structure de données *GeoDataFrame* qui est une extension du *DataFrame* de Pandas. Un *GeoDataFrame* peut contenir une ou plusieurs colonnes contenant des géométries (GeoPandas developers, 2013).

2.6 OpenStreetMap, Overpass API et quickOSM

OpenStreetMap est un projet collaboratif qui vise à créer une carte éditable par le monde entier, sur le monde entier. Le projet est lancé en 2004 par Steve Coast. Il repose sur une communauté de volontaires qui y contribuent en collectant des données géographiques à partir de diverses sources telles que des images aériennes, des dispositifs GPS et des relevés de terrain pour collecter et mettre à jour les données. OpenStreetMap contient des données sur les routes, bâtiments, adresses, commerces, points d'intérêts, réseaux ferroviaires, transit et bien d'autres (OpenStreetMap Contributors, n.d.).

Le projet est open source, c'est-à-dire que n'importe quel utilisateur est libre de l'utiliser à toute fin, tant qu'il crédite OpenStreetMap et ses contributeurs. Si une modification des données est effectuée de n'importe quelle manière, le résultat doit être contribué sous la même licence (OpenStreetMap Contributors, 2024).

La carte OpenStreetMap est maintenue par près de 5 millions d'utilisateurs enregistrés et plus d'un million de contributeur à travers le monde. Le soutien financier provient des membres de la Fondation OpenStreetMap, y compris des membres corporatifs, ainsi que des dons (OpenStreetpMap Contributors, n.d.).

Il est également possible d'utiliser une API. L'Overpass API est une API en lecture seule qui fournit des parties sélectionnées personnalisées des données cartographiques d'OSM. Elle agit comme une base de données sur le web : le client envoie une requête à l'API et reçoit en retour l'ensemble de données correspondant à la requête. Les différents éléments sont sélectionnés par des critères de recherche tels que la localisation, le type d'objet, les propriétés des tags, la proximité ou des combinaisons de ceux-ci. Un manuel complet d'utilisateur est disponible pour se familiariser avec l'Overpass et il est également possible de le faire via overpass turbo, une interface web interactive (OpenStreetMap Contributors, 2024).

Nous avons besoin de l'Overpass API pour séparer au mieux l'espace urbain et effectuer la tessellation, afin d'utiliser une source de données avec l'emprise spatiale des bâtiments. Momepy permet d'accéder à ces données en se connectant à l'Overpass API (Fleischmann *et al.*, 2018). Pour utiliser les données de l'Overpass API, nous utilisons le package Python OSMnx, qui permet de télécharger, modéliser, analyser et visualiser des caractéristiques géospatiales comme les réseaux routiers ou l'emprise des bâtiments (Boeing, 2024).

De plus, cela nous donne alors un accès à des structures urbaines telles que les routes, les limites administratives, les pistes cyclables, etc. De nombreuses données sont disponibles, et même pas seulement de la structure urbaine. Il est également possible d'y ajouter des rivières, des points d'eau, des parcs naturels… La liste est longue et disponible sur le wiki d'OpenStreetMap (OpenStreetMap Contributors, 2024). Nous reviendrons vers ces structures dans la partie sur la tessellation avec barrières.

QuickOSM sera également utilisé pour visualiser des données OpenStreetMap via QGIS. QGIS est un logiciel SIG libre multiplateforme publié sous licence GPL. Il permet de créer des cartes, d'éditer des couches, ainsi que de les analyser (QGIS, 2024). Il s'agit également d'un projet open source. Concernant QuickOSM, il s'agit d'un plugin qui permet de télécharger les données OSM via l'Overpass API (QGIS, n.d.).

2.7 Base de données PostGIS

Les données de GeoPostcodes sont stockées sur une base de données PostgreSQL. Il s'agit d'un SGBD open source prenant en charge les requêtes SQL relationnelles. (PostgreSQL Tutorial, 2011). Elle possède également une extension de base de données géospatiale : PostGIS.

PostGIS étend les fonctionnalités de la base de données relationnelle PostgreSQL en ajoutant la prise en charge du stockage, de l'indexation et de l'interrogation des données géospatiales. Les fonctionnalités de PostGIS comprennent (PostGIS PSC & OSGeo, 2023) :

- Stockage de données spatiales : stockage de points, de lignes, de polygones et de multi-géométries, à la fois en 2D et en 3D.
- Indexation spatiale : recherche et récupération des données spatiales en fonction de leur emplacement.
- Fonctions spatiales : filtre et analyse de données spatiales, de mesurer des distances et des surfaces, d'intersecter des géométries, de créer des zones tampons, et plus encore.
- Traitement de la géométrie : traitement et manipulation des données de géométrie, tels que la simplification, la conversion et la généralisation.
- Prise en charge des données raster : stockage et traitement des données raster, telles

que les données d'élévation et les données météorologiques.

- Géocodage et géocodage inverse : fonctions de géocodage et de géocodage inverse.
- Intégration : accès à PostGIS en utilisant des outils tiers tels que QGIS, GeoServer, ArcGIS, etc.

L’extension PostGIS nous permettra d’effectuer des requêtes spatiales sur nos résultats de tessellation, directement à partir de la base de données de GeoPostcodes.

3 MÉTHODOLOGIE

3.1 Les données des codes postaux fournies par GeoPostcodes.

Pour rappel, la BD de GeoPostcodes fournit des tables de codes postaux de différents pays. Nous avons premièrement une table de géométries sous forme de polygones, comprenant les premiers caractères d'un code postal et deuxièmement, nous avons des tables de codes postaux, comprenant l'entièreté des caractères, sous forme d'un semi de points.

3.2 But du mémoire et proposition d'un programme informatique

Dans le cadre de ce mémoire, nous proposons un programme informatique codé en python. Le but de ce programme va être d'aller au-delà des codes postaux actuels globaux (sous forme de polygones) en augmentant la granularité et en proposant des polygones plus précis, en se basant sur les différentes tessellations proposées par momepy. Pour cela, nous allons créer des cellules de tessellation avec les trois tessellations que nous avons décrites auparavant : la tessellation morphologique, la tessellation avec barrières et la tessellation basée sur les blocs. Ces tessellations vont nous permettre de créer de nombreux polygones, chacun centré sur un bâtiment (dans le cas de la tessellation morphologique et de la tessellation avec barrières) ou sur un bloc de bâtiments (pour la tessellation basée sur les blocs).

Ensuite, il faudra attribuer des codes postaux aux cellules créées par la tessellation. Ces codes postaux sont originaires du semi de points de GeoPostcodes. Les cellules seront alors fusionnées en fonction des codes postaux qui leur auront été attribués. Cela permettra donc d'aller plus loin dans la granularité, en se basant sur la tessellation pour partitionner l'espace. Nous le verrons, certaines tessellations seront préférables à d'autres pour cette tâche, bien que cela dépendra du contexte.

Une demande majeure de GeoPostcodes est d'éviter au maximum des « trous » dans la tessellation. Une zone sur laquelle une tessellation est appliquée doit être recouverte au maximum.

Ainsi, notre programme proposera à l'utilisateur de choisir un lieu. Le lieu choisi devra correspondre à la nomenclature acceptée par l'Overpass API[5].

Ensuite l'utilisateur se verra proposer plusieurs choix de tessellation : morphologique, avec barrières, basée sur les blocs. Pour la tessellation avec barrières et celle basée sur les blocs, l'utilisateur se verra également proposer d'ajouter des barrières additionnelles, en plus de la barrière primaire[6]. Nous reviendrons sur ces barrières lorsque nous analyserons plus en détail la tessellation avec barrières et la tessellation basée sur les blocs.

Pour chacune des tessellations qui ont été acceptées par l'utilisateur, celui-ci se voit alors proposer de sauvegarder les tessellations résultantes dans la BD de GeoPostcodes[7]. Enfin, il lui sera également proposé la possibilité de combiner les différentes cellules de tessellations en fonction des codes postaux[8]. Le résultat sera alors sauvegardé dans la BD de GeoPostcodes. Nous verrons qu'il est possible que certaines cellules de tessellation n'aient pas de points attribués par GeoPostcodes, soit parce que la cellule ne contient pas de bâtiments (ce sera souvent le cas de la tessellation fermée), soit parce que GeoPostcodes n'a pas encore de données postales sur les bâtiments. Dans ces cas, nous attribuerons à ces cellules les codes postaux plus généraux (ceux issus des polygones déjà existants), suivis du terme *undefined*. Nous y reviendrons dans la partie 3.5.

Pour la suite de la méthodologie, nous nous concentrons sur l'explication de certaines parties du code, celui-ci étant disponible en annexe.

3.3 Décomposition des polygones

Dans la documentation momepy, il est expliqué que les bâtiments considérés comme des multipolygones viennent généralement de données imprécises. Par exemple des bâtiments très fins ou très longs. Même si l'analyse peut potentiellement marcher avec ce type de géométrie, cela ne fait pas toujours sens. Il est alors conseillé de fixer les données en pré-traitement

[5] Si l'input d'un lieu est accepté sur le site web OpenStreetMap et que le résultat correspond aux attentions, alors il fonctionnera dans notre programme.
[6] Généralement, la barrière primaire sera un réseau de rues.
[7] Le code a été pensé pour la société GeoPostcodes. Si un utilisateur souhaite utiliser ses propres données et sa propre base de données, il devra effectuer des changements dans le code.
[8] Pour cela, nous utiliserons une requête SQL, directement dans le code python.

(Fleischmann *et al.*, 2018). Il faut donc décomposer les géométries téléchargées par OSMnx lors du téléchargement des bâtiments.

```
def explode_geometry(geom):
    if geom.geom_type == 'GeometryCollection':
        return [explode_geometry(part) for part in geom.geoms]
    elif geom.geom_type == 'MultiPolygon':
        return [explode_geometry(part) for part in geom.geoms]
    elif geom.geom_type in ['Polygon']:
        return [geom]
    else:
        return []
```

Extrait de code 1 : Décomposition des géométries

Dans cette fonction, nous décomposons les collections de géométries et les multipolygones, jusqu'à ce que les géométries ne soient plus que des polygones.

3.4 Les 3 différents types de tessellations exploitées

Dans le code, 3 types de tessellations sont proposées : la tessellation morphologique, la tessellation avec barrières[9] et la tessellation basée sur les blocs. L'utilisateur choisit un lieu ainsi que la ou les tessellations qu'il désire.

3.4.1 La tessellation morphologique

```
def tessellation():
    print(f"Exécution de la tessellation morphologique")
    try:
        print(f"Tessellation morphologique pour : {r_lieu}...")
        gdf = ox.features.features_from_place(r_lieu, tags={'building':True})
    except Exception as e:
        print(f"Erreur de chargement des bâtiments de {r_lieu} sur Openstreet-
Maps():{str(e)}")
        return None
```

Extrait de code 2 : téléchargement des bâtiments

Cette tessellation est la plus basique. Nous utilisons la fonction *ox.features.features_from_place()* de la bibliothèque OSMnx (Boeing, 2024) pour récupérer les

[9] Appelée également tessellation fermée.

données des bâtiments pour un lieu spécifié par l'utilisateur. Les données sont ensuite stockées dans un GeoDataFrame.

Ensuite, nous appliquons la méthode *explode()* pour décomposer les géométries en polygones individuels. Les collections de géométries ainsi que les multipolygones sont alors décomposés en simples polygones. Nous appliquons cette méthode pour éviter de nous retrouver avec des collections de géométries ou des multipolygones, que la documentation momepy conseille de ne pas traiter pour la tessellation.

```
    try:
        gdf = gdf.explode()
        geometries = gdf.geometry.apply(explode_geometry)
        gdf = gdf.loc[geometries.astype(bool)]
        gdf.geometry = geometries[geometries.astype(bool)].apply(lambda x:
x[0])
        print(f"Décomposition des géométries réussie !")
    except Exception as e:
        print(f" Erreur lors de la décomposition :{str(e)}")
        return None
```

Extrait de code 3 : Décomposition des géométries

Nous utilisons ensuite la fonction *ox.projection.project_gdf()* pour projeter le GeoDataFrame dans un système de coordonnées approprié. En règle générale, il est judicieux d'utiliser le système de coordonnées WGS84, qui est le système de coordonnées standard GPS (GISGeography, 2024).

La fonction *momepy.preprocess()* est alors utilisées pour prétraiter les bâtiments avant d'effectuer la tessellation morphologique. L'objectif principal de cette fonction est de gérer les cas où les bâtiments sont séparés en plusieurs éléments au lieu d'être représentées par une seule géométrie (Fleischmann *et al.*, 2018).

Ensuite, un identifiant unique est attribué aux bâtiments et une limite est définie. Cette limite peut être définie par la fonction *momepy.buffered_limit()*, qui va définir un buffer maximum autour des bâtiments où la tessellation aura lieu. Enfin, nous appliquons la tessellation à l'aide de la fonction *momepy.Tessellation()*, celle-ci ayant besoin de la couche des bâtiments, de l'identifiant unique des bâtiments ainsi que de la limite que nous avons précédemment définie.

Nous ajoutons également un attribut *has-buildings*[10]. La fonction retourne ensuite les cellules de tessellation.

```
    gdf_projected = ox.projection.project_gdf(gdf, to_crs = 3857)

    buildings = momepy.preprocess(gdf_projected.reset_index(), size=30,
                                  compactness=0.2, islands=True)

    buildings['uID'] = momepy.unique_id(buildings)
    limit = momepy.buffered_limit(buildings, buffer=500)

    tessellation = momepy.Tessellation(buildings, unique_id='uID',
limit=limit)
    tessellation_gdf = tessellation.tessellation
    tessellation_gdf['has_buildings'] = tessellation_gdf.geometry.apply(lambda
x: any(buildings.intersects(x)))

    return tessellation_gdf
```

Extrait de code 4 : Création des cellules de la tessellation morphologique

Pour rappel, la tessellation morphologique utilise la tessellation de Voronoï. Cela créée des cellules de tessellation où chaque point de la cellule est le plus proche du bâtiment à l'origine de la tessellation.

3.4.2 La tessellation avec barrières/fermée

La tessellation avec barrières, appelée également tessellation fermée (ou *enclosed tessellation* en anglais) a pour objectif de proposer plusieurs barrières qui fermeront nos cellules de tessellations. Dans notre cas, il s'agira principalement des rues ainsi que d'autres barrières additionnelles choisies par l'utilisateur.

Premièrement, comme pour la tessellation morphologique, nous téléchargeons les données des bâtiments grâce à la bibliothèque OSMnx pour un lieu spécifié par l'utilisateur. Ensuite, nous utilisons la fonction *ox.graph_from_place()* pour récupérer le réseau routier de la zone spécifiée. Dans cette fonction, le paramètre *network_type* nous permet de sélectionner le type de rues qui sera pris en compte dans le réseau routier. Nous l'avons défini sur *all* pour pouvoir prendre en

[10] Nous y reviendrons dans la partie 3.5 sur l'attribution de codes postaux aux cellules de tessellation.

compte tous les types de réseaux routiers[11]. Dans certains cas, nous verrons qu'il est également utile de le définir sur *drive*. Cela reprend uniquement les routes utilisables par des véhicules motorisés.

```
try:
    #streets_graph = ox.graph_from_place(r_lieu, network_type='all')
    streets_graph = ox.graph_from_place(r_lieu, network_type='drive')
    streets_graph = ox.projection.project_graph(streets_graph, to_crs = 3857)
    streets = ox.graph_to_gdfs(streets_graph, nodes=False, edges=True,
                                node_geometry=False, fill_edge_geometry=True)
    print(f"Inclusion des rues pour enclosed tessellation...")
except Exception as e:
    print(f"Erreur pour inclusion des rues pour l'enclosed tessellation: {str(e)}")
```

Extrait de code 5 : Inclusion de routes dans la tessellation fermée

Les barrières supplémentaires

Deuxièmement, l'utilisateur est invité à choisir s'il souhaite inclure des barrières supplémentaires autres que les routes. Les options proposées par ce code sont notamment[12] :

- Les autoroutes
- Les rivières
- Les voies ferrées
- Les canaux
- Les murs.

Nous avons également inclus des barrières reprenant les chemins pour piétons et les pistes cyclables. Ces deux barrières sont reprises dans la barrière primaire *streets* (qui constitue le réseau de rues) lorsque nous paramétrons *network_type* sur *all*. Dans les cas où nous ne voudrons pas prendre en compte les chemins pour piétons[13] ou les pistes cyclables, il est donc intéressant de paramétrer notre *network_type* uniquement sur les routes accueillant des véhicules (*drive*) et proposer les chemins pour piétons ainsi que les pistes cyclables en option.

[11] Les types de réseaux routiers de *network_type* sont définis par la bibliothèque OSMnx et sont accessibles dans sa documentation (Boeing, 2024).
[12] D'autres barrières, placées en tant que commentaires, sont également disponibles dans le code. Elles correspondent à des besoins plus spécifiques.
[13] Nous verrons dans les résultats qu'il n'est pas toujours désirable d'inclure les chemins pour piétons parallèles aux rues empruntées par les voitures, car cela peut créer plusieurs barrières parallèles.

Ces barrières ont été sélectionnées sur base d'observations et de tests effectués durant la réalisation de ce mémoire. La sélection de barrières dépend du contexte urbanistique et des objectifs de l'utilisateur, mais il dépend également de la réaction de la tessellation à l'inclusion de certaines barrières. En effet, certaines barrières donnent parfois des résultats peu satisfaisants. Par exemple, l'inclusion des parcs urbains dans les barrières formera des « trous » dans la tessellation car les parcs urbains ne contiennent généralement pas de bâtiment. Comme la tessellation est effectuée autour des bâtiments, les parcs correspondront à des zones vides[14].

Pour visualiser les barrières qui peuvent structurer une zone, il est tout à fait possible de se rendre sur le site web d'OpenStreetMap. Ensuite, il faut sélectionner l'outil *query features*. Celui-ci permet à un utilisateur de cliquer sur un endroit de la carte et d'afficher les éléments proches de l'endroit sélectionné.

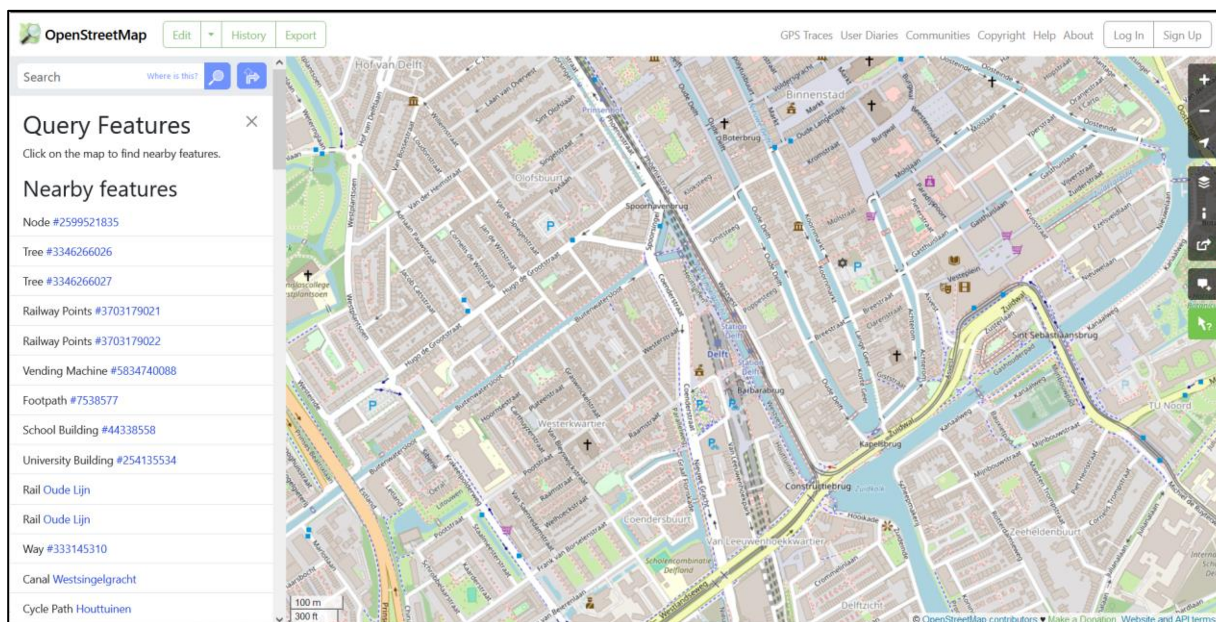

Figure 13 : Query features : détection des barrières (OpenStreetMap, 2024)

Ainsi, il est aisé de détecter les différents éléments qui structurent la ville. Sur la Figure 13, nous nous trouvons dans la ville de Delft, aux Pays-Bas et nous observons des plans d'eaux linéaires qui structurent la ville.

[14] Pour rappel, GeoPostcodes souhaite obtenir une couverture complète du territoire. Les barrières formant des zones vides dans la tessellation n'ont donc pas été prises.

Figure 14 : Exemple tag Delft, Pays-Bas (OpenStreetMap, 2024)

Avec l'outil *query features*, nous pouvons voir qu'il s'agit de canaux. En cliquant sur les objets en question, cela nous donne le tag correct, que nous pouvons ensuite appliquer dans notre tessellation. Par exemple le tag de ces canaux est : *waterway : canal*. Cette méthode fut utilisée dans de ce mémoire pour déterminer les barrières pertinentes à utiliser afin de limiter la tessellation.

Par exemple, nous avons les rivières, que nous pouvons également lier avec les canaux. Certaines villes peuvent être structurées par des canaux ou des cours d'eau et la tessellation avec barrières permet alors d'utiliser ces structures comme barrières pour séparer les zones postales. Un cas typique est la ville de Delft au Pays-Bas, possédant de nombreux canaux. Nous y reviendrons dans la partie 4.1.2, qui concerne les résultats de la tessellation fermée.

Les vois ferrées sont également présentes dans de nombreuses villes et il paraissait évident de les inclure. Il faut tout de même différencier les nombreux types de voies ferrées que nous pouvons trouver. Par exemple dans certaines villes, les voies ferrées peuvent structurer la zone mais celles-ci peuvent être toujours en activité ou désuètes. Elles peuvent également être de différents types en fonction du type de transport passant sur les rails. Les voies ferrées possèdent de nombreux tags dans la documentation d'OpenStreetMap.

Nous pourrions être tentés d'utiliser le tag plus général *route : railway* pour toutes les prendre en compte, malheureusement cela ne marche pas car l'Overpass API requiert des tags précis. Par exemple, la gare *Amsterdam Centraal* se situe dans le centre d'Amsterdam et pour prendre en compte les voies ferrées, nous devons obligatoirement utiliser le tag *railway : rail*, qui correspond aux voies ferrées en activité. Le tag *route : railway* ne fonctionnera pas (voir Figure 15).

Toutefois, les voies ferrées ne donnent pas toujours des résultats optimaux. Dans le cas d'une gare par exemple, où de multiples rails se trouvent, nous aurons plusieurs barrières qui seront les unes à côté des autres.

Les murs ont également été inclus, ceux-ci pouvant entourer des blocs de bâtiments.

Bien sûr, de nombreuses autres barrières peuvent être envisagées. Les tags sont disponibles sur le Wiki d'OpenStreetMap (voir Figure 17).

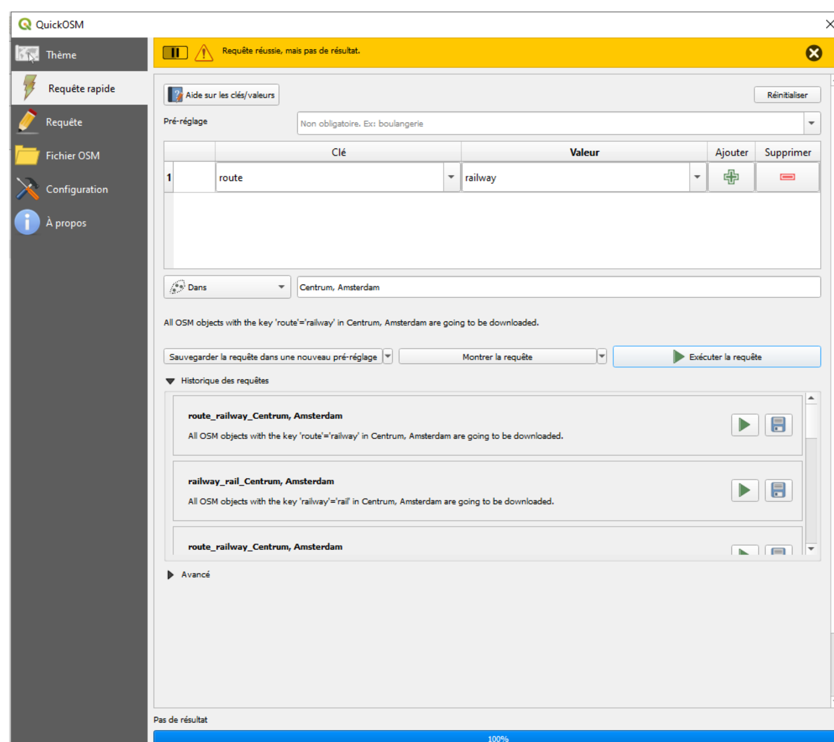

Figure 15 : Utilisation du tag route : railway pour télécharger les voies ferrées dans le quartier de la gare Amsterdam Centraal, aux Pays-Bas. Cette méthode ne renvoie aucun résultat (QGIS, 2024).

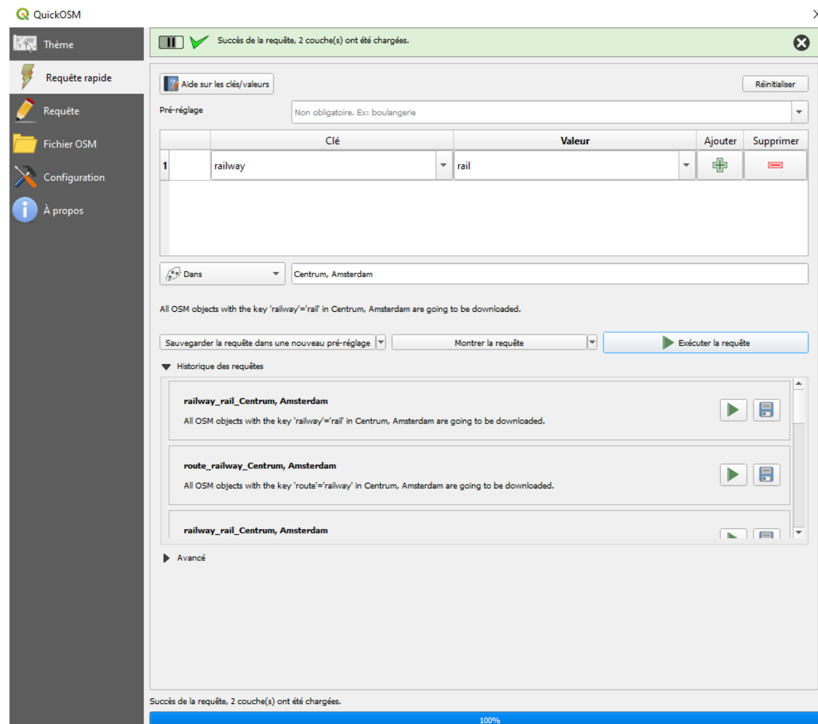

Figure 16 : Utilisation du tag railway : rail pour télécharger les voies ferrées dans le quartier de la gare Amsterdam Centraal, aux Pays-Bas. Cette méthode fonctionne car le tag correspond à celui de l'Overpass API (QGIS, 2024).

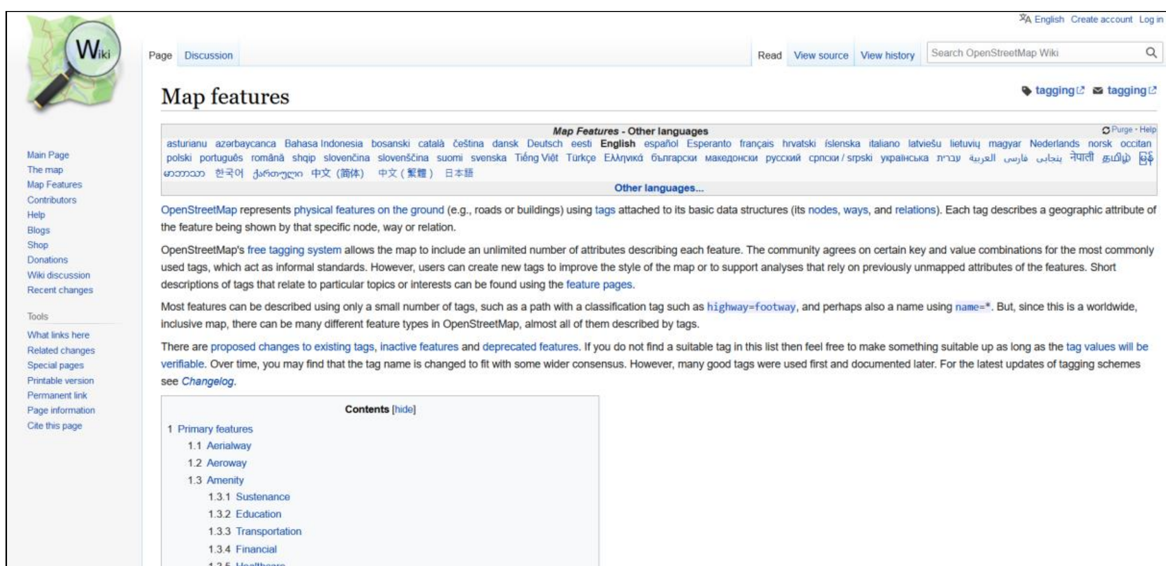

Figure 17 : Les tags sont disponibles sur cette page (OpenStreetMap Contributors, 2024).

Parmi toutes les barrières choisies, nous l'avons déjà mentionné, l'inclusion des parcs urbains ne donnait pas de résultats satisfaisants. Nous avions également choisi d'autres barrières qui semblaient être utiles mais qui n'ont pas donné de résultats. Par exemple, dans les cas où les territoires se trouvent à proximité de points d'eau, comme la mer ou encore des lacs, il aurait été intéressant de créer une barrière sur la limite terrestre pour éviter que la tessellation déborde sur l'eau (voir Figure 18).

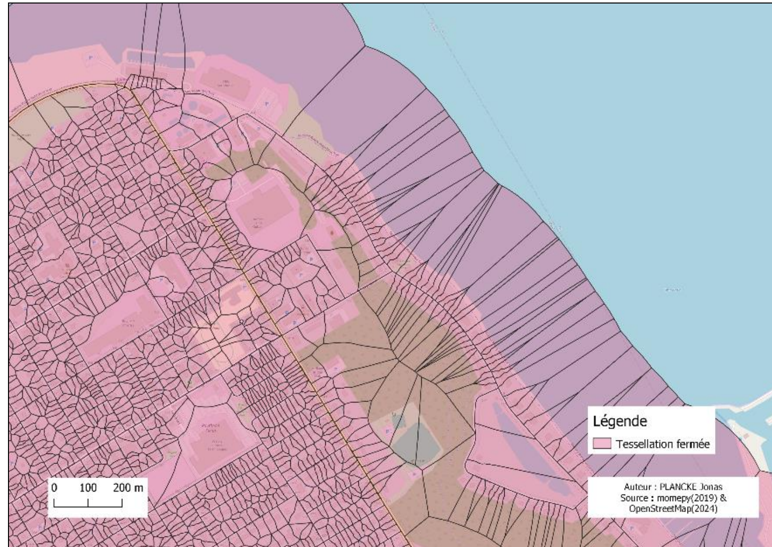

Figure 18 : tessellation débordant sur un lac à Midland, Canada (momepy(2019) & OpenStreetMap(2024)).

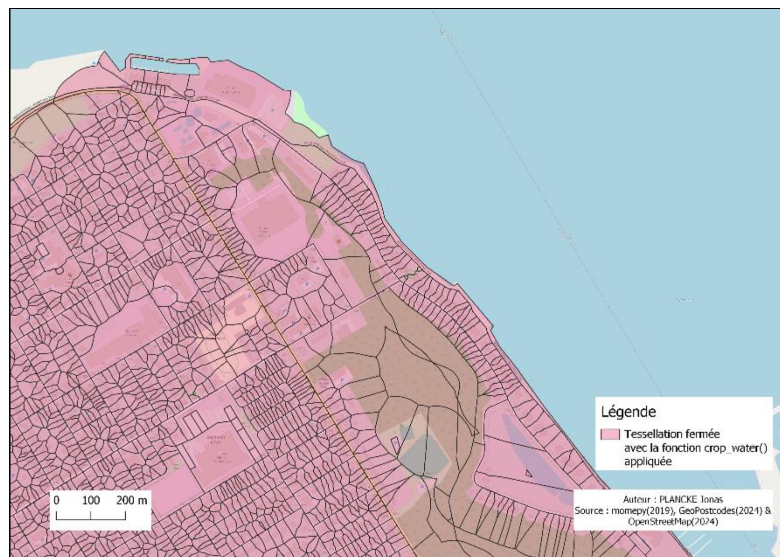

Figure 19 : Correction de la tessellation grâce à la fonction crop_water() de GeoPostcodes (momepy(2019), GeoPostcodes(2024) & OpenStreetMap(2024)).

Certains tags sont disponibles pour opérer cette séparation. Il y a par exemple *natural=coastline*, tag utilisé pour représenter la ligne de côte, ou encore *natural=water,* qui est utilisé pour représenter n'importe quelle étendue d'eau, comme les océans, les mers, les lacs, les rivières, etc. L'utilisation de ces tags n'a malheureusement rien donné avec l'Overpass API avec des temps de traitements trop longs. Pour contrecarrer ce problème, GeoPostcodes possède une fonction *crop_water()*, qui permet de découper avec précision les polygones en bordure d'eau (voir *Figure* 19).

```
        # Rivières
        demande_river = input(f"Voulez-vous inclure les rivières dans l'enclosed tessellation ? (o/n)")
        if demande_river.lower() == 'o':
            try:
                print(f"Téléchargement des rivières pour enclosed tessellation...")
                rivers = ox.features_from_place(r_lieu, tags={"waterway": "river"})
                rivers_projected = ox.projection.project_gdf(rivers, crs = 3857)
                additional_barriers.append(rivers_projected)
                print(f"Inclusion des rivières pour enclosed tessellation effectuée !")
            except Exception as e:
                print(f"Erreur pour l'inclusion des rivières : {str(e)}")
```

Extrait de code 6 : ajout d'une barrière

De manière générale, si une personne souhaite rajouter une barrière au code, il suffit de procéder comme l'exemple de barrière montré sur l'Extrait de code 6. Il faut remplacer le tag par celui qui intéresse l'utilisateur et changer les noms en conséquence : le nom de l'*input* (ici : demande_river) pour savoir si l'utilisateur souhaite ou non inclure ladite barrière, ainsi que les noms de la barrière et de la barrière projetée[15]. La nouvelle barrière sera incluse dans l'ensemble des barrières, appelé *additional_barriers* dans ce code.

<u>Les barrières non sélectionnées</u>

En effet, initialement, un plus grand nombre de barrières avaient été sélectionnés. Toutefois, elles ne réagissaient pas comme ce qui aurait pu être attendu. Voici quelques barrières qui furent initialement incluses :

- Les niveaux administratifs. Tag OpenStreetMap : *admin_level : (number)*. Cela correspond à : « Définit le niveau de la division décrite par la frontière dans le système hiérarchique auquel appartient la division. Généralement dans la gamme de 1 à 10 (sauf pour plusieurs pays, où il se situe dans la gamme de 1 à 11 – Bolivie, Allemagne, Mozambique, Pays-Bas, Philippines, Pologne, Turkménistan, Venezuela). » (Source : OpenStreetMap Contributors, 2024).
- Les parcs urbains. Tag OpenStreetMap : *leisure : park*. Cela correspond à : « Un parc, généralement situé en milieu urbain (municipal), créé pour la détente et les loisirs. ») (Source : OpenStreetMap Contributors, 2024).
- Les points d'eaux. Tag OpenStreetMap : *natural : water.* Cela correspond à : « Tout plan d'eau intérieur, qu'il soit naturel comme un lac ou un étang, ou artificiel comme un fossé ou un canal (Source : OpenStreetMap Contributors, 2024).

[15] On reprojette la barrière dans le système de coordonnées désiré.

- Le littoral. Tag OpenStreetMap : *natural : water.* « La ligne des plus hautes eaux (marées de printemps) entre la mer et la terre (avec l'eau du côté droit du tracé). ». (Source : OpenStreetMap Contributors, 2024).
- Etc.

Nous avons déjà mentionné le problème des parcs urbains, pourtant fréquents dans les villes : puisque ces éléments ne possèdent généralement pas de bâtiments, il n'y aura pas de cellule de tessellation créée et ces éléments serviront juste de barrières pour stopper les tessellations extérieures. On se retrouve alors avec des cas typiques de « trous ». Nous pouvons extrapoler ce problème en incluant tous les éléments formants des polygones qui ne contiennent pas de bâtiments comme les parkings, les plaines de jeux, les lacs, les forêts, etc.

Pour les points d'eaux, le problème vient du temps de réponse de l'Overpass API. Comme celle-ci prenait trop de temps à répondre, la barrière n'a pas pu être incluse dans les barrières finales. Pourtant, cette barrière « plus générale » permet de prendre en compte plusieurs autres barrières. Par exemple, on peut retrouver des zones avec plusieurs points d'eau : des lacs, des rivières, des canaux, etc. Au lieu d'utiliser plusieurs tags pour les définir, il aurait suffi d'utiliser *natural : water* pour considérer tous les points d'eau comme des barrières.

Dans la version finale du code, les canaux ont été sélectionnés comme barrière mais celle-ci ne peut en réalité pas prendre en compte tous les canaux car certains peuvent avoir des tags plus précis ou légèrement différents sur OpenStreetMap. Il est alors tentant d'utiliser des tags plus généraux, mais ceux-ci correspondront à une demande plus intensive pour l'Overpass API.

Pour le littoral, la raison est la même que celle-ci-dessus : le temps finit par dépasser la limite de l'Overpass API et la barrière ne peut pas se charger. Pourtant, cela aurait pu éviter d'avoir des tessellations qui « débordent » au-delà de la bande de terre, comme nous l'avons vu sur la Figure 18.

Concernant les limites administratives, celles-ci avaient deux objectifs : être utilisées comme simples barrières mais également délimiter la portée maximale de la tessellation en servant de limite à celle-ci. En effet, la documentation de momepy précise que : « Comme la tessellation de Voronoï tend vers l'infini pour les points de bord, nous devons définir une limite pour la tessellation. Cela peut être la zone de votre étude de cas représentée sous forme de Polygone ou de MultiPolygone, ou vous pouvez utiliser *momepy.buffered_limit()* pour générer une telle

limite comme une distance maximale définie à partir des bâtiments. » (Fleischmann *et al*., 2018). Dans le cadre de ce mémoire, les limites administratives auraient été définies comme la limite maximale indépassable par la tessellation. Dans certains cas, les limites administratives correspondent entre l'Overpass API et le site web OpenStreetMap. C'est le cas de la ville de Delft, aux Pays-Bas comme nous pouvons le voir sur la Figure 20 et la Figure 21.

Dans les cas où l'Overpass API permet de définir clairement la limite administrative choisie par l'utilisateur pour une zone, il devient théoriquement aisé de définir comme barrière ou limite la dite-zone et avoir une tessellation correcte. Nous verrons que ce n'est pas forcément le cas dans la partie 4.2.

De plus, il peut y avoir d'autres zones où les résultats ne correspondent pas entre le site web et l'Overpass API. C'est par exemple le cas de la ville de Midland, au Canada comme nous pouvons le constater sur la Figure 22 et la Figure 23.

Dans ces cas-là, il est alors impossible de définir comme barrière ou comme limite la limite administrative de la zone qui intéresse l'utilisateur. En règle générale, les niveaux administratifs sont numérotés de 1 à 10, à l'exception de plusieurs pays où les numérotations vont de 1 à 11, comme c'est le cas de la Bolivie, de l'Allemagne, du Mozambique, des Pays-Bas, des Philippines, de la Pologne, du Turkménistan et du Venezuela (OpenStreetMap Contributors, 2024). Un niveau moins élevé correspond à une zone plus large. En raison de différences culturelles et politiques, les niveaux administratifs des différents pays ne correspondent qu'approximativement les uns aux autres. Par exemple, une entité avec *admin_level=7* qui se trouve dans une juridiction de niveau *admin_level=4* (juridiction sous-nationale), se trouvant également dans un pays (*admin_level=2*) n'est pas exactement le même type d'entité qu'une entité avec *admin_level=7* dans une autre juridiction sous-nationale, même si elles sont dans le même pays (OpenStreetMap Contributors, 2024).

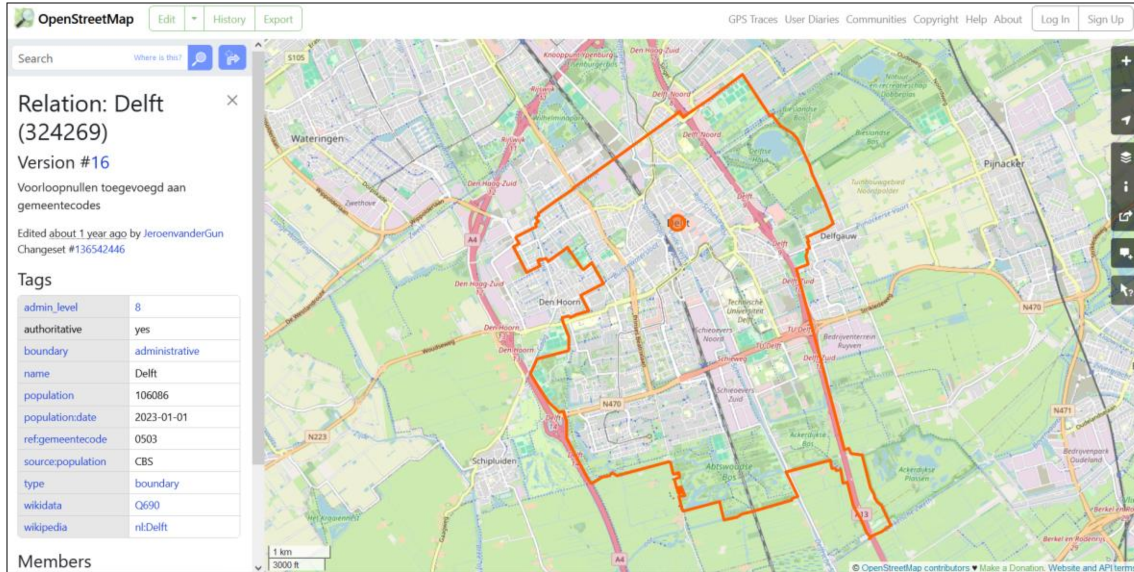

Figure 20 : Niveau administratif pour Delft, Pays-Bas (Openstreetmap, 2023)

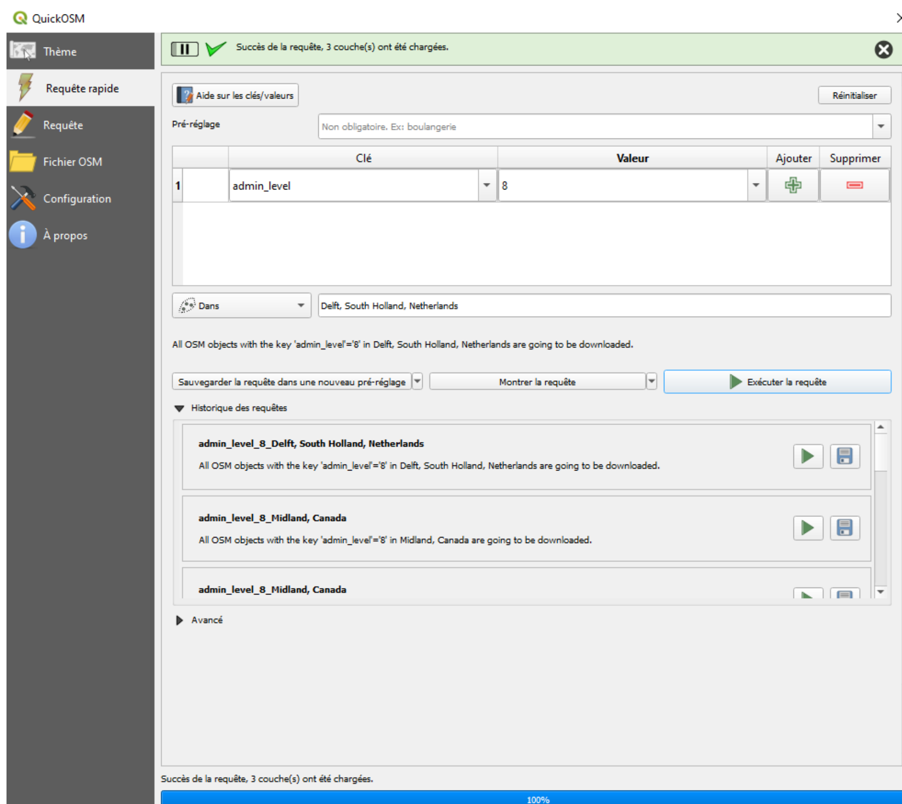

Figure 21 : exécution sur QuickOSM via QGIS pour le niveau administratif de Delft, Pays-Bas. Nous voyons que la requête a été exécutée avec succès pour le même niveau administratif (QGIS, 2023).

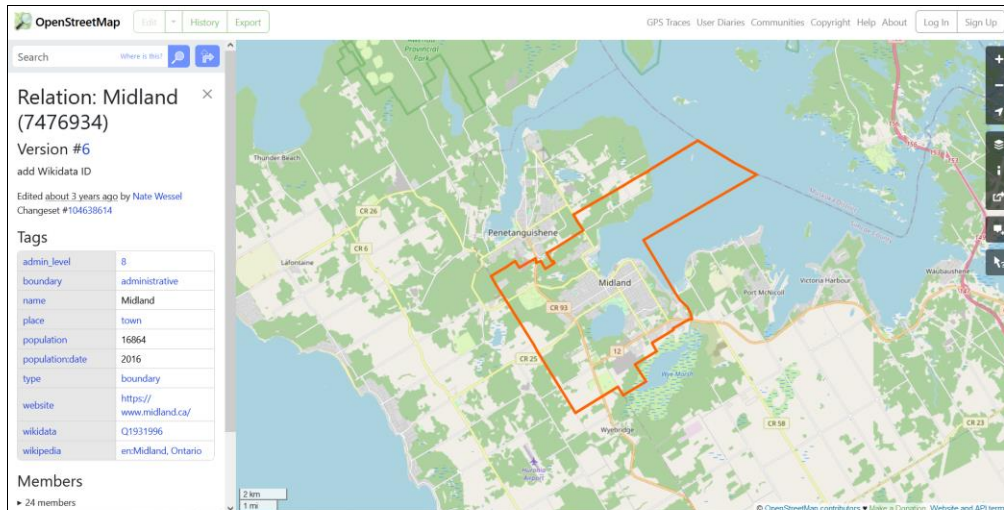

Figure 22 : Niveau administratif pour Midland, Canada (Openstreetmap, 2023)

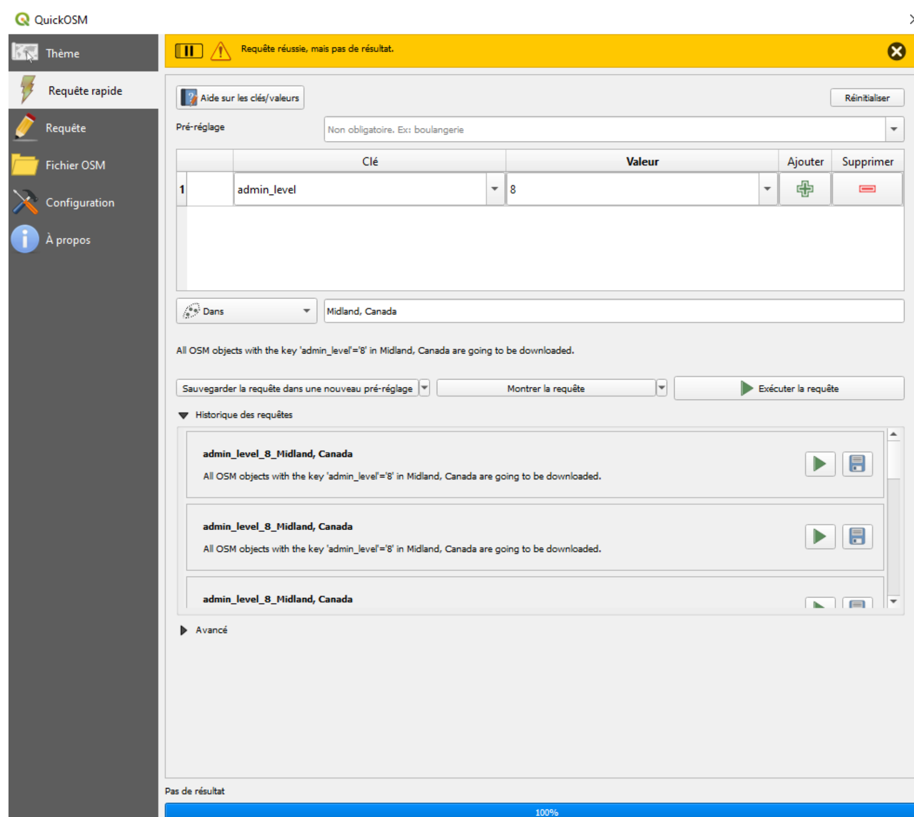

Figure 23 : exécution sur QuickOSM via QGIS pour le niveau administratif de Midland, Canada. Nous voyons que la requête n'a pas été acceptée (QGIS, 2023).

Dans notre code, nous avons beaucoup travaillé sur des villes. Par exemple, la ville de Delft correspond à un niveau administratif de 10, la ville de Salvador a un niveau administratif de 8, la ville de Québec a un niveau 8, la ville d'Erechim a un niveau 8, la ville de Vancouver a un niveau 8, etc. Bien qu'on puisse trouver des niveaux similaires, il n'est pas possible d'en faire une règle

générale et de définir un niveau administratif dans notre code qui marchera pour chaque ville, indépendamment du lieu choisi.

Toutefois, si GeoPostcodes souhaite effectuer la tessellation pour un pays entier, les niveaux administratifs correspondent généralement au niveau 2. Ainsi, en incluant le niveau administratif comme limite de la tessellation sur un pays entier, celle-ci devrait pouvoir être exécutée sans encombre. Malheureusement, il était impossible de tester cela dans le cadre de ce mémoire car l'exécution de la tessellation pour un pays demande une puissance de calcul et un temps trop élevé pour la machine utilisée. De plus, nous le verrons dans les résultats, les géométries des limites ne correspondent pas toujours entre ce qu'on peut observer sur le site web OpenStreetMap et ce qui est téléchargé via l'Overpass API.

C'est pour cela que les niveaux administratifs n'ont pas été retenus comme barrière pertinente pour la tessellation avec barrières, ni comme limites de la tessellation. Toutefois, les niveaux administratifs ont été laissés en commentaire dans le code[16].

Dans la partie dédiée aux résultats, nous montrerons plus en détail les problèmes que certaines barrières peuvent engendrer, aussi bien par leur présence, que par l'absence de certains tags.

<u>Quelques remarques et conclusion générale sur les barrières</u>

Les tags peuvent donc être utilisés pour définir des barrières de manière efficace. Encore faut-il que ces barrières se rejoignent dans une boucle fermée. Dans ces cas-là, les parties qui ne rejoignent pas le reste du réseau ne seront pas utilisées comme barrières par momepy. Un exemple est montré dans la documentation de momepy (Fleischmann *et al.*, 2018).

Momepy propose également des outils pour « réparer » le réseau de rues en prolongeant les segments pour les connecter entre eux ou leur faire toucher les bords de la zone.

[16] Dans la fonction de la tessellation fermée.

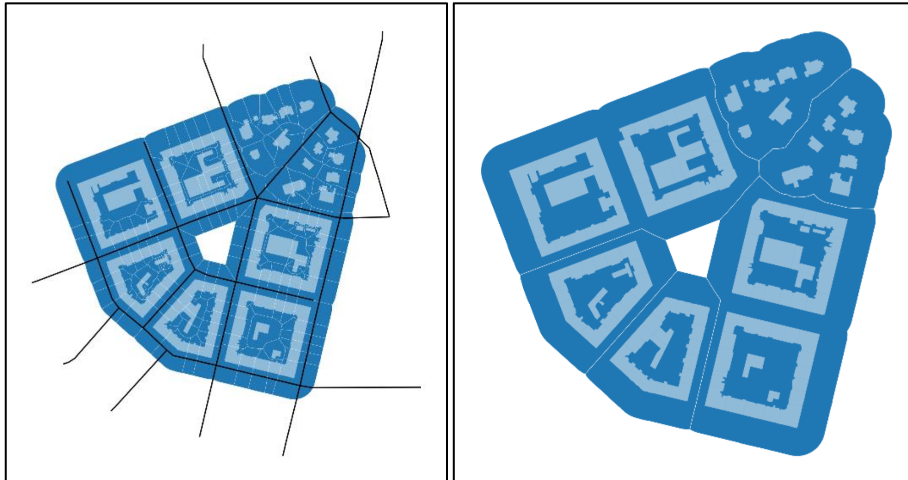

Figure 24 : Réseau routier incomplet : based blocks (Fleischmann et al., 2018)

Ainsi sur la Figure 24, nous voyons que le réseau routier ne rejoint pas la limite de la zone et au lieu d'avoir des blocs séparés par la route qui les traverse, nous avons une fusion de ces blocs car la route n'est pas prise en compte[17]. Une solution proposée dans la documentation de momepy est donc de prolonger ces « impasses » (Fleischmann *et al*., 2018). Pour cela, il faut utiliser la fonction *momepy.extend_lines.* Cette fonction prolonge les segments de lignes d'un GeoDataFrame pour les connecter à d'autres segments, ou à une cible spécifique définie dans la fonction. Si une cible est spécifiée, les lignes sont prolongées jusqu'à cette cible (exemple : la zone d'étude) et si aucune cible n'est spécifiée, les lignes sont prolongées pour se rejoindre elles-mêmes. Une barrière peut être également spécifiée pour empêcher que les lignes prolongées ne traversent certains objets, comme des bâtiments. Si une intersection avec la barrière est détectée, la ligne initialement prolongée n'est pas incluse dans le résultat final (Fleischmann *et al*., 2018).

Sur la Figure 25, les routes incomplètes ont été prolongées, ce qui permet de délimiter de nouveaux blocs. Ainsi, dans certains cas spécifiques où des rues forment des impasses mais peuvent être prolongées sans toucher des bâtiments, cette méthode peut être efficace et permettre d'éviter d'enlever des éléments structurants de la ville qui auraient pu servir de barrière dans momepy. Toutefois, nous n'avons pas testé cette méthode avec des données venant de l'Overpass API[18].

[17] Cet exemple vient de la documentation de momepy sur la tessellation basée sur les blocs (Fleischmann *et al*., 2018). Un exemple sur la tessellation fermée se trouve dans les résultats, sur la Figure 38, page 65. Tous les canaux ne sont pas reliés au reste des barrières car certains finissent en impasse.

[18] Une tentative de la méthode se trouve dans le code, en commentaire.

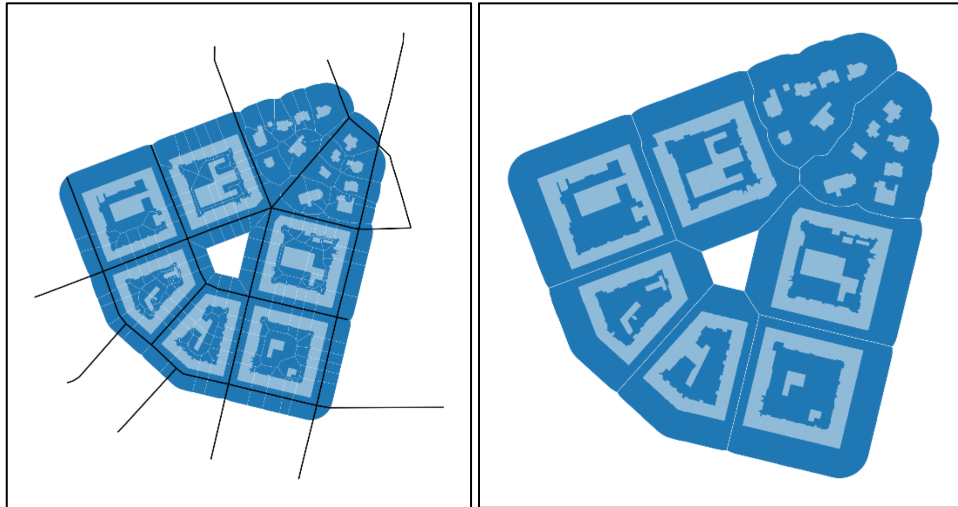

Figure 25 : Réseau routier complet : based blocks (Fleischmann et al., 2018)

Pour résumer, la tessellation avec barrières sera toujours basée sur le réseau routier. Ensuite, diverses barrières sont proposées à l'utilisateur pour qu'il puisse choisir en fonction du contexte urbanistique. La tessellation fermée peut-être utile dans des endroits spécifiques avec une structure urbaine particulière, comme dans des villes structurées par des canaux, ou des zones coupées par des voies ferrées, par exemple. L'avantage de la tessellation fermée est qu'elle est plus précise que la tessellation morphologique et est plus proche de la réalité parcellaire.

3.4.3 La tessellation basée sur les blocs

Initialement, ce type de tessellation va générer des blocs basés sur les bâtiments, la tessellation morphologique autour de ces bâtiments puis sur le réseau routier. La fonction utilisée est *momepy.Blocks()*. Cette fonction va fusionner les cellules de tessellation en fonction de polygones définis par le réseau routier pour former des blocs.

Cette méthode produit une tessellation qui s'apparente à la tessellation morphologique, tout en y ajoutant une dimension supplémentaire. En plus de diviser l'espace autour des bâtiments individuels, elle crée également des blocs de bâtiments en fonction du réseau de rues environnant. Ainsi, les rues en impasse prennent une importance particulière. En effet, les segments de rue qui ne rejoignent pas d'autres segments ou qui ne s'étendent pas jusqu'aux limites de la zone d'étude ne servent pas de délimitation de bloc. Cette particularité donne un résultat semblable à celui observé sur la Figure 24.

Une variante intéressante serait d'utiliser non pas la tessellation morphologique comme base de la tessellation basée sur les blocs, mais d'utiliser la tessellation avec barrières. En effet, se baser

sur la tessellation morphologique peut donner des résultats imprécis, car mêmes si les blocs se forment à partir du réseau routier, ils se basent sur la tessellation morphologique. Utiliser la tessellation avec barrières permettrait d'avoir des résultats plus robustes et plus précis.

Pour cela, le procédé est semblable à celui de la tessellation avec barrières. La tessellation qui structure la tessellation basée sur les blocs sera donc la tessellation avec barrières. Les rues sont toujours imposées dans le code comme barrière principale, et l'utilisateur peut ensuite choisir les autres barrières à sa convenance pour générer les enclosures et la tessellation en elle-même[19]. L'utilisateur peut également choisir s'il souhaite utiliser la tessellation morphologique ou la tessellation avec barrières comme base de la tessellation basée sur les blocs.

```
enclosures_block = momepy.enclosures(streets, limit=gpd.GeoSeries([limit]), additional_barriers = addi-
tional_barriers)
enclosed_tess_block = momepy.Tessellation(buildings, unique_id='uID', enclosures=enclosures_block).tes-
sellation
```

Extrait de code 7 : création des barrières pour la tessellation-based blocks

Cette tessellation revisitée peut se montrer très efficace pour les zones comprenant des bâtiments enregistrés dans OpenStreetMap. En effet, la fonction *momepy.Blocks()* se base non seulement sur le réseau routier mais également sur les bâtiments. Ainsi, dans une zone ne comprenant pas de bâtiment enregistré, les blocs ne seront pas générés. Nous y reviendrons dans la partie dédiée aux résultats.

3.5 Attribution de codes postaux aux cellules de tessellation

Les trois types de tessellations ont été décrits ci-dessus. Maintenant que nous avons obtenu des cellules de tessellation, il faut leur attribuer des codes postaux. L'objectif est donc d'attribuer des codes postaux pour chaque cellule à partir du semi de points de GeoPostcodes (chacun ayant un code postal attribué) et de fusionner les cellules qui ont les mêmes codes postaux.

Pour cela, nous devons créer une requête SQL. Chaque cellule se verra donc attribuer le code postal du ou des points qui s'y trouvent. Dans le cas où il n'y a qu'un seul point, ou plusieurs points avec les mêmes codes postaux, la cellule prend la valeur du code postal associé à ces

[19] Bien que nous ayons proposé des barrières supplémentaires, tous les résultats de cette tessellation « *based blocks enclosed* » se sont uniquement basés sur le réseau routier.

points. Dans le cas où la cellule contient des points avec différents codes postaux, la cellule prend la valeur du code postal le plus représenté.

Dans le cas où une cellule a une égalité entre différents codes postaux, nous attribuons à cette cellule le code postal d'un de ces codes postaux qui est le plus proche de notre cellule.

Toutefois, il peut y avoir des cellules qui ne possèdent pas de bâtiments ou qui n'ont pas de point avec des codes postaux attribués. Il faut donc gérer ces cas spécifiques. Deux cas ont été repris :

- Le cas où une cellule de tessellation possède des bâtiments en son sein mais pas de point avec un code postal attribué par GeoPostcodes.
- Le cas où une cellule de tessellation n'a pas de bâtiment en son sein ni de point avec un code postal attribué par GeoPostcodes.

Dans le premier cas, il a été choisi d'attribuer à la cellule de tessellation le code postal général dans lequel elle se trouve, suivi du terme *undefined-points*.

Dans le deuxième cas, il a été choisi d'attribuer à la cellule de tessellation le code postal général dans lequel elle se trouve, suivi uniquement du termine *undefined*.

De manière plus générale, il faut également que la requête SQL fonctionne quel que soit le lieu choisi par l'utilisateur. Nous avons donc créé une fonction pour normaliser les noms de pays saisis par l'utilisateur ainsi que pour sélectionner dynamiquement les tables appropriées pour les requêtes SQL.

```
def normaliser_pays(pays):
    pays = unidecode(pays)
    pays = pays.lower()
    pays = pays.replace('-', ' ')
    return pays
```

Extrait de code 8 : normalisation des pays

La fonction *normaliser_pays()* est conçue pour standardiser les entrées utilisateur concernant les noms des pays. Pour cela, elle utilise la fonction *unidecode()* pour supprimer les accents et les caractères spéciaux (Solc, 2024). Elle va également changer tous les caractères en minuscules et

remplacer les tirets par des espaces. Par exemple, « Pays-Bas », « PAYS Bas » ou « pays-bas » deviennent tous « pays bas ».

Ensuite, la fonction *combiner_polygones()* utilise la normalisation des pays pour sélectionner les tables correspondantes pour les polygones de codes postaux généraux et pour le semi de points. Deux dictionnaires ont été définis :

1. *dict_points* : associe chaque pays normalisé à la table de points correspondante.
2. *dict_poly* : associe chaque pays normalisé à la table de polygone correspondante.

Une table appropriée est ensuite utilisée en fonction du pays qui sera identifié dans l'input utilisateur. Par exemple, si l'utilisateur choisit comme lieu : « Quebec, Canada », l'input sera normalisé dans son entièreté et la fonction cherchera dans l'input le pays pour choisir les points et les polygones correspondants. Les pays valables pour le code sont : le Brésil, le Canada, la Grande-Bretagne, Israël, les États-Unis et les Pays-Bas.

Cela nous permet d'insérer dans la requête SQL des paramètres qui prendront les valeurs des tables de points et de polygones du pays choisi par l'utilisateur.

Sur l'Extrait de code 9, nous constatons que le dictionnaire associé aux tables des semis de points possède des noms différents pour le Brésil, le Canada, les Pays-Bas et les États-Unis. En réalité, les semis de points de GeoPostcodes peuvent être de 3 types : postcode[20], street[21] et adresse[22].

[20] Moyenne de tous les éléments associés à un code postal.
[21] Centroïde de la rue, associé à un code postal.
[22] Bâtiment associé à un code postal.

```
def combiner_polygones(nom_table, nom_resultat):
    # Dictionnaire associant chaque pays à la table de points correspondante
    dict_points = {
        'bresil': 'jonas."points_br_adresse"',
        'canada': 'jonas."points_ca_adresse"',
        'grande bretagne': 'jonas."points_gb"',
        'israel': 'jonas."points_il"',
        'etats unis': 'jonas."points_us"',
        'pays bas': 'jonas."points_nl_adresse"'
    }
    # Dictionnaire associant chaque pays à la table de polygones correspon-
dante
    dict_poly = {
        'bresil': 'jonas."shape_zips_br"',
        'canada': 'jonas."shape_zips_ca"',
        'grande bretagne': 'shape_zips_gb"',
        'israel': 'jonas."shape_zips_il"',
        'etats unis': 'jonas."shape_zips_us"',
        'pays bas': 'jonas."shape_zips_nl"'
    }

    #Normaliser nom du pays
    pays_normalise = normaliser_pays(r_lieu)

    # Rechercher le pays dans l'input utilisateur pour les points
    table_points = 'jonas."points_br_adresses"'  #valeur par défaut
    for pays_key in dict_points.keys():
        if pays_key in pays_normalise:
            table_points = dict_points[pays_key]
            break

    # Rechercher le pays dans l'input utilisateur pour les polygones
    shape_zip = 'jonas."shape_zips_br"'  #valeur par défaut
    for pays_key in dict_points.keys():
        if pays_key in pays_normalise:
            shape_zip = dict_poly[pays_key]
            break
```

Extrait de code 9 : dictionnaires des points et des polygones

Puisque nous utilisons la tessellation qui se base sur les codes postaux des bâtiments, nous n'avons conservé que les données sur les adresses, que nous avons mises dans des nouvelles tables[23]. De plus, comme nous l'avons mentionné dans l'introduction de ce mémoire, nous avons effectué nos tests sur le Brésil, le Canada, les Pays-Bas et les États-Unis, ce qui explique que le nom des autres tables reste inchangé.

[23] Nous avons créé des index spatiaux sur ces tables pour accélérer les requêtes.

Explication de la logique de la requête SQL

Nous rentrons désormais dans la requête SQL, intégrée dans le code python. Comme celle-ci est relativement longue, nous avons choisi de ne pas la placer directement dans le texte, mais plutôt d'expliquer la logique de celle-ci. Elle reste disponible dans le code en annexe.

D'abord, nous créons une table qui va compter le nombre de points dans chaque géométrie issue de la tessellation, par code postal.

Ensuite, nous identifions les géométries où il y a une égalité dans le nombre maximum de points pour différents codes postaux. Cela se fait en créant d'abord une table qui trouve le nombre maximum de points pour chaque géométrie, puis une table qui identifie les géométries où plus d'un code postal a ce nombre maximum de points.

Pour les géométries avec égalité, nous analysons les codes postaux des géométries voisines. Une table est créée, et celle-ci compte la fréquence des codes postaux dans les géométries voisines pour chaque géométrie avec égalité.

Nous passons ensuite à l'attribution des codes postaux finaux. Pour les géométries sans égalité, le code postal avec le plus de points est choisi. Pour les géométries avec égalité, le code postal le plus fréquent parmi les voisins est sélectionné. Dans le cas des géométries sans points, nous distinguons deux situations : si elles ont des bâtiments, le code postal du polygone[24] le plus proche est attribué suivi du terme *undefined-points*. Si elles n'ont pas de bâtiments, le code postal du polygone le plus proche est attribué suivi du terme *undefined*.

Dans la dernière partie, les cellules avec le même code postal final sont fusionnées et groupées. Cela crée la table finale avec les géométries fusionnées pour chaque code postal.

Le but de cette requête était de suivre la volonté de GeoPostcodes : avoir une couverture la plus complète possible et éviter au maximum des incohérences. Lorsque des cellules n'ont pas de bâtiment, ni de point, des valeurs leurs sont tout de même attribué (*undefined*). Lorsque des cellules ont le même nombre de points pour différents codes postaux, nous choisissons le code postal du voisin le plus proche. Cela permet généralement d'homogénéiser nos résultats. Nous en reparlerons dans la partie dédiée aux résultats (point 4.4).

[24] Ici, les polygones les plus proches correspondent aux polygones « globaux », reprenant les premiers caractères des codes postaux. Il s'agit d'une des tables fournies par GeoPostcodes.

3.6 Programme principal

Pour résumer comment fonctionne le programme principal, celui-ci demande d'abord à l'utilisateur le lieu où il souhaite effectuer la tessellation. L'utilisateur doit ensuite entrer un lieu qui peut être lu par l'Overpass API. Les trois types de tessellations lui sont proposés et il peut choisir celle(s) qu'il souhaite.

```
print(f"A quel lieu voulez-vous appliquer la ou les différente(s) tessellation(s)
?")
print(f"Les pays valides sont les suivants : Brésil, Canada, États-Unis, Grande-
Bretagne, Israël, Pays-Bas.")
r_lieu = input("Entrez le lieu désiré (ville, pays):")
normaliser_pays(r_lieu)
```

Extrait de code 10 : Choisir le lieu pour effectuer la ou les tessellation(s)

Dans l'Extrait de code 10, le lieu est ensuite normalisé pour être utilisable dans la requête SQL, comme nous l'avons vu auparavant.

```
    print(f"Le programme propose 3 types de tessellations :  la
tessellation morphologique, l'enclosed tessellation ou la tessellation
block-based")
    reponse = input("Voulez-vous effectuer la tessellation morphologique ?
(o/n)")
    if reponse == "o":
        print(f"Vous avez choisi la tessellation morphologique")
        try:
            tessellation_results = tessellation()
            tes1 = "true"

            print(f"Sous quel nom voulez-vous sauvegarder le GeoDataFrame
de la tessellation morphologique ?")
            nom_t = input("Entrez le nom désiré :")
        except Exception as e:
            print(f"ERREUR pour l'exécution de tessellation(): {str(e)}")
    else:
        print(f"Vous n'avez pas choisi la tessellation morphologique")
```

Extrait de code 11 : Exemple code tessellation dans le programme principal

L'Extrait de code 11 nous montre comment est appliqué la tessellation dans le programme principal. Si l'utilisateur choisit d'appliquer la tessellation, la fonction *tessellation()*, définie dans notre code, s'exécute. Un paramètre est ensuite défini sur *true*. Celui-ci nous servira pour

déterminer quelle tessellation a été effectuée (morphologique, fermée ou blocs), et ainsi savoir quelle fonction de sauvegarde doit être appliquée (voir Extrait de code 12 et Extrait de code 13).

```
def sauvegarde_postgis_tessellation(tessellation):
    tessellation.to_postgis(nom_t, engine, schema='jonas', if_exists='fail')

def sauvegarde_postgis_enclosed_tessellation(enclosed_tessellation):
    enclosed_tessellation.to_postgis(nom_et, engine, schema='jonas', if_exists='fail')

def sauvegarde_postgis_based_block(tessellation_block):
    tessellation_block.to_postgis(nom_tb, engine, schema='jonas', if_exists='fail')
```

Extrait de code 12 : Définitions des sauvegardes dans la BD en fonction de la tessellation

Il est ensuite proposé à l'utilisateur de sauvegarder le GeoDataFrame dans la base de données, sous le nom désiré. Les opérations sont semblables pour la tessellation fermée et la tessellation basée sur les blocs.

```
    if tes1 == "true":
        try:
            print(f"Tentative de sauvegarde des Geodataframes pour la 
tessellation...")
            sauvegarde_postgis_tessellation(tessellation_results)
            print(f"Sauvegarde dans PostGIS réussie pour la tessellation !")
        except Exception as e:
            print(f"Erreur lors de l'exécution de sauvegarde_postgis() pour la 
tessellation: {str(e)}")
```

Extrait de code 13 : Sauvegarde de la table dans la base de données

L'Extrait de code 13 nous montre la sauvegarde effectuée si l'utilisateur a accepté de sauvegarder le GeoDataFrame.

```
    #COMBINAISON ZIP

    combiner_zip = input("Voulez-vous combiner les polygones résultants de la 
tesselation en fonction de leurs codes postaux ? (o/n)")

    if combiner_zip == "o":
        if tes1 == "true":
            nom_resultat_t = f"{nom_t}_zip_combined"
            combiner_polygones(nom_t, nom_resultat_t)
```

Extrait de code 14 : Attribution et combinaison des codes postaux des cellules de tessellation

Enfin, pour attribuer des codes postaux à nos cellules de tessellation, ainsi que pour les fusionner, nous demandons d’abord à l’utilisateur s’il souhaite combiner les polygones. Ensuite, nous utilisons la fonction *combiner_polygones()*, décrite auparavant. À noter que la requête SQL effectuée dans la fonction ne peut fonctionner que si la table avec les cellules de tessellation a été sauvegardée dans la base de données. C’est pour cela que nous vérifions bien que l’argument tes1 est *true*. Celui-ci est *true*, uniquement si l’utilisateur a choisi de sauvegarder la table de la tessellation résultante dans la base de données.

4 RÉSULTATS

Dans la section dédiée aux résultats, nous présenterons un échantillon des tessellations obtenues. GeoPostcodes vise à s'appuyer sur le code développé dans ce mémoire pour générer des tessellations à grande échelle et les associer à leurs codes postaux. Nous mettrons donc en avant des exemples réussis de tessellations, mais aussi des cas problématiques qui pourraient surgir lors de l'exécution du code sur leurs propres systèmes. En détaillant les divers obstacles rencontrés selon les contextes urbains, notre objectif est de mettre en lumière les erreurs à éviter dans certaines situations.

Pour cela, nous allons appliquer les différentes tessellations sur plusieurs villes de divers pays, avec des contextes urbains variés. L'objectif n'est pas de déterminer la meilleure tessellation par pays, mais plutôt par contexte. Il serait possible d'en tirer une conclusion intuitive, mais ce n'est pas le but de ce mémoire. Nous observerons comment les tessellations réagissent face à des zones sans bâtiments sur OpenStreetMap, des zones denses, des zones structurées par certains types de barrières, etc.

4.1 Résultats avec les différentes tessellations

Dans cette partie, nous allons montrer quelques cas de tessellations. Premièrement, nous comparons les trois tessellations pour un même lieu. Ensuite nous montrerons quelques exemples typiques de chaque tessellation, en mettant en avant les problèmes potentiels qui peuvent survenir. Le but est de montrer à GeoPostcodes comment les différentes tessellations se comportent dans des cas « non idéaux ».

4.1.1 Différences entre les 3 types de tessellation

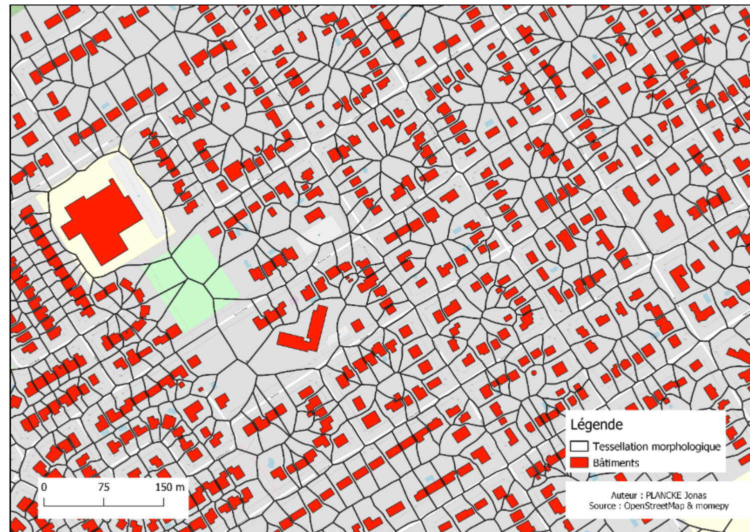

Figure 26 : tessellation morphologique, Midland, Canada (momepy(2019) &OpenStreetMap (2024)).

La Figure 26 montre l'application de la tessellation morphologique dans la ville de Midland, au Canada. Les bâtiments sont affichés en rouge et mis en évidence grâce à QuickOSM. La tessellation de Voronoï est appliquée autour de chaque bâtiments.

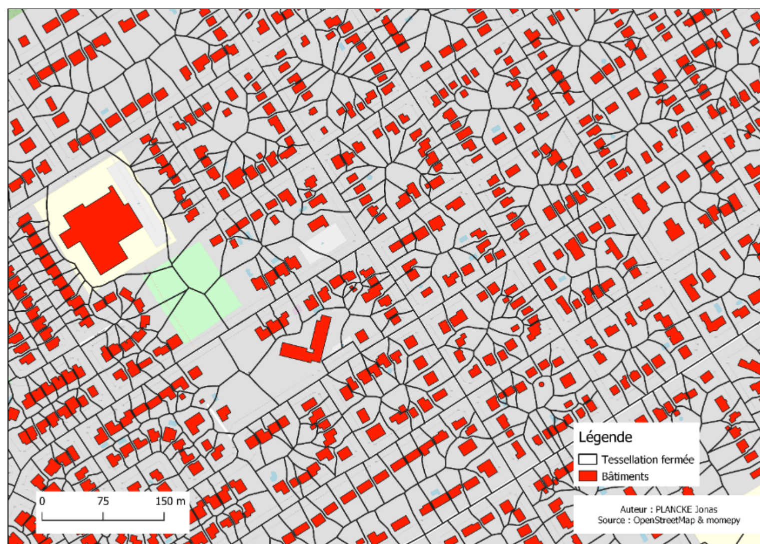

Figure 27 : tessellation fermée, Midland, Canada (momepy(2019) &OpenStreetMap (2024)).

La Figure 27 nous montre la tessellation fermée, avec barrières. En comparaison à la figure précédente, nous pouvons voir que la tessellation suit le réseau routier qui structure la ville et sont ainsi plus proches des véritables parcelles urbaines.

Figure 28 : tessellation basée sur les blocs (basée sur une tessellation fermée), Midland, Canada (momepy(2019) &OpenStreetMap (2024)).

La Figure 28 montre une tessellation basée sur les blocs, sur base d'une tessellation fermée. Les bâtiments sont correctement associés en blocs. Les blocs sont précisément limités par le réseau routier. Pour les rues qui ne forment pas des boucles, nous aurons des blocs plus importants.

4.1.2 Tessellation morphologique

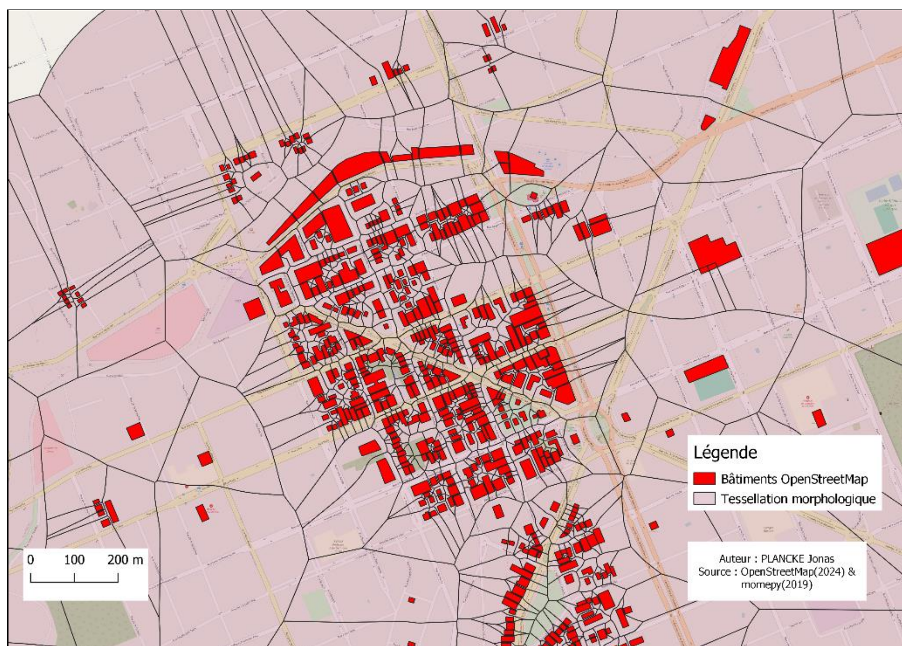

Figure 29 : Tessellation morphologique à Erechim, Brésil (momepy(2019) &OpenStreetMap (2024)).

La Figure 29 est une tessellation morphologique à Erechim, au Brésil. Cette tessellation basique délimite correctement les différents bâtiments au centre de l'image, ceux-ci étant nombreux. A l'extérieur du centre, beaucoup moins de données de bâtiments sont disponibles sur OpenStreetMap. La tessellation commence alors à s'étendre jusqu'à ce qu'elle atteigne sa limite

maximale définie par le code. Elle s'arrête également lorsqu'elle touche une autre cellule de tessellation.

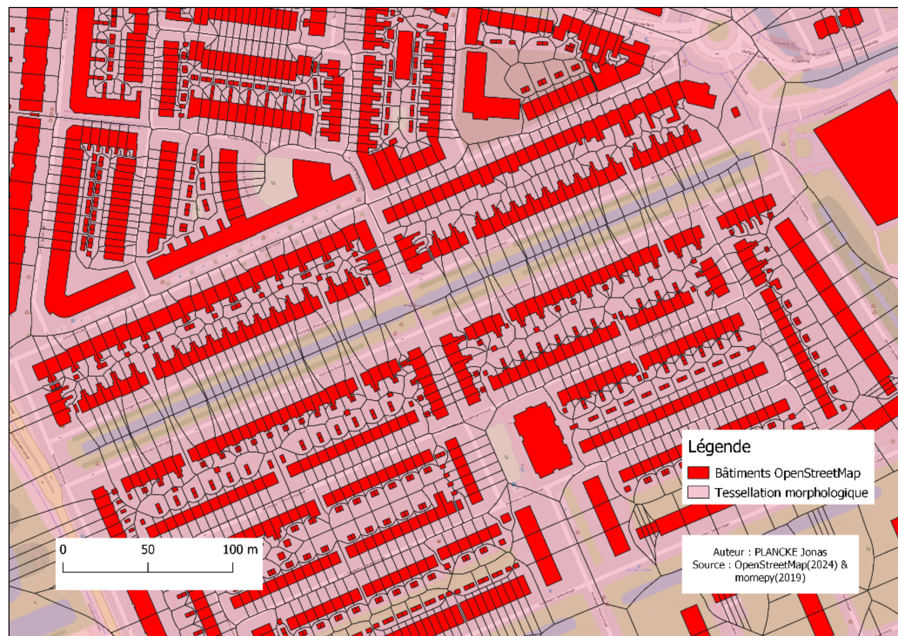

Figure 30 : Tessellation morphologique à Delft, Pays-Bas (momepy(2019) &OpenStreetMap (2024)).

La Figure 30 est une tessellation morphologique dans le quartier de Wippolder, dans la ville de Delft (Pays-Bas). Comme les bâtiments sont denses, la tessellation morphologique donne des résultats pertinents, proche de ce qui pourrait être obtenu avec une tessellation fermée. Cependant, contrairement à cette dernière, elle ne suit pas le réseau routier.

4.1.3 Tessellation avec barrières/fermée

Dans cette partie, nous allons montrer des exemples de résultats de tessellations avec barrières.

La Figure 31 est une tessellation fermée, dans la ville de Delft, aux Pays-Bas. Comme barrière primaire, nous avons inclus tous les types de routes. Cette ville possède également de nombreux canaux et nous avons donc utilisé cette barrière. Comme nous pouvons le constater, les barrières suivent les tracés des routes et des canaux. Il arrive régulièrement que des routes se trouvent à proximité de ces canaux et les suivent de manière parallèle. Nous pouvons voir ce cas au centre de la carte. Dans ces cas, la barrière formée par le canal n'a pas réellement d'utilité car elle ne limite pas la tessellation autour des bâtiments.

La Figure 32 est une tessellation fermée dans le quartier financier de Manhattan, aux États-Unis. Nous constatons que de nombreuses barrières se sont formées, notamment au niveau des

routes. Dans ce cas-ci, cela est lié au fait que nous avons indiqué à notre barrière primaire, qui correspond au réseau de rues, qu'elle devait prendre tous les types de réseau via *network_type = 'all'*.

Les routes, mais également les chemins empruntés par les piétons, sont prises en compte dans cette tessellation. Dans certaines villes, ces routes empruntées par les piétons ne sont généralement pas affichées sur OpenStreetMap, mais dans le cas de villes ou de quartiers plus importants, elles le sont. On retrouve par exemple la même situation à Vancouver, au Canada (Figure 33). Sur cette carte, nous voyons que les routes principales contiennent plusieurs barrières qui correspondent aux routes empruntées par les véhicules ainsi qu'aux chemins empruntés par les piétons. La Figure 34 montre également un exemple similaire au Brésil où les chemins pour les piétons sont également inclus.

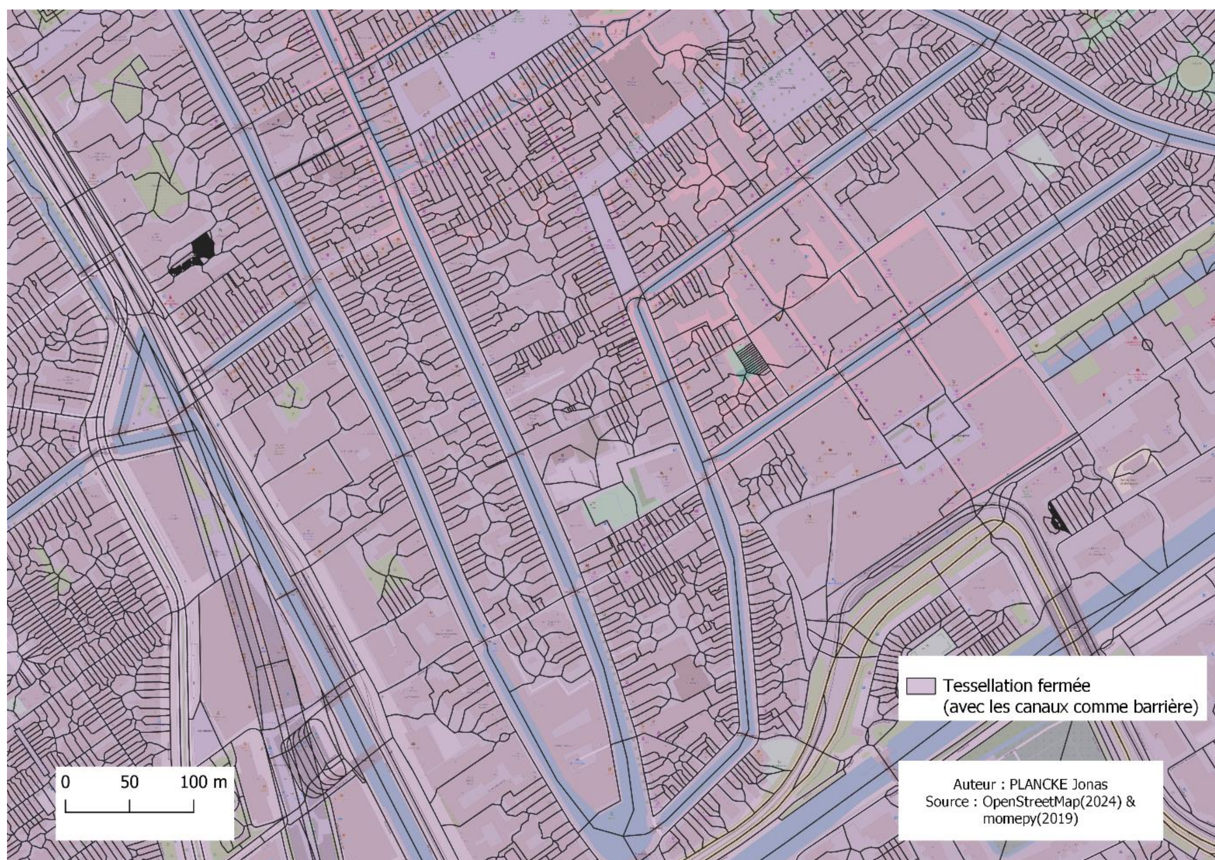

Figure 31 : Tessellation fermée (inclusion des canaux en tant que barrière), Delft, Pays-Bas (OpenStreetMap (2024) & momepy(2019))

Figure 32 : tessellation fermée dans le quartier financier de Manhattan, New York (OpenstreetMap (2024) & momepy (2019))

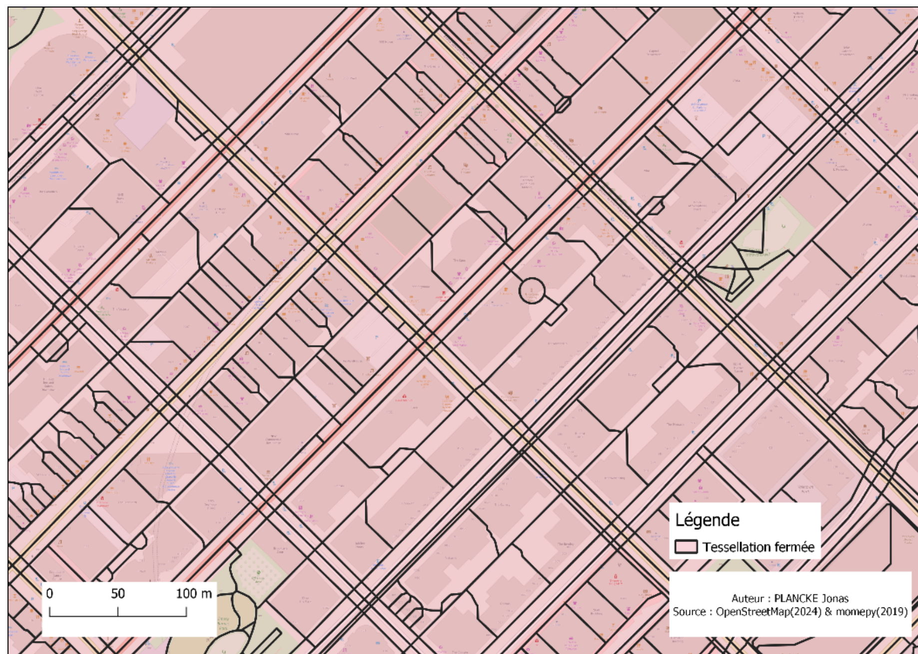

Figure 33 : tessellation fermée dans le centre-ville de Vancouver, Canada (OpenstreetMap (2024) & momepy (2019))

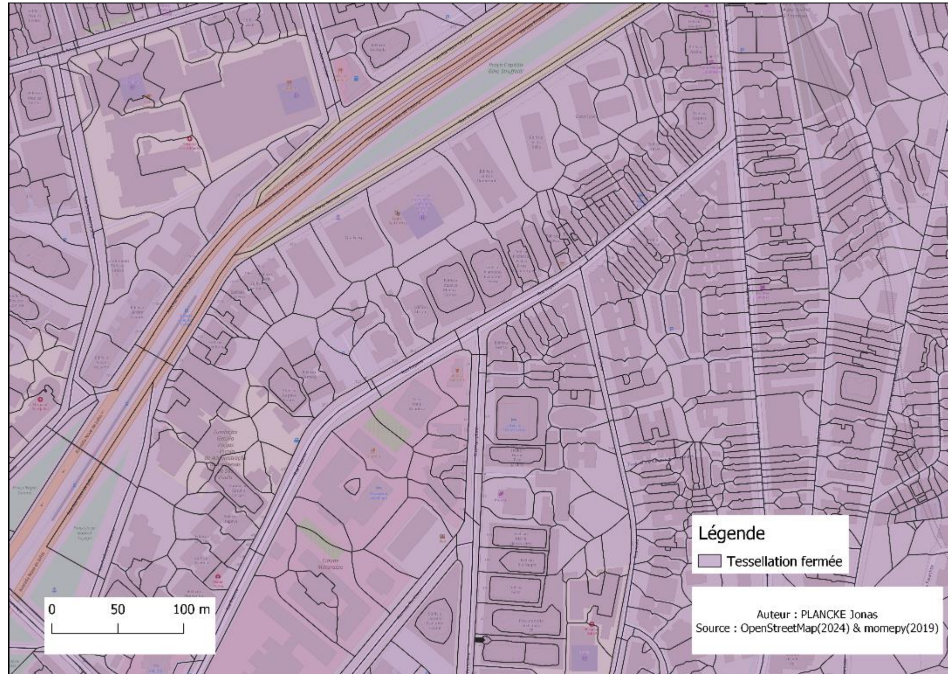

Figure 34 : tessellation fermée dans le quartier de Bela Vista à Sao Paulo, Brésil (OpenstreetMap (2024) & momepy (2019))

De plus, sur la Figure 34, nous voyons sur la partie gauche de la carte une autoroute avec plusieurs voies parallèles. La tessellation fermée va donc créer plusieurs barrières parallèles, et non pas une seule.

Figure 35 : tessellation fermée dans le quartier financier de Manhattan, New York. Les chemins pour piétons et les pistes cyclables n'y sont pas inclus (OpenstreetMap (2024) & momepy (2019))

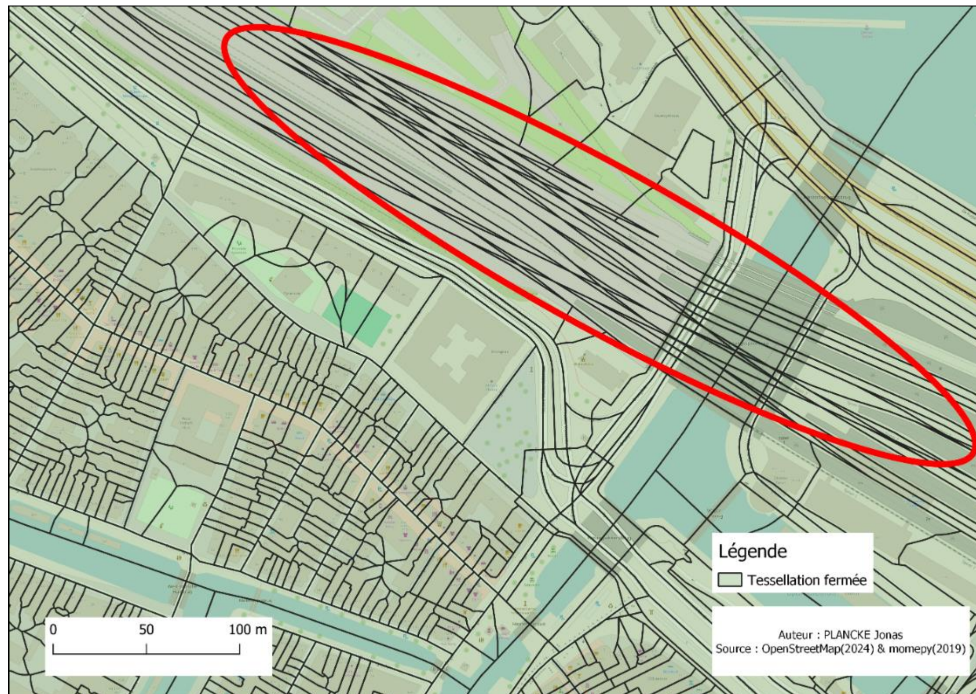

Figure 36 : tessellation fermée dans le centre d'Amsterdam, Pays-Bas. L'ellipse rouge met en évidence les barrières créées par le réseau ferroviaire (OpenStreetMap(2024) & momepy(2019))

Il reste possible de corriger cela, dans les cas des routes et des chemins pour piétons. En mettant le réseau de rue (barrière primaire) sur *network = drive*, ne reprenant ainsi que les routes utilisées par les véhicules, nous corrigeons le problème des routes parallèles. La Figure 35 est une nouvelle version de la Figure 32, dans le quartier financier de Manhattan. Cette fois, nous n'avons pas inclus les chemins pour piétons (ni les pistes cyclables) et la plupart des barrières parallèles qui apparaissaient ne le font désormais plus. En bas à gauche de la carte, ainsi qu'en bas à droite, des barrières parallèles apparaissent tout de même, mais il s'agit d'un réseau de routes parallèles, comme nous pouvons souvent en rencontrer dans les grandes villes.

La Figure 36 montre une tessellation fermée dans le centre d'Amsterdam, aux Pays-Bas. Nous mettons en évidence les barrières créées par le réseau ferroviaire. Le problème est donc similaire à celui des routes parallèles : lorsque plusieurs voies ferrées sont parallèles, la tessellation va donc créer plusieurs barrières.

La Figure 37 nous montre une tessellation fermée dans le village néerlandais de Giethoorn. Ce village possède de nombreux canaux qui structurent le village. Sur cette carte, nous effectuons la tessellation sans prendre en compte les canaux, ni les rivières. Sur la figure suivante, nous incluons les canaux pour comparer les deux résultats.

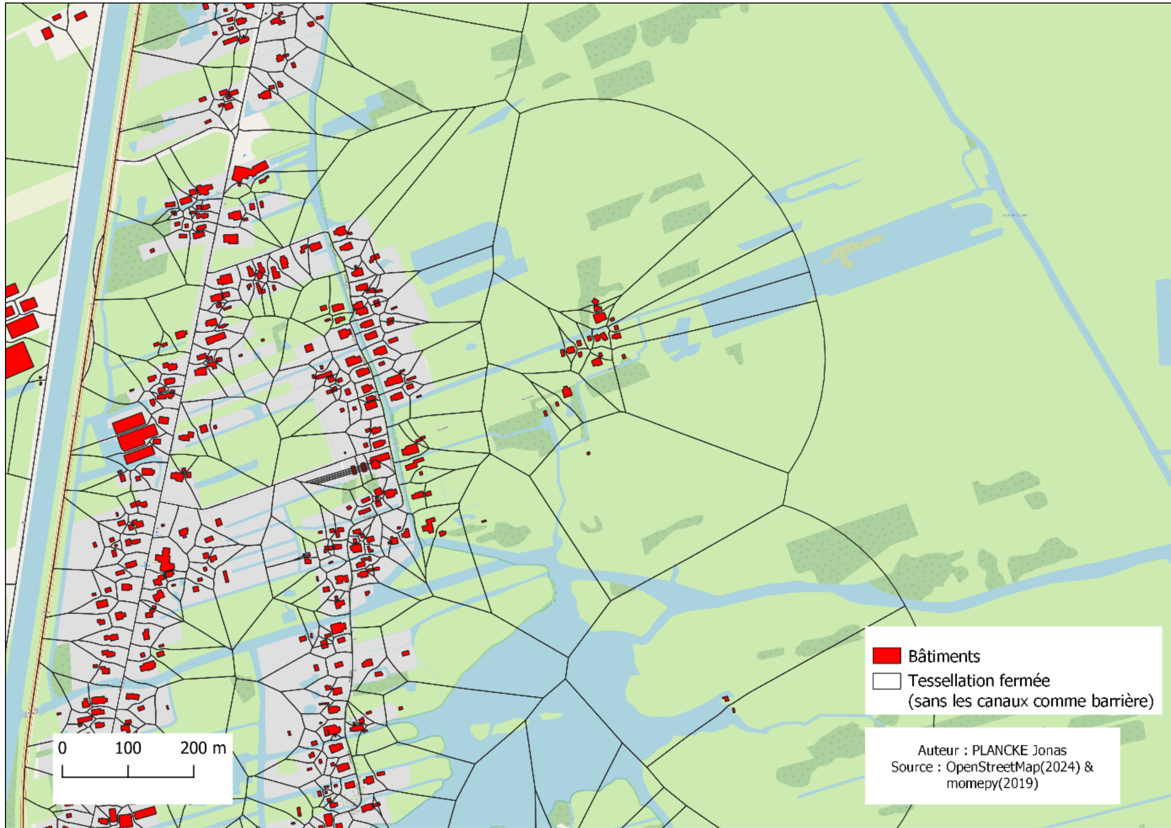

Figure 37 : Tessellation fermée (sans les canaux comme barrière), Giethoorn, Pays-Bas (OpenStreetMap (2024) & momepy(2019))

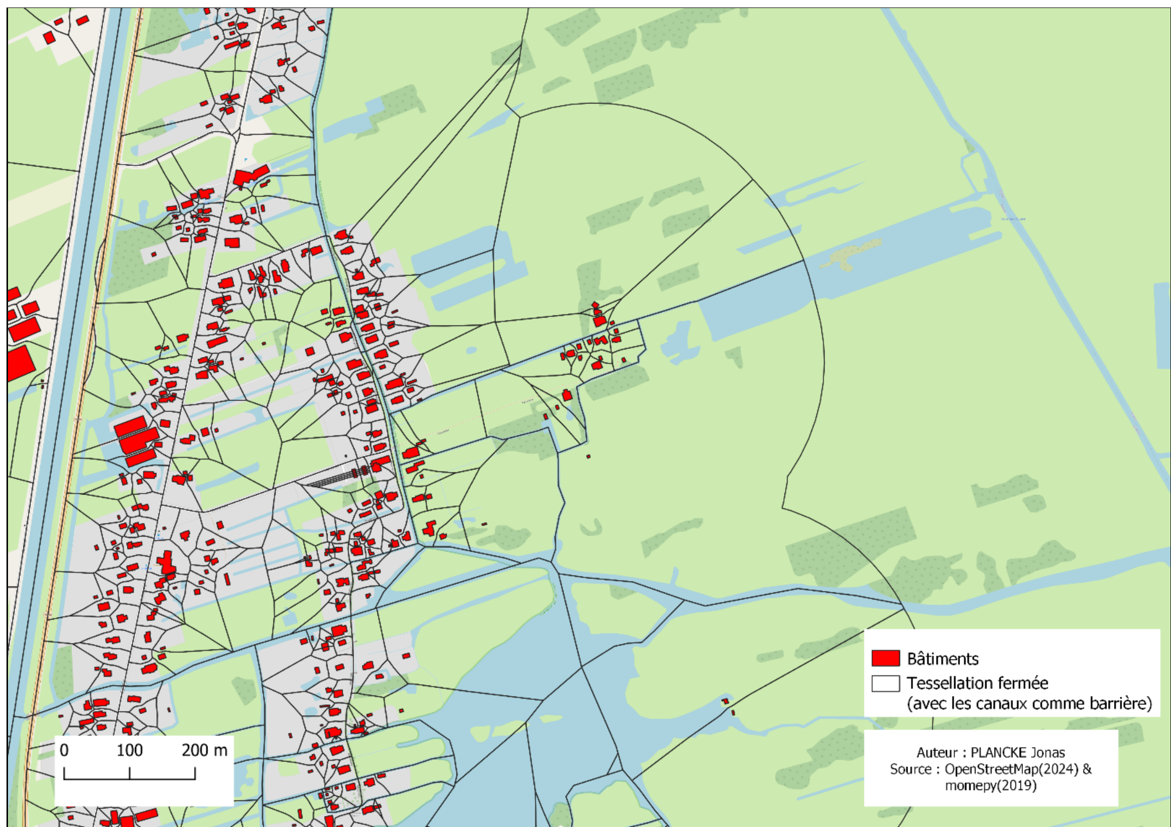

Figure 38 : Tessellation fermée (inclusion des canaux en tant que barrière), Giethoorn, Pays-Bas (OpenStreetMap (2024) & momepy(2019))

Ainsi, la Figure 38 nous montre la tessellation fermée lorsque nous prenons en compte les canaux et les rivières. Au milieu de la carte, ainsi qu'en haut à gauche, nous voyons que la tessellation suit bien les différents canaux. À proximité des habitations, sur la gauche de la carte, nous pouvons constater que la tessellation ne suit pas toujours les canaux. Ces canaux ne sont en réalité pas entièrement reliés au reste du réseau de barrières et ne peuvent pas former de barrières polygonales, car ils terminent en impasse. Pour former une barrière, ces canaux auraient dû être reliés au reste du réseau de barrière aussi bien d'un côté que de l'autre.

La Figure 39 est une tessellation fermée exécutée sur le quartier de Kralingen-Crooswijk à Rotterdam. Les barrières incluses sont toutes celles du code. Sur la Figure 40, la tessellation fermée a été appliquée au même endroit mais cette fois, nous avons inclus une barrière supplémentaire : les parcs urbains. La carte indique leur emplacement, en vert. En incluant cette barrière, nous voyons des zones vides qui apparaissent dans la tessellation, aux emplacements de ces parcs urbains.

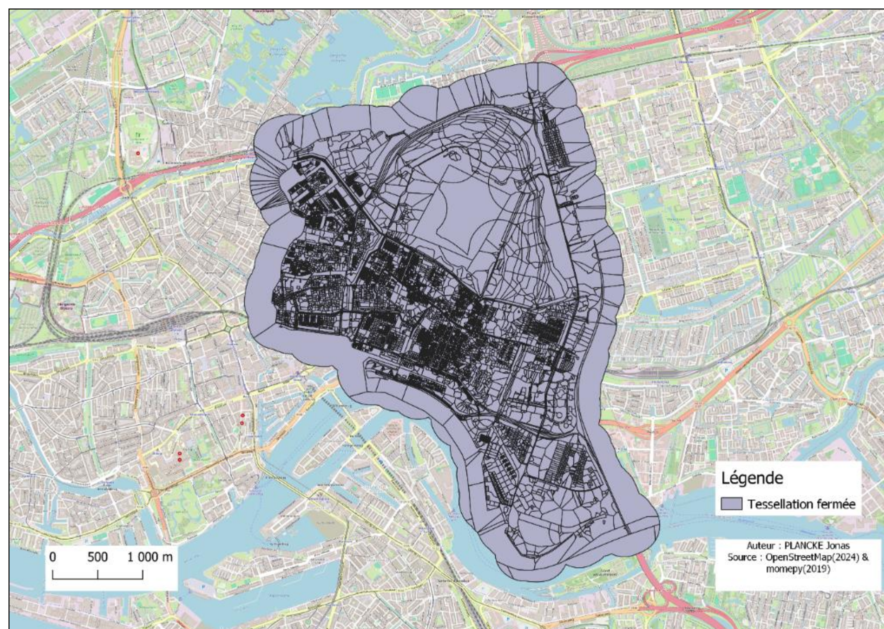

Figure 39 : Tessellation avec barrières, Kralingen-Crooswijk, Rotterdam (OpenStreetMap (2024) & momepy(2019))

Inclure des barrières fermées qui ne possèdent pas de bâtiments en leur sein formera régulièrement des zones vides dans la tessellation. Bien que ces zones ne contiennent pas de bâtiments et n'aient donc parfois pas de codes postaux qui leur sont attribués, GeoPostcodes préfère une couverture maximale du territoire, sans vide dans la tessellation. Nous ne conseillons donc pas d'inclure ces barrières dans le cadre de ce mémoire.

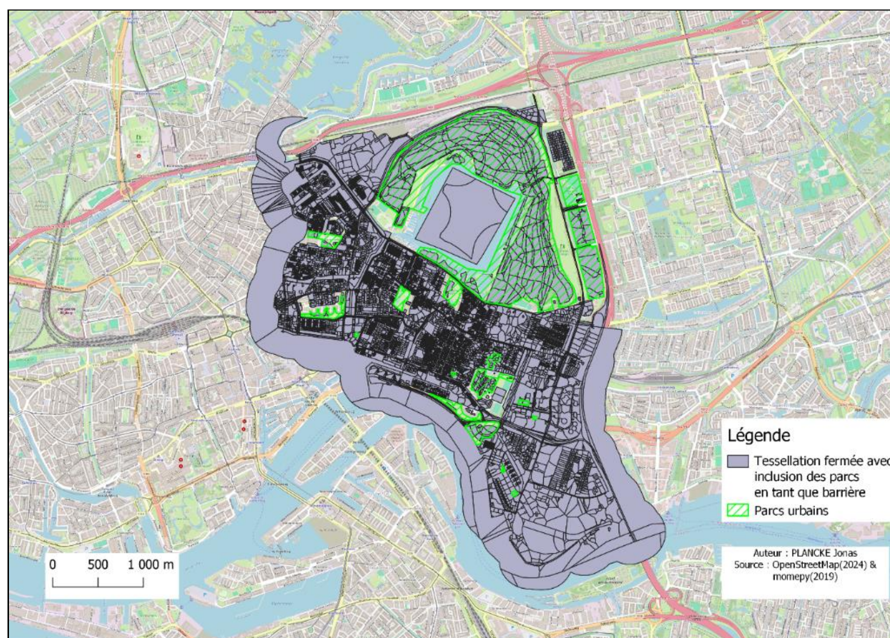

Figure 40 : Tessellation avec barrières (inclusion des parcs urbains en tant que barrière), Kralingen-Crooswijk, Rotterdam (OpenStreetMap (2024) & momepy(2019))

4.1.4 Tessellation basée sur blocs

Dans cette partie, nous montrons quelques exemples de tessellation basée sur les blocs.

La Figure 41 nous montre une tessellation basée sur les blocs dans un cas « idéal ». Nous avons choisi le quartier financier de la ville de Manhattan, pour son réseau de rues denses et ses nombreux bâtiments, également tous disponibles sur OpenStreetMap. La tessellation morphologique se forme initialement autour des bâtiments avant de former des blocs en fonction du réseau de rues. Comme cette tessellation se base sur la tessellation morphologique, elle n'est pas limitée par le réseau de rues comme le serait une tessellation fermée. En réalité, les différentes tessellations morphologiques autour des bâtiments sont fusionnées en fonction du réseau de rue. Comme mentionné dans la méthodologie, il est également possible d'utiliser la tessellation fermée et non la tessellation morphologique comme base.

La Figure 42 nous montre également le quartier financier de Manhattan, sur lequel nous avons appliqué une tessellation basée sur les blocs, sur base d'une tessellation fermée. Cette fois, le territoire n'est pas entièrement recouvert et nous nous retrouvons avec des blocs séparés les uns des autres par une partie du réseau routier.

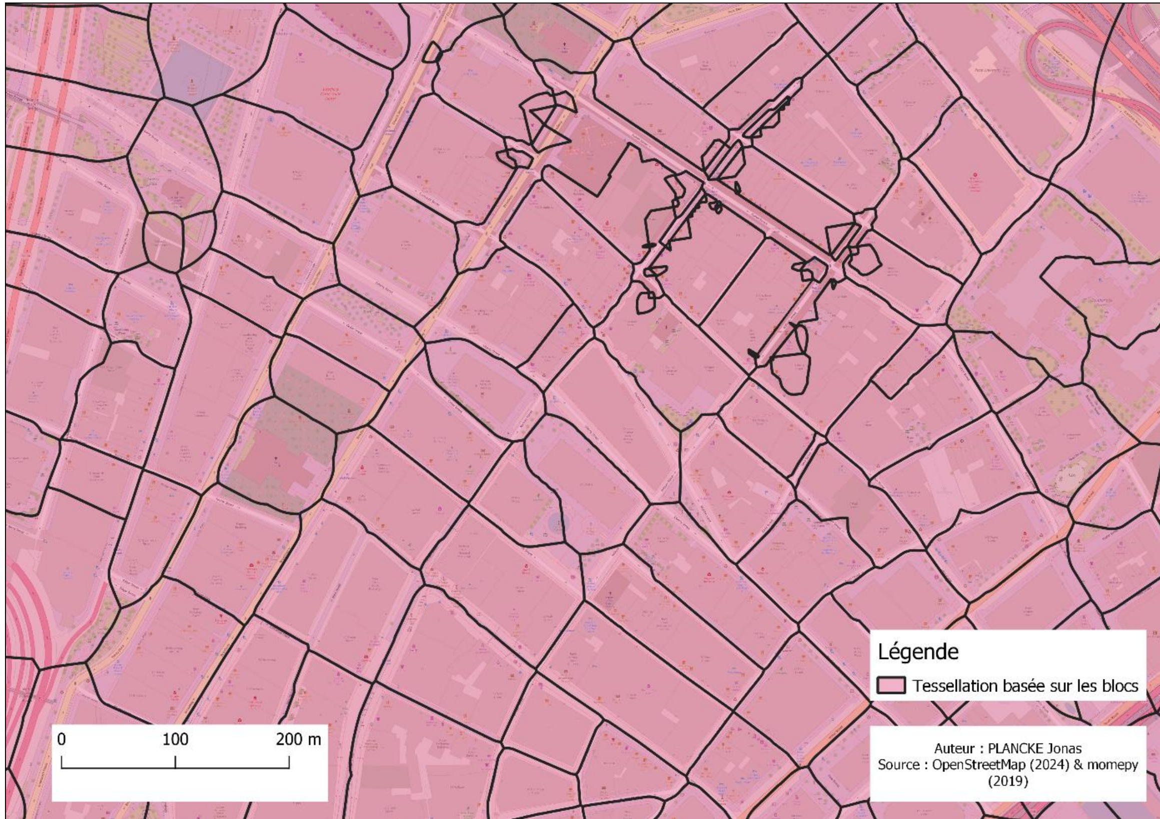

Figure 41 : tessellation basée sur les blocs dans le quartier financier de Manhattan, New York (OpenstreetMap (2024) & momepy (2019))

Figure 42 : tessellation basée sur les blocs (sur base d'une tessellation fermée) dans le quartier financier de Manhattan, New York (OpenStreetMap (2024) & momepy (2019))

Figure 43 : tessellation basée sur les blocs (sur base d'une tessellation fermée sans les chemins pour piétons) dans le quartier financier de Manhattan, New York (OpenstreetMap (2024) & momepy (2019))

Sur la Figure 42, les barrières vont stopper les tessellations, ce qui provoque la séparation des blocs. Par exemple, en plus des routes principales, les trottoirs sont considérés comme une barrière par la barrière primaire *streets* (car nous avons défini notre *network_type* sur *all*). De plus, il y a également parfois des rues à double sens, qui vont créer deux barrières et stopper la tessellation. Nous pouvons corriger ce problème de la même façon que la tessellation fermée en ne prenant pas en compte les chemins pour piétons. La version corrigée se trouve sur la Figure 43.

Nous constatons un bloc plus important que les autres au centre de la Figure 43. Il s'agit d'un piétonnier. Puisque nous n'avons pas inclus les chemins pour piétons, le piétonnier n'est pas utilisé pour la tessellation. Les zones vides sont des zones séparées par le réseau routier, qui ne contiennent pas de bâtiment.

La Figure 44 est une tessellation basée sur les blocs dans la ville d'Erechim, au Brésil. Nous avons choisi cette ville car il y a de nombreux bâtiments non repris par OpenStreetMap. Comme la tessellation basée sur les blocs se base sur une tessellation morphologique autour de bâtiments

entourés par un réseau de rues pour former des blocs, les résultats sont satisfaisants lorsqu'il y a une forte densité de bâtiments.

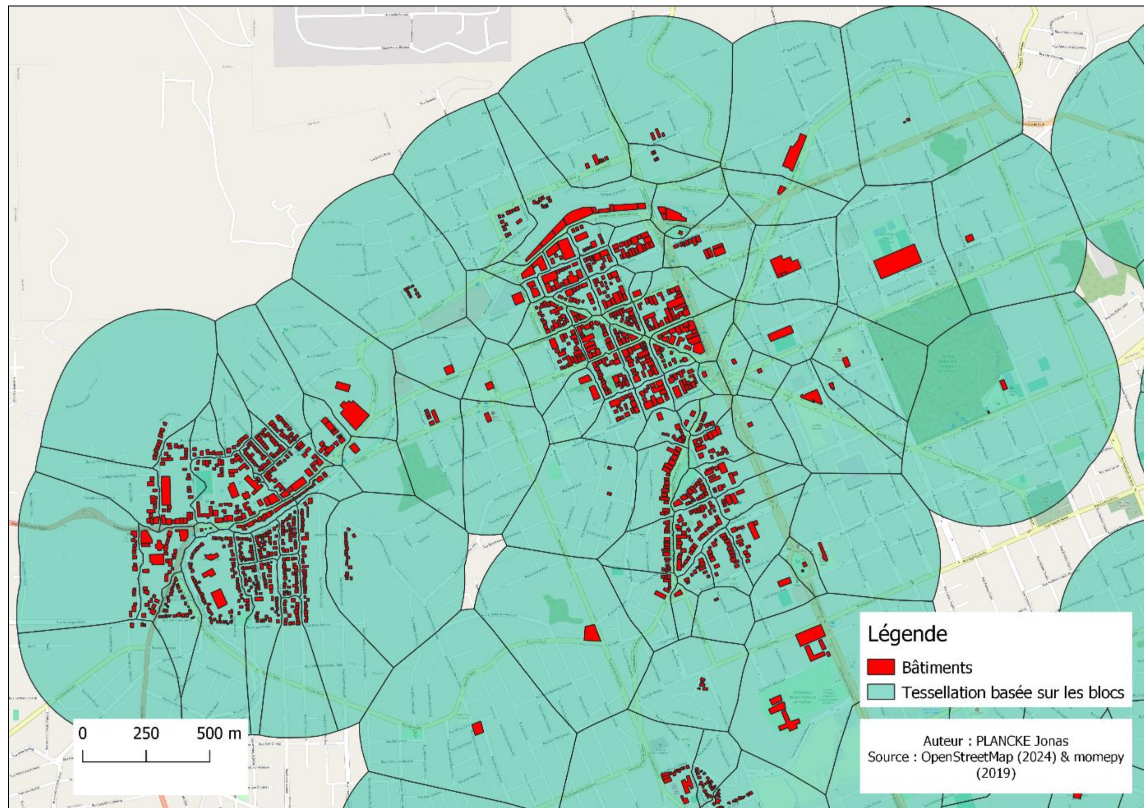

Figure 44 : tessellation basée sur les blocs à Erechim, Brésil (OpenStreetMap (2024) & momepy (2019))

Sur la carte, il y a trois zones de bâtiments importantes, et nous voyons que la tessellation forme des blocs. Pour les bâtiments isolés, la tessellation par bloc agit comme une tessellation morphologique classique et ne s'arrête que lorsqu'elle rencontre une autre cellule de tessellation. Cette tessellation permet tout de même de couvrir tout le territoire (les zones vides correspondent simplement au buffer maximum autour des bâtiments, défini dans le code à 500 mètres), et reste relativement efficace dans les zones denses.

La Figure 45 est également une tessellation basée sur les blocs, mais cette fois, elle se base sur la tessellation fermée. En plus des barrières indiquées dans le code, nous avons également inclus les parcs urbains, pour tester la réaction de la tessellation basée sur les blocs avec l'inclusion de barrières polygonales qui ne contiennent pas de bâtiment en leur sein. Nous voyons donc à droite de la carte, qu'une zone vide s'est créée autour d'un parc urbain.

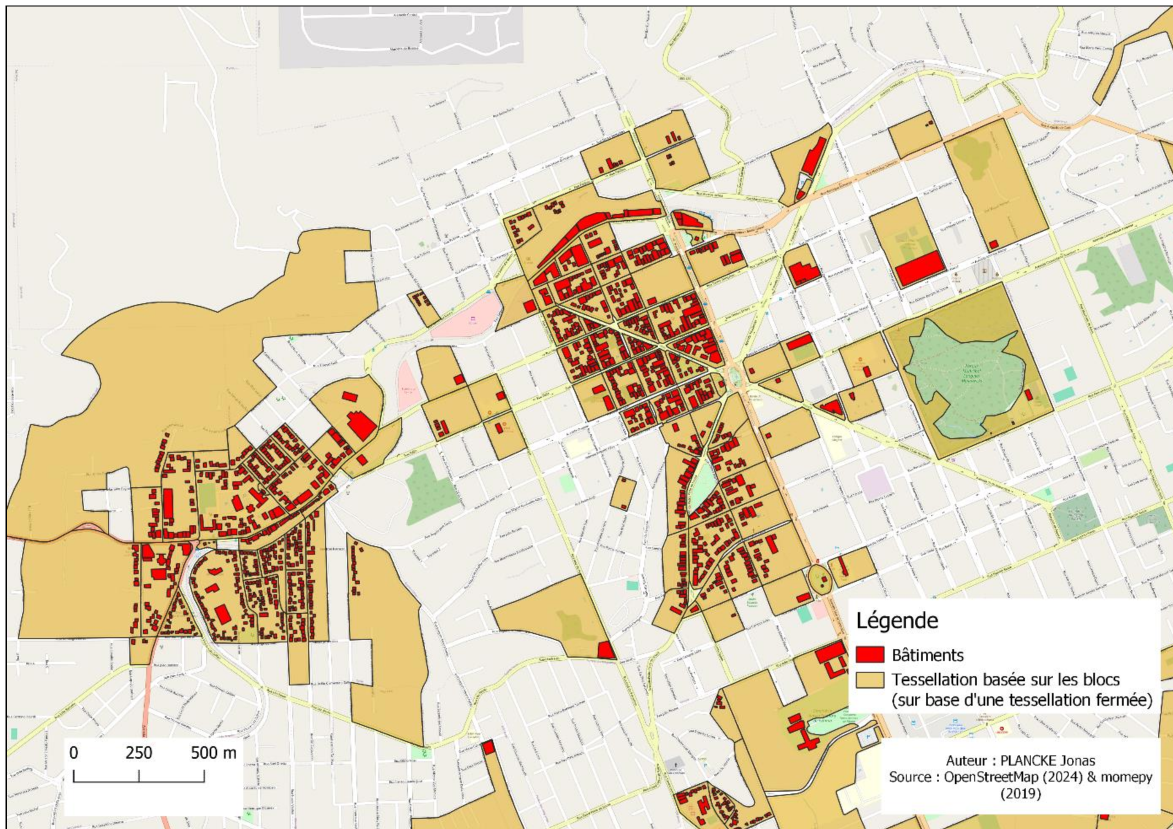

Figure 45 : tessellation basée sur les blocs (sur base d'une tessellation fermée) à Erechim, Brésil (OpenStreetMap (2024) & momepy (2019))

Contrairement à la Figure 44, tout le territoire n'est pas couvert par cette tessellation. En effet, puisque nous nous sommes basés sur la tessellation fermée, les tessellations des bâtiments vont être limitées par les barrières, notamment le réseau de rues. Ainsi, se baser sur la tessellation fermée peut être efficace dans les cas où nous possédons une couche de bâtiments complète mais dans les cas où celle-ci vient à manquer, nous obtenons une couverture incomplète. En comparaison, nous montrons une tessellation fermée à Erechim (Figure 46), qui couvre tout le territoire, malgré le manque de bâtiments. En effet, la tessellation fermée va se baser sur le réseau de rues et tout de même créer une tessellation pour les zones sans bâtiments.

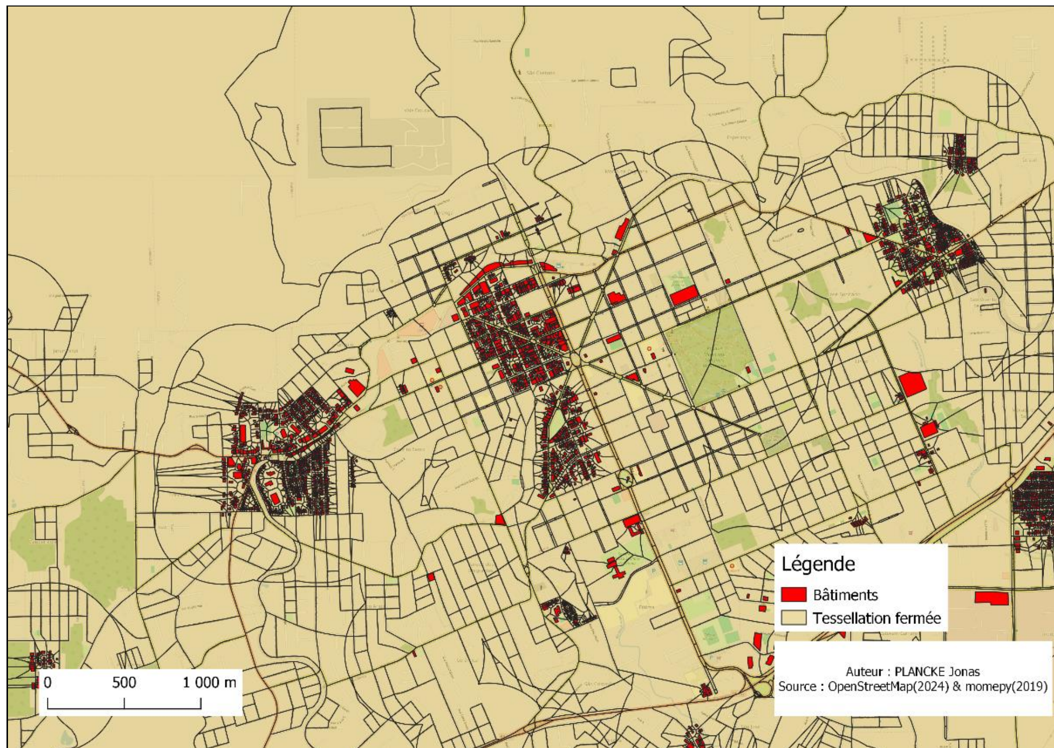

Figure 46 : tessellation fermée à Erechim, Brésil (OpenStreetMap (2024) & momepy (2019))

4.2 Limiter la tessellation

Dans cette partie, nous montrons la différence entre les limites définies par la fonction *momepy.buffered_limit()*, qui délimite la tessellation maximale à l'aide d'un buffer autour des bâtiments ainsi que la limite définie par le niveau administratif de la zone, quand cela est possible à l'aide de l'Overpass API.

En règle général, nous utilisons la fonction *momepy.buffered_limit()*. Cette fonction calcule un buffer autour des bâtiments et retourne un polygone ou un multipolygone qui définit la limite de la tessellation. Nous avons défini cette limite à 500 mètres dans le code. Il est également possible de définir ce buffer sur « adaptative », pour que celui-ci soit calculé en fonction du graphe de Gabriel (Fleischmann *et al.*, 2018).

Comme GeoPostcodes souhaite obtenir une couverture maximale du territoire, nous avons tenté d'utiliser les limites définies par les niveaux administratifs pour servir de limite à la tessellation. Malheureusement, celles-ci n'ont pas donné des résultats probants.

Figure 47 : Limite administrative de niveau 8, Lunenburg, Canada (OpenStreetMap, 2024)

Sur la Figure 47, nous avons indiqué le niveau administratif qui correspond à la limite de la ville de Lunenberg, au Canada. Pour obtenir cette limite, nous avons utilisé QuickOSM, un plugin QGIS. Cela nous permet de vérifier que cette limite existe belle et bien et est téléchargeable par l'Overpass API. Toutefois, lorsque nous utilisons le niveau administratif (via le tag *admin_level*) pour utiliser ce polygone comme limite de la tessellation et que nous choisissons le même que celui de la zone qui nous intéresse, il ne s'agit pas du même polygone qui est téléchargé, mais un polygone comprenant une zone plus large.

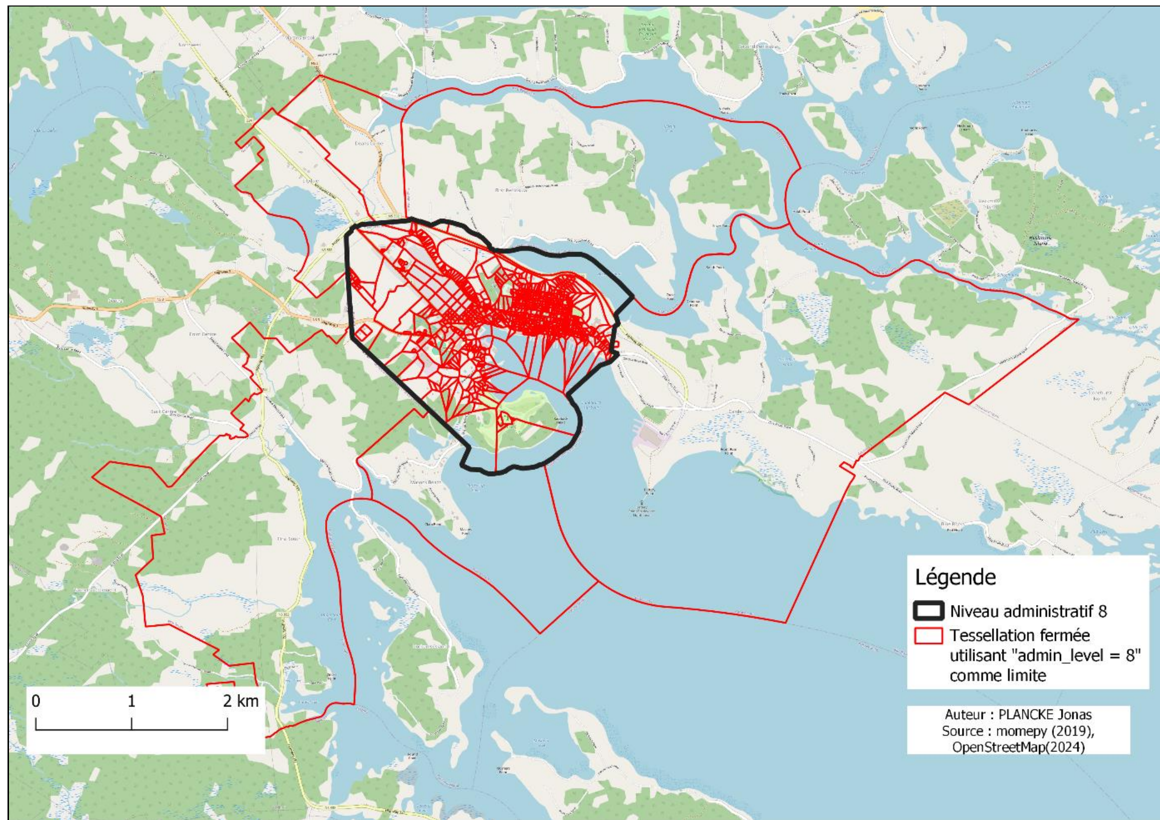

Figure 48 : Tessellation fermée utilisant admin_level = 8 comme limite à Lunenburg, Canada (momepy(2019), OpenStreetMap(2024))

La Figure 48 illustre notre propos. Au lieu d'utiliser la même limite que QuickOSM, qui est également la même que celle sur le site web OpenStreetMap, le multipolygone téléchargé correspond à une zone plus grande. Utiliser un niveau administratif correspondant à 9 ne permet pas de résoudre le problème.

La tessellation fermée est générée à l'intérieur de notre zone et à l'extérieur de notre zone, de manière séparée. En fait, lorsque nous entrons le nom de la ville dans notre code, les bâtiments de l'entité sont téléchargés, et les barrières téléchargées correspondront également uniquement à celles qui se trouvent dans la ville. Toutefois, puisque notre limite maximale de tessellation est définie par le multipolygone téléchargé par quickOSM, celle-ci s'arrêtera aux limites définies par celui-ci.

Il reste bien sûr possible de supprimer les polygones extérieurs à notre tessellation principale, notamment via QGIS.

4.3 Résumé des 3 types de tessellations

Nous avons vu les 3 types de tessellations : morphologique, fermée et basée sur les blocs.

La tessellation morphologique est la plus basique mais ne permet pas d'utiliser des barrières pour limiter la tessellation. Cela peut l'amener à s'étendre à l'infini ou jusqu'à la limite maximale définie dans le code. Lorsque de nombreux bâtiments sont disponibles via l'Overpass API, ce n'est généralement pas problématique car les tessellations se limitent mutuellement. Cependant, dans le cas où des bâtiments censés être présents ne le sont pas dans les données, ils ne peuvent pas jouer leur rôle de limitation pour la tessellation d'autres bâtiments existants. Cela peut entraîner une expansion excessive des tessellations pour les bâtiments présents, ne reflétant pas fidèlement la réalité du terrain. Malgré ces limitations, cette approche assure une couverture complète de la zone sélectionnée.

La tessellation fermée offre l'avantage d'utiliser des barrières pour délimiter la zone de tessellation. Ces barrières s'adaptent efficacement à la morphologie urbaine du territoire. Cette méthode permet une couverture complète de la zone sélectionnée[25], ce qui correspond bien à l'objectif de GeoPostcodes de maximiser la couverture territoriale. Cependant, il convient d'exclure certaines barrières, notamment les polygones dépourvus de bâtiments, car leur inclusion crée des « trous » dans la tessellation. Une attention particulière doit également être portée aux nombreux cas de barrières parallèles. Malgré ces points de vigilance, cette tessellation s'avère relativement robuste et permet une structuration précise de l'espace urbain.

La tessellation basée sur les blocs offre généralement une couverture étendue du territoire. Sa génération peut se baser sur deux approches : la tessellation morphologique et celle directement basée sur les blocs. Lorsqu'elle se base sur la tessellation morphologique, des blocs se forment selon le réseau de rues et peuvent couvrir l'ensemble du territoire, dans la limite maximale de tessellation définie. Néanmoins, cette méthode peut manquer de précision si l'Overpass API ne dispose pas de données suffisantes sur les bâtiments.

Dans le cas où elle se base sur la tessellation fermée, cette approche nécessite une attention particulière concernant les barrières parallèles, susceptibles de produire des résultats incompatibles avec notre objectif. En effet, nous pouvons obtenir des blocs de bâtiments, non

[25] Bien que cela dépende des tags sélectionnés.

liés les uns aux autres, à cause de multiples barrières les séparant. De plus, contrairement à la tessellation morphologique, elle ne couvrira pas les zones où l'Overpass API manque de données sur les bâtiments. En effet, les tessellations sont ici contraintes par les barrières, ce qui limite leur expansion.

4.4 Tessellation associée aux codes postaux

Dans cette section, nous analysons les résultats de la fusion entre notre tessellation et les codes postaux de GeoPostcodes. Notre approche consiste à examiner des cas représentatifs dans diverses villes, sans prétendre à l'exhaustivité. Pour évaluer précisément l'efficacité de l'attribution des codes postaux, nous nous concentrons sur des zones très restreintes. Il est important de noter que notre objectif n'est pas de comparer directement ces résultats avec les tessellations présentées au point 4.1, mais plutôt d'évaluer l'efficacité de notre méthode. Chaque résultat final pourrait faire l'objet d'une analyse approfondie, difficilement réalisable sur des cartes statiques. Une visualisation dynamique sur un logiciel SIG serait plus appropriée pour mettre en évidence les cohérences et les incohérences de nos résultats. Les exemples potentiels sont nombreux, mais notre attention reste sur la validation de notre méthode plutôt que sur une comparaison détaillée avec les tessellations initiales.

Nous avons choisi de ne pas traiter tous les types de tessellations dans cette analyse. Par exemple, la tessellation morphologique, étant la plus basique, n'est pas incluse dans le cadre des associations de codes postaux. Nous nous concentrons sur les tessellations fermée et par blocs, qui sont plus précises et correspondent davantage aux attentes de GeoPostcodes.

4.4.1 Exemple 1 : tessellation fermée

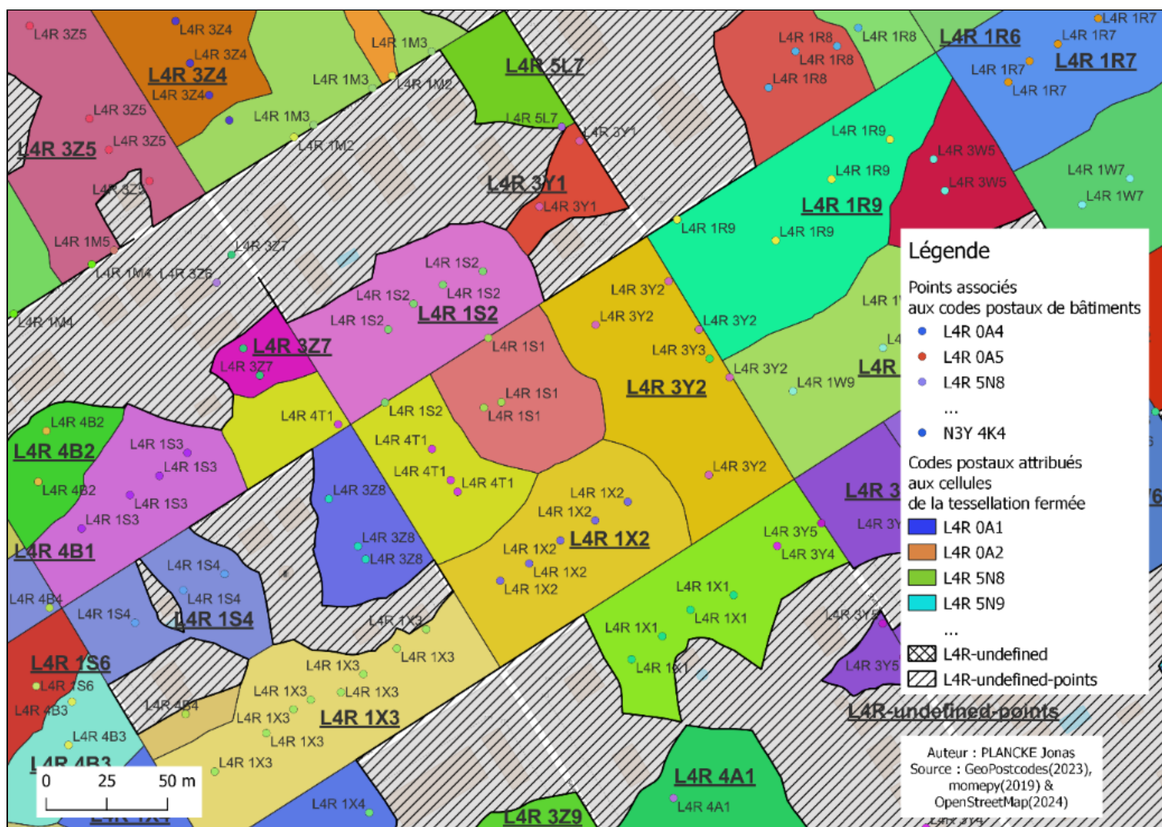

Figure 49 : Codes postaux attribués aux cellules de la tessellation fermée à Midland, Canada (GeoPostcodes(2023), momepy(2019) & OpenStreetMap(2024))

La Figure 49 montre l'attribution des codes postaux aux cellules de tessellation fermée dans la ville de Midland, au Canada. Nous avons affiché les codes postaux attribués à nos cellules en gras et nous les avons soulignés. Nous avons également affiché les points de GeoPostcodes que nous avons catégorisé en fonction des codes postaux pour visualiser la précision de notre requête SQL. Nous voyons que les codes postaux les plus fréquents des points de GeoPostcodes attribuent aux cellules les codes postaux adéquats[26].

Nous pouvons exprimer quelques remarques. Concentrons-nous sur le polygone du code postal L4R 3Y2, au centre de la Figure 49. Nous voyons que la plupart des points correspondent à l'intitulé de ce polygone, sauf un : L4R 3Y3. Si nous regardons maintenant la même zone sur la Figure 50, nous voyons une cellule contenant deux points : L4R 3Y2 et L4R 3Y3. Nous avons donc une égalité. Dans ce cas, la requête SQL choisit d'attribuer le code postal qui touche le plus notre

[26] La carte comprenant de nombreuses données, il était difficile d'afficher correctement toutes les étiquettes sur QGIS.

cellule avec une égalité. Comme nous sommes entourés des codes postaux L4R 3Y2, la cellule de tessellation prend ce code postal. Ensuite, les cellules sont fusionnées en fonction de leur code postal. Cela nous permet d'obtenir des résultats plus cohérents.

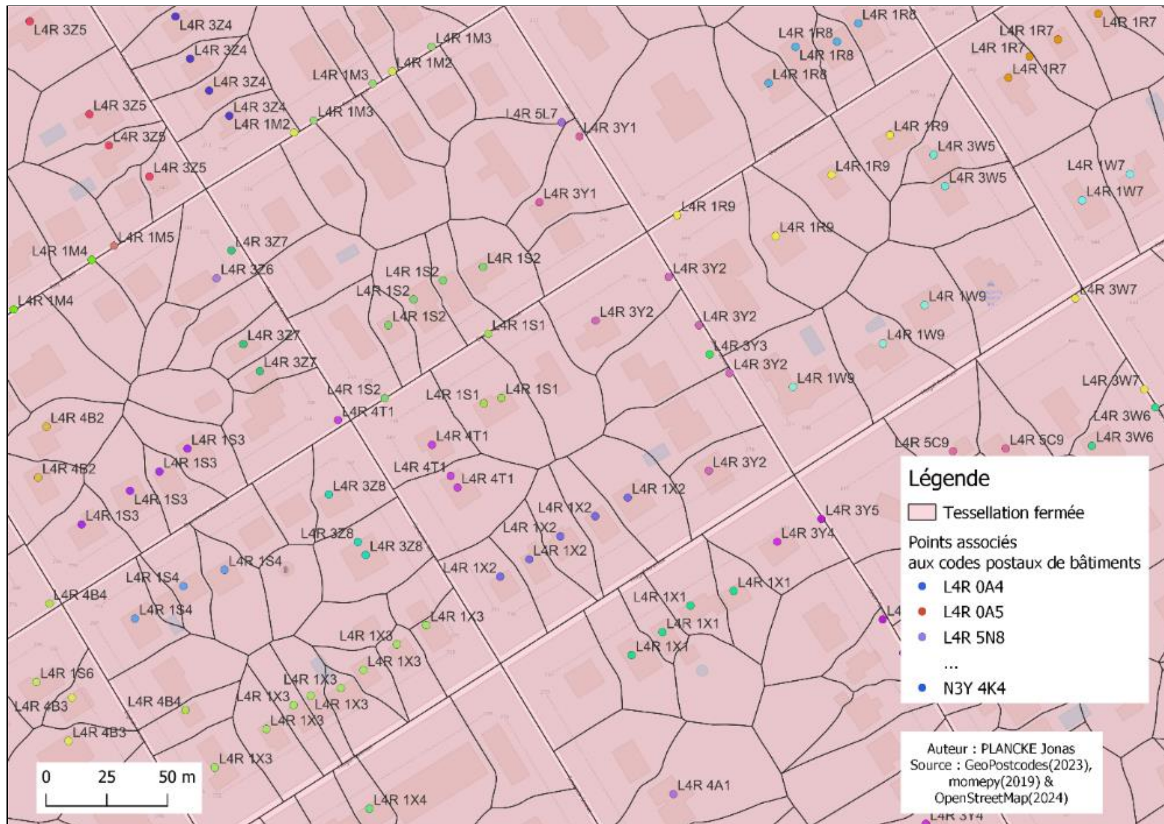

Figure 50 : Tessellation fermée et points associés aux codes postaux de bâtiments (GeoPostcodes(2023), momepy(2019), OpenStreetMap(2024))

Nous constatons également que parfois, des points ne se trouvent pas exactement sur des bâtiments. Cela est dû à une différence entre les données d'OpenStreetMap et celles de GeoPostcodes. C'est justement pour contrer ces occurrences que nous avons décidé d'appliquer cette logique à la requête SQL. Par exemple, le point du code postal L4R 3Y3 aurait probablement dû se trouver dans une autre cellule, de l'autre côté de la rue. Si nous n'avions pas implémenté notre logique d'égalité dans la requête SQL, celle-ci aurait pu choisir aléatoirement entre les deux codes postaux et donner un résultat erroné.

Cette logique peut également amener quelques erreurs. Par exemple, en haut à gauche de la carte, nous pouvons constater que des cellules ont été attribuées du terme *L4R-undefined-points* alors que des points s'y trouvent. Pour rappel, lorsque nous avons des cellules contenant des bâtiments d'OpenStreetMap mais pas de points de GeoPostcodes, nous donnons le nom du code postal plus général, suivi du terme *undefined-points*. Lorsqu'une cellule ne contient ni bâtiment,

ni points, nous utilisons le terme *undefined*. Ici, nous sommes dans code postal L4R[27]. Pour comprendre ce résultat, il faut se référer à la Figure 50. En réalité, nous voyons que deux points se situent sur la cellule de la tessellation fermée : L45 326 et L4R 327. Nous avons donc pour cette cellule, une égalité entre deux codes postaux. Dans ce cas, notre requête SQL choisit d'attribuer à la cellule le code postal de la cellule de tessellation la plus proche. Ici, la cellule la plus proche était *L4R-undefined-points*, d'où cette attribution.

4.4.2 Exemple 2 : tessellation basée sur les blocs

Analysons maintenant la tessellation basée sur les blocs à Midland (Figure 51).

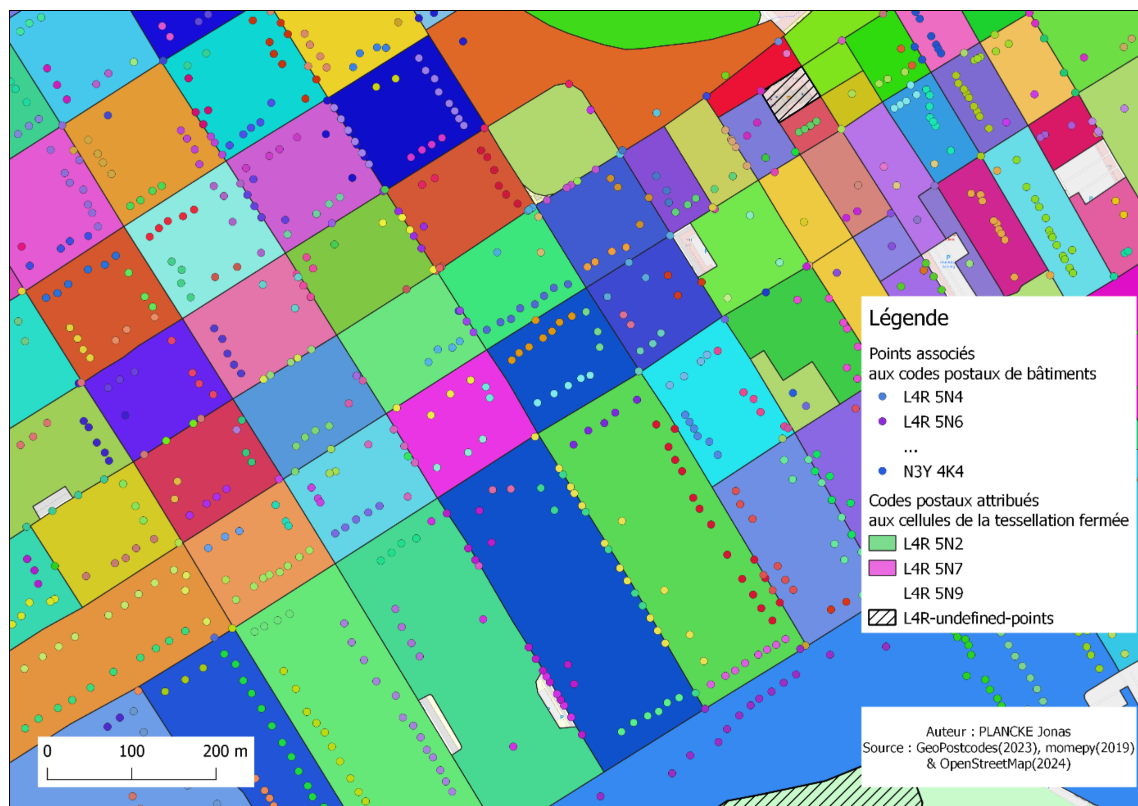

Figure 51 : Codes postaux attribués aux cellules de la tessellation basée sur les blocs à Midland, Canada (GeoPostcodes(2023), momepy(2019) & OpenStreetMap(2024))

La tessellation basée sur les blocs nous donne des codes postaux bien structurés par blocs. Toutefois, ceux-ci ne présentent pas des résultats corrects. Nous voyons que les polygones contiennent des points de différents codes postaux. Malgré une structure en apparence idéale, la tessellation basée sur les blocs ne permet pas de capturer la structure des codes postaux.

[27] Ce code postal est plus général car il ne contient que les 3 premiers caractères du code postal complet.

4.4.3 Exemple 3 : situation idéale

Figure 52 : Codes postaux attribués aux cellules de la tessellation fermée à Delft, Pays-Bas (GeoPostcodes(2023), momepy(2019) & OpenStreetMap(2024))

Sur la Figure 52, nous avons un exemple quasi idéal de tessellation. Pour une meilleure visibilité, nous n'avons pas affiché les intitulés de codes postaux directement sur la carte. Les polygones sont bien définis, et des points de même codes postaux sont visibles sur ceux-ci[28]. La situation est idéale car les points sont correctement assignés aux bâtiments et ce, de manière précise. Les polygones *undefined-points* correspondent à des bâtiments qui existent sur OpenStreetMap mais qui n'ont pas eu de points attribués. Ici, il s'agit en réalité de bâtiments qui ne sont pas des habitations et qui se trouvent dans des jardins. Il est donc naturel de ne pas avoir de code postal attribué. Toutefois, puisqu'ils sont répertoriés comme des bâtiments par OpenStreetMap, des cellules de tessellations été créées et combinées pour former un code postal *undefined-points*. Quant aux polygones *undefined*, il s'agit généralement de zones sans bâtiment, ni point attribué par GeoPostcodes : des parcs, des canaux, etc.

[28] Une couleur = un code postal.

4.4.4 Exemple 4 : points confondus

Figure 53 : Codes postaux attribués aux cellules de la tessellation fermée à Erechim, Brésil (GeoPostcodes(2023), momepy(2019) & OpenStreetMap(2024))

Sur la Figure 53, nous sommes dans la ville d'Erechim, au Brésil. OpenStreetMap ne contient pas de bâtiment pour une importante partie de la ville. La tessellation fermée est donc générée uniquement grâce aux rues. Pour certains polygones, les codes postaux sont correctement attribués. Par exemple, en bas de la carte, le polygone correspondant au code postal 99709-472 est correctement attribué vu l'occurrence des points ayant le même code postal. Pour d'autres, à première vue, cela ne semble pas le cas. Pourtant les résultats sont corrects. Pour mieux comprendre ce qu'il se passe, zoomons au centre de l'image, sur le code postal 99711-102 (Figure 54).

La Figure 54 nous montre le polygone ayant comme code postal attribué : 99711-102. Sur cette carte, il ne semble y avoir qu'une seule occurrence de ce code postal chez les points, avec une multitude d'autres points possédant d'autres codes postaux. En réalité, ce n'est pas le cas. Il s'agit en fait de points confondus. Les points les plus fréquents sont effectivement ceux associés au code postal 99711-102, comme nous pouvons le voir dans la table d'attribut, sur la Figure 55. Ce

cas est très fréquent au Brésil et demande une attention particulière.

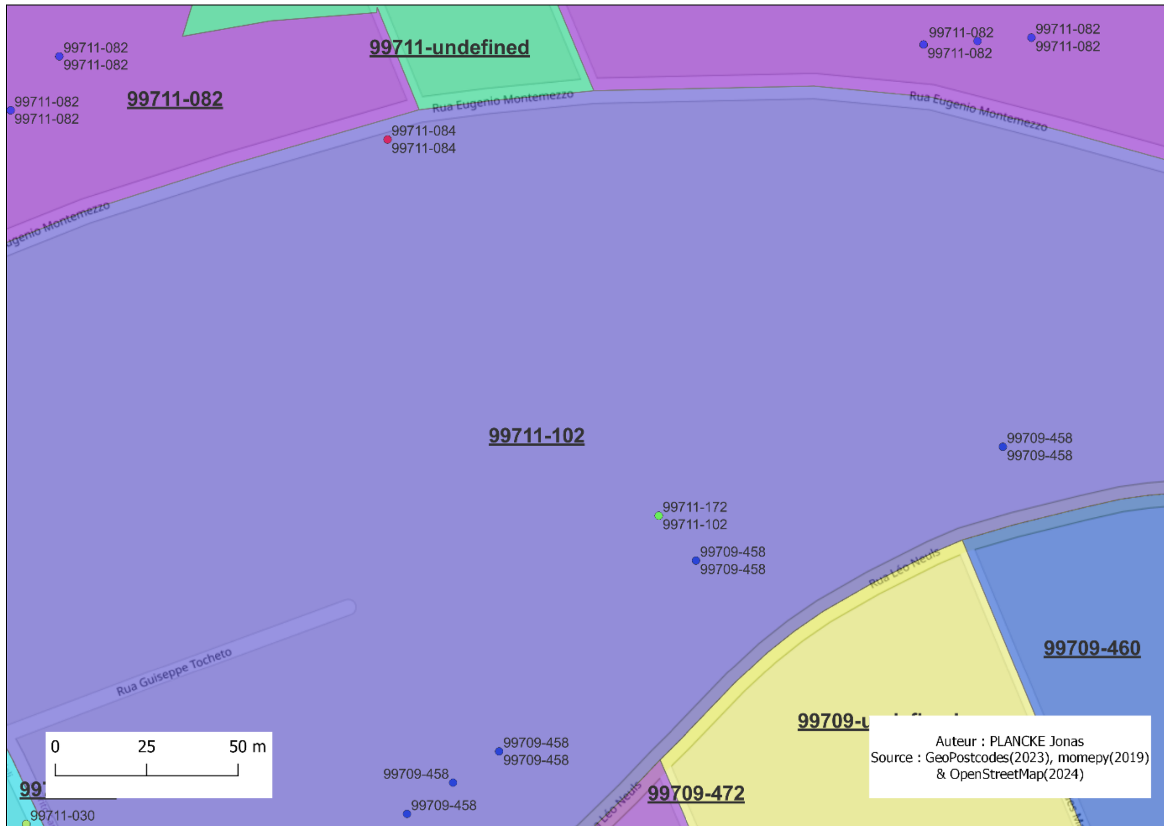

Figure 54 : Zoom sur un code postal avec une « erreur » ? Erechim, Brésil (GeoPostcodes(2023), momepy(2019) & OpenStreetMap(2024)).

	iso	zip	type	id
11101	BR	99711-084	addresse	14795495
11102	BR	99711-102	addresse	33387268
11103	BR	99711-102	addresse	33477131
11104	BR	99711-102	addresse	33694096
11105	BR	99711-102	addresse	33694097
11106	BR	99711-102	addresse	33694098
11107	BR	99711-102	addresse	33694099
11108	BR	99711-102	addresse	33694100
11109	BR	99711-102	addresse	33694101
11110	BR	99711-102	addresse	33694102
11111	BR	99711-102	addresse	33694103
11112	BR	99711-102	addresse	33694105
11113	BR	99711-102	addresse	5263661
11114	BR	99711-102	addresse	5704168
11115	BR	99711-102	addresse	23976638
11116	BR	99711-102	addresse	23976645
11117	BR	99711-102	addresse	23976647
11118	BR	99711-102	addresse	23976648
11119	BR	99711-102	addresse	23976650
11120	BR	99711-102	addresse	23976652
11121	BR	99711-102	addresse	23976653
11122	BR	99711-102	addresse	23976654
11123	BR	99711-102	addresse	23976655
11124	BR	99711-102	addresse	23976660
11125	BR	99711-102	addresse	23976661
11126	BR	99711-102	addresse	28003889

Figure 55 : Bâtiment avec plusieurs codes postaux (GeoPostcodes, 2023)

Nous ne testons pas la tessellation basée sur les blocs pour Erechim. Comme nous l'avons vu dans la partie sur les résultats de la tessellation basée sur les blocs (4.1.4, Figure 45), la tessellation basée sur les blocs ne fonctionnent pas optimalement dans les cas où OpenStreetMap ne possède pas de données sur les bâtiments pour notre zone.

4.4.5 Exemple 5 : incohérence ou manque de données

Figure 56 : Codes postaux attribués aux cellules de la tessellation fermée à Sao Paulo, Brésil (GeoPostcodes(2023), momepy(2019) & OpenStreetMap(2024))

Cette carte révèle plusieurs problèmes potentiels, présentés ici sans légende pour une meilleure visibilité. Premièrement, on constate une incohérence entre la majorité des points et des bâtiments. Bien que ces points soient enregistrés comme des adresses dans la base de données de GeoPostcodes, ils apparaissent au centre des rues plutôt que sur les bâtiments, ce qui est inattendu. Il y a une incohérence entre les bâtiments d'OpenStreetMap et les données de GeoPostcodes.

De plus, une part importante des points associés aux bâtiments est manquante. Cela génère de nombreuses cellules *undefined-points*, car les bâtiments sont bien présents sur OpenStreetMap mais absents de la BD de GeoPostcodes ou alors, il existe un décalage entre les deux sources de

données. Cette situation, combinée à l'utilisation d'une tessellation fermée, entraîne un découpage irrégulier des polygones et une attribution incorrecte des codes postaux.

Enfin, on observe que certaines tessellations sont imbriquées dans d'autres. Ce phénomène résulte de la prise en compte de bâtiments complexes comme plusieurs entités distinctes, en fonction de la forme de leur toit. La tessellation se forme alors autour de ces pseudo-bâtiments ainsi que sur le bâtiment principal, produisant des données erronées.

5 DISCUSSION ET PERSPECTIVES

De nombreuses perspectives peuvent se dessiner pour la suite de l'utilisation de momepy pour affiner les codes postaux.

Une première perspective concerne les barrières parallèles multiples. Pour les voies ferrées ou pour certaines routes parallèles les unes ou autres, plusieurs barrières seront créées. Il est possible de pas prendre en compte certaines barrières pour éviter ce problème. Par exemple, si des routes sont parallèles à des chemins empruntés par des piétons, nous pouvons ne pas prendre en compte les chemins empruntés par les piétons. Cela permet d'avoir une barrière unique, qui structure correctement le contexte urbain. Toutefois, les cas des routes et des voies ferrées parallèles présentent un défi plus important. Il faudrait donc trouver une manière de supprimer ces multiples entités pour n'en former qu'une.

Une deuxième perspective consiste à étendre certaines impasses grâce à la fonction *momepy.extend_lines()*. Cette méthode permettrait de prolonger ces barrières jusqu'à leur intersection avec le réseau existant, formant ainsi un maillage complet. Dans le contexte de la tessellation fermée, cela se traduirait par l'ajout de nouvelles barrières. Pour la tessellation basée sur les blocs, cette méthode permettrait de fermer certaines boucles de rues incomplètes et d'en créer de nouvelles, formant ainsi des blocs supplémentaires pour la tessellation.

Une troisième perspective peut être la question de l'assignation de codes postaux pour les données incohérentes ou imprécises. Dans les faits, nous avons plusieurs cas de figures :

1. Une cellule issue de la tessellation n'a pas de bâtiment enregistré sur OpenStreetMap alors que des bâtiments y existent. Des points de GeoPostcodes peuvent y être attribués.
2. Une cellule issue de la tessellation contient des bâtiments sur OpenStreetMap mais n'a pas de point de GeoPostcodes. Les points peuvent également ne pas correspondre aux données de bâtiments d'OpenStreetMap (incohérence entre les données).
3. Une cellule issue de la tessellation n'a pas de bâtiment sur le terrain et donc pas de point attribué par GeoPostcodes.

Dans le premier cas, nous avons vu que la tessellation morphologique et la tessellation basée sur les blocs ne sont pas efficaces pour ce genre de problème. La tessellation fermée permet d'obtenir des résultats plus pertinents. En effet, comme cette tessellation se base notamment

sur le réseau de rues, cela permet de compenser le manque de données sur les bâtiments et de tout de même créer des cellules de tessellations même si certains bâtiments manquent à l'appel. Il serait toutefois pertinent de chercher une autre source de données de bâtiments plus complète. Une alternative à l'utilisation d'OpenStreetMap pourrait être des données du projet *Microsoft Global ML Building Footprints*. Il s'agit d'une initiative de cartographie à grande échelle qui utilise l'intelligence artificielle pour identifier et délimiter automatiquement les contours de bâtiments dans le monde entier (Microsoft, 2024). Une autre alternative peut être l'utilisation de *Overture Maps*. Il s'agit également d'une initiative qui vise à créer une base de données cartographiques mondiales fiable, facile d'utilisation et interopérable (Overture Maps Foundation, 2024).

Dans le deuxième cas, les ensembles de bâtiments qui n'ont pas de point attribué formeront des cellules *undefined-points*. Pour les données incohérentes entre les bâtiments et les points de GeoPostcodes, une alternative est également d'utiliser une source de données plus précises qu'OpenStreetMap comme *Microsoft Global ML Building Footprints* ou *Overture Maps*. Pour les bâtiments qui ne possèdent pas de données de GeoPostcodes, l'ajout progressif de nouvelles données permettra d'améliorer la précision de la tessellation.

Dans le troisième cas, les cellules sans code postal ni bâtiment se voient attribuer le terme *undefined*. Comme nous l'avons précédemment mentionné, GeoPostcodes privilégie une couverture maximale du territoire. Il est pertinent de s'interroger sur l'approche optimale pour les zones dépourvues de bâtiment. Est-il judicieux d'attribuer un code postal à une zone qui ne contient pas (encore) de bâtiment ? Une forêt sans zone habitable devrait-elle avoir un code postal ? Faut-il en attribuer un à un lac ? D'un côté, l'attribution d'un code postal à toutes les zones d'un pays peut s'avérer utile, en prévision d'éventuels développements futurs. Une habitation pourrait un jour s'implanter là où il n'y en a pas actuellement. Nous aurions pu opter pour l'attribution des codes postaux les plus proches à ces cellules sans bâtiment ni point. Cependant, dans un souci de précision, nous avons choisi de leur assigner des codes postaux spéciaux, facilitant ainsi leur identification.

Enfin, une quatrième perspective peut être le téléchargement d'un jeu de données directement dans la BD de GeoPostcodes. Un problème récurrent était le temps d'attente de l'Overpass API.

En effet, il arrivait fréquemment que la quantité de géométries demandées soient imposantes et dépassaient alors les temps d'attentes autorisés. Pour contrer cela, nous avons dû régulièrement nous focaliser sur des quartiers particuliers au lieu de villes entières. Il serait idéal que GeoPostcodes puisse se passer de l'Overpass API en ayant directement un jeu de données à disposition, qu'il soit originaire d'OpenStreetMap, Microsoft Global ML Building Footprints, Overture Map ou d'une autre source de données.

6 CONCLUSION

Dans le cadre de ce mémoire, nous avons développé un programme permettant à GeoPostcodes d'exploiter la bibliothèque Python momepy pour générer trois types de tessellations : morphologique, fermée et basée sur les blocs. Ces tessellations s'inspirent de la tessellation de Voronoï, mais se forment autour de bâtiments plutôt que de points. Le programme peut générer ces tessellations pour toute zone spécifiée par l'utilisateur, à condition qu'elle respecte la nomenclature de l'Overpass API et que le pays soit précisé. Les résultats sont sauvegardés dans leur base de données, puis combinés avec leurs propres données de codes postaux via une requête SQL, aboutissant à des polygones auxquels sont attribués des codes postaux.

Parmi ces trois approches, la tessellation fermée présente un avantage important. Elle offre une précision supérieure en créant des cellules limitées par des barrières, définies par des tags d'OpenStreetMap que nous pouvons choisir en fonction du contexte urbain. Ces barrières doivent généralement être linéaires plutôt que polygonales, car les polygones sans bâtiments créent des vides dans la tessellation. Cependant, l'utilisation de polygones contenant des bâtiments reste pertinente, la tessellation s'arrêtant alors à la limite du polygone[29].

La tessellation morphologique, plus basique, n'utilise pas de barrière. Elle est efficace dans les zones densément bâties, mais s'éloigne de la structure parcellaire dans d'autres contextes. La tessellation basée sur les blocs, bien qu'esthétiquement plaisante, ne capture pas toujours fidèlement la structure des codes postaux.

Un avantage de la tessellation fermée apparaît lorsque certains bâtiments sont absents d'OpenStreetMap, comme c'est souvent le cas au Brésil. Dans ces situations, elle génère des résultats plus cohérents en se basant sur le réseau de barrières spécifié, contrairement aux deux autres méthodes qui produisent alors des résultats moins convaincants.

L'attribution des codes postaux via la requête SQL fonctionne généralement bien pour nos tessellations. Les problèmes rencontrés proviennent principalement de données manquantes ou d'incohérences entre les données de GeoPostcodes et celles d'OpenStreetMap. Ces difficultés

[29] Bien que nous n'ayons pas inclus d'exemple spécifique dans le mémoire, notre code intègre une fonctionnalité testée en interne qui illustre cette approche. Nous avons notamment expérimenté avec la barrière *historic : castle* sur un château situé dans le Vieux-Québec, au Canada.

pourraient être résolues en enrichissant les données de points ou en utilisant des sources plus précises qu'OpenStreetMap, comme les projets d'*Overture Maps* ou de *Microsoft Global ML Building Footprints.*

Somme toute, nous avons démontré la possibilité d'utiliser les tessellations de momepy pour améliorer la granularité des polygones actuels de GeoPostcodes, en attribuant les codes postaux des points de GeoPostcodes aux cellules de tessellation. Pour optimiser les résultats, il est nécessaire de sélectionner judicieusement les barrières via l'Overpass API en fonction du contexte urbain et de les ajuster en pré ou post-traitement pour certains cas problématiques, comme les nombreuses barrières parallèles. Avec des données complètes et cohérentes sur la localisation des bâtiments, les résultats finaux gagneront en précision, permettant à GeoPostcodes d'atteindre une granularité fine pour l’attribution de codes postaux.

7 BIBLIOGRAPHIE

Boeing, G. (2024). *User Reference - OSMnx 1.9.4 documentation*. https://osmnx.readthedocs.io/en/stable/user-reference.html. Consulté en février 2024.

Boeing, G. (2024). Modeling and Analyzing Urban Networks and Amenities with OSMnx. Working paper. https://geoffboeing.com/publications/osmnx-paper/. Consulté en mai 2024.

DataGenetics (2017). *Voronoi tessellations*. http://datagenetics.com/blog/may12017/index.html. Consulté en février 2024.

Codigo-postal.org. (2024). *Brazil ZIP Code*. https://codigo-postal.org/en-us/brazil/. Consulté en avril 2024.

Du, Q., Faber, V., and Gunzburger, M. (1999). Centroidal Voronoi Tessellations. *Applications and Algorithms. SIAM Review*, 41(4): 637-676.

Erwig, M. (2000). The graph Voronoi diagram with applications. *Networks,* 36(3): 156–163.

Fleischmann, M. *et al.* (2018). *momepy documentation — momepy 0.8.1.dev0+gd33e67f.d20240722 documentation*. https://docs.momepy.org/en/stable/. Consulté en novembre 2023.

Fleischmann, M., (2019). Momepy: Urban Morphology Measuring Toolkit. *Journal of Open Source Software*, 4(43): 1807.

Fleischmann, M., Feliciotti, A., Romice, O. and Porta, S. (2020). Methodological Foundation of a Numerical Taxonomy of Urban Form. *Environment and Planning B: Urban Analytics and City Science*, 49 (4): 1283-1299.

Fleischmann, M., Feliciotti, A., Romice, O., *et al.* (2020), Morphological tessellation as a way of partitioning space: Improving consistency in urban morphology at the plot scale. *Computers, Environment and Urban Systems*, 80: 101441.

Fleischmann, M., and Arribas-Bel, D. (2022). Geographical characterisation of British urban form and function using the spatial signatures framework. *Scientific Data*, 9(1): 546.

Fleischmann, M., Feliciotti, A. and Kerr, W. (2022), Evolution of Urban Patterns: Urban Morphology as an Open Reproducible Data Science. Geogr Anal, 54: 536-558. https://doi.org/10.1111/gean.12302

Fleischmann, M. (2024). *Martin Fleischmann*. https://martinfleischmann.net/. Consulté en juin 2024.

GeoPandas developers. (2013). *About GeoPandas*. https://geopandas.org/en/stable/about.html. Consulté en mai 2024.

GeoPandas developers. (2013). *geopandas.GeoDataFrame — GeoPandas 1.0.1+0.g747d66e.dirty documentation*. (n.d.). https://geopandas.org/en/stable/docs/reference/api/geopandas.GeoDataFrame.html. Consulté en février 2024.

GeoPandas developers. (2013). *geopandas.GeoDataFrame.to_postgis — GeoPandas 1.0.1+0.g747d66e.dirty documentation*. (n.d.). https://geopandas.org/en/stable/docs/reference/api/geopandas.GeoDataFrame.to_postgis.html. Consulté en mars 2024.

GeoPostcodes. (2023). *Postgres* [Base de données]. GeoPostcodes. Données fournies directement par GeoPostcodes.

GeoPostcodes. (2024). *GeoPostcodes : International Postal & ZIP Code Database*. https://www.geopostcodes.com. Consulté en décembre 2023.

GISGeography. (2024). World Geodetic System (WGS84). *GIS Geography.* https://gisgeography.com/wgs84-world-geodetic-system/. Consulté en mars 2024.

Keeler, P. (2023). Voronoi tessellations. H. Paul Keeler. https://hpaulkeeler.com/voronoi-dirichlet-tessellations/. Consulté en janvier 2024.

Lecaché, J. (2023). How to build a postal Code polygon Database. *GeoPostcodes.* https://www.geopostcodes.com/blog/postal-code-polygon-database/. Consulté en novembre 2023.

Lynch, P. (2017). How Voronoi diagrams help us understand our world. *The Irish Times*. https://www.irishtimes.com/news/science/how-voronoi-diagrams-help-us-understand-our-world-1.2947681. Consulté en janvier 2024.

Microsoft (2024). microsoft/Global ML Building Footprints: Worldwide building footprints derived from satellite imagery. *GitHub.* https://github.com/microsoft/GlobalMLBuildingFootprints. Consulté en mai 2024.

NetworkX (2024). *NetworkX documentation*. https://networkx.org. Consulté en mars 2024.

Okabe, A., Boots, B., and Sugihara, K. (1992). *Spatial Tessellations: Concepts and applications of Voronoi diagrams*. http://ci.nii.ac.jp/ncid/BA47125014. Consulté en juin 2024.

OpenStreetMap Contributors. (2024). About OpenStreetMap. *OpenStreetMap Wiki*. https://wiki.openstreetmap.org/wiki/About_OpenStreetMap. Consulté en février 2024.

OpenStreetMap Contributors. (2024). Map features. *OpenStreetMap Wiki*. https://wiki.openstreetmap.org/wiki/Map_features. Consulté en février 2024.

OpenStreetMap Contributors. (2024). OpenStreetMap. https://www.openstreetmap.org.

OpenStreetMap Contributors. (2024). Overpass API. *OpenStreetMap Wiki*. https://wiki.openstreetmap.org/wiki/Overpass_API. Consulté en février 2024.

OpenStreetMap Contributors. (n.d.). What is OpenStreetMap?. *OpenStreetMap Wiki*. https://welcome.openstreetmap.org/what-is-openstreetmap/. Consulté en février 2024.

Overture Maps Foundation. (2024), *Linux Foundation Project*. https://overturemaps.org/. Consulté en mai 2024.

Postal Codes in Canada. (n.d.). *Canadian postal codes and Address Lookup*. https://www.postalcodesincanada.com/. Consulté en décembre 2023.

PostGIS PSC and OSGeo. (2023). About PostGIS. *PostGIS*. http://postgis.net/. Consulté en février 2024.

PostgreSQL Tutorial. (2011). What is postgresql ?. *Getting Started with Postgresql.* https://www.postgresqltutorial.com/postgresql-getting-started/what-is-postgresql/. Consulté en février 2024.

QGIS. (n.d.). QuickOSM QGIS Python Plugins Repository. *QGIS Web Site* https://plugins.qgis.org/plugins/QuickOSM/. Consulté en mars 2024.

QGIS. (2024). Spatial without Compromise. *QGIS Web Site*. https://qgis.org/. Consulté en mars 2024.

Solc, T. (2024). *Unidecode* (Version 1.3.8) [Python Package]. PyPI. https://pypi.org/project/Unidecode/

Wang, J., Fleischmann, M., Venerandi, A., Romice, O., Kuffer, M., and Porta, S. (2023). EO+ Morphometrics: Understanding cities through urban morphology at large scale. *Landscape and Urban Planning*, 233: 104691.