Exploitation de l’Open Data avec Python par l’architecte

Démonstration par la pratique des apports potentiels au sein de la conception architecturale

ABSTRACT

Ce mémoire a pour but d'aborder les enjeux de l'appropriation du langage de programmation Python par les architectes afin d'exploiter les données issues de plateformes en Open Data. Plus précisément, le potentiel de ce langage sera démontré à travers une approche pratique de plusieurs scripts répondant aux principaux enjeux autour de l'appropriation des données ouvertes par les architectes. Comment manipuler de manière universelle les formats de données ouverts et leur structure ? De quelle manière peut-on les intégrer dans l'environnement de travail de la conception architecturale ? Peut-on finalement se servir de Python pour mener ses propres analyses ?

SOMMAIRE

INTRODUCTION

Au cours des dernières années, un déploiement prolifique de jeux de données est en train d'avoir lieu sous l'égide de « l'Open Data ». Sur le territoire Français, des dispositifs mis en place par le gouvernement tel qu'« Etalab », chargé de la coordination et la mise en place de l'ouverture de jeux de données (par décret du 30 Octobre 2019) incarnent cette volonté de faciliter la diffusion de données ouvertes, tout en promouvant leur réutilisation.[1]

Le gouvernement définit l'Open Data comme « l'effort que font les institutions, notamment gouvernementales, qui partagent les données dont elles disposent ».[2] En effet, c'est avant tout une stratégie prônant l'ouverture du plus grand nombre de bases de données au public, les rendant ainsi totalement accessibles. A l'instar des autres mouvements du même type, tel que « l'Open Source », le traitement et la rediffusion des données sont autorisées, voir même encouragées comme c'est le cas par le gouvernement français : « les données partagées trouvent des réutilisateurs qui les intègrent dans de nouveaux services à forte valeur ajoutée économique ou sociale. ». Les règles relatives à leur réutilisation font l'objet d'une licence publique et universelle, ne réclament pas ou peu de démarches pour se l'approprier.

Ainsi, de nombreuses plateformes mises en place par diverses instances opérant dans des domaines très variés ont vu le jour au cours des dernières années, allant d'organismes spécialisés dans les données géographiques comme l'Institut national de l'information géographique et forestière (IGN)[3] jusque dans le domaine des transports comme Ile de France Mobilités,[4] en passant par l'environnement et l'écologie tel que l'ADEME.[5]

---

[1] *Etalab - Qui sommes-nous. Le blog d'Etalab* [en ligne]. [s. d.]. [Consulté le 17 janvier 2021]. Disponible à l'adresse : https://www.etalab.gouv.fr/qui-sommes-nous.

[2] *L'ouverture des données publiques. Gouvernement.fr* [en ligne]. [s. d.]. [Consulté le 17 janvier 2021]. Disponible à l'adresse : https://www.gouvernement.fr/action/l-ouverture-des-donnees-publiques.

[3] *Géoservices | Accéder au téléchargement des données libres IGN* [en ligne]. [s. d.]. [Consulté le 15 septembre 2020]. Disponible à l'adresse : https://geoservices.ign.fr/documentation/diffusion/telechargement-donnees-libres.html.

[4] *Portail Open data Île-de-France Mobilités* [en ligne]. [s. d.]. [Consulté le 17 janvier 2021]. Disponible à l'adresse : https://data.iledefrance-mobilites.fr/pages/home/.

[5] *Portail open data de l'ADEME* [en ligne]. [s. d.]. [Consulté le 17 janvier 2021]. Disponible à l'adresse : https://data.ademe.fr/.

Bien que cette nécessité étatique de partager l'information publique ne date pas de l'apparition du Web (comme l'explique la loi Cada de 1978), Ce dernier a permis, au-delà de la dispense de tout intermédiaire (notamment humain) entre le fournisseur et l'utilisateur, d'exploiter de nouvelles formes d'accès et surtout de consommation, en particulier via des scripts ou des algorithme écrits dans un langage de programmation afin d'automatiser la récupération de données depuis les formats de fichiers ouverts.

Ainsi, quiconque cherche à mettre en place un travail d'analyse le plus exhaustif possible d'un contexte donné peut, grâce aux plateformes et moyens cités ci-dessus, disposer très rapidement de données riches et abondantes.

De ce point de vue-là, il paraît extrêmement pertinent pour les métiers issus de l'architecture et de l'urbanisme, et en particulier le métier d'architecte, de se saisir des données issues de l'Open Data afin de renforcer leur compréhension du territoire sur lequel ils construisent, que cela soit par la simple analyse statistique ou bien la récupération d'informations géométriques d'un site.

Or, les architectes ont tendance à préférer, de par leur expertise orientée sur la conception, réclamant un esprit de synthèse affûté, les résultats explicites d'analyse de données plutôt que les données en elles-mêmes. De plus, les outils numériques sur lesquels les architectes se forment relèvent très majoritairement des domaines du dessin, de la modélisation ou de la communication plutôt que de l'analyse en elle-même, qui accentue leur besoin de résultats synthétiques « préfabriqués ».

Cependant, il existe depuis les années 2010 un certain essor des travaux de recherche basés sur des données issues en partie ou totalement de l'Open Data, et ce, grâce à un langage de programmation en particulier, dont la simplicité de la syntaxe couplée à une profusion de bibliothèques (comparables à des « plug-in ») spécialisées dans le traitement de données informatiques en ont fait un outil populaire pour la recherche

d'aujourd'hui, le Python. A juste titre, ce langage est aujourd'hui très répandu au sein des Systèmes d'Informations Géographiques (SIG) tels que ArcGIS, où ses caractéristiques mentionnés ci-dessus permettent de manière accessible de mener des travaux complexes autour des données géographiques ouvertes, de l'analyse et la datavisualisation[6] à l'entraînement de modèles de prédiction.[7] Comme l'illustre l'exemple du travail de recherche « CityEngine - Twitter » mené au « Centre for Advanced Spatial Analysis » de Londres[8] , proposant une cartographie urbaine de densité basée sur des Tweets géolocalisés dans cette même ville, un seul et unique script en Python permet à la fois de récupérer les messages sur une plage de 24 heures (via une bibliothèque , nommée « Tweepy » , permettant au code d'interagir avec l'API de Twitter), de les trier et d'en extraire leurs coordonnées et leur horaire de publication, et enfin de fournir ces données directement à l'outil de génération de modèles 3D urbains « CityEngine » (publié par l'ESRI) afin que ce dernier puisse constituer une carte procédurale (animée selon le nombre de tweets sur une plage de 24 heures).

Une telle étude étant désormais possible sur des données massives privées, ce type d'exploitation peut encore plus aisément être mis en place lorsque les données utilisées sont totalement ouvertes et avec accès illimité.

Ainsi, grâce à des données massives accessibles (tant en termes de tarifs qu'en terme de facilité d'extraction) couplées à un langage de programmation comme Python, développer ses propres analyses par exploitation de données brutes est désormais à la portée des chercheurs, sans avoir besoin d'un bagage informatique conséquent.

Dès lors, face à la complexité des enjeux auxquels la conception architecturale fait appel (climatique, socio-

[6] *Spatial and temporal distribution of service calls using big data tools | ArcGIS for Developers* [en ligne]. [s. d.]. [Consulté le 3 février 2021]. Disponible à l'adresse : https://developers.arcgis.com/python/sample-notebooks/spatial-and-temporal-trends-of-service-calls/.

[7] *Automate Road Surface Investigation Using Deep Learning | ArcGIS for Developers* [en ligne]. [s. d.]. [Consulté le 3 février 2021]. Disponible à l'adresse : https://developers.arcgis.com/python/sample-notebooks/automate-road-surface-investigation-using-deep-learning/.

[8] HÜGEL, Stephan et ROUMPANI, Flora. *CityEngine-Twitter* [logiciel]. [S. l.] : Zenodo, 14 mai 2014. [Consulté le 28 décembre 2020]. DOI 10.5281/ZENODO.9795.

économique, écologique ou structurel par exemple), il semble pertinent d'envisager que des architectes se saisissent de ce type d'outil, dans le but de construire, au prisme de leurs propres volontés d'intervention (même complexes), leurs propres modèles de compréhension du territoire. Ce nouveau regard, personnalisé par l'architecte, pourrait alors apporter à ce dernier des éléments susceptibles de le guider de manière bien plus significative, en particulier dans les premières phases d'esquisse, afin d'améliorer la qualité de sa production.

Ainsi, dans quelle mesure l'exploitation de données issues de l'Open Data grâce au langage Python représente-t-elle un avantage certain pour l'architecte ? Après avoir initialement démontré l'intérêt du langage Python dans l'extraction et la manipulation des données issues des plateformes accessibles en Open Data à travers l'élaboration complète d'un script de récolte de données, ce dernier sera complété à travers un aperçu constitué d'exemples clés de la capacité de Python à produire des documents de travail utiles à l'architecte (cartographie, dessin et modélisation). Enfin, ce travail d'exploitation sera abouti en montrant la prodigieuse capacité du langage Python à permettre de manière accessible l'analyse complexe de ces données ainsi que la mise en place d'algorithmes de prédiction.

1 Etude de cas des données des « volumes bâtis » de l'Open Data Paris : le script Python comme outil unique pour gérer la complexité.

Tel que le stipule le portail européen de données, au-delà de l'accessibilité en elle-même des données, la question de la lisibilité des structures de données et des formats de fichiers disponibles sur les plateformes relevant de l'Open Data est d'importance cruciale : « On peut utiliser les données car elles sont disponibles sous une forme commune et lisibles par des machines. ».[9] Cet organisme relève également un autre aspect primordial, celui de la facilité du traitement des données par les outils informatiques. En effet, elles ont davantage vocation à faire l'objet de manipulations automatiques (synthèse, tri, etc…) plutôt que d'être simplement lues par un utilisateur humain.

Pour partager des données tabulaires (sous forme de tableur) par exemple, là où un utilisateur humain préfèrera un format Excel (.XLSX) (en y incluant notamment couleurs et styles de polices pour améliorer sa lisibilité), le portail européen des données recommande plutôt d'autres formats comme le .CSV (Comma Separated Values), format ouvert constitué de texte brut séparé par des caractères spéciaux, compatible avec un large panel d'outils logiciels capable d'opérations de traitement.

Face à ce besoin de compréhension et de manipulation de données brutes, les langages de programmation de haut niveau d'abstraction (possédant une syntaxe plus lisible et concise pour l'humain, rendant leur utilisation accessible) et en particulier le Python apparaissent alors comme des outils offrant la souplesse et la puissance nécessaire pour répondre à cette problématique.

Au sein de cette section, le jeu de données « Volumes bâtis » de la plateforme Open Data Paris sera étudié de près en tant qu'exemple type, à travers une approche concrète de sa complexité. Afin de permettre son exploitation, un script Python sera élaboré en fin de section. Ce travail servira également de base pour

[9] *What is open data ?* [en ligne]. [s. d.]. [Consulté le 28 décembre 2020]. Disponible à l'adresse : https://www.europeandataportal.eu/elearning/en/module1/#/id/co-01.

aborder les concepts plus approfondis des chapitres suivants.

1.1 L'Open Data : entre nomenclature et variables

La plupart des plateformes distribuant des données en Open Data proposant directement en ligne des moyens de prévisualiser un jeu de données, cela semble constituer un moyen pratique de discerner et comprendre son contenu en détail. C'est le cas sur la plateforme Open Data Paris, qui nous permet de prévisualiser le jeu de données des volumes bâtis sous la forme d'un tableau, mais aussi d'une carte, laquelle formera un premier contact avec les données en elle-mêmes.

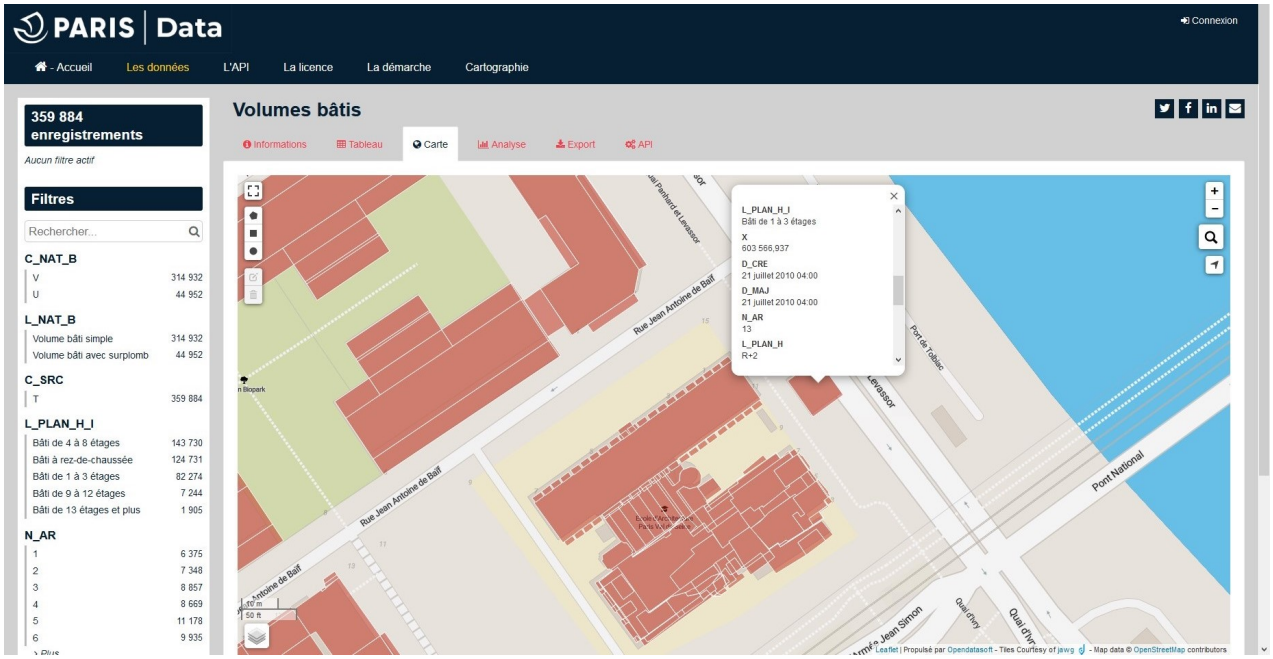


FIG. 1 :

1.1.1 Variables et typologies des valeurs

Le premier constat que l'on peut réaliser après avoir brièvement interagi avec la carte est que le jeu de donnée associe un ensemble de **variables** (dont la dénomination est commune à l'ensemble de ce jeu) et leurs **valeurs** (possédant elles aussi une notation spécifique) avec une **forme géométrique géolocalisée** sur un fond de carte (en l'occurrence, ce sont des **polygones**, formes géométriques les plus à même de représenter l'emprise en plan des différents bâtiments).

Afin de permettre une lecture plus complémentaire, il paraît intéressant de consulter le tableau fin d'avoir une vue plus "centrée" sur les différentes variables et leurs valeurs.

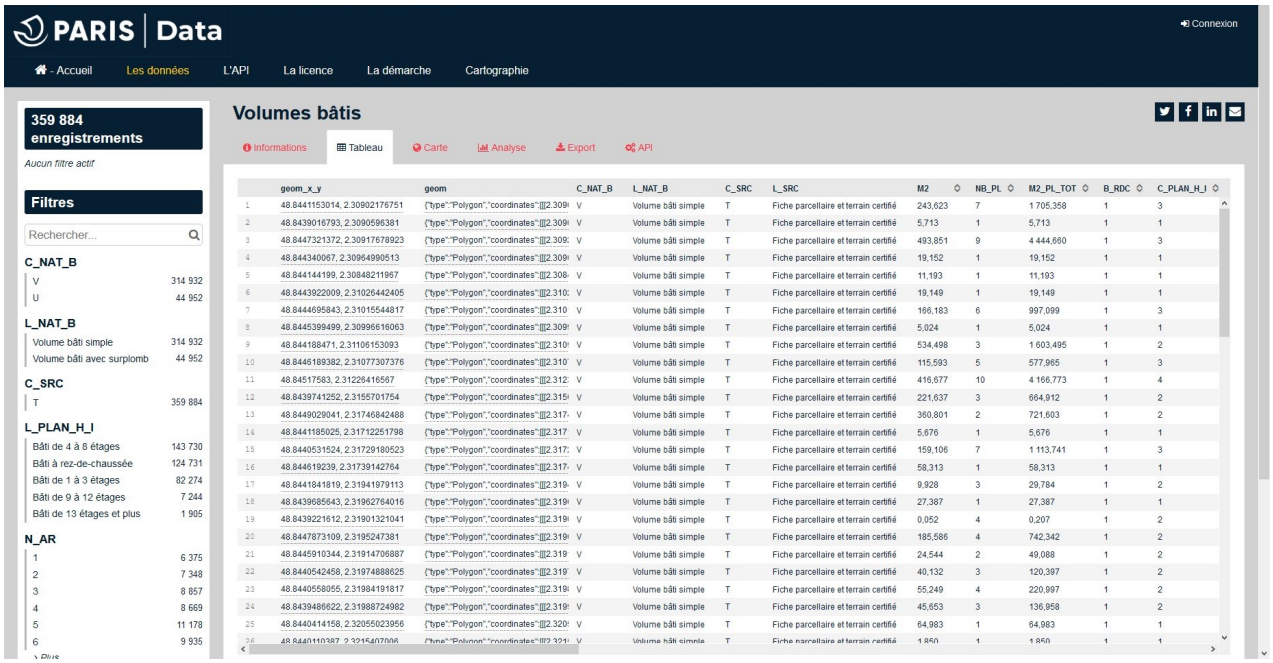


FIG. 2 :

Chacune d'entre elles est ici représentée par une **colonne**, chaque ligne correspondant à un **volume bâti**. Tout d'abord, la variable *geom* est celle qui contient les informations géométriques, nous renseignant sur le type de géométrie employée, ainsi que les coordonnées des points qui la définissent. En l'occurence, la

typologie géométrique “Polygon” se base sur les types primitifs de références des Systèmes d’Informations Géographiques (SIG), et ses coordonnées sont définies en **latitude/longitude** (ce que confirme la variable *geom_x_y*).

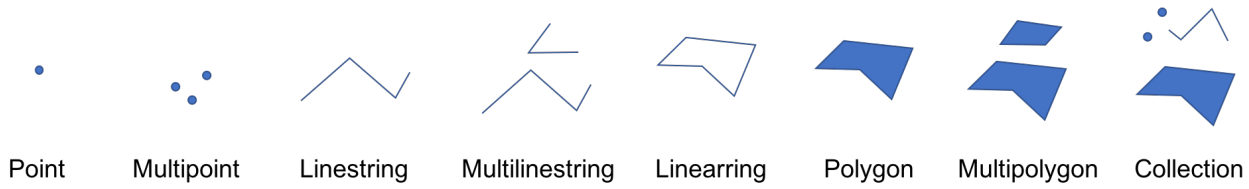


FIG. 3 : Primitives géométriques en SIG

Nous pouvons également noter que certaines variables comme *L_NAT_B* ou *L_SRC* sont exprimées sous forme de texte, qualifié alors de **“chaîne de caractères”** (“string” ou “str” en anglais) d’un point de vue informatique. Elles semblent également **catégoriques**, c’est à dire ne pouvant prendre qu’un nombre défini de valeurs possibles. Bien que les noms de ces variables ne soient pas explicites, leurs valeurs permettent d’avoir une première idée de ce qu’elles renseignent.

A l’inverse, d’autres variables comme *B_RDC* ne possèdent ni un nom explicite, ni une valeur permettant de suggérer sa signification, étant catégorique mais notée sous forme d’**entiers**.

Enfin, d’autres variables comme *M2* ou *NB_PL* sont notées **numériquement**, pouvant à priori prendre une infinité de valeurs , respectivement sous la forme de nombres à virgule (qualifiés de **“float”** en anglais), et d’entiers (“int”). Bien que l’on puisse deviner que *M2* semble représenter la surface d’un volume bâti, cela reste une supposition.

Rappelons égalament que toutes ces observations sont faites sur un échantillon visible d’un jeu de données massif. Certaines subtilités présentes plus loin dans le tableau peuvent encore échapper à cette lecture préliminaire.

Dès lors, chaque variable possédant sa propre nomenclature, et étant plus ou moins explicite dans sa dénomination, une première difficulté de lecture émerge. Heureusement, les jeux de données en Open Data disposent généralement d'informations complémentaires capables de renseigner l'utilisateur sur cette nomenclature, afin qu'il puisse exploiter les données.

1.1.2 Les métadonnées : clé de compréhension des données

Comme c'est le cas ici, la majorité des jeux de données accessibles en Open Data disposent d'un document annexe de référence, dont le but est à minima de fournir à l'utilisateur qui souhaite se saisir des données contenues les explications nécessaires à la compréhension des variables constituant le jeu de données en question. Ce sont **les métadonnées**.

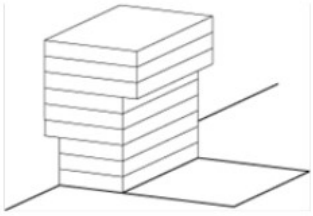
Elles peuvent également contenir des informations complémentaires concernant le fournisseur, la manière dont les données ont été acquises ou encore d'éventuelles limites de précision et recommandations d'utilisation par exemple.

VOLUME BATI	**Producteur** : Ville de Paris / Direction de l'Urbanisme Département de la Topographie et de la Documentation Foncière **Actualité** : avril 2017

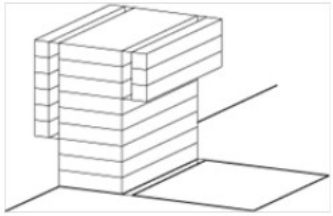
1 DÉFINITION

1.1 Définition de l'objet

Donnée vecteur qui décrit les bâtiments de manière détaillée en différenciant les bâtis en fonction de leur hauteur et des parties en saillie ou en retrait, définissant ainsi des volumes.



Remarques
Les volumes bâtis décrivent les bâtiments tels que représentés sur le plan parcellaire raster géré jusqu'en 2015 par le Service de la Topographie et de la Documentation Foncière (STDF). La couche vecteur des volumes bâtis décrit les parties en saillie sur la voie publique, alors qu'elles ne sont pas figurées sur le plan parcellaire raster.

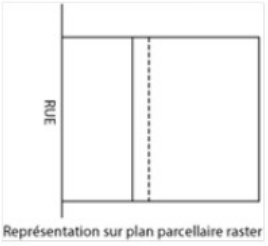


1.2 Étendue géographique

Paris.

1.3 Contraintes géographiques et topologiques

Un volume bâti est formé d'un seul polygone.
Cas des surfaces à trou : actuellement la couche ne gère pas les objets à trou.



Représentation sur plan parcellaire raster

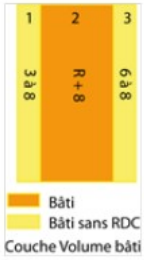
1.4 Identification et clés

Nom informatique de l'objet : VOLUME_BATI

Code court de l'objet : VB

Identification informatique : L'identifiant (N_SQ_VB) est un numéro séquentiel unique.

Clé sémantique : Pas de clé sémantique.



Couche Volume bâti

1.5 Changement d'identifiant et disparition d'objets

Changement de l'identifiant :

Conservation de l'identifiant : l'identifiant ne change pas lors de modifications géométriques des objets

Disparition de l'identifiant : l'identifiant disparait quand le bâtiment est démoli.

1.6 Limite d'usage et remarques

Néant

FIG. 4 :

En l'occurence, la première page nous renseigne de manière plus exhaustive sur la manière dont ont été tracés les différents polygones, à travers quelques schémas, ainsi qu'un paragraphe exprimant la source de ces tracés. Premièrement, ce document explique sa logique de séparer un bâtiment "réel" en plusieurs volumes fictifs, suivant s'ils sont en porte à faux ou non, permettant d'apporter une certaine précision.

Dès lors, les deux informations primordiales associées à chaque polygone est sa **hauteur**, ainsi que sa **plages de hauteur** s'il est en porte à faux. cette fiche indique également le contexte géographique, ainsi que les limitations géométriques (empêchants ici de représenter un polygone "évidé", obligeant à le sectionner si l'on veut représenter de manière correcte un patio par exemple).

2 CARACTÉRISTIQUES DESCRIPTIVES

2.1 Type de donnée

Les données Volumes bâtis sont des données surfaciques et descriptives

2.2 Définition des attributs et des liens

Attributs :

Nom	Libellé	Type	O	Valeurs possibles
N_SQ_VB	Identifiant séquentiel du volume bâti	N	O	
C_NAT_B	Code nature du volume bâti	C 1	O	V : Volume bâti simple U : Volume bâti avec surplomb
L_NAT_B	Nature du volume bâti	C 30		
C_SRC	Code nature de la source d'information	C 1		F : fiche parcellaire T : fiche parcellaire et terrain certifié C : fiche parcellaire et terrain non certifié
L_SRC	Nature de la source d'information	C 50		
M2	Surface graphique (m²)	N		
NB_PL	Nombre de planchers	N		
M2_PL_TOT	Surface totale de planchers (m²)	N		
H_ET_MAX	Hauteur max (nb étages/sol)	N		
C_PLAN_H_I	Classification du plan Hauteur d'étages	N		1 : Bâti à rez-de-chaussée 2 : Bâti de 1 à 3 étages 3 : Bâti de 4 à 8 étages 4 : Bâti de 9 à 12 étages 5 : Bâti de 13 étages et plus
L_PLAN_H_I	Libellé des classes du plan Hauteur d'étages	C 50		
L_PLAN_H	Description du plan Hauteur d'étages	C 10		Ex : R+2
B_RDC	Présence d'un RDC ? (1=vrai ; 0 = faux)	N		
L_B_U	Détail du volume avec surplomb	C 100		Ex : 1 à 5
X	Coord. X centre du polygone	N		
Y	Coord. Y centre du polygone	N		
N_AR	Numéro d'arrondissement (Bois séparés)	N		
N_QU	Numéro de quartier (Bois séparés)	N		
D_CRE	Date de constitution	D		
D_MAJ	Date de la dernière modification	D		Renseignée à la date de constitution de la donnée en l'absence de modification.

FIG. 5 :

Enfin, la seconde page contient les informations cruciales concernant les données à exploiter.

En effet, le tableau-ci-dessus renseigne sur le **libellé** de chaque variable (son contenu explicite), son **type** (ici, **C*n*** où *n* est un entier signifique que les valeurs possibles sont sous la forme d'une chaîne de caractères, contenant *n* caractères, tandis que **N** désigne simplement des données numériques), ainsi que ses valeurs possibles (servant à distinguer les variables **catégoriques**).

Dès lors, il est possible de repérer les deux variables les plus pertinentes si l'on souhaite extraire la hauteur des différents volumes. En l'occurence, ce seront les variables **H_ET_MAX** ainsi que **L_B_U** (pour les volumes en porte à faux), toutes deux exprimées en nombre d'étages. La surface de plancher totale **M2_PL_TOT** est également intéressante à extraire.

Ainsi, les métadonnées offrent les clés de compréhension nécessaires à l'utilisateur afin de comprendre le contenu d'un jeu de données en profondeur. Cette prise de connaissance permet désormais de manipuler les données en elle-mêmes, au sein d'un script en Python.

1.2 Le Python pour extraire et comprendre la structure des données

Afin d'exploiter ce jeu de données constitué d'objets géolocalisées et leurs données, le langage de programmation **Python** sera exclusivement employé. Comme mentionné dans l'introduction, ce dernier possède toutes les fonctionnalités nécessaires pour manipuler simplement ce type de données. La plateforme Open Data Paris permettant de définir un périmètre directement sur la carte, cette fonction sera utilisée afin de télécharger un échantillon du jeu de données (en l'occurence localisé autour de l'ENSAPVS), en premier lieu au format .CSV.

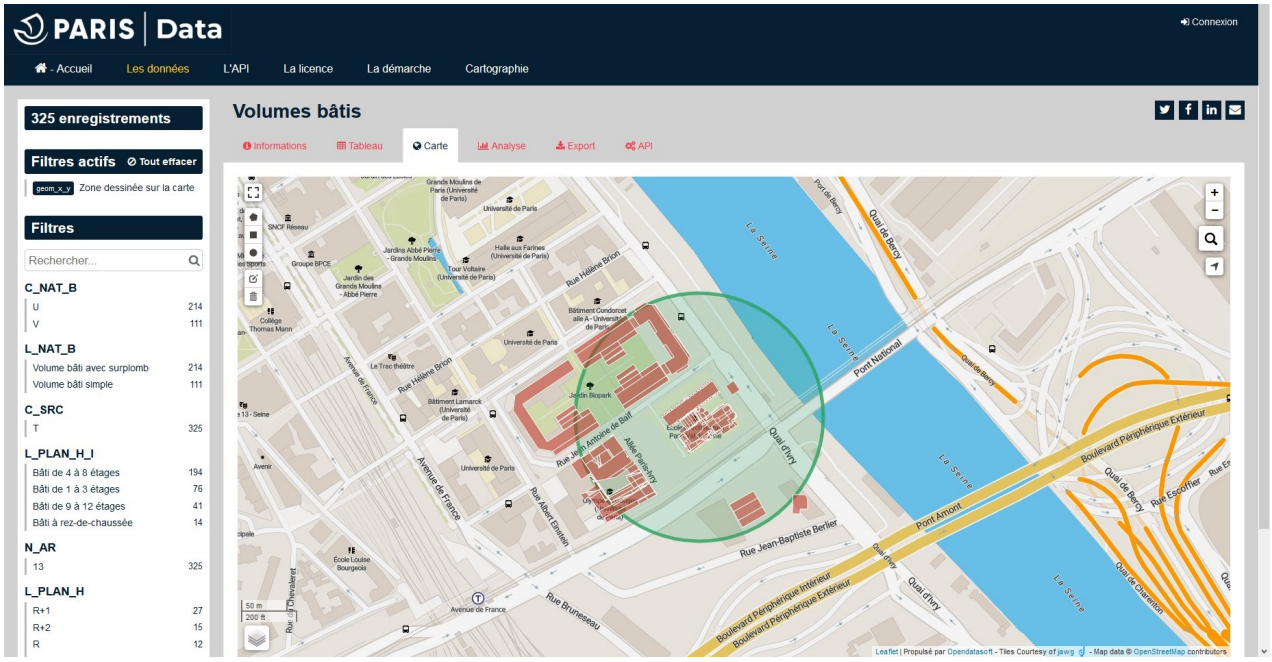


FIG. 6 :

1.2.1 Le format JSON : hiérarchiser pour dépasser les limites du tableur

Une fois téléchargé, le fichier .CSV peut être importé dans un script Python, grâce à la bibliothèque **Pandas**, spécialisée dans la manipulation de bases de données formatées sous forme de tableur. Afin de ne séléctionner que les variables les plus pertinentes, le paramètre *usecols* sera utilisé pour indiquer au code de ne garder que les colonnes **geom**,**H_ET_MAX**,**L_B_U** et **M2_PL_TOT**.

Enfin, la fonction `print(data.head(5))` nous permettra d'afficher seulement les 5 premiers enregistrements.

```
import pandas as pd
data = pd.read_csv("DONNEES/volumesbatisparis.csv",sep=";",usecols=["geom","H_ET_MAX","
    L_B_U","M2_PL_TOT"])
print(data.head(5))
```

```
##                                                 geom   M2_PL_TOT        L_B_U  H_ET_MAX
## 0  {"type": "Polygon", "coordinates": [[[2.384619...  281.214948          NaN       3.0
## 1  {"type": "Polygon", "coordinates": [[[2.384753...    9.842935          5a8       8.0
## 2  {"type": "Polygon", "coordinates": [[[2.384930...   52.982843  Ra1_et_3a9       9.0
## 3  {"type": "Polygon", "coordinates": [[[2.383544...  229.771164          2a8       8.0
```

```
## 4  {"type": "Polygon", "coordinates": [[[2.383038...  310.513295         3a9       9.0
```

Dès lors, une limitation directement liée au choix de format tableur apparaît.

En effet, comme montré lors de la première lecture, la colonne *geom* comporte en réalité plusieurs valeurs par case. Cela pose problème dans un format de type tableur, car chacune d'entre elle est reconnue comme **une seule chaîne de caractères** ("str"), comme le prouve la commande ci-dessous.

```
print(type(data["geom"][0]))
```

```
## <class 'str'>
```

Il est donc impossible d'en extraire des données en l'état. Heureusement, le langage Python possèdant une riche collection de bibliothèques internes, une bonne connaissance de ces dernières permet dans la majorité des cas de palier à ce genre de problèmes. Ainsi, grâce au code ci-dessous, les deux valeurs peuvent être extraites individuellement.

```
import ast
geom0 = ast.literal_eval(data["geom"][0])
print(geom0["type"])
```

```
## Polygon
```

```
print(geom0["coordinates"])
```

```
## [[[2.38461935101913, 48.82753330722759], [2.384628217027156, 48.827532989502515], [2.38
   46363958367522, 48.82753065095026], [2.384645732926233, 48.82752608251942], [2.38465232
   6040948, 48.82752179316423], [2.384656649382123, 48.827517191739766], [2.38466102315034
   42, 48.82751039520838], [2.38466170017297, 48.827501160346245], [2.3846595300456173, 48
   .827494714176034], [2.3846568869734233, 48.827489906925564], [2.384651922306437, 48.827
   48583150989], [2.384644105454289, 48.827480816864174], [2.384636766433263, 48.827477788
   690516], [2.384629481798523, 48.82747615392299], [2.384622830912725, 48.827475528812954
   ], [2.3846147562394, 48.82747539282936], [2.384604096304909, 48.82747686151168], [2.384
   597659243542, 48.827478431967464], [2.384591275760087, 48.827481117919454], [2.38457042
   9741702, 48.82747103121795], [2.384564292829435, 48.8274680613895], [2.384626028289865,
    48.82741133372302], [2.384605980177239, 48.8274017376928], [2.384599619398239, 48.8273
   98692965495], [2.384590133231389, 48.82740735276743], [2.384491575487986, 48.8274973259
   48926], [2.384502471066941, 48.827502550714016], [2.384490396493715, 48.82751330891048]
   , [2.384495045310945, 48.82751552280666], [2.38450719147041, 48.82750446998423], [2.384
   5183543743262, 48.82750978156396], [2.384548210954412, 48.827482266947136], [2.38455493
   5749008, 48.82748540078928], [2.384576016445481, 48.82749522428569], [2.384573845038873
   , 48.827504696086415], [2.384575044805671, 48.82751060663542], [2.384577439054112, 48.8
   2751606734975], [2.384581597086672, 48.82752085451733], [2.384586015415385, 48.82752447
   7438544], [2.384592041213091, 48.827527975524475], [2.384603039948636, 48.8275315018979
   94], [2.384612013974757, 48.82753299606446], [2.38461935101913, 48.82753330722759]]]
```

Bien que cela fonctionne, ce n'est pas une bonne pratique à adopter en règle général. En effet, il peut arriver qu'une des valeurs inscrites dans une valeur *"mère"* se décompose elle-même en un ensemble de valeurs. Il faudrait alors répéter la commande précédente autant de fois que nécessaire, tout en étant sûr à 100% que chaque enregistrement soit construit de manière identique sur l'ensemble du jeu de données.

Ainsi, il est plus judicieux de s'orienter vers un format capable de représenter cette hiérarchie dans sa structure. Le format JSON (JavaScript Object Notation) est un des formats d'échanges hiérarchisés les plus courant (au côté du XML). Ce dernier peut se décomposer en deux éléments primaires : le **dictionnaire**, un ensemble formé de couples **attribut : valeur** (aussi appelés *"objets"*), et la **liste**, une collection d'objets.[10]

```
# exemple de structure d'un script json
js = [ # début de la liste
        { # dictionnaire 1
          "attribut"  : "valeur", # couple attribut  : valeur
          "attribut2"  : [0,1,2,3,4] # une liste peut être une valeur
        },
        { # dictionnaire 2
          "attribut"  : { "attribut3"  : "valeur3" }, # un dictionnaire contenant lui même
              des objets peut être une valeur
          "attribut2"  : 99
        }
      ] # fin de la liste
```

Ainsi, ces deux constituantes suffisent à représenter des hiérarchies pouvant être complexes, tout en restant facile à lire et écrire pour les humains.

Or, la liste et le dictionnaire sont très courants au sein des langages de programmation modernes. Ainsi, l'import au sein de Python est extrêmement limpide. La syntaxe afin d'accéder à une valeur donnée s'effectue simplement en saisissant l'index des listes ou bien le nom des attributs en partant de la base de l'arborescence (de gauche à droite) entre crochets.

[10] *JSON* [en ligne]. [s. d.]. [Consulté le 3 février 2021]. Disponible à l'adresse : https://www.json.org/json-fr.html.

```
# 1er objet de la liste (dictionnaire 1), puis "attribut2"
print(js[0]["attribut2"])
```

```
## [0, 1, 2, 3, 4]
```

```
# 2e objet de la liste (dictionnaire 1), puis "attribut2" (dictionnaire 3), puis "
   attribut3"
print(js[1]["attribut"]["attribut3"])
```

```
## valeur3
```

La méthodologie est identique lors de l'import d'un fichier JSON externe, qui sera en l'occurence le même jeu de données téléchargé depuis le site Open Data Paris, mais au format .JSON.

```
import json
data = json.load(open("DONNEES/volumesbatisparis.json","r"))
# premier objet de la liste de volumes
print(data[1])
```

```
## {'datasetid' : 'volumesbatisparis', 'recordid' : 'c30d8245b2d55a552f4c03a4785bfa19c9d47c7
   3', 'fields' : {'objectid' : 536130, 'n_sq_pf' : 750031416, 'd_maj' : '2010-07-21T04 :00 :00+
   02 :00', 'l_src' : 'Fiche parcellaire et terrain certifié', 'l_b_u' : '5a8', 'h_et_max' : 8
   .0, 'b_rdc' : 0.0, 'n_ar' : 13, 'm2_pl_tot' : 9.8429349, 'd_cre' : '2010-07-21T04 :00 :00+02 :
   00', 'm2' : 2.4607337, 'l_plan_h_i' : 'Bâti de 4 à 8 étages', 'n_sq_qu' : 750000050, '
   geom_x_y' : [48.8272100647, 2.38474285824], 'l_nat_b' : 'Volume bâti avec surplomb', '
   n_qu' : 50.0, 'c_plan_h_i' : 3.0, 'shape_area' : 0.0, 'c_src' : 'T', 'nb_pl' : 4.0, 'n_sq_vb
   ' : 750169701, 'shape_len' : 0.0, 'c_nat_b' : 'U', 'geom' : {'type' : 'Polygon', '
   coordinates' : [[[2.384753219690729, 48.82722001182662], [2.384761632106771, 48.82721264
   787777], [2.384739182508318, 48.827203171721116], [2.3847246454617173, 48.8272024871575
   6], [2.384721921583155, 48.82720510470582], [2.384753219690729, 48.82722001182662]]]},
   'n_sq_ar' : 750000013, 'y' : 125193.0588305, 'x' : 603540.4980132}, 'geometry' : {'type' : '
   Point', 'coordinates' : [2.38474285824, 48.8272100647]}, 'record_timestamp' : '2020-12-24
   T11 :41 :01.963+01 :00'}
```

1.2.2 L'hétérogénéité : caractéristique intrinsèque aux données ouvertes

Cette lecture d'un des volumes du jeu de données au format JSON permet de repérer ses différents niveaux hiérarchiques, tout en observant les information supplémentaires dont on dispose alors.

Tout d'abord, les variables du jeu de données d'origine sont ici présentes sous l'attribut **"fields"**, et sera à référencer lors de l'extraction. En son sein, la variable **geom** est ici complètement représentée, contenant elle-même un dictionnaire avec ses deux valeurs (type et coordonnées). Nous pouvons également noter que les coordonnées sont constituées d'une **liste contenant des listes à deux valeurs**, représentant tout simplement

une collection de points (définis par x et y). Cependant, elle est elle même contenue dans une liste englobante superflue, que nous allons supprimer par la suite.

Ainsi, il est possible d'extraire les variables initialement souhaitées pour chacun des volumes avec une **boucle**, qui itèrera à travers la liste des volumes. Les résultats obtenus seront ajoutés dans un nouveau dictionnaire (avec des attributs nommés plus explicitement), lui même ajouté à une liste vide.

```
liste = [] # liste vide
for volume in data :
  liste.append({
    # Le [0] à la fin permet de supprimer la liste englobante superflue
    "coords"  : volume["fields"]["geom"]["coordinates"][0],
    "surface"  : volume["fields"]["m2_pl_tot"],
    "hauteur"  : volume["fields"]["h_et_max"],
    "hauteur_paf"  : volume["fields"]["l_b_u"]
  })
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords): KeyError: 'l_b_u'

##

## Detailed traceback:

##   File "<string>", line 7, in <module>
```

Cependant, nous obtenons l'erreur *KeyError : "l_b_u"*, signifiant que l'attribut "l_b_u" n'est pas défini pour certains objets. En effet, là où dans un tableur, un attribut non-défini se traduit par une valeur nulle ou une case vide, les structures hiérarchisées comme le JSON permettent à chaque objet de posséder certains attributs qui lui sont propres. Ainsi, les volumes n'étant pas en porte à faux ne possèdent pas cet attribut.

D'un point de vue plus général, cette hétérogénéité est très courante dans les jeux de données ouvertes. Au-delà d'être causée quand **certains enregistrements** possèdent **des attributs qui leurs sont propres** sans toutefois justifier la création d'un jeu de données supplémentaire (comme les volumes en porte à faux ici, difficilement dissiociables des autres), elle peut être tout simplement provoquée par **certaines valeurs manquantes** dans différents enregistrements (dans le cas d'une enquête aussi complexe et étendue que celle des hauteurs

des bâtiments de Paris, cela est compréhensible). Enfin, comme c'est le cas ici, cette hétérogénéité se révèle pendant l'extraction, car pas toujours mentionnée dans les métadonnées.

En conséquences, les fonctions de gestion d'erreur **try** et **except** seront utilisées afin de détécter l'absence de l'attribut **"l_b_u"**.

```
liste = [] # liste vide
for volume in data :
  try :
    liste.append({
      "coords"  : volume["fields"]["geom"]["coordinates"][0],
      "surface"  : volume["fields"]["m2_pl_tot"],
      "hauteur"  : volume["fields"]["h_et_max"],
      "hauteur_paf"  : volume["fields"]["l_b_u"]
    })
  except KeyError :
    liste.append({
      "coords"  : volume["fields"]["geom"]["coordinates"][0],
      "surface"  : volume["fields"]["m2_pl_tot"],
      "hauteur"  : volume["fields"]["h_et_max"]
    })

print(liste[1])
```

```
## {'coords' : [[2.384753219690729, 48.82722001182662], [2.384761632106771, 48.827212647877
   77], [2.384739182508318, 48.827203171721116], [2.3847246454617173, 48.82720248715756],
   [2.384721921583155, 48.82720510470582], [2.384753219690729, 48.82722001182662]], '
   surface' : 9.8429349, 'hauteur' : 8.0, 'hauteur_paf' : '5a8'}
```

```
print(liste[7])
```

```
## {'coords' : [[2.38351222440612, 48.826747420640146], [2.383527749099538, 48.826732649332
   93], [2.383165045737739, 48.82659869275863], [2.383149529111281, 48.82661346765992], [2
   .38351222440612, 48.826747420640146]], 'surface' : 121.4416013, 'hauteur' : 1.0}
```

Ainsi, le langage Python a permis de mettre en lumière à la fois l'importance de la hiérarchisation dans un jeu de données complexe, dont une certaine hétérogénéité persiste (sans être forcément repérable en amont de la manipulation) réclamant une souplesse de la part des outils d'extraction à cet égard.

1.3 Aperçu de la souplesse des fonctions de manipulation de données

Après la phase d'extraction précédente, il est nécessaire que les données etraites en l'état soient exploitables pour la suite. En l'occurence, il serait souhaitable d'exprimer les hauteurs en **mètres** plutôt qu'en nombre de niveaux.

```
h_etage = 3
print(liste[0]["hauteur"] * h_etage)
```

```
## 9.0
```

Cette opération est triviale pour l'attribut **"hauteur"**, étant exprimé numériquement. Il suffit alors de définir une **hauteur d'étage type** et d'effectuer une multiplication.

Cependant, l'attribut **"hauteur_paf"** étant exprimé sous la forme d'une chaîne de caractères (texte décrivant les plages de hauteur), il est nécessaire de convertir cette notation numériquement.

1.3.1 Approche compréhensive des différentes notations grâce à Python

Heureusement, Python est capable de reconnaître des morceaux de texte au sein de chaînes de caractères, permettant ainsi de les remplacer par leur équivalent numérique.

```
if "da" in "data" :
  print(1)
else :
  print(0)
```

```
## 1
```

Dès lors, afin de comprendre les différentes manières d'exprimer les hauteurs en porte à faux, il est possible de les énumérer par **longueur**.

```
notations = {}
for volume in liste :
  try :
    notations[len(volume["hauteur_paf"])] = volume["hauteur_paf"]
  except KeyError :
    pass

print(notations.values())
```

```
## dict_values(['2a4', '2a4_et_7a8', 'encorbt_au_5', 'Ra1_et_encorbt_au_5', 'R_et_3a9', '
   Auvent_n1', '4a11', 'R_et_encorbt_au_3', 'Ra1_et_3_et_5a8'])
```

Dès lors, certains éléments constituants peuvent être notés :

- Une plage de hauteur est principalement renseignée par ses **deux niveaux de hauteur** séparées par un *a*
- et est utilisé pour renseigner **plusieurs plages de hauteur**.
- *"encorbt_au_N"* désigne une plage de hauteur du niveau **N** au niveau maximal (exprimé par la variable "hauteur"). S'ils désignent le même niveau, la plage sera du niveau $\mathbf{N^{-1}}$ au niveau maximal.
- *auvent* désigne une plage du niveau **N** à $\mathbf{N^{+1}}$.
- Enfin, la présence de *R* exprimé seul suggère que **R** désigne une plage d'une **seule hauteur d'étage partant du sol**, tandis que **Ra1** désigne **deux hauteurs d'étages partant du sol**.

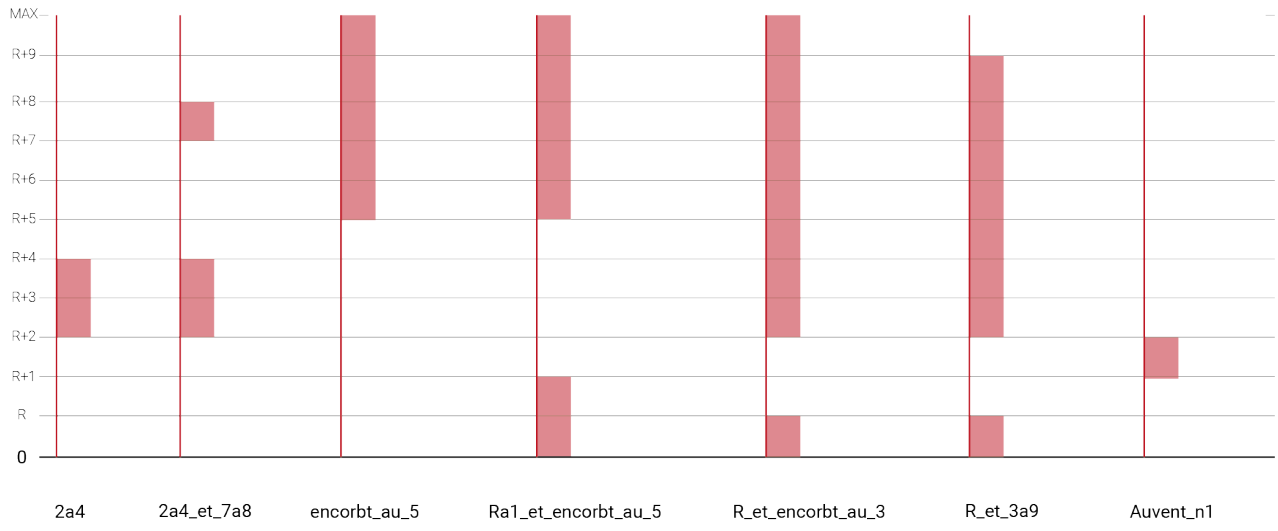


FIG. 7 :

Cette dernière observation est cruciale : si l'on fixe **R = 0**, alors la plage **Ra1** devient **0a1**. Or, si l'on multiplie ces deux bornes par la hauteur d'étage type de 3 mètres, le volume aura une plage de hauteur de **0 à 3m**, tandis qu'il faudrait obtenir **0 à 6m**. Dès lors, il faut **ajouter 1 à tout nombre de niveau sauf R** avant multiplication par la hauteur d'étage type.

1.3.2 De la chaîne de caractère à la valeur numérique

A la lumière de toutes ces subtilités, il est désormais possible de créer une fonction capable de **transformer des chaînes de caractères** en **valeurs numériques**. Le code ci-dessous illustre la transformation d'une notation **N^1aN^2**. Après s'être assuré que la notation contient bien un *a*, le code sépare les deux bornes à cet endroit, puis les convertit en **entiers**. Enfin, chaque borne est incrémentée de **1** avant multiplication par la hauteur d'étage type, sauf *R* qui prend la valeur 0.

```
# Rappel  : h_etage = 3 mètres
h_paf =  "Ra8"
if "a" in h_paf :
  hauteur = h_paf.split("a")
  hauteur = [(int(h)+1)*h_etage if h  != "R" else 0 for h in hauteur]

print(hauteur)
```

```
## [0, 27]
```

Suivant ce principe, le bloc de code suivant opère ce type de transformation suivant les différents notations relevées précédemment. L'incrément de **1** sera également appliqué à l'attribut **hauteur**

```
# Rappel  : h_etage = 3 mètres
for volume in liste :
  intervalles = []
  try :
    intervs = volume["hauteur_paf"].split("_et_")
    for interv in intervs :
        if "encorbt_au_" in interv :
          n = int(interv.strip("encorbt_au_"))
          if n == volume["hauteur"] :
            intervalles.append([n*h_etage,(n+1)*h_etage])
          else :
            intervalles.append([(n+1)*h_etage,(volume["hauteur"]+1)*h_etage])
        elif "a" in interv :
            if "R" in interv :
                intervalles.append([0,(int(interv.strip("Ra"))+1)*h_etage])
            else :
                intervalles.append([(int(h)+1)*h_etage for h in interv.split("a")])
        elif interv == "R" :
            intervalles.append([0,h_etage])
    volume["hauteur_paf"] = intervalles
  except KeyError :
    pass

  volume["hauteur"] = (volume["hauteur"]+1) * h_etage

print(liste[1])
```

```
## {'coords' : [[2.384753219690729, 48.82722001182662], [2.384761632106771, 48.827212647877
   77], [2.384739182508318, 48.827203171721116], [2.3847246454617173, 48.82720248715756],
   [2.384721921583155, 48.82720510470582], [2.384753219690729, 48.82722001182662]], '
   surface' : 9.8429349, 'hauteur' : 27.0, 'hauteur_paf' : [[18, 27]]}
```

Enfin, ce jeu de données apurées peut être sauvegardé au format JSON, afin de pouvoir le réutiliser facilement dans le chapitre suivant.

```
json.dump(liste,open("DONNEES/liste_apuree.json","w"))
```

Comme démontré au cours de ce chapitre, le langage de programmation Python possède une souplesse lui permettant d'extraire et de manipuler facilement les formats de données courants en Open Data, et de palier sans réelle difficulté aux éventuelles subtilités présentes dans des jeux de données hétérogènes, le tout bénéficiant d'une syntaxe claire tout au long du code. **A la fois outil de compréhension et de traitement, il se révèle extrêmement précieux lorsque l'on souhaite exploiter des jeux de données en Open Data.**

Dans le contexte de la profession architecturale, l'obtention de ces données n'a cependant que peu de valeur si l'architecte ne peut l'intégrer dans son environnement de travail, ce à quoi le prochain chapitre est dédié.

2 Synthétiser et intégrer les données ouvertes au sein du « workflow » de l'architecte avec Python : du graphique au modèle 3D

La production de documents synthétiques, en particulier les éléments graphiques faisant partie intégrante du « workflow » de l'architecte, il est primordial de s'y intéresser au sein de ce mémoire.

En effet, c'est via ce type de document que l'architecte est capable de non seulement communiquer sa production ou encore sa démarche de conception, mais également de se documenter au cours de sa démarche.

Or, il s'avère que le langage Python regorge de bibliothèques spécialisées dans la datavisualisation tel que *Matplotlib*[11] ou encore *Seaborn*,[12] comme le montre la plateforme *The Python Graph Gallery*[13] offrant énormément de possibilités de représentation, du graphique statistique au modèle 3D.

Ce chapitre présentera ainsi plusieurs variantes d'exploitation du script obtenu à la fin du chapitre précédent dans le but d'obtenir des documents synthétiques à partir des données des bâtiments obtenues. Elles seront respectivement consacrées à l'élaboration de graphiques statistiques, puis d'une carte interactive, d'un dessin vectorisé avec gestion des calques, puis aboutir à un modèle 3D du contexte bâti.

[11] *Matplotlib : Python plotting* [en ligne]. [s. d.]. [Consulté le 16 janvier 2021]. Disponible à l'adresse : https://matplotlib.org/.

[12] *Seaborn : statistical data visualization* [en ligne]. [s. d.]. [Consulté le 28 janvier 2021]. Disponible à l'adresse : https://seaborn.pydata.org/.

[13] *The Python Graph Gallery : Visualizing data with Python* [en ligne]. [s. d.]. [Consulté le 28 janvier 2021]. Disponible à l'adresse : https://python-graph-gallery.com/.

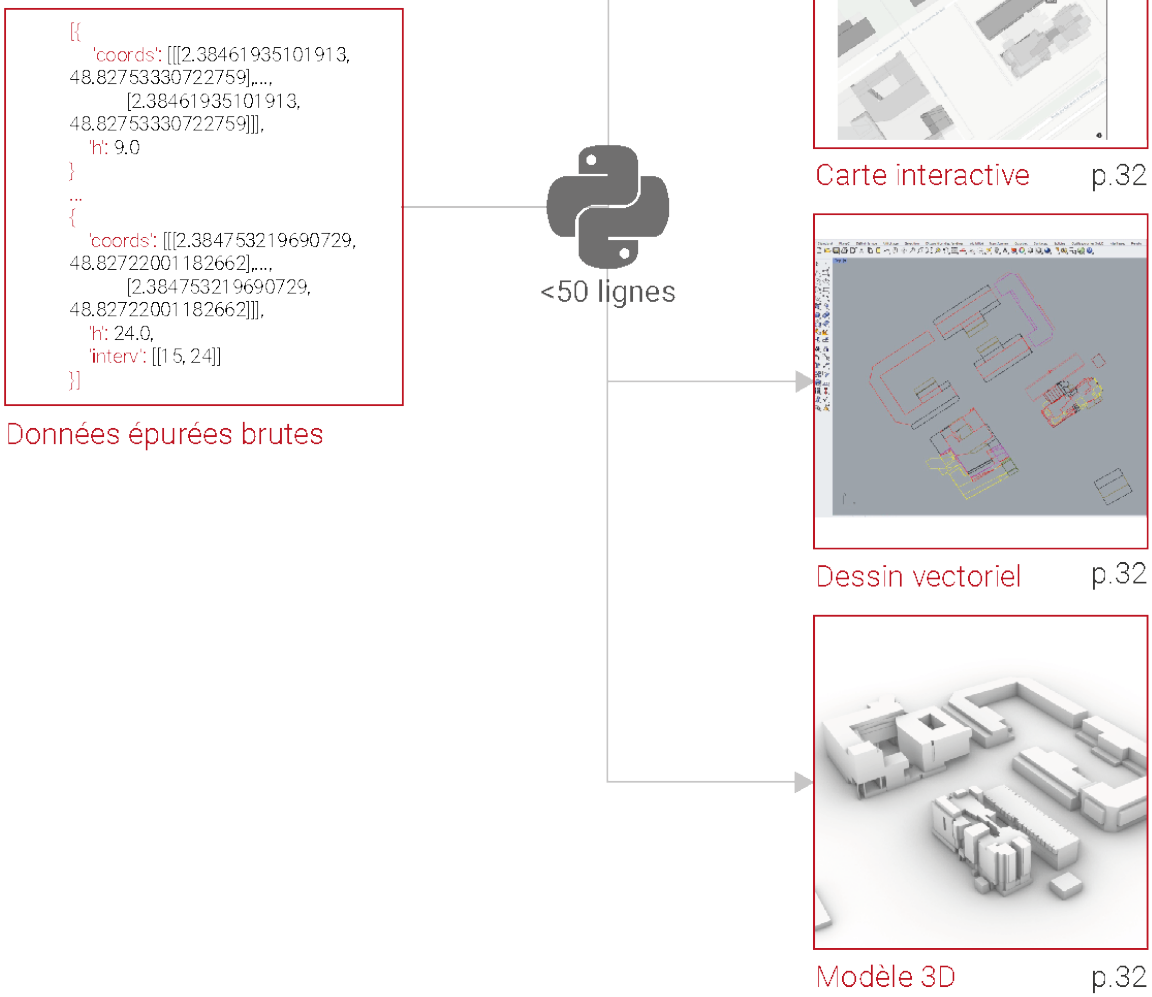

[{
'coords': [[[2.38461935101913,
48.82753330722759],...,
[2.38461935101913,
48.82753330722759]]],
'h': 9.0
}
...
{
'coords': [[[2.384753219690729,
48.82722001182662],...,
[2.384753219690729,
48.82722001182662]]],
'h': 24.0,
'interv': [[15, 24]]
}]
Données épurées brutes
<50 lignes
Répartition des hauteurs
Graphiques p.32
Carte interactive p.32
Dessin vectoriel p.32
Modèle 3D p.32

FIG. 8 :

2.1 Production de documents synthétiques interactifs

Cette section présentera deux méthodes afin de visualiser les données extraites. Au sein de cette section, la bibliothèque *Plotly* sera employée, possédant davantage d'options graphiques ainsi que de possibilités d'export (dont des options d'interactivité) que son concurrent plus répandu *Matplotlib* mentionné précédemment.

2.1.1 Visualisation statistique des données

Lorsque l'on aborde la question de la synthèse de données quelles qu'elles soient, la représentation statistique par des graphiques semble représenter l'approche la plus intuitive et la plus directe.

```
import json
import plotly.graph_objects as go
volumes = json.load(open("DONNEES/liste_apuree.json","r"))
```

Le premier graphique créé sera un histogramme de **répartition du nombre de volumes par hauteur**. Le code ci-dessus permet d'importer la bibliothèque *Plotly* (par l'intermédiaire d'un de ses « sous-module » nommé « graph_objects », que nous appellerons ici avec « go » tout au long du script), et charge également la liste des volumes enregistrée à la fin du chapitre 1.

```
n_par_hauteur = {}
for volume in volumes :
  try :
    n_par_hauteur[volume["hauteur"]] += 1
  except KeyError :
    n_par_hauteur[volume["hauteur"]] = 1

print(n_par_hauteur)
```

```
## {12.0 : 22, 27.0 : 66, 30.0 : 37, 21.0 : 17, 6.0 : 30, 24.0 : 28, 15.0 : 19, 18.0 : 68, 9.0 : 24
   , 3.0 : 14, 33.0 : 5, 36.0 : 5}
```

Cette liste des volumes est ensuite analysée dans le code ci-dessus à travers une boucle. Le but est en effet d'en récupérer le **nombre de volumes par hauteur**. La fonction **try**/**except** permet d'inclure une nouvelle hauteur dans le dictionnaire **n_par_hauteur** si elle n'existe pas encore. Si elle existe déjà, sa valeur est incrémentée.

Enfin, quelques lignes de codes permettent à la fois la construction d'un graphique, ainsi que son export. La **liste des différentes hauteurs** (soit celle des *attributs* du dictionnaire **n_par_hauteur**) représentera l'axe *x*, tandis que les différents **nombres de volumes** associés formeront les valeurs à renseigner pour l'axe *y*. Quelques paramètres graphiques servent à définir un titre général, des libellés pour les deux axes ainsi qu'une résolution d'export. Ici, deux exports possibles seront montrés, à savoir un export **statique** sous forme d'image au format PNG, ainsi qu'un export **interactif** au sein d'une page web au format HTML.

```
# traçage du graphique
fig = go.Figure([go.Bar(x=list(n_par_hauteur.keys()), y=list(n_par_hauteur.values()),
    opacity=0.8,marker_color='rgb(200,0,0)')])
fig.update_xaxes(categoryorder='category ascending',tickvals=sorted(list(n_par_hauteur.
    keys())))
```

```
fig.update_layout(title="Répartition des hauteurs",xaxis_title="hauteur (m)",yaxis_title="
    nombre de volumes",width=600,height=600)
```

```
fig.write_image("OUTPUT/graphique_hauteurs.png")
fig.write_html("OUTPUT/graphique_hauteurs.html")
```

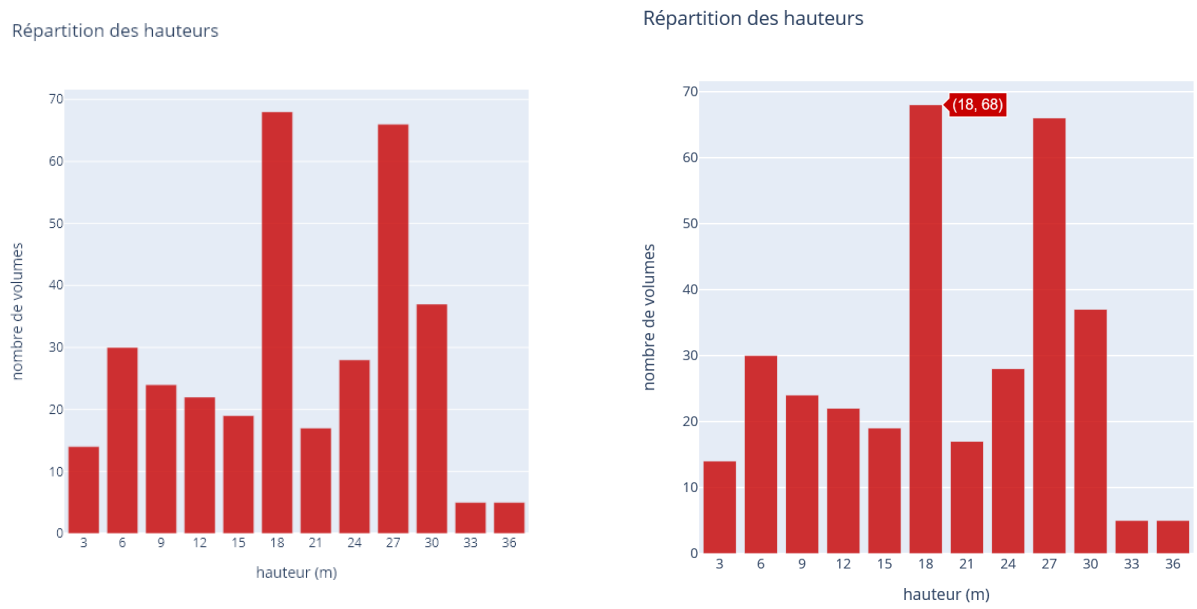


FIG. 9 :

Il est également possible de créer de la même manière une multitude d'autres graphiques que *Plotly* permet de construire. Un second graphique plus complet peut être construit, en s'intéressant cette fois-ci à une répartition **des volumes suivant leur hauteur, leur surface et s'ils sont en porte à faux**. Ici, le sous-module *express* de *Plotly*

sera employé avec *Pandas*, permettant une mise en forme plus condensée et une meilleur gestion des données. Il sera possible de réutiliser directement les éléments de la liste des volumes. Cependant, un attribut devra être ajouté à chaque volume, spécifiant s'il est en porte à faux ou pas, afin de pouvoir être traité dans *Plotly*.

```
import plotly.express as px
import pandas

for volume in volumes :
  try :
    _ = volume["hauteur_paf"]
    volume["is_paf"] = 1
  except KeyError :
    volume["is_paf"] = 0

df = pandas.DataFrame(volumes).drop(columns=["coords","hauteur_paf"])
fig2 = px.parallel_coordinates(df, color="hauteur",color_continuous_scale=px.colors.
   sequential.amp)
fig2.update_layout(font={"size" :20},width=1800,height=800)
```

```
fig2.write_image("OUTPUT/graphique_repart.png")
fig2.write_html("OUTPUT/graphique_repart.html")
```

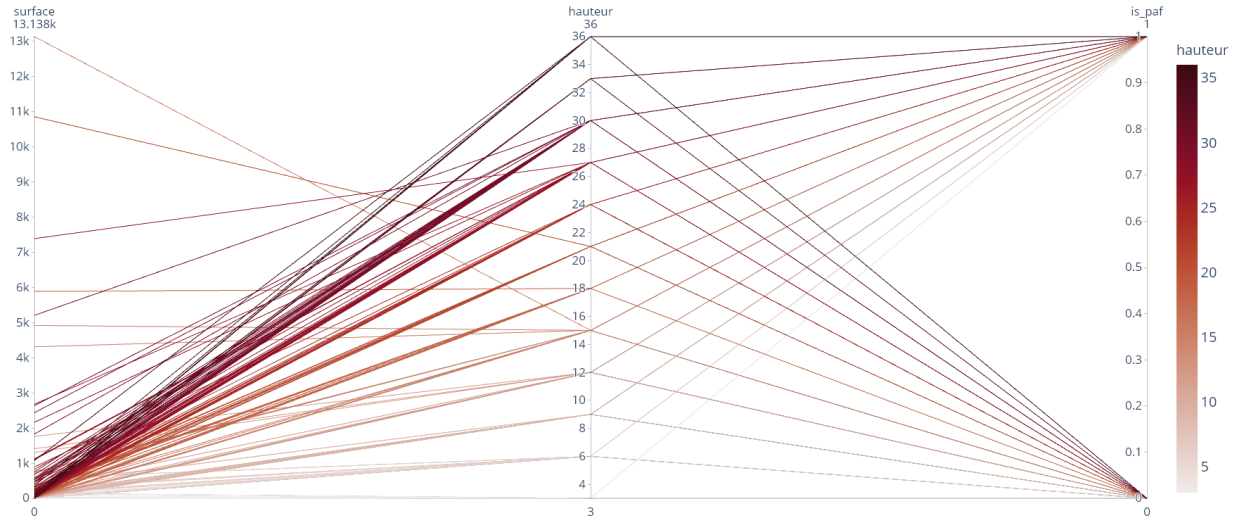


FIG. 10 :

En l'occurrence, le jeu de données extrait contient davantage de petites surfaces ainsi qu'un nombre équlibré de volumes en porte à faux. Cependant, cette souplesse dans la création de représentations graphiques que permet *Plotly* rend également possible de dessiner les **emprises des volumes en eux-mêmes**, qui sera également augmentée avec de l'interactivité.

2.1.2 Cartographier de manière interactive

Cette sous-section a pour but de présenter une fonctionnalité particulièrement utile pour la profession architecturale. En effet, *Plotly* est capable de créer de représenter des **formes géométriques sur un fond de carte** (d'une manière similaire aux solutions de SIG sans avoir besoin de convertir les coordonnées en latitude/longitude), avec laquelle il est possible d'**interagir**, à travers notamment des fonctionnalités telles que le **zoom**, l'**affichage/masquage** d'élements légendés ainsi que l'affichage de caractéristiques au **survol avec le curseur**.

Ainsi, l'objectif sera ici de générer une **carte interactive des volumes par hauteur**.

```
import json
import plotly.graph_objects as go
volumes = json.load(open("DONNEES/liste_apuree.json","r"))
```

Après avoir chargé le jeu de données de base, la première étape est de **grouper les volumes par hauteur**. Ceci sera effectué comme dans la section précédente, à l'exception que les **volumes ainsi que leurs caractéristiques** seront ajoutés à une liste par hauteur au lieu d'être simplement comptés. Le tout sera trié selon la hauteur par **ordre croissant**.

```
volumes_par_hauteur = {}
for volume in volumes :
  try :
    volumes_par_hauteur[volume["hauteur"]].append(volume)
  except KeyError :
    volumes_par_hauteur[volume["hauteur"]] = [volume]

volumes_par_hauteur = dict(sorted(volumes_par_hauteur.items()))
```

Ensuite, une **boucle** permet d'itérer à travers chaque **hauteur et ses volumes** afin de les tracer. Une **couleur** sera préalablement attribuée pour chacune d'entre elles, calculées selon une échelle de gris en RGB (plus le volume est **haut**, plus il sera **clair**). La seule subtilité ici est de devoir séparer **les listes de coordonnées latitude/longitude** en **deux listes séparées**, et ainsi avoir une liste pour les valeurs de **latitude** et une autre pour les valeurs de **longitude**, contenant des valeurs **nulles** pour séparer les volumes lors du traçage.

```
traces = []
for h,volumes in volumes_par_hauteur.items() :
  couleur = "rgb(" + ",".join([str(h/max(volumes_par_hauteur.keys())*255)]*3) +")"
  X = []
```

```
  Y = []
  for vol in volumes :
    for coords in vol["coords"] :
      X.append(coords[0]) # longitude
      Y.append(coords[1]) # latitude
    X.append(None) #Séparations entre chaque volume
    Y.append(None)
    # Traçage des volumes
  traces.append(go.Scattermapbox(
    name=str(h) +"m",
    mode="lines",
    line = {"width"  : 0.5, "color"  : couleur},
    lon=X,
    lat=Y,
    opacity=1.0,
    hoverinfo="name",
    fill="toself"))
```

Enfin, la carte complète (avec chaque couche de volumes pour chaque hauteur) est créée à partir de la liste *traces*, en configurant le titre, la légende ainsi que le style de fond de carte. Le tout est sauvegardé au format .HTML, permettant de l'ouvrir dans un navigateur afin de permettre l'interactivité et la **navigation libre** dans le fond de carte.

```
fig = go.Figure(traces)
fig.update_layout(title="Hauteurs des volumes bâtis",legend_title="Hauteur", autosize=True
    ,
    mapbox = {'style' :"carto-positron",'center' : {'lon' : 2.3848515, 'lat' : 48.8272092}, "
        zoom"  : 16.6})
# Export sous forme d'une page web interactive
```

```
fig.write_html("OUTPUT/carte_des_hauteurs.html")
```

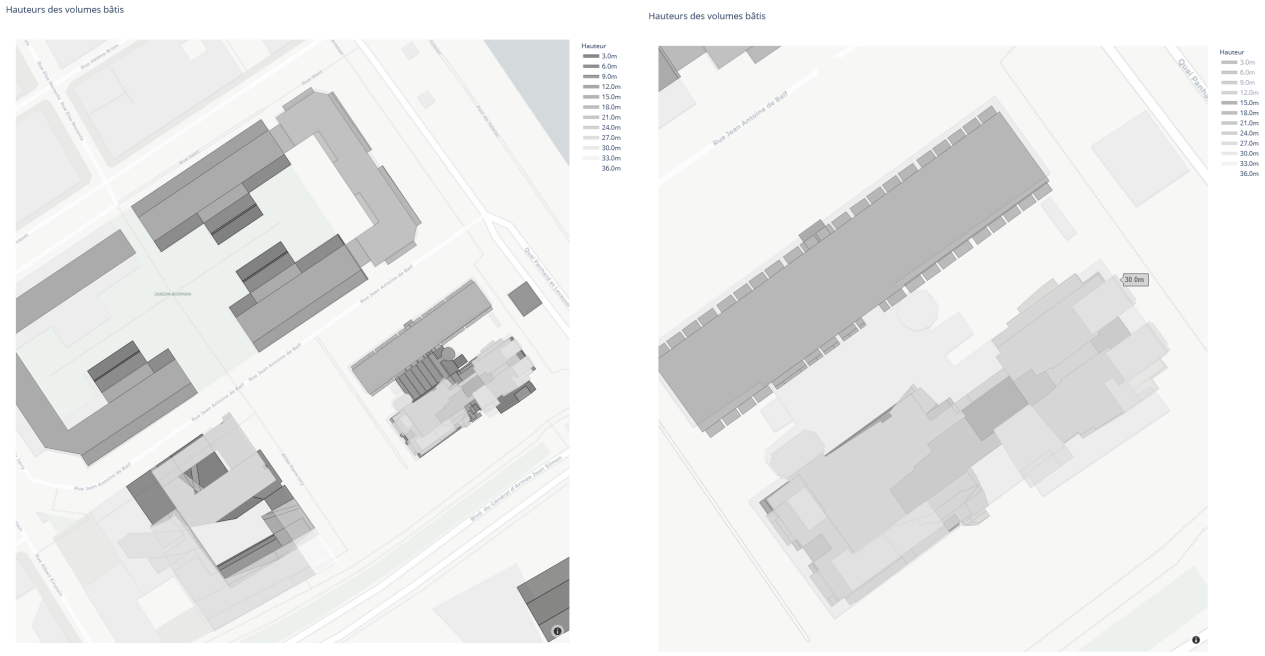


FIG. 11 :

Dès lors, une telle capacité à cartographier avec souplesse des données brutes présente un intérêt certain au sein de la pratique architecturale.

Ainsi, ces aperçus attestent de la capacité du langage Python à produire facilement de multiples représentations graphiques synthétiques à partir de données brutes, du graphique statistique aux cartes interactives, renforcant ainsi la pertinence de son utilisation dans le cadre de l'exploitation de données issues de l'Open Data pour les architectes.

2.2 Génération automatique de documents techniques.

Les capacités de synthèse graphique et de cartographie des données montrées précédemment sont certes intéressantes, mais le domaine de la conception architecturale est surtout concerné par la production de dessins, modèles 3D et autres représentations techniques à l'échelle dans le cadre d'un projet. Cette section permettra de répondre à cet enjeu grâce à Python.

2.2.1 Fichier CAD vectorisé et hiérarchisé

Le premier aperçu livré dans cette section, sera de **produire un document vectorisé au format .DXF** (format d'échange de dessin vectorisé similaire au .DWG, répandu en CAO), où seront tracés les **différents volumes sous forme de polylignes**. Chacun d'entre eux sera également classé dans un **calque correspondant à sa hauteur**, avec pour chaque calque une couleur différente. Pour ce faire, le module **ezdxf** sera employé. Ce dernier permet la plupart des fonctions de dessin vectoriel que propose d'autres outils de CAD tels qu'*AutoCAD*, dont la création de calques entre autres.

Un second module nommé **utm** sera également utilisé pour manipuler les coordonnées géographiques. En effet, il est ici indispensable de convertir les **coordonnées géodésiques** exprimées en *degrés de latitude/longitude* en **coordonnées cartésiennes** exprimées en unités de grandeur terrestres. La notation UTM (*Universal Transverse Mercator*) étant exprimée en **mètres** et dont le calcul prend en compte la rotondité de la Terre, elle se révèle donc essentielle pour pouvoir dessiner dans un repère. (Une petite subtilité sera d'inverser l'ordre des degrés de latitude/longitude dans le code, afin que l'axe *x* corresponde à la **longitude**, et l'axe *y* à la **latitude**).

```
import json
import ezdxf
import utm
volumes = json.load(open("DONNEES/liste_apuree.json","r"))
```

Après l'import des modules nécessaires et le chargement du jeu de données de base, un nouveau dessin est initialisé. Ensuite, **un calque par hauteur** sera créé en amont des tracés des volumes, de sorte à ce qu'ils apparaîssent dans un ordre croissant au sein du futur fichier. Tout comme dans la carte interactive présentée dans la session 2.1.2, une **teinte de couleur proportionnelle à chaque hauteur** sera créée (en l'occurence, sur du rouge).

```
# initialisation d'un nouveau dessin
doc = ezdxf.new(dxfversion='R2010')
msp = doc.modelspace()
# boucle sur une ligne pour récupérer les hauteurs
hauteurs = [volume["hauteur"] for volume in volumes]
# permet de supprimer les doublons et de trier par ordre croissant
hauteurs = sorted(list(set(hauteurs)))
for h in hauteurs :
```

```
  calque = doc.layers.new(str(h) + "m")
  calque.rgb = (255*(h/hauteurs[-1]),0,0)
```

Ceci effectué, une boucle itèrera à travers chaque volume. Pour chacun d'entre eux, leurs **coordonnées seront préalablement converties** des degrés de latitude/longitude en mètres grâce au module *utm*. Une **polyligne** peut ainsi être tracée à partir des points aux coordonnées converties pour représenter l'emprise de chaque volume, puis affectée au calque correspondant à la hauteur du volume. Enfin, le document est exporté au format .DXF.

```
for volume in volumes :
    coords_cart = []
    # Conversion des coordonnées géodésiques (lat/lon) en cartésiennes (x/y)
    for coords in volume["coords"] :
        c = utm.from_latlon(coords[1],coords[0])
        coords_cart.append((c[0],c[1]))

    calque = str(volume["hauteur"]) + "m"
    msp.add_polyline2d(coords_cart, dxfattribs={'layer' : calque})
```

```
doc.saveas("OUTPUT/plan_bati.dxf")
```

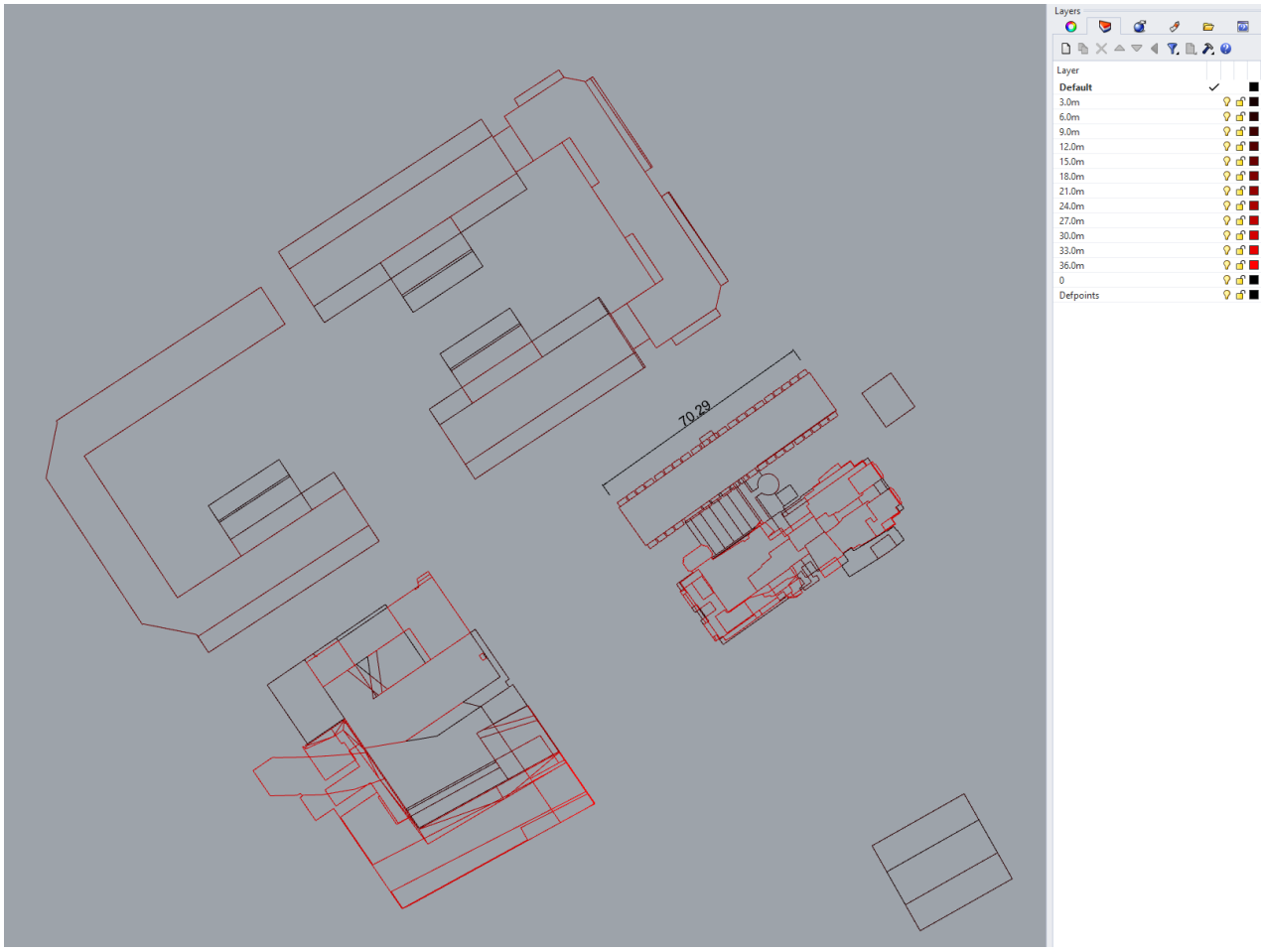


FIG. 12 :

Le document fourni en sortie peut ensuite être importé dans n'importe quel logiciel supportant le format DXF, ce qui est le cas pour la majorité des outils de dessin vectoriel des agences d'architecture. Ce document est d'autant plus exploitable qu'il reste **géoréférencé** (les volumes conservent leurs coordonnées UTM dans le dessin), et **à l'échelle** (puisque exprimés en mètres).

De plus, les options de **personnalisation** des calques permettent d'organiser les données dont on dispose en amont d'une exploitation "manuelle".

Ainsi, le langage Python prouve également son efficacité lorsqu'il est question de générer des documents vectoriels graphiques, pleinement exploitables comme support de travail.

2.2.2 Construction d'un modèle 3D

Cette sous-section aura pour but de clore le chapitre en exploitant à son plein potentiel le jeu de donné extrait. En effet, puisque l'on dispose à la fois d'**emprises géométriques traçables** ainsi que de **hauteurs**, construire un **modèle 3D** automatiquement se présente comme un aboutissement certain, d'autant plus qu'il représente tout naturellement une forte utilité pour l'architecte.

Bien que Python possède des bibliothèques permettant de manipuler de la 3D telles que **PyOpenGL**[14] ou **Open3D**,[15] ces dernières sont plutôt orientées pour effectuer des rendu en images de synthèses ou de la reconstruction sur des maillages. Or, ces fonctionnalités sont assez éloignées de l'usage souhaité, où l'on cherche plutôt à **construire un modèle 3D** à partir de données brutes puis les exporter dans des formats possédant des **capacités d'organisation du modèle** (comme des calques par exemple) et en évitant le **maillage**, afin de les rendre compatibles avec l'usage qu'en feraient un architecte. Ce n'est pas le cas ici, les bibliothèques citées privilégiant les formats d'échange maillés comme le format OBJ.

De plus, même si ce retard a tendance à diminuer avec leur évolution, la quasi-totalité des bibliothèques orientées 3D pour Python sont basées sur des versions codées dans des langages de programmation plus complexes

[14] *PyOpenGL – The Python OpenGL Binding* [en ligne]. 24 janvier 2021. Disponible à l'adresse : http://pyopengl.sourceforge.net/.
[15] ZHOU, Qian-Yi, PARK, Jaesik et KOLTUN, Vladlen. Open3D : A Modern Library for 3D Data Processing. *arXiv :1801.09847*. 2018.

comme le *C++* pour des raisons de performances, ce qui fait qu'il persiste toujours un "retard" entre le moteur natif et son intégration destinée à Python, comme le présente plus en détail l'ouvrage *Python & OpenGL for Scientific Visualization* (P.ROUGIER,2018).[16]

C'est pourquoi la bibliothèque **RhinoInside**[17] sera ici utilisée. C'est une solution logicielle Open Source développée sous l'initiative de la société développant Rhinoceros (McNeel) associée au modeleur 3D Rhinoceros depuis la version 7. Elle a pour but de **connecter Rhinoceros à un autre programme**, afin de pouvoir transférer de manière directe des données entre ces deux éléments.

La version Python de cet outil permet de connecter une instance de Rhinoceros sans interface graphique (mais permettant tout de même toute opération possible manuellement dans le logiciel) à un script, en important un module nommé «rhinoinside». Ainsi, il sera possible directement dans un même script de créer un fichier au format natif de Rhinoceros (.3DM), puis y dessiner et extruder des tracés pour enfin le sauvegarder sur son disque dur.

```
import json
import utm
volumes = json.load(open("DONNEES/liste_apuree.json","r"))
# chargement d'une instance de RhinoInside
import rhinoinside
from pathlib import Path
rhino_path = Path("C :/Program Files/Rhino 7/System")
rhinocore_path = Path("C :/Program Files/Rhino 7/System/RhinoCore.dll")
rhinoinside.load(str(rhino_path))
import System
import Rhino
```

La séquence d'import est un peu plus conséquente, puisqu'il faut référencer le chemin d'installation de Rhino 7. Le module *utm* sera également utilisé ici.

```
# Création du document
DOC = Rhino.RhinoDoc.Create("")
DOC.ModelUnitSystem = Rhino.UnitSystem.Meters
# Création d'un calque principal
calque_bati =  Rhino.DocObjects.Layer()
calque_bati.Color = System.Drawing.Color.FromArgb(255,0,0,0)
calque_bati.Name = 'batiments'
DOC.Layers.Add(calque_bati)
```

[16] P.ROUGIER, Nicolas. *Python & OpenGL for Scientific Visualization*. Bordeaux : [s. n.], 2018. Disponible à l'adresse : https://www.labri.fr/perso/nrougier/python-opengl/.

[17] *Rhino - Rhino.Inside* [en ligne]. 25 janvier 2021. Disponible à l'adresse : https://www.rhino3d.com/features/rhino-inside/.

```
# Définition de ce calque comme actuel

c_actuel = DOC.Layers.FindByFullPath("calque_bati",-1)
DOC.Layers.SetCurrentLayerIndex(c_actuel,False)
```

Une première étape consiste à créer une **nouvelle instance d'un document rhino**, spécifier ses unités (ici, en accord avec les unités des coordonnées UTM, soit le *mètre*), ainsi que créer un calque dans lequel seront dessinés les volumes. Ce dernier se verra attribuer un **nom** et une **couleur**.

```
for volume in volumes :
  pts = System.Collections.Generic.List[Rhino.Geometry.Point3d]()
  # Conversion des coordonnées géodésiques (lat/lon) en cartésiennes (x/y)
  for coords in volume["coords"] :
    c = utm.from_latlon(coords[1],coords[0])
    pts.Add(Rhino.Geometry.Point3d(c[0],c[1],0.0))

  poly = Rhino.Geometry.Polyline(pts)
  try :
    # Si le volume est en porte à faux
    for intervalle in volume['hauteur_paf'] :
      poly.SetAllZ(float(intervalle[0]))
      polyz = poly.ToPolylineCurve()
      h_cible = float(intervalle[1])-float(intervalle[0])
      extr = Rhino.Geometry.Extrusion.Create(polyz,-1*h_cible,True)
      DOC.Objects.AddExtrusion(extr)
  except KeyError :
    # Si le volume repose au rdc
    polyz = poly.ToPolylineCurve()
    extr = Rhino.Geometry.Extrusion.Create(polyz,-1*float(volume["hauteur"]),True)
    DOC.Objects.AddExtrusion(extr)
```

```
success = DOC.SaveAs('OUTPUT/modele.3dm')
```

La boucle ci-dessus effectue deux opérations distinctes sur chaque volume.

La première étape consiste à **convertir les coordonnées** en latitude/longitude en coordonnées *utm* afin de pouvoir les exploiter dans le repère de Rhino, puis de les grouper au sein d'une liste (nommée *pts*) afin d'en générer une **polyligne**.

Ensuite, si le volume est en **porte à faux**, les différents **intervalles de hauteur** qu'il occupe dans son emprise en plan (exemple : 6 à 15 mètres) sont récupérés. Pour chacun d'entre eux, la polyligne créée précédemment sera déplacée en hauteur pour atteindre la borne "basse" de l'intervalle. Une **extrusion** est alors créée depuis cette base pour atteindre la hauteur définie par la borne "haute" de l'intervalle, et sera enfin écrite dans le fichier Rhinoceros. En revanche, en cas d'un volume simple reposant au rez-de-chaussée, une seule extrusion sera

créée afin d'atteindre la hauteur définie par l'attribut *hauteur*, puis sera écrite dans le fichier.

Enfin, le fichier est sauvegardé au format 3DM, que l'on peut ensuite visualiser et parcourir directement dans Rhinoceros.

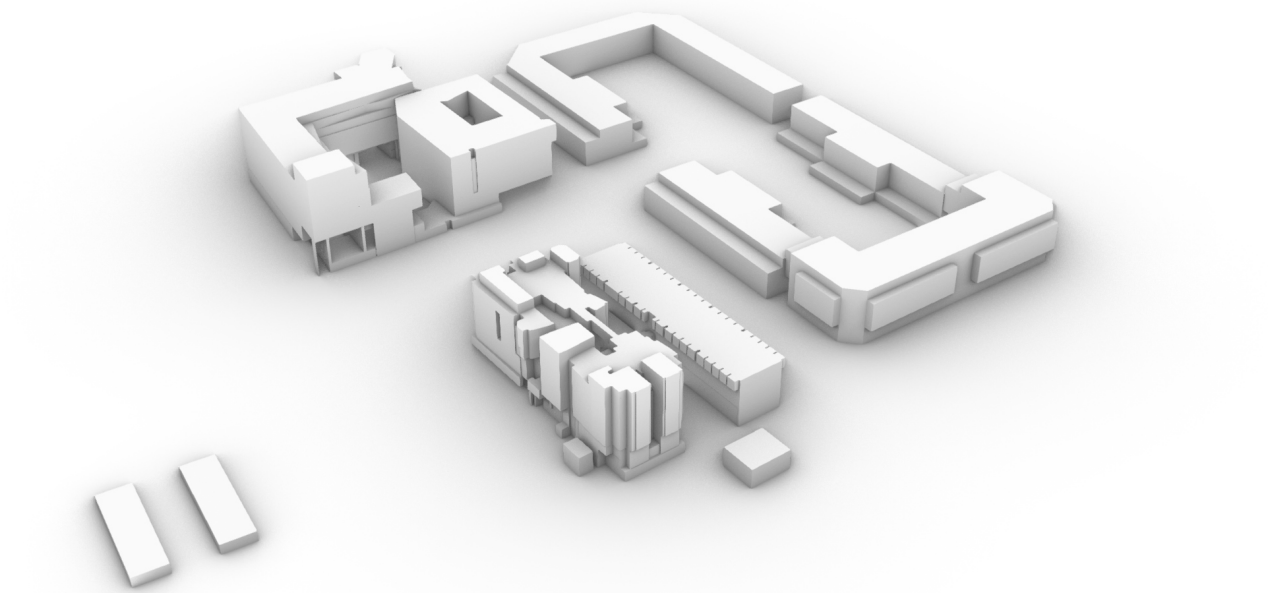


FIG. 13 :

Le résultat révèle à la fois une **précision** et un **degré de complexité** certains du jeu de données initial, jusque-là non-visualisables.

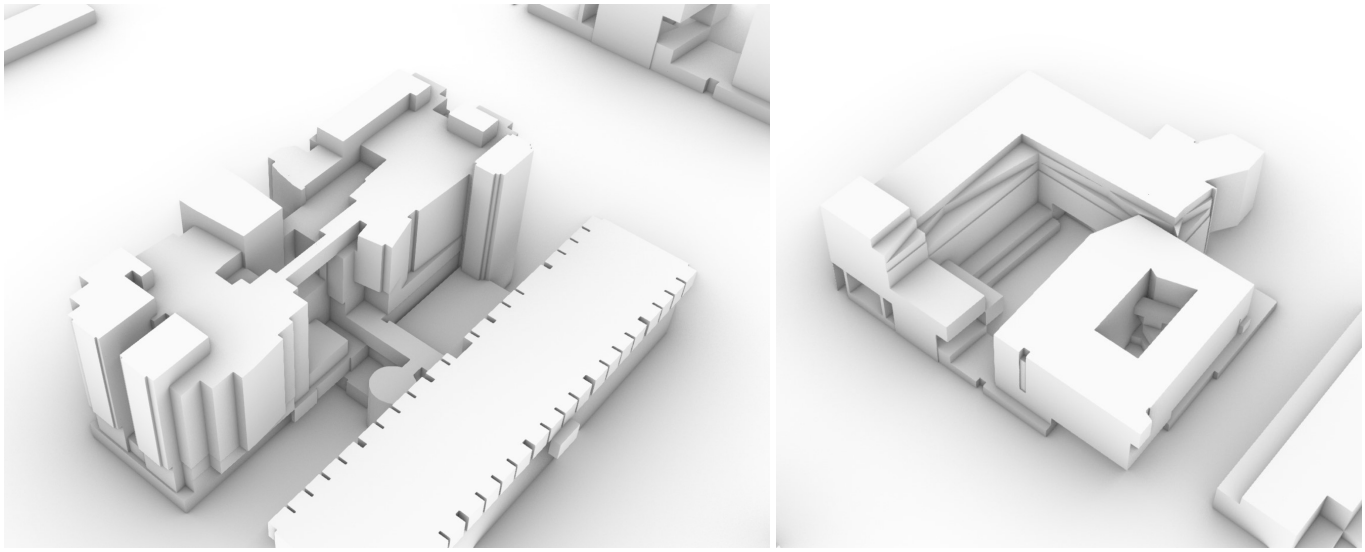


FIG. 14 :

Bien que la syntaxe de **Rhinoinside** soit plus ardue que celle de la section précédente (dû au fait qu'il faille s'appuyer sur de la documentation écrite pour le langage de programmation natif de Rhinoceros, le *C#*), un architecte possédant une bonne connaissance du logiciel Rhinoceros, et surtout de la manière logique de résoudre un problème de modélisation dans cet outil précis (surtout vis à vis des commandes) peut établir une stratégie viable.

Cette solution a donc permis d'exploiter les données collectées à leur plein potentiel.

Ainsi, au-delà de permettre une simple visualisation statistique des jeux de données en Open Data, le langage Python apporte à l'architecte des solutions techniques variées afin de convertir les jeux de données bruts en fichiers synthétiques dans des formats pleinements exploitables dans son environnement de travail, de la visualisation des données interactives au modèle 3D.

En outre, toutes ces solutions permettent d'obtenir chacun de ces éléments en **un temps d'éxécution très faible (quelques minutes d'exécution tout au plus)**, permettant un **gain de temps considérable** une fois qu'ils ont été développés, qui ne fera que **s'accroître** à mesure qu'on le **réutilise**. De plus, les scripts présentés au cours

de ce chapitre démontrent de par leur **longueur maîtrisée** (50 lignes maximum) que ce genre de manipulations complexes peuvent être mises en place de manière clarifiée, en prolongement des concepts basiques abordés dans le chapitre 1.

Jusqu'à lors, l'exercice a été de manipuler une infime portion d'un jeu de données ciblé autour d'un site, dans une optique « traditionnelle » de se renseigner sur un contexte immédiat. Le prochain chapitre sera dédié à une approche plus approfondie de l'utilisation de données ouvertes, recélant cependant un intérêt certain pour la conception architecturale.

3 Analyse approfondie d'une masse de données en Open Data : révéler et prédire des liens pour aiguiser sa conception

Montrer l'exemple de l'outil de prédiction de gisement de matériaux développé dans le cadre du PFE, en tant qu' « aboutissement » de ce que l'on est actuellement capable de tirer des données en Open Data.

CONCLUSION

Grâce à l'approche pratique proposée au sein de ce mémoire, nous avons pu démontrer que le langage de programmation Python constitue un outil souple et performant permettant de d'accompagner l'architecte tout au long de son exploitation des données issues de l'Open Data.

En effet, permettant en premier lieu de rendre compréhensible et manipulable aux plus novices les structures de données couramment rencontrées, en particulier les données hiérarchisées hétérogènes, le langage est doté d'outils de production de documents synthétiques tout à fait capables de convertir les données extraites en formats de fichiers courants pour l'architecte, élément primordial au sein du « workflow » de l'architecte. Ainsi, au-delà des graphiques statistiques, Python permet la génération de cartographies interactives, mais surtout des dessins techniques hiérarchisés et même des modèles 3D de manière automatique et personnalisée, représentant un gain de temps considérable.

Enfin, l'architecte devient capable grâce à Python de mener ses propres analyses de données afin de les mettre à profit dans son activité de conception, grâce à une capacité d'identification de corrélations entre différentes données agrégées, ou encore celle de pouvoir prédire l'impact de son intervention grâce au Machine Learning.

De plus, épaulé par divers modules permettant tous ces usages de manière simplifiée tout au long du processus d'exploitation des données ouvertes, le Python acquiert un véritable caractère universel, autant dans le sens où il est capable de se suffire à lui-même que pour qualifier sa compatibilité extraordinaire avec d'autres outils et services (comme Rhinoceros).

Au-delà du cadre de l'exploitation des données issues de l'Open Data, ce langage représente un véritable pivot afin d'accompagner les agences vers les nouveaux outils plus intelligents.

Premièrement, l'expérience acquise durant la manipulation des jeux de données ouverts (en particulier sur les notions des structures de données) sera extrêmement précieuse lorsqu'il s'agira d'exploiter des jeux de données plus directement liés à la pratique architecturale en elle-même, comme depuis un parc de « Smart Buildings » qu'il faudra surveiller et analyser par exemple.

Au-delà de cet aspect, des solutions logicielles émergentes tel que « Générative Design in Revit » publiée par Autodesk,[18] désormais intégrée à Revit 2021 proposent déjà à l'architecte de travailler aux côtés d'algorithmes évolutifs et prédictifs, que ce soit de manière simplifiée par interface graphique ou bien personnalisable grâce au langage de programmation visuelle Dynamo. La collaboration entre des agences d'architecture et des solutions intégrant de l'Intelligence Artificielle commence également à se développer, tel que l'agence Viguier et son partenariat avec « SpaceMaker AI »,[19] société norvégienne spécialisée dans la conception urbaine générative.

Toutes ces opportunités nouvelles requièrent une certaine familiarité avec le fonctionnement algorithmique, et surtout une assimilation de sa manière de fonctionner afin de l'intégrer efficacement à son « workflow »

Dès lors, de par son statut de langage de programmation lui conférant un caractère universel quant aux notions de bases informatiques et surtout algorithmiques (même avec une syntaxe simplifiée), le Python se présente comme un atout « futur-proof » face à ces nouvelles interactions architecte-machine. De plus, l'adoption massive du Python par les architectes permettrait d'amorcer une solution efficace afin de lutter contre la crainte de la montée en puissance de l'Intelligence Artificielle au sein des métiers du bâtiment, par une atténuation considérable de l'effet « Boite noire » qu'elle suscite. Ceci faciliterait donc grandement la transition du corps architectural dans la numérisation du domaine du bâtiment.

[18] *Generative Design in Revit now available. Revit* [en ligne]. 8 avril 2020. [Consulté le 17 janvier 2021]. Disponible à l'adresse : https://blogs.autodesk.com/revit/2020/04/08/generative-design-in-revit/. Section : What's New.

[19] *VIGUIER noue un partenariat avec SPACEMAKER AI* [en ligne]. [s. d.]. [Consulté le 17 janvier 2021]. Disponible à l'adresse : https://www.viguier.com/fr/actualites/journal/viguier-noue-un-partenariat-avec-spacemaker-ai.

Au-delà de cette appropriation, une certaine motivation de créer des outils logiciels par les architectes pour les architectes intégrant ces nouvelles compétences pourrait émerger, permettant alors de réaffirmer la place du métier d'architecte au sein d'un écosystème de plus en plus techno-centré.

BIBLIOGRAPHIE

Automate Road Surface Investigation Using Deep Learning | ArcGIS for Developers [en ligne]. [s. d.]. [Consulté le 3 février 2021]. Disponible à l'adresse : https://developers.arcgis.com/python/sample-notebooks/automate-road-surface-investigation-using-deep-learning/

Etalab - Qui sommes-nous. Le blog d'Etalab [en ligne]. [s. d.]. [Consulté le 17 janvier 2021]. Disponible à l'adresse : https://www.etalab.gouv.fr/qui-sommes-nous

Generative Design in Revit now available. Revit [en ligne]. 8 avril 2020. [Consulté le 17 janvier 2021]. Disponible à l'adresse : https://blogs.autodesk.com/revit/2020/04/08/generative-design-in-revit/. Section : What's New

Géoservices | Accéder au téléchargement des données libres IGN [en ligne]. [s. d.]. [Consulté le 15 septembre 2020]. Disponible à l'adresse : https://geoservices.ign.fr/documentation/diffusion/telechargement-donnees-libres.html

JSON [en ligne]. [s. d.]. [Consulté le 3 février 2021]. Disponible à l'adresse : https://www.json.org/json-fr.html

L'ouverture des données publiques. Gouvernement.fr [en ligne]. [s. d.]. [Consulté le 17 janvier 2021]. Disponible à l'adresse : https://www.gouvernement.fr/action/l-ouverture-des-donnees-publiques

Matplotlib : Python plotting [en ligne]. [s. d.]. [Consulté le 16 janvier 2021]. Disponible à l'adresse : https://matplotlib.org/

Portail open data de l'ADEME [en ligne]. [s. d.]. [Consulté le 17 janvier 2021]. Disponible à l'adresse : https://data.ademe.fr/

Portail Open data Île-de-France Mobilités [en ligne]. [s. d.]. [Consulté le 17 janvier 2021]. Disponible à l'adresse : https://data.iledefrance-mobilites.fr/pages/home/

PyOpenGL – The Python OpenGL Binding [en ligne]. 24 janvier 2021. Disponible à l'adresse : http://pyopengl.sourceforge.net/

Rhino - Rhino.Inside [en ligne]. 25 janvier 2021. Disponible à l'adresse : https://www.rhino3d.com/features/rhino-inside/

Seaborn : statistical data visualization [en ligne]. [s. d.]. [Consulté le 28 janvier 2021]. Disponible à l'adresse : https:

//seaborn.pydata.org/

Spatial and temporal distribution of service calls using big data tools | ArcGIS for Developers [en ligne]. [s. d.]. [Consulté le 3 février 2021]. Disponible à l'adresse : https://developers.arcgis.com/python/sample-notebooks/spatial-and-temporal-trends-of-service-calls/

The Python Graph Gallery : Visualizing data with Python [en ligne]. [s. d.]. [Consulté le 28 janvier 2021]. Disponible à l'adresse : https://python-graph-gallery.com/

VIGUIER noue un partenariat avec SPACEMAKER AI [en ligne]. [s. d.]. [Consulté le 17 janvier 2021]. Disponible à l'adresse : https://www.viguier.com/fr/actualites/journal/viguier-noue-un-partenariat-avec-spacemaker-ai

What is open data ? [en ligne]. [s. d.]. [Consulté le 28 décembre 2020]. Disponible à l'adresse : https://www.europeandataportal.eu/elearning/en/module1/#/id/co-01

HÜGEL, Stephan et ROUMPANI, Flora. *CityEngine-Twitter* [logiciel]. [S. l.] : Zenodo, 14 mai 2014. [Consulté le 28 décembre 2020]. DOI 10.5281/ZENODO.9795

P.ROUGIER, Nicolas. *Python & OpenGL for Scientific Visualization*. Bordeaux : [s. n.], 2018. Disponible à l'adresse : https://www.labri.fr/perso/nrougier/python-opengl/

VANNIEUWENHUYZE, Aurélien. *Intelligence artificielle vulgarisée : le Machine Learning et le Deep Learning par la pratique*. [S. l.] : [s. n.], 2019. ISBN 978-2-409-02073-5. OCLC : 1127535504

ZHOU, Qian-Yi, PARK, Jaesik et KOLTUN, Vladlen. Open3D : A Modern Library for 3D Data Processing. *arXiv :1801.09847*. 2018

ICONOGRAPHIE

ANNEXE : MÉTHODOLOGIE DE RÉDACTION

Ce mémoire a été écrit en utilisant le langage **R markdown**, sous la forme d'un seul script.