La réalisation du transfert des localisations du monde actuel vers les périodes géologiques passées par le biais d'un système d'information géographique

Certificat complémentaire en géomatique (2021)

Université de Genève, Suisse

ZHANG Haodong

Le 17 janvier 2022

Directeurs :

Gregory Giuliani

Christian Vérard

Table des matières

Résumé

Les géologues ont toujours voulu essayer de comprendre l'évolution de la Terre et ont proposé plusieurs théories pour expliquer comment la Terre est devenue ce qu'elle est aujourd'hui. Depuis l'acceptation de la théorie de la tectonique des plaques il y a quelques décennies, de nombreux modèles de tectoniques des plaques ont été réalisés afin de reconstituer leurs mouvements au fil du temps, y compris le projet PANALESIS. Ce modèle est toujours en train d'évoluer et nécessite un grand nombre de données de terrain pour s'améliorer. Dans ce contexte, une plateforme interactive a été proposée pour que des utilisateurs puissent transférer des localisations du monde actuel vers des ères géologiques du passé et ainsi comparer leurs données de terrain au modèle de la tectonique des plaques intégré dans PANALESIS. Ayant pour but d'explorer une première possibilité du développement d'une telle plateforme, ce travail s'est effectué à l'aide du logiciel de système d'information géographique ArcGIS Pro, et a produit un script issu du langage de programmation Python qui exécute le transfert de localisations vers le passé. Avec une quantité limitée de données en entrée, le script développé s'avère efficace et assez satisfaisant, mais exigera sans aucun doute plus de tests pour accroître sa stabilité et sa fiabilité pour garantir l'établissement de ladite plateforme interactive.

Introduction

Depuis la fin des années 1960, la théorie de la tectonique des plaques a été acceptée par le monde scientifique, ce qui a offert une nouvelle optique d'observation de la Terre que nous habitons (Palin & Santosh, 2021). Les plaques tectoniques constituent la représentation fondamentale du mouvement et de la déformation de la surface de la Terre, et des modèles de plaques peuvent être construits sur des logiciels informatiques (Wessel et Müller, 2015 ; Gurnis et al., 2018). La reconstruction des tectoniques des plaques étant une méthode couramment utilisée dans les géosciences, elle peut nous aider par exemple à prédire la structure du manteau d'aujourd'hui ou à construire des modèles de mouvements des plaques tectoniques (Gurnis et al., 2012). De tels modèles permettent de comprendre l'évolution de la Terre au fur et à mesure des périodes géologiques, mais restent complexes parce qu'ils mobilisent de multiples domaines de géosciences, y compris la paléontologie, la sédimentologie et la géophysique (Vérard, 2019a). Pourtant, de nombreux modèles globaux ont déjà été construits, à la fois pour des fins académiques et commerciales. Les modèles commerciaux développés par des entreprises privées généralement difficiles à se procurer, Vérard (2019a) a élaboré une liste des modèles académiques de plaques tectoniques dans son travail, entre autres le projet de PALEOMAP [1] construit par Christopher Scotese et celui d'EarthByte [2] dirigé par R. D. Müller, qui ont d'ailleurs largement poussé les recherches en la matière dans plusieurs disciplines de géosciences. En se basant sur les modèles existant, Vérard (2019b) a proposé un nouveau modèle de reconstructions paléogéographiques, baptisé PANALESIS, qui combine la reconstruction des plaques tectoniques, la reconstruction topographique, la reconstruction climatique ainsi que celle de la biosphère.

Palin & Santosh (2021) soulignent qu'en tant que masses de la lithosphère, les plaques tectoniques peuvent être complètement continentales, océaniques, ou les deux en même temps. Différents modèles de tectoniques des plaques proposeraient différentes façons de les délimiter (Vérard, 2019a), et cette délimitation doit se refaire à chaque nouvelle reconstruction (Vérard, 2019b). Sachant que la délimitation des tectoniques des plaques constitue un travail fondamental pour leurs reconstructions, elle est donc essentielle pour le modèle PANALESIS susmentionné. Comme clarifié par Vérard (2019b), le modèle des tectoniques des plaques de PANALESIS relève d'un modèle géodynamique global en perpétuel développement, et exige d'éventuelles améliorations qui demandent de considérables quantités de données de terrain. Dans ce contexte, il serait pratique de mettre en œuvre une plateforme informatique qui permet aux utilisateurs de

[1] http://www.scotese.com
[2] https://www.earthbyte.org

fournir leurs données pour pouvoir les comparer au modèle de tectoniques des plaques intégré dans PANALESIS. Cette plateforme, toujours selon Vérard (2019b), est censée être capable de transférer les données du passé vers leurs propres périodes géologiques pour qu'elles puissent être correctement analysées. Le travail de ce mémoire a donc été mené dans l'objectif d'explorer de façon préliminaire la possibilité de la mise en œuvre de ladite plateforme. Plus précisément, au lieu de produire une version interactive qui serait idéalement un site web, ce travail a été entièrement réalisé à l'aide d'un système d'information géographique (SIG), dans lequel un script Python a été développé pour réaliser le transfert de localisations du monde actuel vers les périodes géologiques passées.

Ce travail de mémoire se déroulera comme suit : l'outil et les données utilisés pour le développement du script Python seront abordés en premier lieu, suivis par la méthodologie ainsi que les résultats obtenus. Puis, une discussion sera ouverte pour parler des limites de ce travail et des éléments qui pourraient être améliorés avant la conclusion finale.

Outil et données

La reconstruction de tectoniques des plaques requérant des traitements d'information géographique, le logiciel ArcGIS Pro a été utilisé dans ce travail en vue de développer l'outil dédié au transfert des localisations du monde actuel vers le passé.

Les données utilisées se divisent en deux catégories. D'un côté, trois couches de points ont été ajoutées au projet d'ArcGIS Pro pour le développement de l'outil susmentionné, dont une provenant du projet de PALEOMAP de Christopher Scotese qui comprend des villes principales du monde d'aujourd'hui et qui a été notamment employée pour développer l'outil ; et deux autres qui sont issues des projets de forages océaniques scientifiques internationaux – respectivement le *Deep Sea Drilling Project* (DSDP, 1968-1983) et l'*Ocean Drilling Program* (ODP, 1985-2003) (Zhong, Zhang & Zhao, 2021) – et qui ont été utilisées pour tester l'outil développé. De l'autre côté, des couches de plaques tectoniques de différentes périodes géologiques (d'il y a plus de 500 millions d'années jusqu'à aujourd'hui) fournies par C. Vérard ont également servi au développement de l'outil.

Méthodologie

Le langage de programmation Python étant le langage d'exécution d'ArcGIS Pro, un script Python a été intégré dans ModelBuilder pour achever la tâche. Comme montré par la figure 1 à droite, le script *Tourner les points* prend trois couches en entrée, à savoir une entrée de points, une entrée de plaques tectoniques du monde présent (ci-après « couche d'aujourd'hui »), et enfin une entrée de plaques tectoniques d'une ère géologique passée choisie par l'utilisateur (ci-après « couche du passé »). Les localisations du monde actuel à transférer sont représentées par des entités de points dans ArcGIS Pro, et les plaques tectoniques des polygones. Le script crée cinq couches en sortie, dont les détails sont présentés dans le tableau 1.

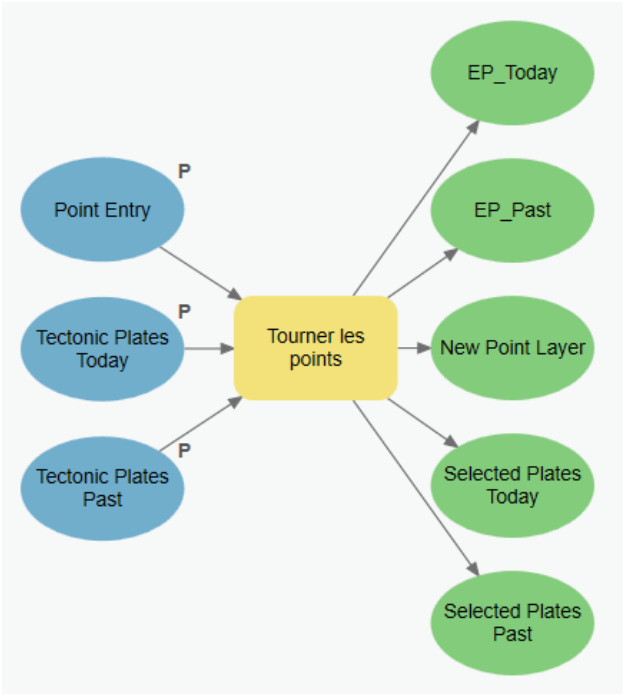

Figure 1 : Le modèle construit dans ModelBuilder d'ArcGIS Pro.

Une explication détaillée du script *Tournes les points* sera présentée dans la section suivante. Avant tout, il est à mentionner que le script utilise deux paquets : (1) *arcpy* qui appelle les fonctions disponibles dans ArcGIS Pro [3], et (2) *math* qui se charge de tous les calculs mathématiques nécessaires au script. La projection utilisée par défaut dans le script est WGS 1984.

La méthodologie est divisée en cinq parties : (1) la préparation des données en entrée, (2) la sélection de plaques tectoniques, (3) le calcul des pôles d'Euler, (4) la rotation des points et finalement, (5) la génération du message d'avertissement.

[3] https://pro.arcgis.com/en/pro-app/2.8/arcpy/get-started/what-is-arcpy-.htm

Tableau 1 : Détails des couches en sortie créées par le script.

Couche *Nom dans le script*	**Description**
EP Today *ajd_f*	Les plaques tectoniques de la couche d'aujourd'hui en entrée ayant été traitées avec l'outil *Sommets d'entités vers points* [4], la couche en sortie se compose de points. Avec la couche *EP Past*, ces deux couches ensemble permettent le calcul des pôles d'Euler pour chaque plaque tectonique. Cette couche peut être supprimée après le transfert des points mais est gardée pour une éventuelle vérification des résultats.
EP Past *pas_f*	Les plaques tectoniques de la couche du passé en entrée ayant été traitées avec l'outil *Sommets d'entités vers points*, la couche en sortie se compose de points. Avec la couche *EP Today*, ces deux couches ensemble permettent le calcul des pôles d'Euler pour chaque plaque tectonique. Cette couche peut être supprimée après le transfert des points mais est gardée pour une éventuelle vérification des résultats.
New Point Layer *pte_f*	Il s'agit de la couche en sortie des points transférés vers le passé. Les points de la couche en entrée qui ne peuvent pas être transférés vers le passé pour diverses raisons sont supprimés pendant le processus de transfert, et n'existent donc pas dans la couche en sortie. Les raisons d'échec seront abordées en détail plus tard.
Selected Plates Today *ajd2*	Couche de polygones, elle se compose de toutes les plaques tectoniques qui contiennent spatialement au moins un point qui arrive à être transféré vers le passé avec succès. Ces plaques tectoniques existent certainement dans la couche du passé. Cette couche peut être supprimée après le transfert des points mais est gardée pour une éventuelle vérification des résultats.
Selected Plates Past *pas1*	Couche de polygones, elle se compose de toutes les plaques tectoniques qui contiennent au moins un point qui arrive à être transféré vers le passé avec succès.

Partie 1 : Préparation des données en entrée

Le transfert de localisations du monde actuel vers le passé a essentiellement besoin du pôle d'Euler qui doit se calculer grâce à une certaine plaque tectonique et à son emplacement du passé. Il est à noter que le pôle d'Euler est un centre de rotation qui décrit les mouvements à la surface d'une sphère [5]. Il peut donc être utilisé pour modéliser les mouvements des plaques tectoniques de la Terre.

Cette étape consiste à préparer les couches en entrée susmentionnées, y compris la couche des points à transférer, la couche d'aujourd'hui et la couche du passé,

[4] *Feature Vertices To Points (Data Management)*
[5] Source : Wikipédia. https://fr.wikipedia.org/wiki/Pôle_eulérien. Consulté le 17 janvier 2022.

pour que le calcul des pôles d'Euler puisse bien se dérouler après. Afin que les couches en entrée ne soient pas modifiées, l'outil d'ArcGIS Pro *Copier des entités* [6] a d'abord été utilisé sur les trois couches en entrée afin d'en créer une copie temporaire. Les noms attribués aux trois copies temporaires sont respectivement ***pte1*** (**p**oin**t**s en **e**ntrée), ***ajd1*** (**au**jour**d**'hui) et ***pas1*** (**pas**sé).

Pour uniformiser la projection, la couche de points *pte1* est reprojetée vers la projection par défaut avec l'outil *Projeter* [7], et la classe d'entités en sortie est nommée ***pte2***.

Partie 2 : Sélection de plaques tectoniques

Bien que l'utilisateur doive paramétrer toutes les trois couches en entrée dans ModelBuilder avant l'exécution du script (cf. figure 1), il n'y a que la couche des points à transférer qui est une variable véritable ici, car la couche d'aujourd'hui et la couche du passé sont déjà toutes les deux préalablement préparées et l'utilisateur ne fait que désigner vers quelle période géologique il veut transférer ses points. En fonction de différentes couches de points en entrée, les plaques tectoniques contenant au moins un point à transférer changent. Pour simplifier les étapes à venir, les plaques tectoniques qui ne contiennent aucun point à transférer peuvent être supprimées d'*ajd1* en vue d'accélérer l'exécution du script, d'où la sélection de plaques tectoniques.

La sélection de plaques tectoniques se compose de deux étapes. La première est d'effectuer une *jointure spatiale* [8] entre *pte2* et *ajd1*, *pte2* étant les entités cible et *ajd1* les entités à joindre. Cette étape permet de garder les points qui se trouvent dans une plaque tectonique et de supprimer le reste. Pour ce faire, l'opération de jointure est *Joindre un à un*, l'opération de correspondance *Dans*, et seulement les entités cible qui réussissent à trouver une entité à joindre sont conservées dans la classe d'entités en sortie nommée ***pte3***. Un autre avantage de cette étape est qu'elle joint le champ *NAME_TERR* – le nom de la plaque tectonique contenant le point – dans la table attributaire, et ce sera utile lors de la rotation des points.

La deuxième étape consiste à utiliser l'outil *Sélectionner une couche par emplacement* [9] sur *ajd1*, la relation spatiale étant *Contient* et la couche de sélection *pte3*. Autrement dit, il s'agit de sélectionner les plaques tectoniques qui contiennent au moins un point à transférer vers le passé. La sélection a été extraite

[6] *Copy Features (Data Management)*
[7] *Project (Data Management)*
[8] *Spatial Join (Analysis)*
[9] *Select Layer By Location (Data Management)*

vers une nouvelle classe d'entités nommée ***ajd2*** pour ne garder que les plaques tectoniques sélectionnées.

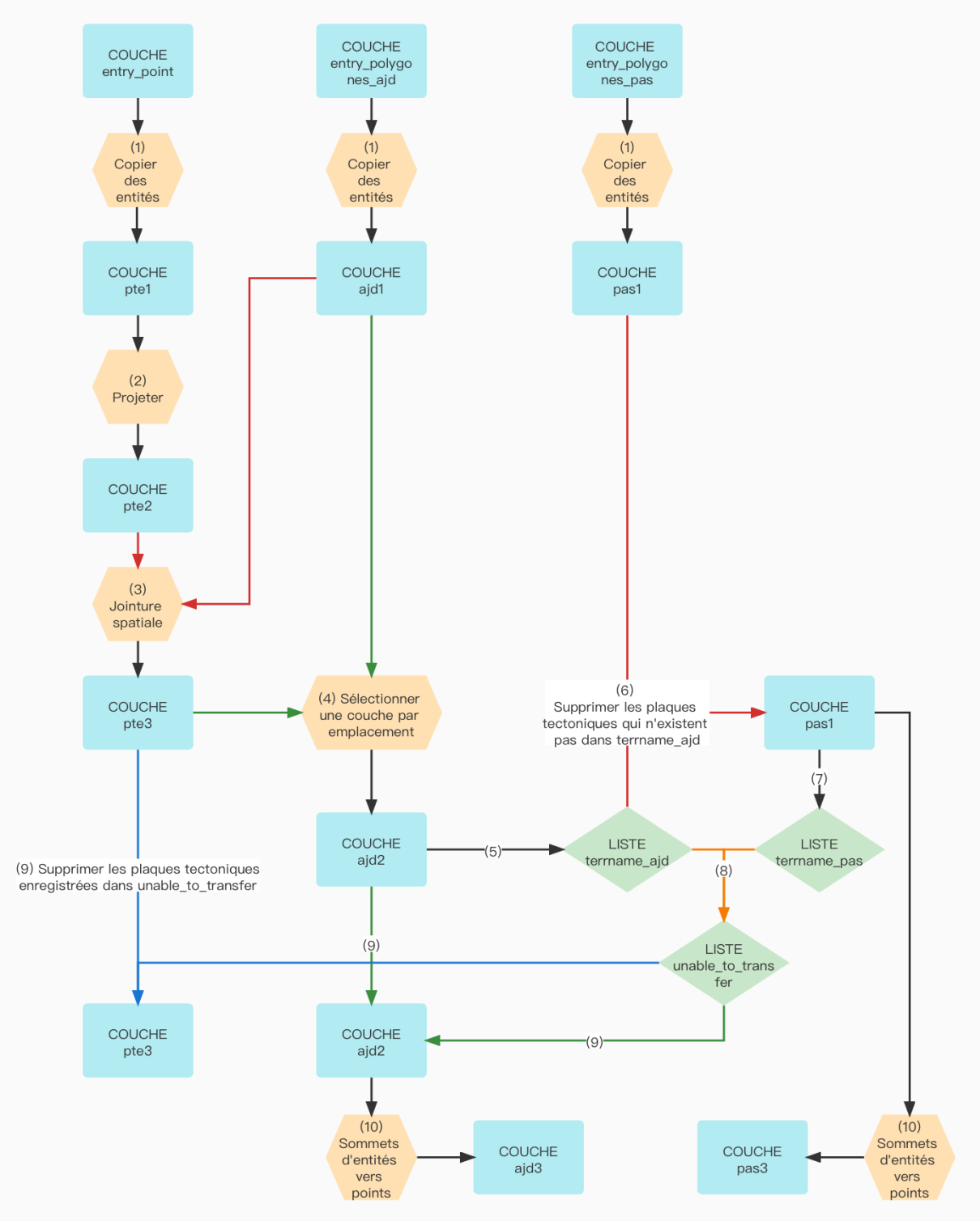

Figure 2 : La procédure en organigramme de la *Partie 1 : Préparation des données en entrée* et de la *Partie 2 : Sélection de plaque tectoniques*. Les couches sont représentées en rectangle cyan, les outils d'ArcGIS Pro en hexagone orange, et les listes créées dans le script Python en losange vert. Les chiffres qui se trouvent à la fois dans les formes et sur les flèches indique le numéro d'étape. Si une étape requiert au moins deux sources, les flèches sont distinguées par différentes couleurs.

Partie 2.1 : Génération des listes des noms de plaques tectoniques

Une liste des noms de plaques tectoniques a été créée à l'aide de l'outil *SearchCursor* à partir d'*ajd2*. Ce dernier lit entité par entité le champ *NAME_TERR* dans la table attributaire d'*ajd2*, et grâce à une boucle *for*, les noms de plaques tectoniques sont enregistrés dans une liste nommée *terrname_ajd*, chaque nom est unique et n'a donc pas de double dans la liste. Similairement, avec *UpdateCursor* qui lit également entité par entité le champ *NAME_TERR* de *pas1*, si le nom de la plaque n'existe pas dans *terrname_ajd*, cette entité sera alors supprimée pour faciliter les prochaines étapes.

Puis, il s'avère que toutes les plaques tectoniques de la couche d'aujourd'hui n'existent pas forcément dans la couche du passé, il faut donc maintenant créer une autre liste des noms de plaques tectoniques à partir de *pas1*, dans le but de décider plus précisément quels points peuvent être transférés vers le passé. Parce qu'une localisation géographique d'aujourd'hui ne peut en aucun cas être transférée à une plaque tectonique qui n'existait pas dans le passé. La création de cette liste s'effectue de la même manière que la création de *terrname_ajd*, mais la nouvelle liste s'appelle *terrname_pas*.

Puis, une autre liste nommée *unable_to_transfer* est créée pour répertorier les plaques tectoniques qui existent dans la couche d'aujourd'hui mais pas dans la couche du passé, autrement dit, il s'agit de la différence entre *terrname_ajd* et *terrname_pas*. Cette liste sera utile à la fin du script pour indiquer à l'utilisateur quels points n'ont pas pu être transférés vers le passé. De plus, les plaques tectoniques enregistrées par cette liste ainsi que les points à transférer qu'elles contiennent sont également supprimés respectivement dans *ajd2* et *pte3*.

Partie 2.2 : Transformation des sommets d'entités des plaques tectoniques en points

Dernière étape préparatoire, cette étape transforme les sommets d'entités des plaques tectoniques sélectionnées, à la fois dans *ajd2* et *pas1*, en points, à l'aide de l'outil *Sommets d'entités vers points*. Les couches en sortie sont respectivement nommées ***ajd3*** et ***pas3***.

Partie 3 : Calcul des pôles d'Euler

La partie principale de la méthodologie consiste à calculer les pôles d'Euler pour les plaques tectoniques qui contiennent au moins un point à transférer vers le passé. Ils se composent d'un axe de rotation et d'un angle de rotation. Pour ce faire,

une copie temporaire a été créée avec *Générer une couche* [10] pour *ajd3* et *pas3*, respectivement sous le nom ***ajd*** et ***pas***. Cette étape est cruciale d'une manière ou d'une autre, car sans elle le script générera des erreurs dont l'origine est inconnue.

À ce moment-là, il est important d'ajouter quelques champs dans la table attributaire d'*ajd* et de *pas* qui seront utiles après. L'outil à utiliser est *Ajouter des champs* [11]. Pour *ajd*, les champs à ajouter incluent :

Tableau 2 : Détails des champs à ajouter dans la table attributaire de la couche d'*ajd*.

Nom	Type	Description
RAD_X	Double	L'abscisse en radians d'un point exprimé en coordonnées géocentriques.
RAD_Y	Double	L'ordonnée en radians d'un point exprimé en coordonnées géocentriques.
RAD_Z	Double	La cote en radians d'un point exprimé en coordonnées géocentriques.
NORM	Double	La norme d'un point exprimé en coordonnées géocentriques. La Terre étant supposée être une sphère parfaite dans ce travail, par défaut elle égale à 1 pour tous les points.
PROD_SCAL	Double	Le produit scalaire entre deux points consécutifs d'une plaque tectonique. Si les *ObjectID*s de ces derniers sont respectivement i et $i + 1$, le produit scalaire entre eux sera associé à i, c'est-à-dire l'*ObjectID* plus petit des deux.
PAIR	Float	L'*ObjectID* d'un point de *pas* qui partage le même produit scalaire qu'un point d'*ajd* donné. Chaque paire est strictement unique.

Tandis que pour *pas*, tous les champs, autre que le dernier, décrits dans le tableau 2 sont à ajouter.

Partie 3.1 : Création des fonctions

Afin d'avoir un script plus concis, plusieurs fonctions Python sont créées. Elles sont décrites dans le tableau 3 ci-dessous. Il faut avant tout clarifier que les points utilisés ou générés par les fonctions, sauf les trois premières, sont toujours exprimés en coordonnées géocentriques sous forme de tuple (x, y, z, n), où x, y et z sont en radians, et n est la norme du point et égale à 1 par défaut.

[10] *Make Feature Layer (Data Management)*
[11] *Add Fields (multiple) (Data Management)*

Tableau 3 : Détails des fonctions.

Nom	Utilité
AT_CalculateXYZ	Calculer les coordonnées géocentriques ainsi que la norme des points d'*ajd* et de *pas* à partir de leurs coordonnées géodésiques, c'est-à-dire la longitude *x* et la latitude *y*, et transcrire les résultats respectivement dans les champs *RAD_X*, *RAD_Y*, *RAD_Z* et *NORM* de la table attributaire. Les coordonnées géocentriques sont exprimées en radians. N. B. : « AT » sont les sigles de la table attributaire en anglais.
XYZtoXY	Convertir les coordonnées géocentriques des points vers les coordonnées géodésiques. Le résultat est un point sous forme de tuple (*x*, *y*, *r*), où *x* est la longitude, *y* la latitude et *r* le rayon moyen de la Terre qui égale à 1 par défaut.
XYtoXYZ	Convertir les coordonnées géodésiques des points vers les coordonnées géocentriques. Le résultat retourné par cette fonction est un point.
VectorProduct	Calculer le produit vectoriel entre deux points. Le résultat retourné par cette fonction est aussi un point.
DotProduct	Calculer le produit scalaire entre deux points. Le résultat de cette fonction est un angle en radians.
RotatePoint	Prendre un axe et un angle de rotation en radians pour tourner un point. Le résultat de cette fonction est un nouveau point. L'axe de rotation est un point et l'angle de rotation doit être en radians.
GetTranslationPole	Calculer entre deux points le produit vectoriel comme axe de rotation, et le produit scalaire comme angle de rotation. Le résultat retourné par cette fonction est un pôle d'Euler encore à vérifier sous forme de tuple « *axe*, *angle* ».
GetSmallCircleAngle	Vérifier le pôle d'Euler retourné par *GetTranslation-Pole*. Son résultat est un angle de rotation en radians.
CheckAngle	Examiner l'angle de rotation issu de *GetSmall-CircleAngle* et décider sa direction. Le résultat retourné est toujours un angle en radians.
AddPoles	Calculer le pôle d'Euler final qui se compose d'un axe et d'un angle de rotation, sous forme de tuple « *axe*, *angle* ».
RotationCheck	Décider si le pôle d'Euler final issu d'*AddPoles* est fiable. Le résultat est un niveau de confiance en pourcentage. Plus le niveau de confiance est proche de 1, plus le pôle d'Euler est fiable.

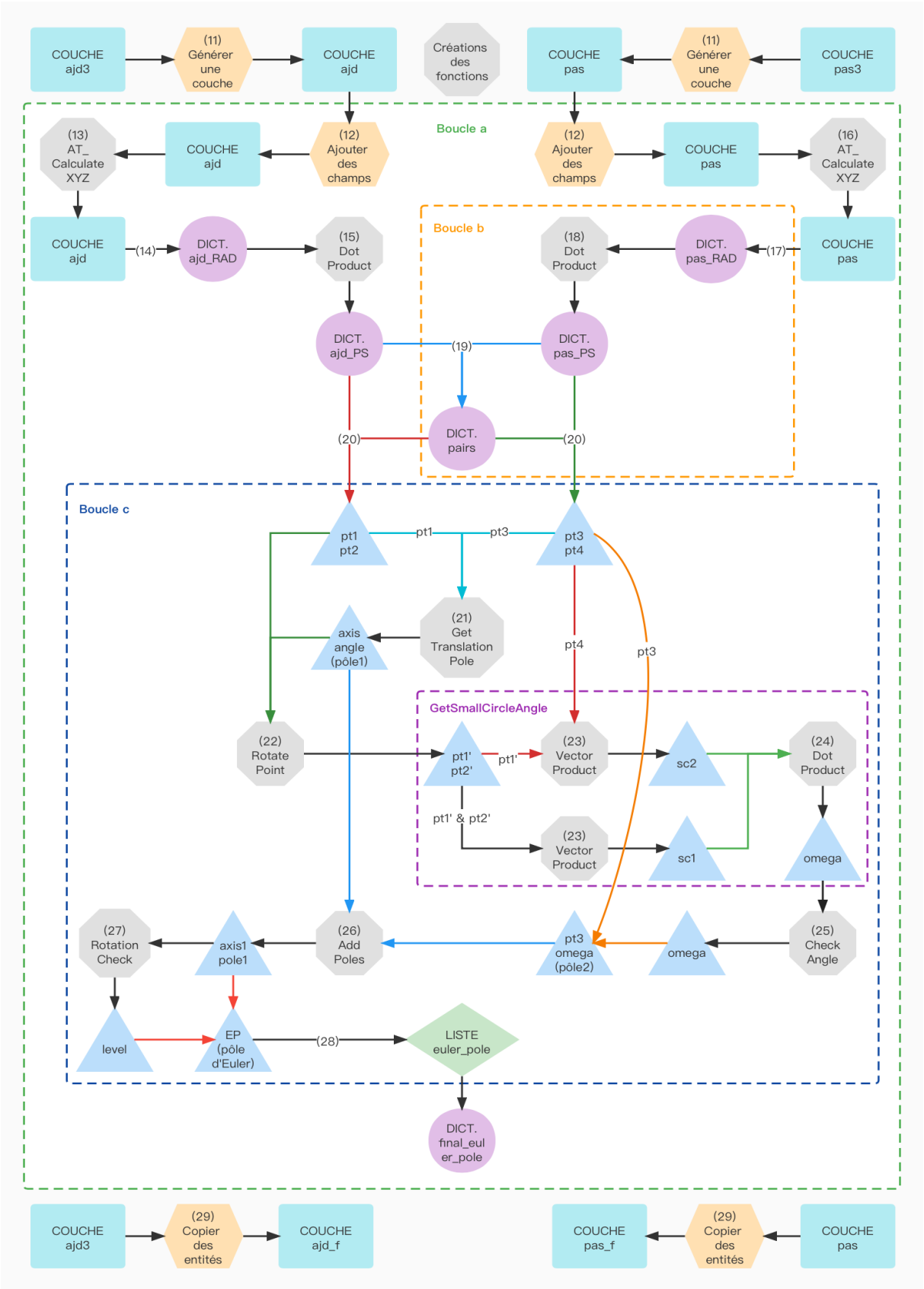

Figure 3 : La procédure en organigramme de la *Partie 3 : Calcul des pôles d'Euler*. À part les éléments déjà expliqués dans la figure 2, les fonctions créées dans le script Python sont représentées en octogone gris, les dictionnaires Python en rond violet, les variables Python en triangle bleu (foncé). Les éléments faisant partie d'un groupe commun sont ceints par des pointillés (boucle *a*, boucle *b*…), et différents groupes sont indiqués en différentes couleurs.

Partie 3.2 : Recherches des paires de points

La recherche des paires de points s'effectue à la fois dans *ajd* et *pas*. Il est à rappeler que ces deux couches se composent des plaques tectoniques sélectionnées dont les sommets d'entités sont transformés en points dans la partie 2. Les points générés lors de ce processus se voient automatiquement attribués un *ObjectID*. Pour une seule plaque tectonique, l'attribution d'*ObjectID*s s'effectue sur les points de manière consécutive en suivant un sens fixe. Cela fait que deux *ObjectID*s consécutifs correspondent toujours à deux points consécutifs dans une seule plaque tectonique. Les paires de points désignent alors deux points consécutifs dans *ajd* et deux autres dans *pas* qui se trouvent dans la même plaque tectonique et qui partagent le même produit scalaire. Cette recherche a pour but de permettre le calcul du pôle d'Euler d'une plaque tectonique. L'idée est de considérer les paires de points disposant du même produit scalaire comme les mêmes points qui existent dans différentes périodes géologiques, ainsi sera-t-il possible de calculer l'axe et l'angle de rotation pour une certaine plaque tectonique. Parce que si est trouvé le pôle d'Euler avec lequel une plaque tectonique s'est tournée au fil du temps, il sera logique de supposer que les localisations contenues par elle se tournent avec le même pôle d'Euler.

Pour identifier les paires de points, quelques dictionnaires et une liste sont créés. Leurs détails sont présentés dans le tableau 4 ci-dessous.

Le dictionnaire dans Python permet de trouver un élément par clef qui est unique dans un dictionnaire donné. Il est alors idéal d'associer des informations à une clef, qui peut être un *ObjectID* ou un nom de plaque tectonique dans le script, pour pouvoir les réutiliser après la création de la clef.

Pour calculer le pôle d'Euler, une boucle *for* est introduite et sera appelée **boucle *a*** ci-après pour plus de commodité. Elle prend à chaque fois une seule plaque tectonique enregistrée dans la liste *terrname_pas*, et calcule son pôle d'Euler et puis procède vers la prochaine. Au lieu de *terrname_ajd*, c'est *terrname_pas* qui est utilisée car il est certain que toutes les plaques tectoniques de ce dernier existent dans la couche d'aujourd'hui dû au traitement décrit plus haut, ce qui est la condition préalable du transfert de points ; tandis que *terrname_ajd* ne pourrait le réaliser car il est possible que cette liste ait des plaques qui n'existent pas dans la couche du passé, ce qui fera échouer le transfert de points.

Le calcul peut être décomposé en quatre parties : (1) le calcul du produit scalaire pour *ajd*, (2) le calcul du produit scalaire pour *pas*, (3) l'identification des paires de points et (4) le calcul du pôle d'Euler. Elles seront abordées l'une après l'autre.

Tableau 4 : Détails des dictionnaires et de la liste créés pour la recherche de pôles d'Euler.

Nom	Nature	Description
ajd_RAD	Dictionnaire	Il enregistre les coordonnées géocentriques en radians ainsi que la norme des points d'*ajd*. L'*ObjectID* étant la clef, la valeur associée à elle est sous forme de (x, y, z, n) comme évoqué plus haut dans la partie 3.1.
pas_RAD	Dictionnaire	Équivalent d'*ajd_RAD*, il enregistre les coordonnées géocentriques en radians et la norme des points de *pas*.
ajd_PS	Dictionnaire	Il enregistre le produit scalaire en degrés entre deux points consécutifs d'une plaque tectonique d'*ajd*. Si les *ObjectID*s de ces deux points consécutifs sont respectivement i et $i + 1$, i sera la clef et le produit scalaire entre les deux points sera la valeur associée. N. B. : « PS » sont les sigles du produit scalaire.
pas_PS	Dictionnaire	Équivalent d'*ajd_PS*, il enregistre le produit scalaire en degrés entre deux points consécutifs d'une plaque tectonique de *pas*.
pairs	Dictionnaire	Il enregistre les paires de points identifiées, avec l'*ObjectID* du point d'*ajd* comme clef et celui du point de *pas* comme valeur associée. Chaque paire permet de calculer un pôle d'Euler. Pour une seule plaque tectonique, au maximum dix paires suffisent pour le calcul et la vérification de son pôle d'Euler.
euler_pole	Liste	C'est une liste des pôles d'Euler calculés pour une seule plaque tectonique. Elle enregistre au maximum dix pôles d'Euler.
final_euler_pole	Dictionnaire	Il enregistre le pôle d'Euler le plus probable d'une plaque tectonique, avec le nom de cette dernière comme clef et le pôle d'Euler comme valeur associée. Il regroupe les pôles d'Euler de toutes les plaques tectoniques sélectionnées plus haut et sera utilisé pour tourner les points dans la partie 4.

Partie 3.2.1 : Calcul du produit scalaire pour la couche d'aujourd'hui

Cette partie n'agit que sur la couche *ajd* et est sujette à la boucle *a*. Pour commencer, il faut calculer les coordonnées géocentriques pour tous les points contenus dans la plaque tectonique courante de la boucle *a* avec la fonction *AT_CalculateXYZ*. Pour rappel, elle calcule les coordonnées géocentriques à partir des coordonnées géodésiques des points, à savoir la longitude x et la latitude y, et transcrit les résultats respectivement dans les champs suivants de la table attributaire : *RAD_X*, *RAD_Y* et *RAD_Z*. Elle calcule également la norme des

coordonnées géocentriques pour la mettre dans le champ *NORM* de la table attributaire. La norme est censée valoir 1 pour tous les points.

Puis, en vue de calculer les produits scalaires, l'outil *SearchCursor* qui permet de lire les informations dans la table attributaire est de nouveau utilisé. Avec une boucle *for* dans l'outil *SearchCursor*, il enregistre les coordonnées géocentriques de tous les points de la plaque tectonique courante, l'un après l'autre, dans le dictionnaire *ajd_RAD*, avec l'*ObjectID* [12] comme clef. Dès que les coordonnées du deuxième point de la plaque ont été enregistrées, il est alors possible de calculer le produit scalaire entre les deux premiers points, qui sera ensuite enregistré dans le dictionnaire *ajd_PS*, toujours avec l'*ObjectID* comme clef. Cette procédure est itérée jusqu'à ce que les coordonnées du dernier point de la plaque courante aient été enregistrées dans *ajd_RAD* et que le produit scalaire entre les deux derniers points ait été calculé et enregistré dans *ajd_PS*. Il est à rappeler que le produit scalaire se calcule avec la fonction *DotProduct* et qu'il est en degrés. La raison pour laquelle il est en degrés sera présentée après.

Les produits scalaires calculés sont ensuite transcrits dans le champ *PROD_SCAL* de la table attributaire avec l'outil *UpdateCursor*. Cette étape pourrait être ôtée mais est gardée en cas d'éventuelles vérifications des résultats.

Partie 3.2.2 : Calcul du produit scalaire pour la couche du passé et l'identification des paires de points

Cette partie n'agit que sur la couche *pas* et est sujette à la boucle *a*. Comme dans *ajd*, il faut aussi calculer les coordonnées géocentriques pour tous les points de la plaque tectonique courante de la boucle *a* avec la fonction *AT_CalculateXYZ*.

Puis, similairement, *pas_RAD* enregistre les coordonnées géocentriques des points de *pas*, une plaque tectonique à la fois, et en même temps *pas_PS* enregistre les produits scalaires calculés, grâce à une boucle *for* imbriquée encadrée dans l'outil *SearchCursor* qui sera appelée **boucle *b***. Contrairement à *ajd*, il n'est pas nécessaire de calculer le produit scalaire pour tous les points de *pas*. Le but ultime est d'identifier les paires de points et au maximum seulement dix paires sont requises, tandis qu'une plaque tectonique en a souvent beaucoup plus. Donc, dans la boucle *b*, une fois qu'un produit scalaire est calculé, il sera comparé aux produits scalaires enregistrés dans *ajd_PS* pour voir s'il y a une valeur identique. Il est à noter qu'une erreur de 0,001 kilomètre est autorisée. Le script prend 40'075 kilomètres comme longueur de l'équateur de la Terre, valeur fournie par Google. 0,001 kilomètre égale donc à 0,001 × 360 / 40'075 degrés, soit à peu près 0,00000898 degrés. Si la valeur absolue de la différence entre le produit scalaire calculé et une

[12] Dans le script, c'est *OID@*.

certaine valeur dans *ajd_PS* est inférieure à 0,00000898 degrés, une paire est considérée identifiée, et elle sera tout de suite enregistrée dans le dictionnaire *pairs*, avec l'*ObjectID* du point d'*ajd* comme clef et celui du point de *pas* comme valeur associée. Cette clef dans *ajd_PS* ainsi que sa valeur associée seront ensuite supprimées pour éviter d'éventuelles confusions lors de l'identification des prochaines paires et assurer qu'une clef n'apparaît pas dans plus d'une paire. Dès que dix paires de points ont été identifiées, la boucle *b* se termine. Or si aucune paire est trouvée, la plaque tectonique en question sera ajoutée à la liste *unable_to_transfer*.

Le travail principal achevé, les paires identifiées et les produits scalaires de *pas* sont respectivement transcrits dans la table attributaire d'*ajd* et de *pas* avec l'outil *UpdateCursor* pour d'éventuelles vérifications des résultats.

Partie 3.3 : Calcul des pôles d'Euler

Cette partie est toujours sujette à la boucle *a* mais une nouvelle boucle *for* imbriquée, qui sera appelée **boucle *c*** ci-après, est nécessaire pour le calcul des pôles d'Euler.

La boucle *c* itère dans le dictionnaire *pairs*. À chaque itération, elle prend deux points consécutifs d'*ajd* et deux autres de *pas*, ce qui se fait en fonction des paires de points enregistrées dans *pairs*. Pour rappel, les clefs de *pairs* sont des *ObjectID*s des points d'*ajd*, et les valeurs associées aux clefs des *ObjectID*s des points de *pas*. Cela dit, prenons la paire « *i : j* (clé : valeur associée) » comme exemple, l'*ObjectID* du premier point d'*ajd* correspond alors à *i*, et celui du deuxième point d'*ajd* est *i + 1*. Analogiquement, l'*ObjectID* du premier point de *pas* correspond à *j*, et celui du deuxième point de *pas* est *j + 1*. Le calcul des pôles d'Euler ayant besoin des coordonnées géocentriques des points, elles sont fournies par *ajd_RAD* et *pas_RAD* antérieurement créés. Les coordonnées des quatre points sont enregistrées dans quatre variables nouvellement créées dans la boucle *c* : ***pt1*** (*i*), ***pt2*** (*i + 1*), ***pt3*** (*j*) et ***pt4*** (*j + 1*). *Pt1* d'*ajd* est d'ailleurs équivalent de *pt3* de *pas*, et *pt2* d'*ajd* équivalent de *pt4* de *pas*.

Afin de trouver le pôle d'Euler, il suffit d'abord d'utiliser la fonction *GetTranslationPole* pour calculer l'axe et l'angle de rotation entre *pt1* et *pt3*, ils sont respectivement appelés ***axis*** et ***angle*** dans le script. *Axis* est sous forme commune utilisée pour les points dans le script, soit (*x*, *y*, *z*, *n*), où *x*, *y* et *z* sont coordonnées géocentriques en radians et la norme *n* égale à 1, et *angle* est aussi en radians. *Axis* et *angle* ensemble constituent le premier pôle.

Puis, *pt1* et *pt2* sont tournés par la fonction *RotatePoint*, avec *axis* comme axe de rotation et *angle* comme angle de rotation, pour produire ***pt1'*** et ***pt2'***. Vu qu'*axis*

et *angle* sont calculés à partir de *pt1* et *pt3*, *pt1'* est censé se superposer sur *pt3*, autrement dit, *pt1'* et *pt3* partagent les mêmes coordonnées géocentriques. Et si l'axe et l'angle de rotation étaient exacts, *pt2'* devrait également se superposer sur *pt4*. Or ce n'est pas cent pour cent garanti, il est donc nécessaire de mesurer à quel point *axis* et *angle* sont précis, et c'est la fonction *GetSmallCircleAngle* qui sert premièrement à cet objectif.

Étant donné que *pt2'* et *pt4* doivent théoriquement se superposer, mais il est probable que ce n'est pas le cas en réalité, il est donc possible de mesurer l'imprécision d'*axis* et d'*angle* par le biais suivant : (1) calculer le produit vectoriel respectivement entre *pt1'* et *pt2'* et entre *pt1'* et *pt4*, leur résultat étant ***sc1*** et ***sc2***, et (2) calculer le produit scalaire, soit l'angle de rotation, entre *sc1* et *sc2*, et cet angle de rotation est nommé ***omega*** dans le script. Ainsi, si *pt2'* se superpose sur *pt4*, *omega* égalera 0. C'est-à-dire que plus *omega* est vers 0, moins l'imprécision d'*axis* et d'*angle* est importante.

Ensuite, la fonction *CheckAngle* examine *omega* et décide sa direction. Elle tourne *pt2'* autour de *pt1'* avec *omega* comme angle de rotation, et le nouveau point sera appelé ***pt2''***. Si la direction de la rotation est correcte, *pt2''* sera censé être proche de *pt4* voire se superposer sur ce dernier. En effet, si *pt2''* se trouve à moins de 0,1 kilomètre de *pt4*, la direction de l'angle de rotation *omega* sera considérée correcte, c'est-à-dire que *pt2'* a été tourné dans le bon sens. Sinon, le signe de la valeur d'*omega* sera inversé. Cela fait, *pt3*, équivalent de *pt1'*, et *omega* constituent le deuxième pôle.

Maintenant que le premier pôle (*axis* et *angle*) et le deuxième pôle (*pt3* et *omega*) sont tous les deux disponibles, la fonction *AddPoles* peut alors les prendre pour calculer le pôle d'Euler tant attendu, qui se compose de l'axe de rotation ***axis1*** et l'angle de rotation ***pole1***. Or ce pôle d'Euler peut être peu précis, il doit encore être examiné par la fonction *RotationCheck*, dernière étape de cette longue histoire. Avant tout, il est à noter que *RotationCheck* ne révoque pas le pôle d'Euler à examiner, mais qu'elle donne un niveau de confiance en pourcentage pour indiquer dans quelle mesure le pôle d'Euler est fiable. Plus le niveau de confiance est vers 1, plus le pôle d'Euler examiné est fiable. En pratique, *RotationCheck* tourne tous les points de *pas* enregistrés dans *pairs* (donc au maximum dix points) autour d'*axis1*, avec *pole1* comme angle de rotation, et voir si la distance entre les points tournés et leur équivalent d'*ajd* est inférieure ou égale à 0,1 kilomètre. Le niveau de confiance du pôle d'Euler examiné augmente au fur et à mesure que de plus en plus de points remplissent ce critère, puisque le niveau de confiance est le résultat du nombre des points remplissant le critère divisé par le nombre total des points tournés. Le niveau de confiance est nommé ***level*** dans le script, et il est rajouté à *axis1* et *pole1* pour former le pôle d'Euler final, baptisé ***EP*** et sous forme de tuple « *axis1*, *pole1*, *level* ». *EP* est ensuite ajouté à la liste *euler_pole*.

La boucle *c* itérant dans *pairs* qui contient au maximum dix paires de points, *euler_pole* possède donc également au maximum dix pôles d'Euler pour une seule plaque tectonique. Et si à l'issue de la dernière itération de la boucle *c*, il n'y a aucun pôle d'Euler qui a été calculé avec succès pour être enregistré dans la liste *euler_pole*, la plaque tectonique en question sera ajoutée à la liste *unable_to_transfer*. Sinon, *euler_pole* contiendra tous les pôles d'Euler issus de la procédure qui vient d'être décrite. Un seul pôle d'Euler suffit néanmoins à une plaque tectonique, celui avec le plus haut niveau de confiance sera donc choisi et mis dans le dictionnaire *final_euler_pole*, avec le nom de la plaque comme clef. Cependant, il ne faut pas ignorer le fait que différents pôles d'Euler qui n'ont pas exactement les mêmes axes ou angles de rotation peuvent tous avoir le plus haut niveau de confiance. Dans ce cas, les éléments dans une liste Python tous indexés à partir de 0, seulement le pôle d'Euler ayant le plus grand index dans la liste *euler_pole* sera gardé pour la plaque tectonique en question.

Partie 3.4 : Fin de la partie 3

La partie 3 sert à trouver le pôle d'Euler des plaques tectoniques sélectionnées dans la partie 2 et les mettre dans le dictionnaire *final_euler_pole*. Avant qu'une itération de la boucle *a* finisse, les dictionnaires, à savoir *ajd_RAD*, *pas_RAD*, *ajd_PS*, *pas_PS* et *pairs*, ainsi que la liste *euler_pole* sont tous vidés pour ne pas compromettre la prochaine itération. Une fois la dernière itération finie, la boucle *a* se termine, et l'outil *Copier des entités* créera deux nouvelles couches *ajd_f* et *pas_f* à partir d'*ajd* et de *pas*. Ces deux nouvelles couches sont parmi les cinq couches en sortie du script.

Partie 4 : Rotation des points

Le calcul des pôles d'Euler fini, il est temps de tourner les points.

La rotation des points nécessite des coordonnées géocentriques. De ce qui est vu plus haut, la fonction *AT_CalculateXYZ* est créée pour calculer et ajouter des coordonnées géocentriques dans la table attributaire. Mais avant cela, il faut ajouter quelques champs dans la table attributaire de *pte3*, couche de points à traiter. À part les champs *RAD_X*, *RAD_Y*, *RAD_Z* et *NORM* déjà vus, les champs propres à *pte3* à ajouter sont présentés en détails par le tableau 5 ci-dessous :

Tableau 5 : Détails des champs à ajouter dans la table attributaire de *pte3*. Ces champs sont tous en lien avec le pôle d'Euler de la plaque tectonique contenant le point à tourner.

Nom	Type	Description
AXIS_X	Double	L'abscisse en radians de l'axe de rotation exprimé en coordonnées géocentriques.
AXIS_Y	Double	L'ordonnée en radians de l'axe de rotation exprimé en coordonnées géocentriques.
AXIS_Z	Double	La cote en radians de l'axe de rotation exprimé en coordonnées géocentriques.
ROTATION_ANGLE	Double	L'angle de rotation en radians.
CONFIDENCE_LEVEL	Float	Le niveau de confiance en pourcentage.

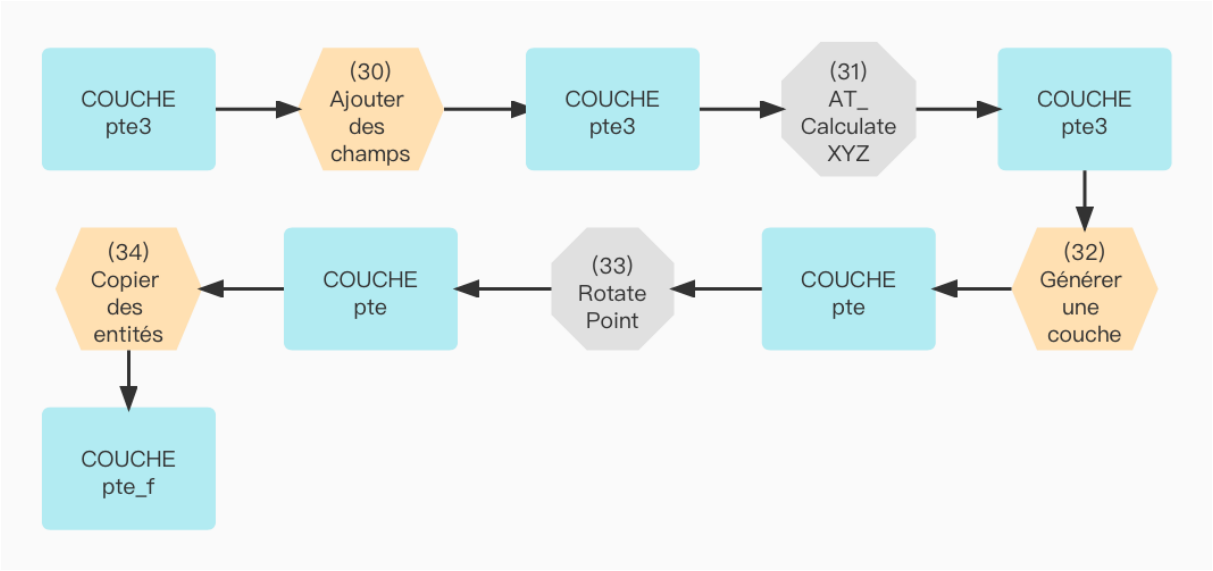

Figure 4 : La procédure en organigramme de la *Partie 4 : Rotation des points*.

Puis, l'outil *Générer une couche* ayant créé ***pte*** à partir de *pte3*, *RotatePoint* tournera les points en raison de leurs coordonnées géocentriques et du champ *NAME_TERR* précédemment joint à la table attributaire de *pte3*, et *UpdateCursor* mettra à jour les coordonnées du point et écrira celles du pôle d'Euler dans les cinq champs affichés dans le tableau 5. Ainsi, les points sont tournés. Si un point ne peut être tourné dû à une erreur, il sera supprimé de *pte*. Finalement, *pte* sera copié vers la couche en sortie *pte_f*.

Partie 5 : Génération du message d'avertissement

Jusque-là, le travail principal du script a déjà pris fin. Or pendant la partie 3, il y a possiblement de nouvelles plaques tectoniques qui sont rajoutées à la liste *unable_to_transfer*, il est donc nécessaire de supprimer de nouveau ces plaques d'*ajd2* et les points qui tombent dans celles-ci. Cela fait, un message sera affiché

pour dire à l'utilisateur si les points ont tous été tournés, sinon, lesquels n'ont pas pu être tournés.

Résultats

Le script a été testé sur trois couches de points et quelques couches de plaques tectoniques du passé. Les trois couches de points en entrée comprennent *ScoteseCities_000*, *DSDPsites_000* et *ODPsites_000*.

Le premier essai a été effectué sur *ScoteseCities_000*. C'est également avec cette couche de points que le script a été développé. Comme montré dans la figure ci-dessous, la couche de points *ScoteseCities_000* a beaucoup de villes dans le monde entier, distribuées dans de multiples plaques tectoniques de l'ère d'aujourd'hui (indiqué par « 000 »). Il y a des points comme *Hawaii* qui ne tombe dans aucune plaque tectonique, il sera donc supprimé lors de l'exécution du script.

Ces points sont transférés vers il y a 103 millions d'années, le résultat de l'exécution du script est démontré par la figure 5 ci-dessous. Les plaques tectoniques qui ne sont pas concernées dans le processus de transfert sont supprimées, et la carte en sortie ne comprend pas *Hawaii*, en confirmant ce qui vient d'être dit plus haut.

La nouvelle carte (figure 7) partage beaucoup de similarité avec la couche de plaques tectoniques en entrée (figure 6), probablement parce que l'ère géologique visée n'est pas encore très éloignée de l'époque actuelle. Afin de bien démontrer l'efficacité du script, un nouvel essai a été lancé pour transférer les points de *ScoteseCities_000* vers il y a 442 millions d'années, et le résultat est montré par la figure 8.

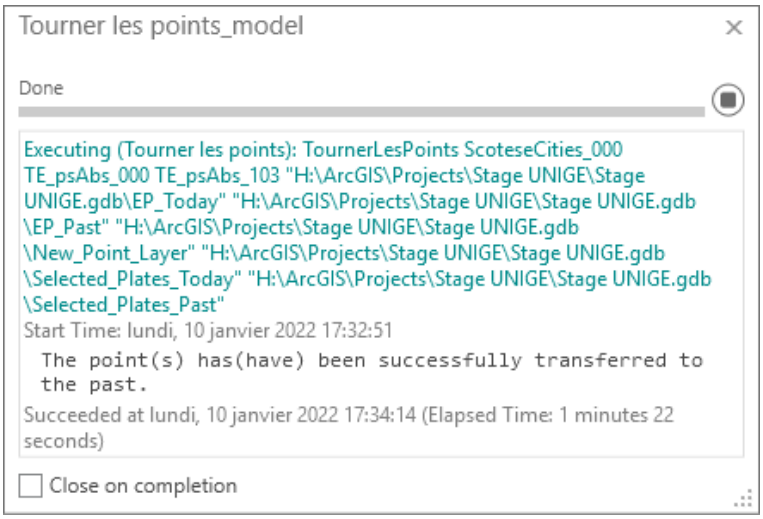

Figure 5 : Le message apparaissant à la fin de l'exécution du script qui indique à l'utilisateur que tous les points ont été transférés vers le passé avec succès.

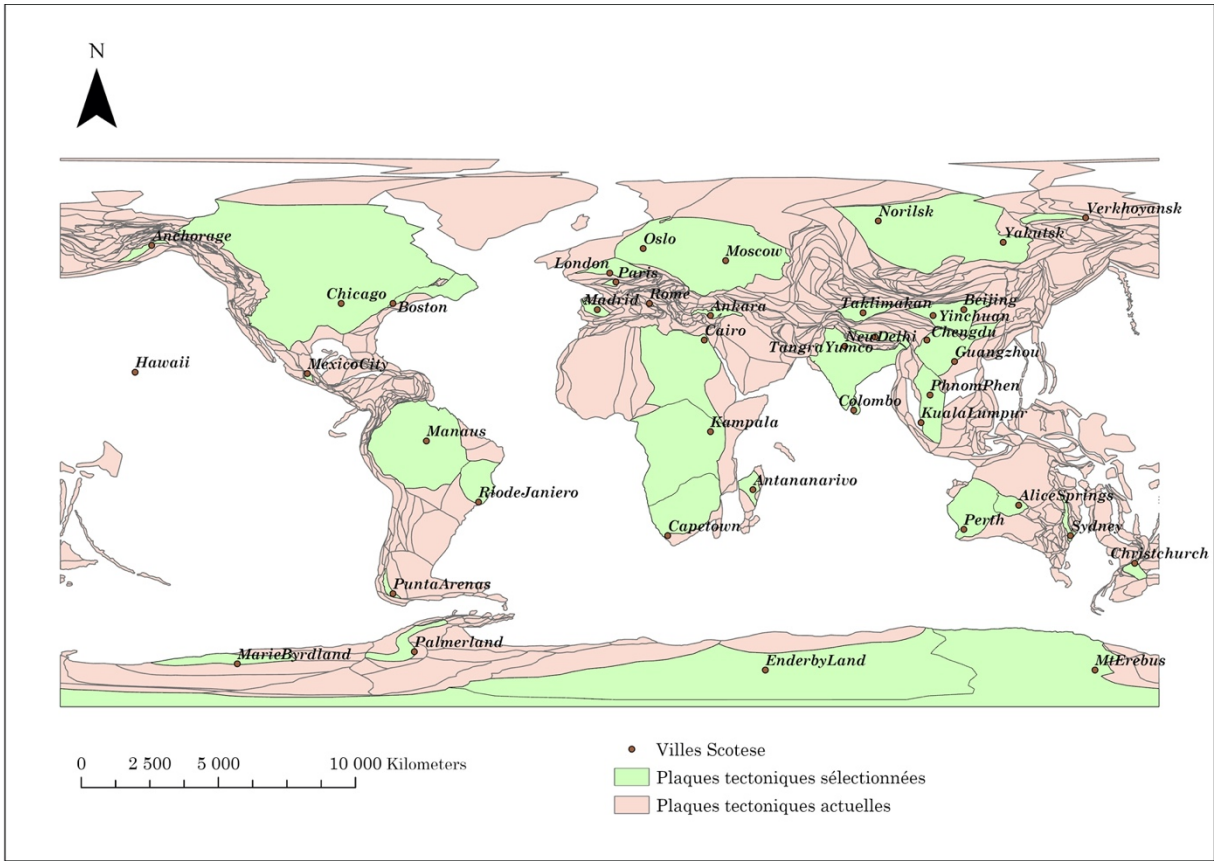

Figure 6 : La couche de points en entrée *ScoteseCities_000* et la couche de plaques tectoniques actuelles.

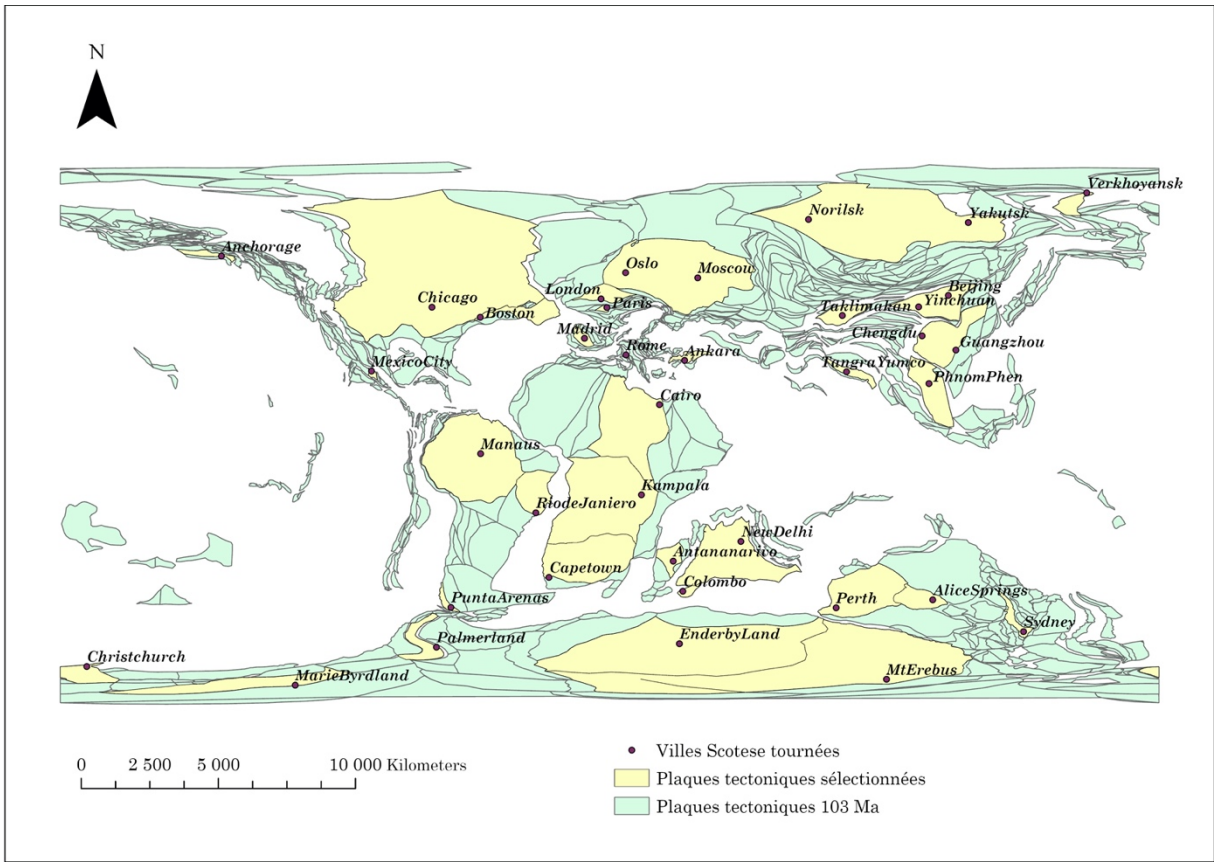

Figure 7 : Le résultat du transfert des points de *ScoteseCities_000* vers il y a 103 millions d'années.

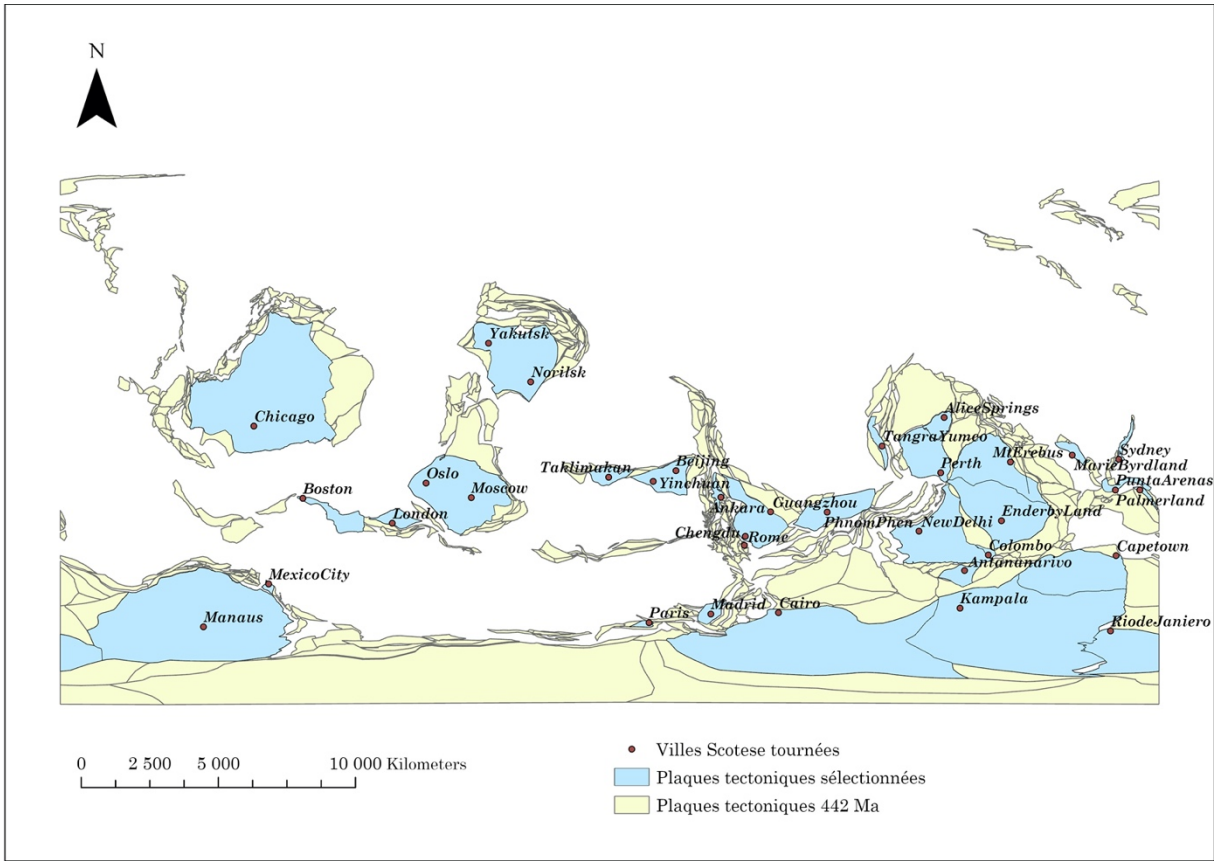

Figure 8 : Le résultat du transfert des points de *ScoteseCities_000* vers il y a 442 millions d'années.

Sur cette nouvelle carte, il est évident que le résultat est très différent de la couche initiale *ScoteseCities_000*. À titre d'exemple, vers le milieu de la nouvelle carte, Rome est très proche de la ville chinoise de Chengdu, ce qui est loin d'être la réalité d'aujourd'hui. Par ailleurs, quelques plaques tectoniques n'existaient pas encore à l'époque, les points qui tombent dans celles-ci ne peuvent donc pas être transférés et sont supprimés pendant l'exécution du script, ce qui est bien indiqué par le message final montré par la figure 9 en haut à droite.

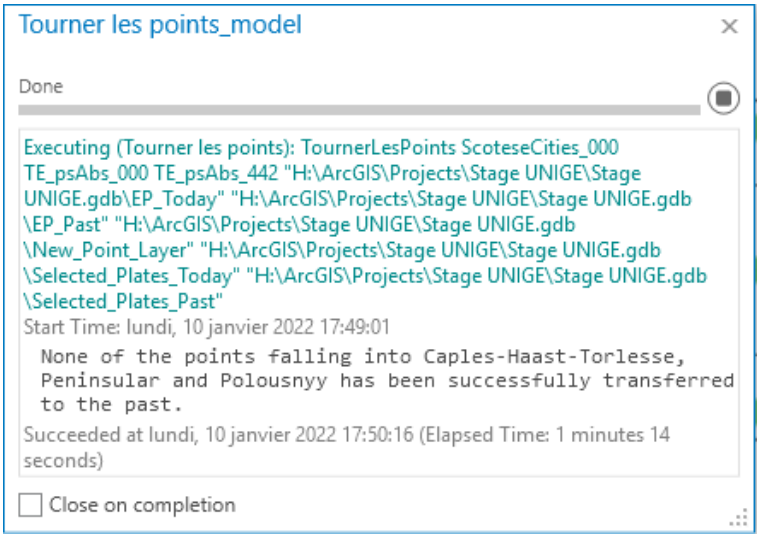

Figure 9 : Le message final annonçant qu'aucun point tombant dans les tectoniques des plaques de *Caples-Haast-Torlesse*, de *Peninsular* et de *Polousny* n'a pu être transféré vers il y a 442 millions d'années.

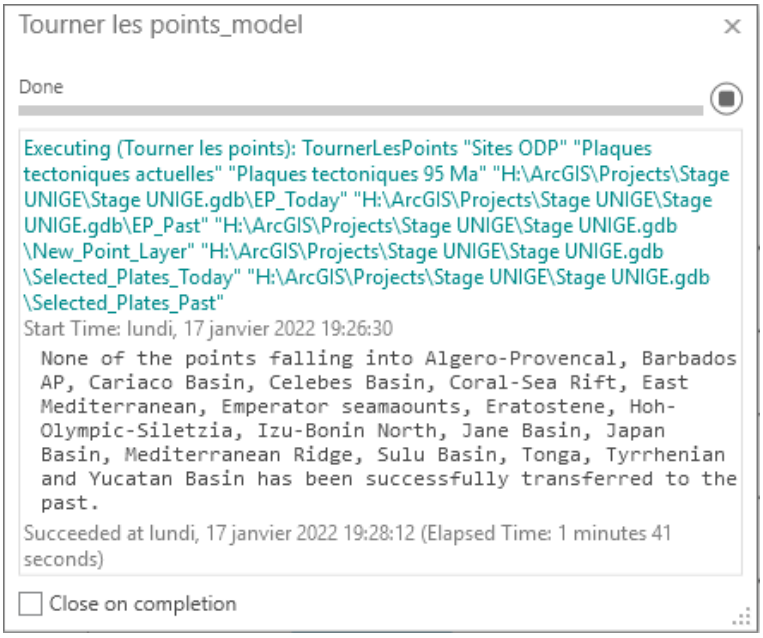

Figure 10 : Le message final qui indique quels points d'*ODPsites_000* n'ont pas pu être transférés vers le passé.

Ensuite, un autre test a été effectué sur la couche de points *ODPsites_000*. L'idée est de transférer ses points il y a 95 millions d'années. Les points et leur position initiale dans les plaques tectoniques d'aujourd'hui sont présentés par la figure 13.

Pour une question de visibilité, les noms de points ne sont pas affichés sur la figure 13. Faute de plaques tectoniques océaniques, seulement un petit nombre de plaques sont sélectionnées en vert. En revanche, il est évident que plusieurs plaques contenant des points ne sont pas sélectionnées. En effet, seules les plaques dont les points ont été transférés vers le passé avec succès sont gardées dans la sélection, c'est-à-dire qu'elles n'ont pas été ajoutées à la liste *unable_to_transfer* pendant l'exécution du script. Il est à rappeler que les points qui tombent dans aucune plaque tectonique sont directement supprimés. Le message qui apparaît à la fin du transfert des points d'*ODPsites_000* apparaît sur la figure 10, et le résultat du transfert est en bas présenté par la figure 14.

La figure 11 illustre la table attributaire de *New Point Layer*, la couche des points transférés nouvellement créée. Elle montre les champs à ajouter décrits plus haut, y compris le nom de la plaque tectonique, les coordonnées géocentriques du point, celles de l'axe de rotation, l'angle de rotation et finalement le niveau de confiance du pôle d'Euler calculé. Ce point se situant dans la plaque *Greenland* a un niveau de confiance de 1, autrement dit, sa rotation est censée être très précise. Ces champs peuvent être visualisés en un simple clic sur les points.

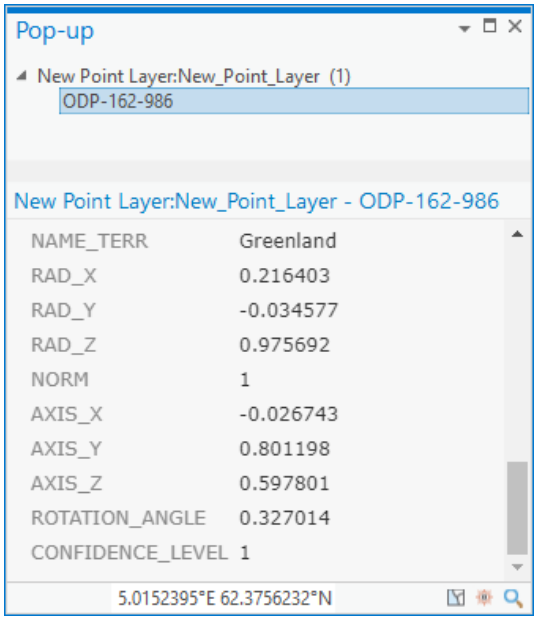

Figure 11 : Les champs rajoutés à la table attributaire de *New Point Layer*, la couche des points transférés créée lors de l'exécution du script. Ils comprennent entre autres le niveau de confiance du pôle d'Euler calculé – *CONFIDENCE_LEVEL* – décrit plus haut.

Finalement, un dernier test a été effectué sur la couche de points *DSDPsites_000* présentée

par la figure 15. Cette fois-ci, les points seront dans un premier temps transférés il y a 57 millions d'années.

Tout comme *ODPsites_000*, il y a énormément de points qui se trouvent dans les océans et qui sont donc directement supprimés lors des traitements. Certaines plaques tectoniques contenant des points ne sont pas sélectionnées en vert, c'est également parce qu'elles sont ajoutées à *unable_to_tranfer* à différentes étapes de l'exécution du script. Dans la figure 12 est communiqué le message concernant le transfert des points de *DSDPsites_000* il y a 57 millions d'années, tandis que la couche en sortie est présentée ci-dessous par la figure 16.

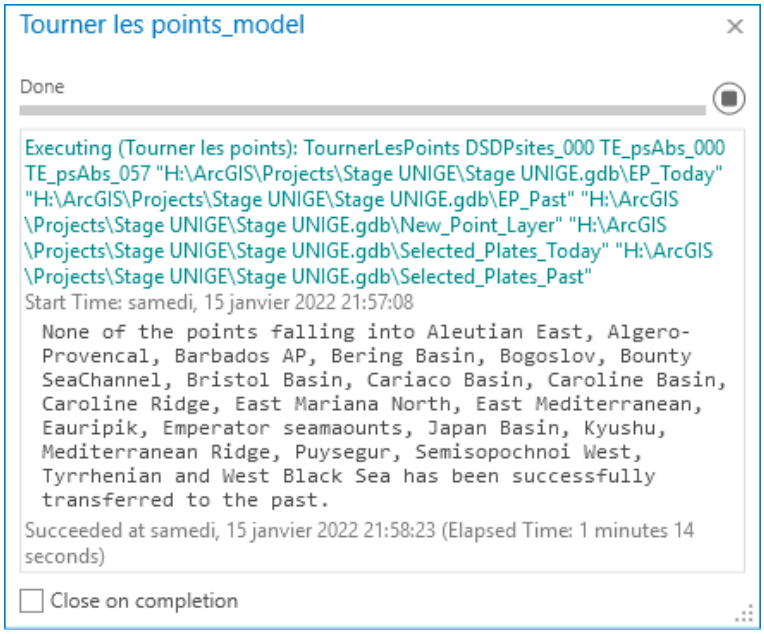

Figure 12 : Le message final qui indique quels points de *DSDPsites_000* n'ont pas pu être transférés vers le passé.

En comparaison avec la couche de points en entrée, uniquement un petit nombre de points ont pu être transférés vers la période visée. Dans la table attributaire dans la couche de points en sortie, le niveau de confiance est de 1 pour tous les points, c'est-à-dire que la rotation de tous les points est fiable. En vue de fournir une comparaison à ce résultat, les points de *DSDPsites_000* sont à nouveau transférés il y a 103 millions d'années, et le résultat est présenté ci-dessous par la figure 17.

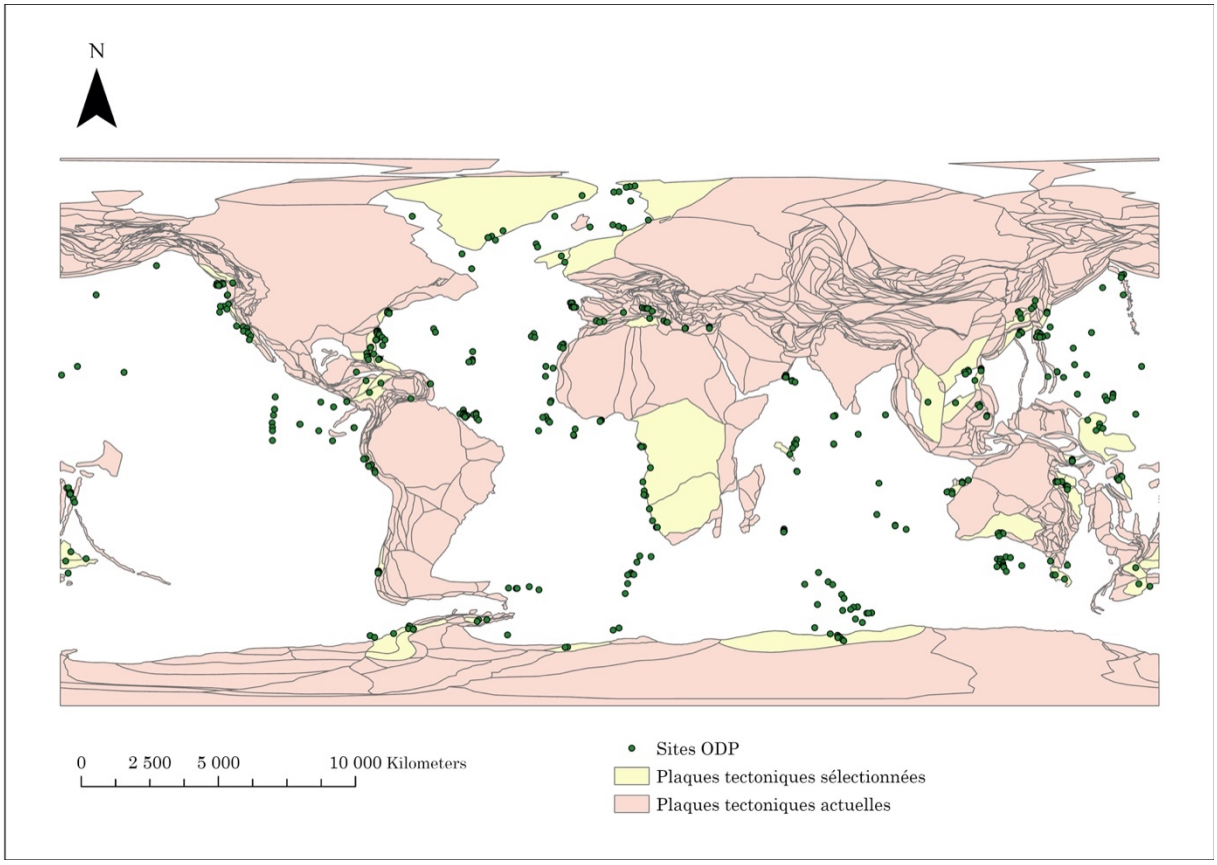

Figure 13 : Les points de la couche de points en entrée *ODPsites_000* et leur position dans le monde actuel.

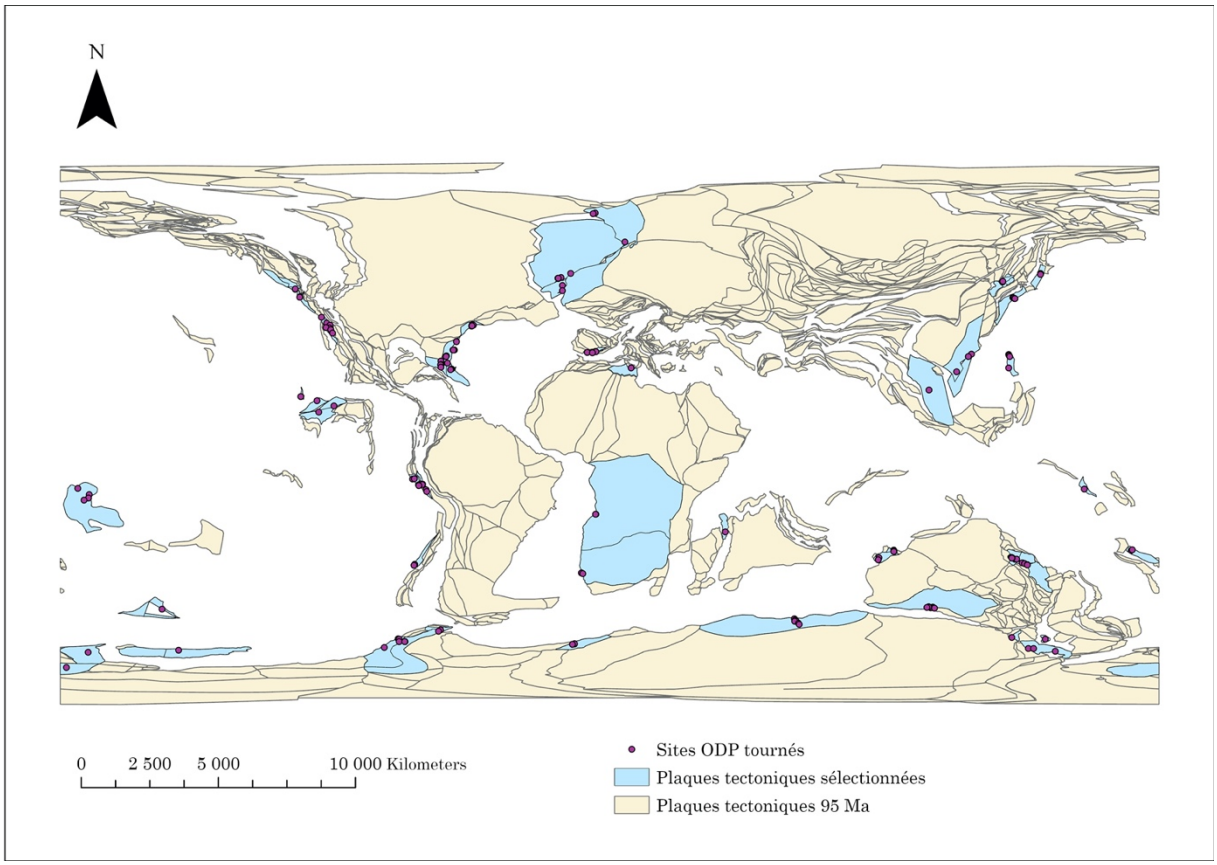

Figure 14 : Le résultat du transfert des points d'*ODPsites_000* vers il y a 95 millions d'années.

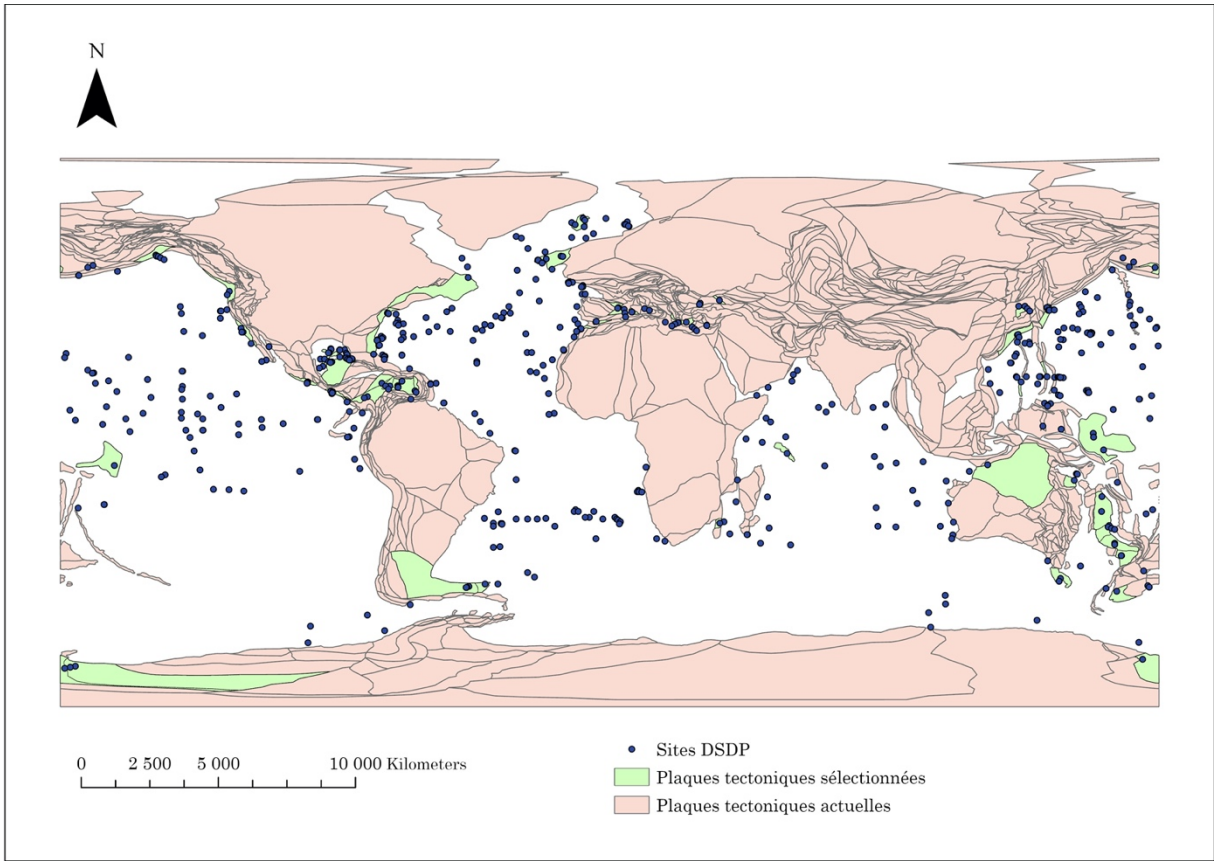

Figure 15 : Les points de la couche de points en entrée *DSDPsites_000* et leur position dans le monde actuel.

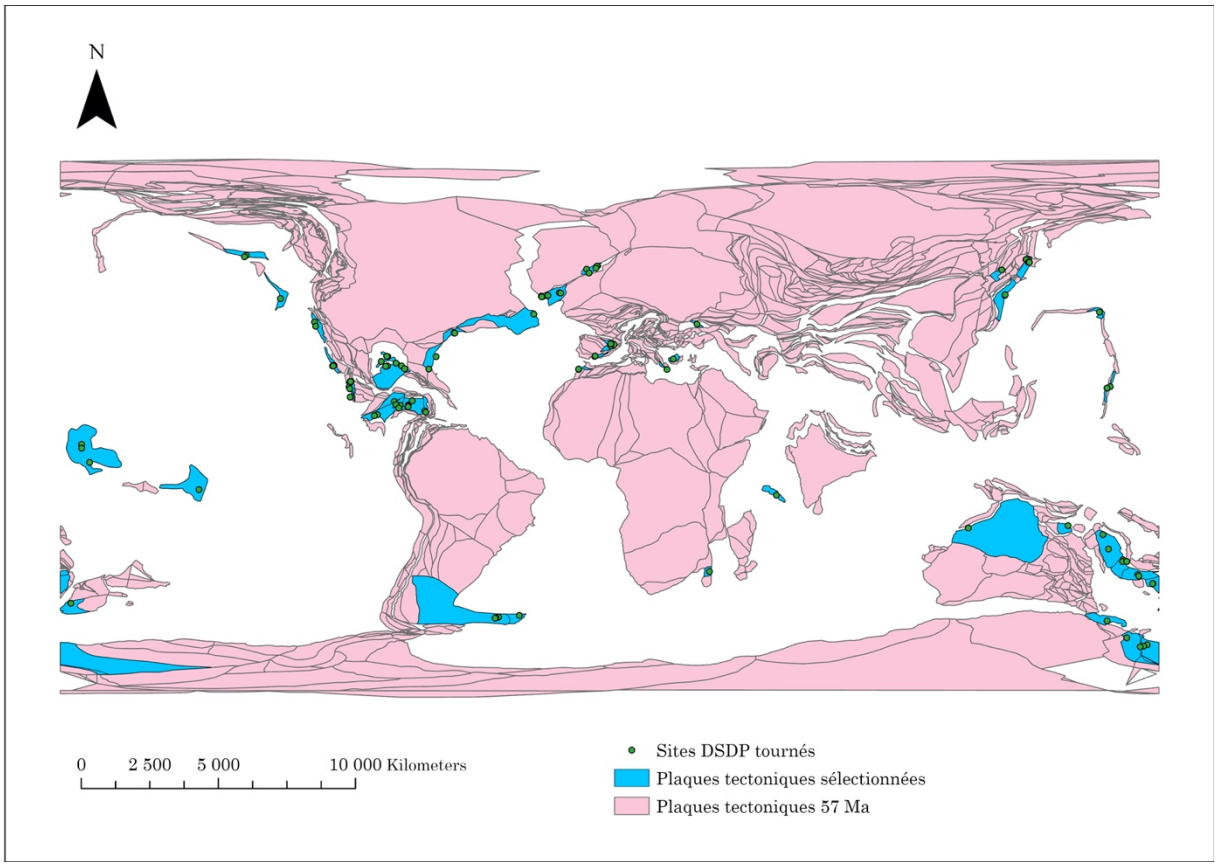

Figure 16 : Le résultat du transfert des points de *DSDPsites_000* vers il y a 57 millions d'années.

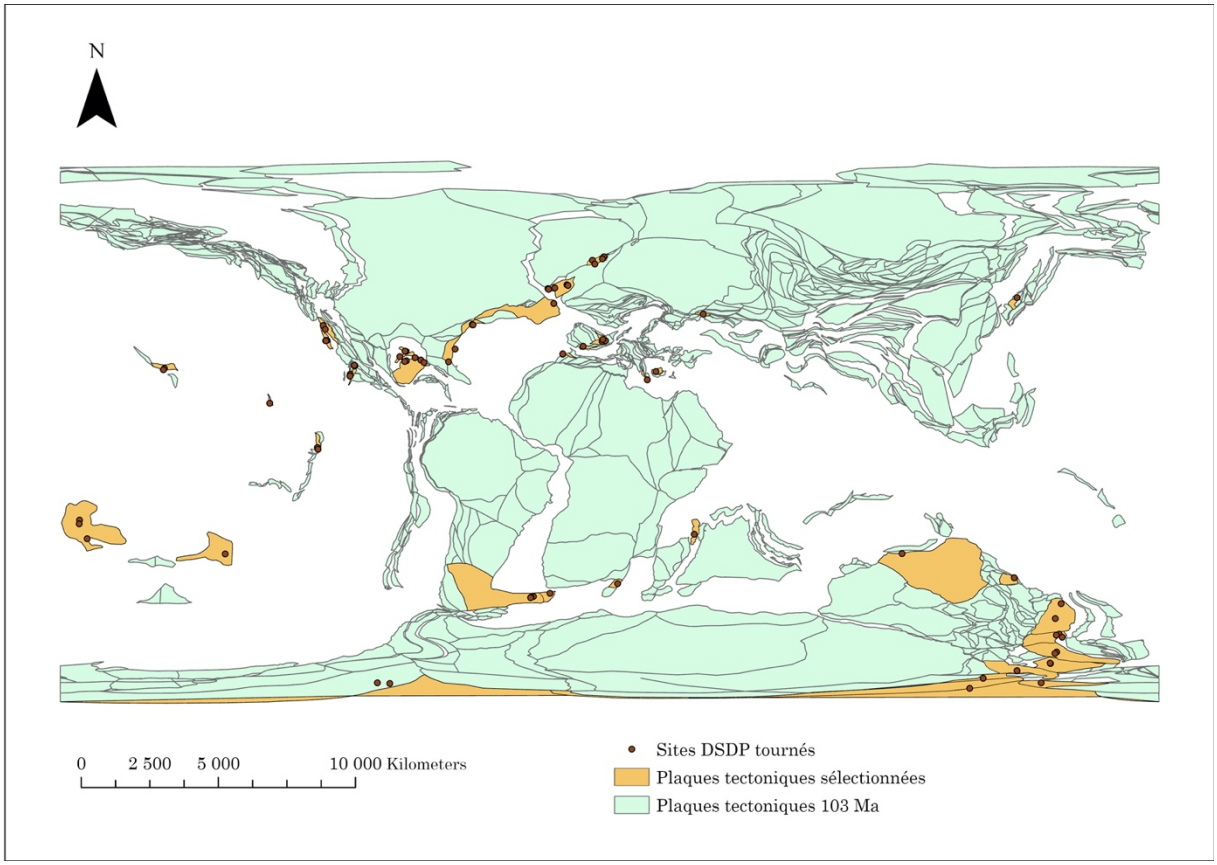

Figure 17 : Le résultat du transfert des points de *DSDPsites_000* vers il y a 103 millions d'années.

Dans la figure 17, il y a deux points en bas vers la gauche qui ne se situent dans aucune plaque tectonique sélectionnée (représentée en orange). Après un examen de la table attributaire de la couche de points en sortie, il s'avère que le niveau de confiance de ces deux points tombant dans la plaque tectonique *Byrd* n'atteint que 0,1, comme montré par la figure 18. Or lors du transfert il y a 57 millions d'années procédé tout à l'heure, tous les points avaient un niveau de confiance de 1. Par conséquent, il est important de consulter le niveau de confiance pour être sûr du résultat de transfert.

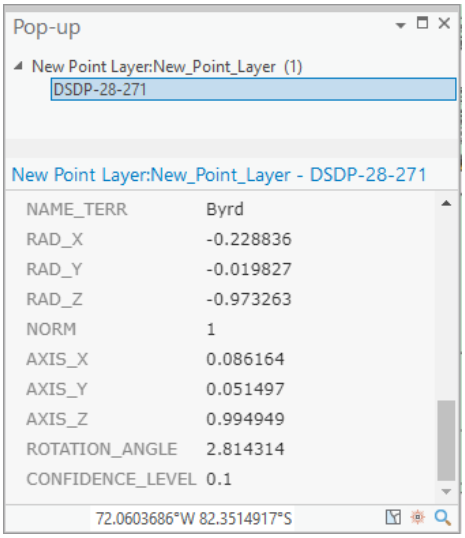

Figure 18 : Le niveau de confiance des deux points qui ne se situent dans aucune plaque tectonique sélectionnée en orange n'est que 0,1.

Discussion et limites

Le script a été testé sur trois couches de points, à savoir *ScoteseCities_000*, *ODPsites_000* et *DSDPsites_000*, ainsi que des couches de plaques tectoniques de différentes périodes géologiques. En général, le script fonctionne bien et donne le résultat désiré. Sur une couche de points en entrée testée, tous les points ont pu être transférés vers le passé ; tandis que sur les deux autres couches de points, les points tombant dans certaines plaques tectoniques n'ont pas été transférés vers le passé avec succès.

En effet, le script rencontre différentes erreurs lors du traitement de différentes couches de points en entrée. Malheureusement les erreurs n'ont pas pu être toutes identifiées ou résolues. Parmi les erreurs identifiées, la précision des calculs mathématiques de Python a parfois fait échouer l'exécution du script. Par exemple, la fonction *math.acos()*, utilisée à maintes reprises dans le script, ne prend que des chiffres dont la valeur absolue n'excède pas 1. Or lors du calcul du produit scalaire des points, la valeur que prend *math.acos()* parfois s'élève à 1,0000000000000002, alors qu'elle est normalement un chiffre égal ou inférieur à 1. Bien que cette valeur soit étroitement proche de 1, Python n'arrive pas à fonctionner correctement et la fonction *round()* a dû être utilisée pour y remédier. Cette erreur en termes de précision trouverait son origine au niveau matériel : les chiffres à virgule flottante exécutés par Python ne sont qu'inexactement représentés par des chiffres binaires relevant d'une approximation. L'erreur est d'ailleurs bien documentée sur le site web officiel de python.org [13]. Des mesures auraient pu être adoptées pour augmenter la précision des résultats du script, mais en raison du fait que c'était la première fois que j'utilisais un langage de programmation pour résoudre un problème réel, une solution simple consistait à utiliser la fonction *round()* pour régler l'erreur.

À part l'imprécision causée par la limite physique du matériel, le script contiendrait également des redondances qui peuvent être optimisées. Même si différentes exécutions visent à transférer différents points vers la même période du passé, le script doit être lu du début à la fin pour obtenir les pôles d'Euler. Si les couches de plaques tectoniques sont prédéfinies et l'utilisateur n'est pas libre de les choisir lui-même, la lecture totale du script ne sera pas forcément nécessaire. Parce que les pôles d'Euler des plaques tectoniques de différentes périodes géologiques peuvent être préalablement calculés et enregistrés quelque part pour la prochaine utilisation, ce qui raccourcirait largement le temps d'exécution du script.

[13] https://docs.python.org/fr/3/tutorial/floatingpoint.html

Le script actuel permet néanmoins de mettre à jour les couches de plaques tectoniques en entrée à tout moment. Par exemple, les couches utilisées pour le développement du script ne comprennent pas encore les plaques tectoniques océaniques. Si à un moment donné cela s'avère nécessaire, elles peuvent être rajoutées directement. En fonction des avancées dans les reconstructions des tectoniques des plaques, la bordure des plaques tectoniques pourrait encore changer, et pourrait ainsi être adaptée en conséquence. De plus, le script actuel permet à l'utilisateur de choisir sa propre version de reconstructions et d'effectuer le transfert de localisations, en lui laissant plus de liberté. Il est encore à noter que la procédure du script pourrait être améliorée afin de rendre son exécution plus rapide.

Par ailleurs, les couches de points testées comprennent des points qui tombent dans aucune plaque tectonique de la couche d'aujourd'hui, et ils sont supprimés lors des traitements pour cette raison. Cependant, il n'existe aucun mécanisme d'avertissement pour indiquer à l'utilisateur quels points ont été supprimés. Aussi, si un point rencontre un problème lors de sa rotation, il sera également supprimé sans aucun avertissement, alors que les autres points qui se trouvent dans la même plaque tectonique pourraient être tournés sans aucun souci. Il serait peut-être judicieux de développer un mécanisme d'indication à cette fin pour permettre plus de traçabilité des points.

Il serait également mieux de pouvoir automatiquement définir une symbologie commune pour les couches en sortie dans le script. Il est à noter que parmi les cinq couches en sortie, seuls *New Point Layer* et *Selected Plates Past* sont affichées à la fin de l'exécution du script, les trois autres étant impertinentes et offrant seulement la possibilité de vérification des résultats en cas de besoin. À part la définition automatique d'une palette de couleurs fixe pour les plaques tectoniques, il est également préférable d'afficher les points tournés en différentes couleurs en raison de leur niveau de confiance de transfert. Comme évoqué plus haut, le niveau de confiance montre si un pôle d'Euler est fiable. Une fois les pôles d'Euler obtenus, ils seront utilisés pour tourner les points, quel que soit leur niveau de confiance. Sans une symbologie appropriée, tous les points tournés sont affichés de la même couleur, ce qui ne donne aucun indice de leur niveau de confiance de transfert. Pour l'utilisateur, cela pourrait être trompeur et la consultation manuelle du niveau de confiance de transfert de chaque point serait une corvée. C'est pourquoi il serait plus facile de définir préalablement une symbologie dans le script avec le langage Python.

Un autre élément qui pourrait améliorer le script et refléter de manière plus appropriée la réalité serait de ne plus considérer la Terre comme une sphère parfaite alors que celle-ci a en réalité une forme elliptique. Dans le script actuel, la norme des coordonnées géocentriques de tous les points égale à 1 par défaut. Si

l'ellipticité de la Terre était prise en compte et que la norme des points sur l'équateur était considérée comme 1, plus un point serait proche des deux pôles, plus sa norme diminuerait. En effet, cela n'altérerait que plus ou moins le résultat des calculs des pôles d'Euler, ainsi que celui de la rotation des points, car les distances du centre de la Terre à l'équateur et aux deux pôles n'ont qu'environ 22 kilomètres de différence [14]. De ce point de vue, bien que le script n'offre qu'un reflet approximatif de la réalité, il est en fin de compte assez proche de cette dernière. Surtout cette approximation suffirait pour développer un prototype d'un outil de transfert de localisations vers le passé.

Finalement, il est évident que la disponibilité des données utilisées pour le développement du script est très limitée. Bien que ce dernier fonctionne déjà de manière satisfaisante, il devra certainement subir plus de tests avec des données en entrée plus diversifiées pour que sa stabilité et fiabilité soient convaincantes. Par exemple, à la préparation des données en entrée, il existe une étape qui uniformise la projection des couches avec l'outil *Projeter* d'ArcGIS Pro. Or en réalité, toutes les données utilisées pour le développement du script avaient déjà été reprojetées vers la même projection, et cette étape n'a été ajoutée que pour des raisons de précaution. Si une plateforme interactive du type site web est à mettre en place, la projection nécessite sans aucun doute de l'attention car une mauvaise projection est rédhibitoire pour la précision du transfert de localisations vers le passé. Plus de tests avec différentes sources de données feraient apparaître d'autres défauts qui n'ont pas encore été identifiés et qui ont besoin d'être résolus afin que le script puisse mieux servir à un grand public dans le futur.

[14] Source : https://www.space.com/17638-how-big-is-earth.html. Consulté le 12 janvier 2022.

Conclusion

Dans le cadre des reconstructions des tectoniques des plaques, ce travail a tenté d'explorer une première possibilité de mettre en place une plateforme interactive qui permet aux utilisateurs de fournir leurs données de terrain et de les comparer au modèle des tectoniques des plaques du projet PANALESIS lancé par C. Vérard. Cette comparaison s'effectue par le biais de transferts de localisations du monde actuel vers des périodes géologiques passées. Le travail s'est complètement déroulé dans ModelBuilder du logiciel SIG d'ArcGIS Pro, dans lequel un script issu du langage de programmation Python a été intégré afin d'achever le transfert de localisations. Le script a été développé à l'aide de trois couches de points qui comprennent des localisations à transférer et de quelques couches de plaques tectoniques représentant différentes périodes géologiques.

En général, le script développé est efficace. Malgré d'éventuelles erreurs, il arrive à transférer des localisations vers le passé à condition que les plaques tectoniques actuelles qui les contiennent existent à la période géologique du passé visée par le transfert. Cependant, le script pourra encore certainement être optimisé avec des données en entrée plus diversifiées que celles testées dans ce travail. Une fois cela réalisé, il pourra garantir que la plateforme interactive peut entrer en opération sans défaut et bénéficier à un grand public.

Bibliographie

Gurnis, M., Turner, M., Zahirovic, S., DiCaprio, L., Spasojevic, S., Müller, R. D., Boyden, J., Seton, M., Manea, V. C., & Bower, D. J. (2012). Plate tectonic reconstructions with continuously closing plates. *Computers and Geosciences, 38*, 35-42. https://doi.org/10.1016/j.cageo.2011.04.014

Gurnis, M., Yang, T., Cannon, J., Turner, M., Williams, S., Flament, N., & Müller, R. D. (2018). Global tectonic reconstructions with continuously deforming and evolving rigid plates. *Computers and Geosciences, 116*, 32-41. https://doi.org/10.1016/j.cageo.2018.04.007

Palin, R. M., & Santosh, M. (2021). Plate tectonics: What, where, why, and when? *Gondwana Research, 100*, 3-24. https://doi.org/10.1016/j.gr.2020.11.001

Vérard, C. (2019a). Plate tectonic modelling: review and perspectives. *Geological Magazine, 156*(2), 208-241. https://doi.org/10.1017/S0016756817001030

Vérard, C, (2019b). PANALESIS: towards global synthetic paleogeographies using integration and coupling of manifold models. *Geological Magazine, 156*(2), 320-330. https://doi.org/10.1017/S0016756817001042

Zhong, G., Zhang, D., & Zhao, L. (2021). Current states of well-logging evaluation of deep-sea gas hydrate-bearing sediments by the international scientific ocean drilling (DSDP/ODP/IODP) programs. *Natural Gas Industry B, 8*(2), 128-145. https://doi.org/10.1016/j.ngib.2020.08.001

Annexes

Le script développé est comme suit :

```
import arcpy
import math

arcpy.env.workspace = "H:\\ArcGIS\\Projects\\Stage UNIGE\\Stage UNIGE.gdb"

''' PART 1: DATA PREPARATION '''
# NO FRENCH CHARACTERS SHOULD APPEAR IN THE FILE, OTHERWISE ERRORS WHEN
OPENING IT
# original layers
entry_point = arcpy.GetParameterAsText(0)
entry_polygones_ajd = arcpy.GetParameterAsText(1)
entry_polygones_pas = arcpy.GetParameterAsText(2)

arcpy.management.SelectLayerByAttribute(entry_point, "CLEAR_SELECTION")
arcpy.management.SelectLayerByAttribute(entry_polygones_ajd,
"CLEAR_SELECTION")
arcpy.management.SelectLayerByAttribute(entry_polygones_pas,
"CLEAR_SELECTION")

pte1 = "Pte_Temp1"
ajd1 = "Ajd_Temp1"
pas1 = "Selected_Plates_Past"
arcpy.management.CopyFeatures(entry_point, pte1)
arcpy.management.CopyFeatures(entry_polygones_ajd, ajd1)
arcpy.management.CopyFeatures(entry_polygones_pas, pas1)

# reproject the point entry layer to the desired projection
# not sure if it is really necessary
pte2 = "Pte_Temp2"
arcpy.management.Project(pte1, pte2,
"GEOGCS['GCS_WGS_1984',DATUM['D_WGS_1984',SPHEROID['WGS_1984',6378137.0,298
.257223563]], PRIMEM['Greenwich',0.0],UNIT['Degree',0.0174532925199433]]")

''' PART 2: SELECTION OF TECTONIC PLATES '''
pte3 = "Pte_Temp3"
arcpy.analysis.SpatialJoin(pte2, ajd1, pte3, "JOIN_ONE_TO_ONE",
"KEEP_COMMON", "", "WITHIN")

# select tectonic plates intersected with the point entry layer
ajd2 = "Selected_Plates_Today"
ajd_extr = arcpy.management.SelectLayerByLocation(ajd1, "CONTAINS", pte3)
arcpy.management.CopyFeatures(ajd_extr, ajd2)

''' PART 2.1: GENERATION OF TERRAIN NAME LISTS '''
terrname_ajd = list()
terrname_pas = list()
unable_to_transfer = list()

# make terrain name list for 000
with arcpy.da.SearchCursor(ajd2, ["NAME_TERR"]) as un:
    for a in un:
        terrname_ajd.append(a[0])

# delete irrelevant terrains for PAST
```

```
with arcpy.da.UpdateCursor(pas1, ["NAME_TERR"]) as deux:
    for b in deux:
        if b[0] not in terrname_ajd:
            deux.deleteRow()

# make terrain name list for PAST
with arcpy.da.SearchCursor(pas1, ["NAME_TERR"]) as trois:
    for c in trois:
        terrname_pas.append(c[0])

# remove the unfound plates in the PAST for 000
with arcpy.da.UpdateCursor(pte3, ["NAME_TERR"]) as dix_huit:
    for r in dix_huit:
        if r[0] not in terrname_pas:
            dix_huit.deleteRow()

with arcpy.da.UpdateCursor(ajd2, ["NAME_TERR"]) as seize:
    for p in seize:
        if p[0] not in terrname_pas:
            seize.deleteRow()

for terrname in terrname_ajd:
    if terrname not in terrname_pas:
        unable_to_transfer.append(terrname)

''' PART 2.2: TURN POLYGONES INTO VERTICES '''
# intermediate layers created by "Feature Vertices to Points"
ajd3 = "Ajd_Temp3"
pas3 = "Pas_Temp3"
arcpy.management.FeatureVerticesToPoints(ajd2, ajd3, "ALL")
arcpy.management.FeatureVerticesToPoints(pas1, pas3, "ALL")
arcpy.management.Delete([pte1, ajd1, pte2])

''' PART 3: EULER POLE CALCULATION '''
# temporary layers
ajd = "Ajd_Temp"
pas = "Pas_Temp"

# final output files
ajd_f = "EP_Today"
pas_f = "EP_Past"

# constant used for transforming kilometers into degrees (360 degrees =
40075 km)
km2deg = 360 / 40075

arcpy.management.MakeFeatureLayer(ajd3, ajd)  # crucial step without which
the following steps cannot work as expected
arcpy.management.MakeFeatureLayer(pas3, pas)
arcpy.management.AddFields(ajd, [["RAD_X", "DOUBLE"], ["RAD_Y", "DOUBLE"],
["RAD_Z", "DOUBLE"], ["NORM", "DOUBLE"], ["PROD_SCAL", "DOUBLE"], ["PAIR",
"FLOAT"]])
arcpy.management.AddFields(pas, [["RAD_X", "DOUBLE"], ["RAD_Y", "DOUBLE"],
["RAD_Z", "DOUBLE"], ["NORM", "DOUBLE"], ["PROD_SCAL", "DOUBLE"]])

''' PART 3.1: FUNCTION CREATION '''
# calculate Cartesian coordinates in the attribute table
def AT_CalculateXYZ(layer):
    with arcpy.da.UpdateCursor(layer, ["SHAPE@XY", "RAD_X", "RAD_Y",
"RAD_Z", "NORM"]) as quatre:
        for d in quatre:
```

```
            x = d[0][0]
            y = d[0][1]
            d[1] = math.cos(math.radians(x)) * math.cos(math.radians(y))
            d[2] = math.sin(math.radians(x)) * math.cos(math.radians(y))
            d[3] = math.sin(math.radians(y))
            d[4] = math.sqrt(d[1] ** 2 + d[2] ** 2 + d[3] ** 2)
            quatre.updateRow(d)

# convert Cartesian coordinates into polar ones (x, y, z should be in
radians)
def XYZtoXY(point):
    x = point[0]
    y = point[1]
    z = point[2]
    r = math.sqrt(x ** 2 + y ** 2 + z ** 2)
    condition = math.sqrt(x ** 2 + y ** 2)
    if condition == 0:
        X = 0
    else:
        X = math.degrees(math.acos(x / condition))
    if y < 0:
        X = -X
    if r == 0:
        Y = 0
    else:
        Y = math.degrees(math.asin(z / r))
    return (X, Y, r)  # in degrees

# convert polar coordinates into Cartesian ones (x, y should be in degrees)
def XYtoXYZ(point):
    x = point[0]
    y = point[1]
    r = point[2]
    X = r * math.cos(math.radians(x)) * math.cos(math.radians(y))  # we
suppose the Earth is a perfect sphere, so r == 1
    Y = r * math.sin(math.radians(x)) * math.cos(math.radians(y))
    Z = r * math.sin(math.radians(y))
    N = math.sqrt(X ** 2 + Y ** 2 + Z ** 2)
    X = X / N
    Y = Y / N
    Z = Z / N
    # N = math.sqrt(X ** 2 + Y ** 2 + Z ** 2)
    return (X, Y, Z, 1)

# calculate rotation axis between point1(000) and point3(PAST)
def VectorProduct(point1, point2):
    x1 = point1[0]
    y1 = point1[1]
    z1 = point1[2]
    x2 = point2[0]
    y2 = point2[1]
    z2 = point2[2]
    x = y1 * z2 - y2 * z1
    y = z1 * x2 - z2 * x1
    z = x1 * y2 - x2 * y1
    r = math.sqrt(x ** 2 + y ** 2 + z ** 2)  # r != 1
    x = x / r  # normalise the vectors so that r == 1
    y = y / r
    z = z / r
    # r = math.sqrt(x ** 2 + y ** 2 + z ** 2)
    return (x, y, z, 1)
```

```
def DotProduct(point1, point2):
    x1 = point1[0]
    y1 = point1[1]
    z1 = point1[2]
    n1 = point1[3]
    x2 = point2[0]
    y2 = point2[1]
    z2 = point2[2]
    n2 = point2[3]
    formule = (x1 * x2 + y1 * y2 + z1 * z2) / (n1 * n2)
    if abs(formule) > 1:
        formule = round(formule)
    if n1 == 0 or n2 == 0:
        angle = 0
    else:
        angle = math.acos(formule)
    return angle  # in radians

def RotatePoint(point, axis, angle):
    x = point[0]
    y = point[1]
    z = point[2]
    xA = axis[0]
    yA = axis[1]
    zA = axis[2]
    cos = math.cos(angle)
    sin = math.sin(angle)
    X = ((cos + (1 - cos) * xA ** 2) * x) + (((1 - cos) * xA * yA - zA *
sin) * y) + (((1 - cos) * xA * zA + yA * sin) * z)
    Y = (((1 - cos) * xA * yA + zA * sin) * x) + ((cos + (1 - cos) * yA **
2) * y) + (((1 - cos) * yA * zA - xA * sin) * z)
    Z = (((1 - cos) * xA * zA - yA * sin) * x) + (((1 - cos) * yA * zA + xA
* sin) * y) + ((cos + (1 - cos) * zA ** 2) * z)
    N = math.sqrt(X ** 2 + Y ** 2 + Z ** 2)
    X = X / N
    Y = Y / N
    Z = Z / N
    # N = math.sqrt(X ** 2 + Y ** 2 + Z ** 2)
    return (X, Y, Z, 1)  # (x, y, z) in rad, N == 1

def GetTranslationPole(point1, point2):
    axis = VectorProduct(point1, point2)  # (xAxis, yAxis, zAxis) in rad, r
== 1
    angle = DotProduct(point1, point2)  # rotation angle in rad
    return axis, angle  # respectively in tuple and number

def GetSmallCircleAngle(point1, point2, point3):
    # between point1 and point2
    sc1 = VectorProduct(point1, point2)
    # between point1 and point3
    sc2 = VectorProduct(point1, point3)
    # calculate dot product between sc1 and sc2
    angle = DotProduct(sc1, sc2)
    return angle

def CheckAngle(point1, axis, angle, point2):
    # rotate [point1]
    point = RotatePoint(point1, axis, angle)
    # calculate distance between [rotated point1] and [point2]
    distance = math.degrees(DotProduct(point, point2))
```

```
    # precision == 0.1 km
    if not distance < 0.1 * km2deg:
        angle = -angle
    return angle

def AddPoles(point1, angle1, point2, angle2):
    point1 = XYZtoXY(point1)
    xp1 = point1[0]
    yp1 = point1[1]
    point2 = XYZtoXY(point2)
    xp2 = point2[0]
    yp2 = point2[1]

    if xp1 == 0 and yp1 == 0 and angle1 == 0 and xp2 == 0 and yp2 == 0 and
angle2 == 0:
        oaxis = (0, 0, 1)  # (x, y, r)
        oaxis = XYtoXYZ(oaxis)
        opole = 0
    elif xp1 == xp2 and yp1 == yp2:
        oaxis = (xp1, yp2, 1)
        oaxis = XYtoXYZ(oaxis)
        opole = math.radians(math.degrees(angle1) + math.degrees(angle2))
    else:
        lamda1 = math.radians(yp1)
        phi1 = math.radians(xp1)
        pX1 = math.cos(lamda1) * math.cos(phi1)
        pY1 = math.cos(lamda1) * math.sin(phi1)
        pZ1 = math.sin(lamda1)
        cos = math.cos(angle1)
        sin = math.sin(angle1)

        R111 = pX1 * pX1 * (1 - cos) + cos
        R121 = pX1 * pY1 * (1 - cos) - pZ1 * sin
        R131 = pX1 * pZ1 * (1 - cos) + pY1 * sin
        R211 = pY1 * pX1 * (1 - cos) + pZ1 * sin
        R221 = pY1 * pY1 * (1 - cos) + cos
        R231 = pY1 * pZ1 * (1 - cos) - pX1 * sin
        R311 = pZ1 * pX1 * (1 - cos) - pY1 * sin
        R321 = pZ1 * pY1 * (1 - cos) + pX1 * sin
        R331 = pZ1 * pZ1 * (1 - cos) + cos

        lamda2 = math.radians(yp2)
        phi2 = math.radians(xp2)
        pX2 = math.cos(lamda2) * math.cos(phi2)
        pY2 = math.cos(lamda2) * math.sin(phi2)
        pZ2 = math.sin(lamda2)
        cos = math.cos(angle2)
        sin = math.sin(angle2)

        R112 = pX2 * pX2 * (1 - cos) + cos
        R122 = pX2 * pY2 * (1 - cos) - pZ2 * sin
        R132 = pX2 * pZ2 * (1 - cos) + pY2 * sin
        R212 = pY2 * pX2 * (1 - cos) + pZ2 * sin
        R222 = pY2 * pY2 * (1 - cos) + cos
        R232 = pY2 * pZ2 * (1 - cos) - pX2 * sin
        R312 = pZ2 * pX2 * (1 - cos) - pY2 * sin
        R322 = pZ2 * pY2 * (1 - cos) + pX2 * sin
        R332 = pZ2 * pZ2 * (1 - cos) + cos

        T11 = R112 * R111 + R122 * R211 + R132 * R311
        T12 = R112 * R121 + R122 * R221 + R132 * R321
```

```
        T13 = R112 * R131 + R122 * R231 + R132 * R331
        T21 = R212 * R111 + R222 * R211 + R232 * R311
        T22 = R212 * R121 + R222 * R221 + R232 * R321
        T23 = R212 * R131 + R222 * R231 + R232 * R331
        T31 = R312 * R111 + R322 * R211 + R332 * R311
        T32 = R312 * R121 + R322 * R221 + R332 * R321
        T33 = R312 * R131 + R322 * R231 + R332 * R331

        xsump = math.degrees(math.atan((T13 - T31) / (T32 - T23)))
        ysump = math.degrees(math.asin((T21 - T12) / math.sqrt((T32 - T23)
** 2 + (T13 - T31) ** 2 + (T21 - T12) ** 2)))
        omegasump = math.degrees(math.atan(math.sqrt((T32 - T23) ** 2 +
(T13 - T31) ** 2 + (T21 - T12) ** 2) / (T11 + T22 + T33 - 1)))

        if T32 - T23 < 0:
            xsump = xsump + 180
        elif T32 - T23 > 0 and T13 - T31 > 0:
            xsump = xsump + 360

        if xsump > 180:
            xsump = xsump - 360

        if T11 + T22 + T33 < 1:
            omegasump = omegasump + 180
        if omegasump > 180:
            omegasump = omegasump - 360

        oaxis = (xsump, ysump, 1)
        oaxis = XYtoXYZ(oaxis)
        opole = math.radians(omegasump)
    return oaxis, opole

def RotationCheck(today, past, pairs, axis, angle):
    # today and past should be ajd_RAD and pas_RAD (points)
    # put smaller point collection as first collection. I think the smaller
collection will always be pas_RAD, considering
    # there are less points in it than in ajd_RAD. But it's possible that
len(pas_RAD) == len(ajd_RAD).
    g1 = past  # create variable [g1] and [g2] to better correspond to
Christian's code. This step is not needed
    g2 = today
    angle = -angle

    good = 0
    # with the dictionary [pairs], the last point in [ajd_RAD] and
[pas_RAD] will never be caught by the code
    # and there are only 10 matches max to check
    # so I think [NChecked] is not necessary here
    for key in pairs:
        pt1 = g1[pairs[key]]
        pt2 = g2[key]
        pt1 = RotatePoint(pt1, axis, angle)
        distance = math.degrees(DotProduct(pt1, pt2))
        # precision = 0.1 km
        if distance <= 0.1 * km2deg:
            good += 1

    ConfidenceRatio = good / len(pairs)
    return ConfidenceRatio

''' PART 3.2: POINT PAIRS IDENTIFICATION '''
```

```
# points' Cartesian coordinates in radians, data format: [point number: (X,
Y, Z)]
ajd_RAD = dict()
pas_RAD = dict()

# dot product, data format: [point number: dot product]
ajd_PS = dict()
pas_PS = dict()

pairs = dict()  # point matches, data format: [point number 000: point
number PAST]
euler_pole = list()
final_euler_pole = dict()

for nom in terrname_pas:
    ''' PART 3.2.1: DOT PRODUCT CALCULATION FOR TODAY'S LAYER '''
    AT_CalculateXYZ(ajd)

    with arcpy.da.SearchCursor(ajd, ["RAD_X", "RAD_Y", "RAD_Z", "NORM",
"OID@"], "NAME_TERR = '{}'".format(nom)) as six:
        for f in six:
            ajd_RAD[f[4]] = (f[0], f[1], f[2], f[3])
            dera = f[4] - 1
            try:
                # calculate the dot product for the 1st point until the
second to last point
                ajd_PS[dera] = math.degrees(DotProduct(ajd_RAD[dera],
ajd_RAD[f[4]]))
            except:
                continue

    # put dot products in the attribute table, for DATA VERIFICATION
    with arcpy.da.UpdateCursor(ajd, ["PROD_SCAL", "OID@"], "NAME_TERR =
'{}'".format(nom)) as sept:
        for g in sept:
            try:
                g[0] = ajd_PS[g[1]]
                sept.updateRow(g)
            except:
                continue

    ''' PART 3.2.2: DOT PRODUCTION CALCULATION FOR THE PAST LAYER AND POINT
PAIRS IDENTIFICATION '''
    AT_CalculateXYZ(pas)

    match = 0  # matches found, max 10 matches are wanted
    with arcpy.da.SearchCursor(pas, ["RAD_X", "RAD_Y", "RAD_Z", "NORM",
"OID@"], "NAME_TERR = '{}'".format(nom)) as huit:
        for h in huit:
            # once 10 matches have been found, exit the for loop
immediately
            if match == 10:
                break

            pas_RAD[h[4]] = (h[0], h[1], h[2], h[3])
            derb = h[4] - 1
            try:
                pas_PS[derb] = math.degrees(DotProduct(pas_RAD[derb],
pas_RAD[h[4]]))
            except:
                continue
```

```
            for key in ajd_PS:
                # precision: 0.001 km
                if abs(ajd_PS[key] - pas_PS[derb]) <= 0.001 * km2deg:
                    pairs[key] = derb
                    match += 1
                    ajd_PS.pop(key)  # delete the key in [ajd_PS] to avoid
possible confusions
                    break

    # if no match is found, the transfer is impossible
    if len(pairs) == 0:
        unable_to_transfer.append(nom)
        break

    # put the pairs in the attribute table of 000
    count = 0
    with arcpy.da.UpdateCursor(ajd, ["PAIR", "OID@"], "NAME_TERR =
'{}'".format(nom)) as neuf:
        for i in neuf:
            if count == len(pairs):
                break
            try:
                i[0] = pairs[i[1]]
                neuf.updateRow(i)
                count += 1
            except:
                continue

    # put dot products in the attribute table of PAST
    count = 0
    with arcpy.da.UpdateCursor(pas, ["PROD_SCAL", "OID@"], "NAME_TERR =
'{}'".format(nom)) as dix:
        for j in dix:
            if count == len(pas_PS):
                break
            try:
                j[0] = pas_PS[j[1]]
                dix.updateRow(j)
                count += 1
            except:
                continue

    ''' PART 3.3: EULER POLE CALCULATION '''
    # take 2 points from 000 and 2 points from PAST
    for pair in pairs:
        try:
            # 000: 1 and 2, PAST: 3 and 4
            pt1 = ajd_RAD[pair]
            pt2 = ajd_RAD[pair + 1]
            pt3 = pas_RAD[pairs[pair]]
            pt4 = pas_RAD[pairs[pair] + 1]

            # GET TRANSLATION POLE
            # calculate rotation axis and rotation angle between [pt1] and
[pt3]
            # axis = vector product (x, y, z, 1)
            # angle = dot product (angle in rad)
            # this is pole1
            axis, angle = GetTranslationPole(pt1, pt3)
```

```
            # rotate pt1 and pt2
            # output (x, y, z) in rad, r == 1
            pt1 = RotatePoint(pt1, axis, angle)  # [rotated pt1]
            pt2 = RotatePoint(pt2, axis, angle)  # [rotated pt2]

            # GET SMALL CIRCLE ANGLE
            # (1) sc1 = vector product of [rotated pt1] and [rotated pt2]
            # (2) sc2 = vector product of [rotated pt1] and [pt4]
            # (3) calculate dot product of [sc1] and [sc2]
            omega = GetSmallCircleAngle(pt1, pt2, pt4)

            # CHECK ANGLE
            # (1) rotate [rotated pt2] with [rotated pt1] as rotation axis
and [omega] as rotation angle
            # (2) calculate distance between [rotated rotated pt2] and
[pt4]
            # (3) condition to determine omega's value
            # pt3 and omega are together pole2
            omega = CheckAngle(pt2, pt1, omega, pt4)

            # ADD POLES
            axis1, pole1 = AddPoles(axis, angle, pt3, omega)

            # ROTATION CHECK
            level = RotationCheck(ajd_RAD, pas_RAD, pairs, axis1, pole1)

            EP = (axis1, pole1, level)  # data format: (axis (X_rad, Y_rad,
Z_rad, norm), angle, confidence level)
            euler_pole.append(EP)

        except:
            continue

        # CHECK SYMETRICAL SOLUTION
        # pt1 = ajd_RAD[pair + 1]
        # pt2 = ajd_RAD[pair]
        # axis, angle = GetTranslationPole(pt1, pt3)
        # pt1 = RotatePoint(pt1, axis, angle)
        # pt2 = RotatePoint(pt2, axis, angle)
        # omega = GetSmallCircleAngle(pt1, pt2, pt4)
        # omega = CheckAngle(pt2, pt1, omega, pt4)
        # axis2, pole2 = AddPoles(axis, angle, pt3, omega)

    # find the pole with the highest confidence level
    # the final pole will be the one with the highest confidence level
    if len(euler_pole) != 0:
        max_level = euler_pole[0][2]
        for pole in euler_pole:
            if pole[2] >= max_level:
                max_level = pole[2]
                final_pole = pole
        final_euler_pole[nom] = final_pole
    else:
        unable_to_transfer.append(nom)

    ''' PART 3.4 '''
    ajd_RAD.clear()
    pas_RAD.clear()
    ajd_PS.clear()
    pas_PS.clear()
    pairs.clear()
```

```
    euler_pole.clear()

arcpy.management.CopyFeatures(ajd, ajd_f)
arcpy.management.CopyFeatures(pas, pas_f)
arcpy.management.Delete([ajd, pas, ajd3, pas3])

''' PART 4: ROTATE POINTS '''
arcpy.management.AddXY(pte3)
arcpy.management.AddFields(pte3, [["RAD_X", "DOUBLE"], ["RAD_Y", "DOUBLE"],
["RAD_Z", "DOUBLE"], ["NORM", "DOUBLE"], ["AXIS_X", "DOUBLE"], ["AXIS_Y",
"DOUBLE"], ["AXIS_Z", "DOUBLE"], ["ROTATION_ANGLE", "DOUBLE"],
["CONFIDENCE_LEVEL", "FLOAT"]])
AT_CalculateXYZ(pte3)

pte = "Pte_Temp"
pte_f = "New_Point_Layer"
arcpy.management.MakeFeatureLayer(pte3, pte)  # very important step

ptct = dict()  # point count for each plate
failed = dict() # count the points who cannot be transferred within a plate
failedOID = list() # for deleting points who cannot be transferred
with arcpy.da.UpdateCursor(pte, ["RAD_X", "RAD_Y", "RAD_Z", "NORM",
"NAME_TERR", "SHAPE@XY", "AXIS_X", "AXIS_Y", "AXIS_Z", "ROTATION_ANGLE",
"CONFIDENCE_LEVEL", "OID@"]) as dix_neuf:
    for s in dix_neuf:
        tec_plate = s[4]
        if tec_plate not in ptct:
            ptct[tec_plate] = 1
        else:
            ptct[tec_plate] += 1

        try:
            pt2rotate = (s[0], s[1], s[2], s[3])
            final_axis = final_euler_pole[tec_plate][0]
            final_angle = final_euler_pole[tec_plate][1]

            pt2rotate = XYZtoXY(RotatePoint(pt2rotate, final_axis,
final_angle))  # data format: (x, y, 1)
            x_new = pt2rotate[0]
            y_new = pt2rotate[1]

            s[5] = (x_new, y_new)
            s[6] = final_axis[0]
            s[7] = final_axis[1]
            s[8] = final_axis[2]
            s[9] = final_angle
            s[10] = final_euler_pole[tec_plate][2]
            dix_neuf.updateRow(s)

        except:
            failedOID.append(s[11])
            if tec_plate not in failed:
                failed[tec_plate] = 1
            else:
                failed[tec_plate] += 1
            continue

for fail in failed:
    # if all the points in one particular plate failed to be transferred to
the past
    if failed[fail] == ptct[fail]:
```

```
        unable_to_transfer.append(fail)
        failed.pop(fail)

arcpy.management.CopyFeatures(pte, pte_f)
arcpy.management.Delete([pte, pte3])

''' PART 5 '''
# delete the plate in the "Selected Polygones Today"
with arcpy.da.UpdateCursor(ajd2, ["NAME_TERR"]) as vingt:
    for t in vingt:
        if t[0] in unable_to_transfer:
            vingt.deleteRow()
# delete the point in the output point layer
with arcpy.da.UpdateCursor(pte_f, ["NAME_TERR", "OID@"]) as vingt_et_un:
    for u in vingt_et_un:
        if u[0] in unable_to_transfer or u[1] in failedOID:
            vingt_et_un.deleteRow()

# final message indicating points that have not been transferred to the
past
messageU = ""
countU = 1
if len(unable_to_transfer) != 0:
    unable_to_transfer.sort()
    for plateU in unable_to_transfer:
        if countU == 1:
            messageU = plateU
            countU += 1
        elif countU != len(unable_to_transfer):
            messageU = messageU + ", " + plateU
            countU += 1
        elif countU == len(unable_to_transfer):
            messageU = messageU + " and " + plateU

messageF = ""
countF = 1
if len(failed) != 0:
    failed.sort()
    for plateF in failed:
        if countF == 1:
            messageF = plateF
            countF += 1
        elif countF != len(failed):
            messageF = messageF + ", " + plateF
            countF += 1
        elif countF == len(failed):
            messageF = messageF + " and " + plateF

if len(unable_to_transfer) == 0 and len(failed) == 0:
    arcpy.AddMessage("The point(s) has(have) been successfully transferred
to the past.")
elif len(unable_to_transfer) == 0 and len(failed) != 0:
    arcpy.AddMessage("At least one point falling into " + messageF + " has
not been successfully transferred to the past.")
elif len(unable_to_transfer) != 0 and len(failed) == 0:
    arcpy.AddMessage("None of the points falling into " + messageU + " has
been successfully transferred to the past.")
else:
    arcpy.AddMessage("At least one point falling into " + messageF + " has
not been successfully transferred to the past. The same goes for all the
points falling into " + messageU + ".")
```