Outils de visualisation et gestion de la topographie

Romain Mencattini

15 juin 2016

Table des matières

1 Introduction

Le but de ce projet est d'enrichir le projet Tetras [10] de simulation volcanique. Ce logiciel permet de simuler une éruption volcanique suivant des paramètres prédéfinis. Cependant, il ne tient pas compte du terrain et ne dispose pas d'outil de visualisation. C'est donc le but de ce travail. Pour faire cela, il a fallu effectuer plusieurs tâches qui vont brièvement être introduites dans cette section. Elles seront bien évidemment détaillés dans le reste du rapport. Les dites tâches sont :

— réaliser un outil pour visualiser un dépôt de cendres sur une carte
— charger des données d'élévations dans le logiciel de simulation
— visualiser des particules dans l'atmosphère

Ces trois tâches ont été réalisées dans différents langages et pour plusieurs raisons : **Python** pour l'outil de visualisation afin d'avoir une certain facilité et car cette application n'a pas d'impératif de performance ; **C++** pour le chargement des données, car il fallait s'interfacer avec le projet qui est également écrit dans ce langage.

Afin de bien traiter l'ensemble du sujet, voici le plan. Tout d'abord, nous allons resituer le contexte en faisant un état de l'art. Pour ce faire l'article intermédiaire, publié par l'équipe, sera très utile . Ensuite, il sera question des formats choisis, que ce soit au niveau des cartes ou bien des dépôts de cendres. Après cela, il y aura deux grandes parties consacrées chacune à une des implémentations. Pour finir, dans le section Discussion, nous discuterons des résultats obtenus et nous conclurons ce rapport.

2 État de l'art

Il est nécessaire d'avoir une idée de l'état du projet Tetras [10], lors de la réalisation du logiciel de visualisation, afin de connaître les enjeux. Pour ce faire, nous allons nous baser sur l'article publié le 5 Février 2016 [10].

Le nom Tetras vient de TEphra TRAnsport Simulator. Il s'agit donc de simuler le transport de *téphra*. Le téphra est le nom donné par les géologues pour les roches et les cendres éjectées lors d'une éruption. Le but est donc d'analyser la manière dont ces particules se dispersent dans l'atmosphère, et la façon dont elles se déposent. Afin de pouvoir comparer les résultats simulés avec des données géologiques réelles, les simulations présentées se basent sur des éruptions ayant eu lieu. Il s'agit de :

— Le volcan Ruapehu, en Nouvelle-Zélande en 1996.
— Le volcan Pululagua, en Équateur il y a 2450 ans.

Il est donc possible de s'appuyer sur des relevés géologiques de concentration de téphra sur le sol.

La motivation pour réaliser Tetras se trouvent dans le fait que les particules se déplacent. Elles peuvent causer des dommages aux animaux, au corps humain, et même engendrer des dégâts économiques sur les exploitations agricoles. Il est important d'avoir un outil

permettant d'estimer les risques, et cela passe par une modélisation du transport et de la sédimentation de téphra.

Le problème ,auquel a été confronté l'équipe, concerne le coût calculatoire. En effet, les éléments peuvent voyager sur de grandes distances suivant leur taille, leur forme et leur densité, ce qui engendre de grandes quantités de calcul. Le coût computationnel est très important. Cependant, avec l'augmentation de la puissance calculatoire et grâce à des algorithmes parallélisant les calculs, ce coût est devenu abordable. Il reste cependant un problème concernant le modèle utilisé. Le modèle Lagrangien qui se base sur une équation avec des variables dynamiques se parallélise mal. Spécialement, quand il y a des interactions entre les particules. Pour Tetras, un modèle hybride a été choisi. Ce modèle combine des particularités du modèle Lagrangien ainsi que du modèle Eulérien. Ce dernier consiste a séparer l'espace de travail, en blocs, avec un domaine discret, et d'échanger les informations via les parois des ces blocs. Il est bien plus facile de paralléliser un tel modèle. La combinaison de ces deux éléments produit le modèle utilisé pour la dispersion et la sédimentation. Il devient donc possible d'avoir un logiciel efficace tout en gardant une bonne flexibilité.

Cette flexibilité est importante car une éruption volcanique dépend d'énormément de paramètres qui peuvent varier. Les modèles d'éruptions font souvent des approximations qui combinées ensembles peuvent fausser la simulation. Il était donc essentiel d'avoir un logiciel flexible pour facilement modifier et ajouter des paramètres afin d'approcher au mieux le monde réel. Une approche modulaire a donc été utilisée afin de pouvoir permettre la modification et l'ajout d'éléments au modèle physique.

Nous allons voir dans la partie qui suit les différents formats utilisés pour le logiciel. Ces choix ont été dictés par les sources de données sus-nommées, il était donc important de les citer.

3 Technologies

3.1 Sources de données

Parlons maintenant des sources de données. Tout ce projet : le logiciel de visualisation ainsi que le code du terrain dans le simulateur, repose sur le fait qu'on ait à notre disposition des relevés topographiques. Cela a été le point de départ. Il y a un site qui s'est vite imposé, car il propose des relevés pour ,quasiment, la globalité de la planète. Ce site est : `http://srtm.csi.cgiar.org` [2]. Ce site dépend du *CGIAR*, qui est un consortium de recherche agricole pour lutter contre la faim. Une de leur branche la *CSI* fournit des informations spatiales. Où *CSI* signifie *Consortium for Spatial Information*. Ce groupe permet d'obtenir de manière gratuite et redistribuable des données d'élévations de la planète. Il manque cependant certaines parties du globe, comme l'Islande, le nord de la Russie et le nord de États-Unis. Pour combler ce manque, il existe une autre

source de données. Il s'agit de : http://viewfinderpanoramas.org/Coverage%20map%20viewfinderpanoramas_org3.htm [8]. Ces données sont également en libre accès, le mainteneur du site ne demandant que la citation de la source.

Il est signalé sur la page du *CGIAR* [16] ; que certaines régions très montagneuses , comme le Népal, n'ont pu être relevées de manières précises. Nous pouvons également citer le Sahara ainsi que certains étendues d'eau. Ce manque de données lors des premières mesures étaient très handicapants. Le consortium a donc ajouté une méthodologie d'interpolation. En voici les principales étapes :

— Fusionner toutes les mesures pour obtenir une surface continue.
— Nettoyer les creux et les pics de la surface.
— Interpoler les points manquants, les "trous".

Il est important de noter que ces deux sites ont pris leurs données du même endroit. En Février 2000, durant 6 jours, deux entités, la NASA et la NGA ont recueilli les données d'élévations depuis une altitude de 233 km. Après cela, les relevés ont été fournis en libre accès sous forme de fichiers matriciels et vectoriels [17].

3.2 Geotiff

3.2.1 Introduction

Le Geotiff est un format de fichier, qui est au coeur du projet. En effet, les données topographiques sont fournies dans ce format. Il est donc utilisé tantôt lors de la visualisation, afin de voir le terrain, tantôt dans le simulateur afin de récupérer les élévations.

Historiquement, le Geotiff a été utilisé pour remplacer le format Tiff. Selon les spécifications officielles [1], au vu des caractéristiques du format Tiff, il fallait inventer quelque chose de nouveau. En effet, l'ancien format était propriétaire, il était donc soumis à des conditions d'utilisations. Ce qui était un frein à l'adoption et à l'expansion. Il apparaissait également qu'il manquait des *tags* pour que celui-ci soit un outil vraiment optimal pour les géographes. Donc aux débuts des années 90 apparaît une spécification d'un : "TIFF geotie" [1]
Une phrase résume précisément qu'est-ce que représente un fichier Geotiff :

> The GeoTIFF spec defines a set of TIFF tags provided to describe all "Cartographic" information associated with TIFF imagery that originates from satellite imaging systems, scanned aerial photography, scanned maps, digital elevation models, or as a result of geographic analyses. [1]

Un format Geotiff est donc une image Tiff, sur laquelle on rajoute des *tags* afin d'ajouter des informations cartographiques utiles. Ces informations peuvent venir de différents supports et analyses.

Il est import de souligner le fait, que malgré cette possibilité apporté par le nouveau format, il n'y a que 6 nouveaux *tags*. Pour bien voir toutes les informations contenues dans un fichier Geotiff, nous allons prendre un exemple et l'examiner.

3.2.2 Manipulation Geotiff

Nous allons donc utiliser l'image "srtm_38_03.tif" [1] qui a été, au préalable, téléchargé sur le site http://srtm.csi.cgiar.org/.

En utilisant, l'outil *gdalinfo* [2]

```
gdalinfo srtm_38_03.tif
```

Nous obtenons un aperçu des Méta-données contenues dans le fichier. Dans notre cas, nous voyons ceci :

```
 Driver: GTiff/GeoTIFF
Files: srtm_38_03.tif
Size is 6001, 6001
Coordinate System is:
GEOGCS["WGS 84",
    DATUM["WGS_1984",
        SPHEROID["WGS 84",6378137,298.257223563,
            AUTHORITY["EPSG","7030"]],
        AUTHORITY["EPSG","6326"]],
    PRIMEM["Greenwich",0],
    UNIT["degree",0.0174532925199433],
    AUTHORITY["EPSG","4326"]]
Origin = (4.999583502803782,50.000417029852201)
Pixel Size = (0.000833333333333,-0.000833333333333)
Metadata:
  AREA_OR_POINT=Area
Image Structure Metadata:
  INTERLEAVE=BAND
Corner Coordinates:
Upper Left  (   4.9995835,  50.0004170) (  4d59'58.50"E, 50
   d 0' 1.50"N)
Lower Left  (   4.9995835,  44.9995837) (  4d59'58.50"E, 44
   d59'58.50"N)
Upper Right (  10.0004168,  50.0004170) ( 10d 0' 1.50"E, 50
   d 0' 1.50"N)
Lower Right (  10.0004168,  44.9995837) ( 10d 0' 1.50"E, 44
   d59'58.50"N)
Center      (   7.5000002,  47.5000004) (  7d30' 0.00"E, 47
   d30' 0.00"N)
Band 1 Block=6001x1 Type=Int16, ColorInterp=Gray
  NoData Value=-32768
```

1. Cette carte correspond à la Suisse et l'est de la France.
2. Commande disponible sur Ubuntu/Debian grâce au paquet gdal-bin

Nous voyons les 6 *tags* avec les informations géographiques :
— "Coordinate System" : définit le format des coordonnées.
— "Origin" : donne l'origine du terrain.
— "Pixel Size" : la distance à laquelle correspond un pixel de l'image.
— "Metadata" : savoir s'il s'agit d'un terrain ou de points.
— "Image Structure Metadata" : nous dit si l'entrelacement se fait par bande ou par pixel.
— "Corner Coordinates" : les coordonnées des quatre points extérieurs de l'image.

Le "WGS 84" ou "World Geodetic System 1984" est un système géodésique. Celui là même utilisé par GPS. Il a été crée par la *NIMA* ("National Imagery and Mapping Agency"). Ce type de représentations géodésique fournit [9] :
"to provide a single, common, accessible 3-dimensional coordinate system for geospatial data collected from a broad spectrum of sources"
Cela nous permet grâce à son système de données, d'avoir un *set* en trois dimensions, afin d'inférer la localisation de l'origine, l'orientation dans un système cartésien et l'échelle.

Voici une liste de commandes permettant de modifier un fichier *gdal* :
— "gdal_translate" : changer le format de l'image
— "gdal_merge.py" : fusionner plusieurs images/cartes
— "gdalwrap" : changer l'image elle-même, comme la résolution par exemple.
— etc.

Les manières de manipuler le fichier, ainsi que la gestion et l'utilisation des méta-données, seront abordées de manières plus précises dans la partie Logiciel de visualisation de dépôts et Code Simulateur.

3.3 HDF

3.3.1 Introduction

Ce format de fichier est utilisé par Tetras [10] afin de stocker les dépôts de cendres ainsi que la position et la vitesse des particules dans l'atmosphère. Nous allons présenter ses caractéristiques.

Le format HDF ,pour "*Hierarchical Data Format* ", est un ensemble de fichiers, permettant de stocker des données. Son avantage est qu'il est optimisé pour de grandes quantités de données et les accès parallèles. Ce qui en fait une solution viable pour le calcul scientifique. Dans notre cas, pour les cendres. Il a été créé par "National Center for Supercomputing Applications". Il est opensource sous licence de type BSD.

Le format HDF est auto-décrit et permet donc des relations complexes entre les éléments [3] Si on se réfère à la documentation officielle [3], le fichier HDF est un conteneur pour plusieurs fichiers qui peuvent être de types différents. Il y a deux objets de bases qui sont :
— *group*
— *dataset*

group **:** Ils contiennent un certain nombre d'instance. Ce nombre peut être nul. Les instances contenues sont des *dataset* ou bien d'autres *group*, le tout avec des méta-données.

dataset **:** Ce sont des tableaux multi-dimensionnels contenants des données, avec des méta-données.

La figure 1 [3.3.1] est une image tirée de la documentation HDF [3]. Elle représente une vue hiérarchique d'un fichier HDF quelconque :

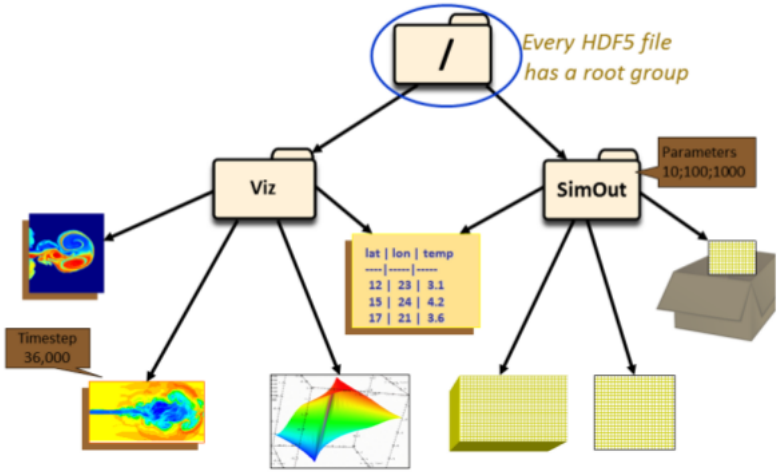

FIGURE 1 – Exemple d'une structure d'un fichier HDF

On voit que les "dossiers" sont les *group* qui contiennent d'autres *group* ou bien des *dataset*. Dans ce cas, les *dataset* sont les images, tableaux etc. présents sur l'exemple.

Comme énoncé plus haut : les *dataset* ont des méta-données en plus des données brutes. En effet, ces dernières permettent de décrire les données présentes. La figure 2 venant de la documentation HDF [3] pour illustrer cela :

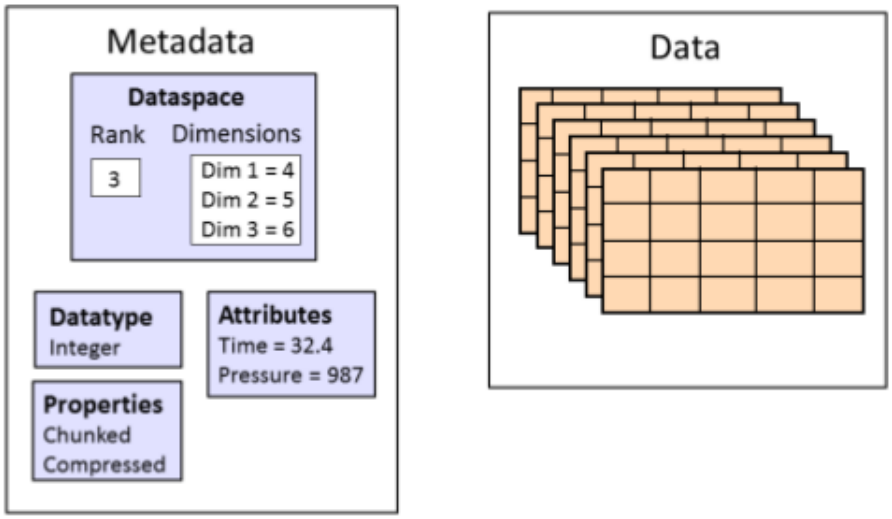

FIGURE 2 – Méta-données et données

Les types de ces *dataset* sont variés, en voici la liste [3] :
— Integer – des entiers.
— Float – en virgule flottante.
— Character – un tableau de caractère encodé de taille 1-byte.
— Variable - des types de séquences de longueurs variables.
— Reference – des références sur d'autres *datasets*.
— Enumeration – une liste de valeurs discrètes avec des noms symboliques.
— Opaque – non interprété.
— Compound (similar to C structs) – une séquence de *datatypes*.
— User-defined (eg, 13-bit integer or fixed/variable length strings) – un type définit par l'utilisateur.
— Nested types - des types imbriqués.

3.3.2 Manipulation HDF

Concernant la manipulation de ce format, la libraire standard fournit une bonne base pour manipuler un fichier .h5 en ligne de commandes sous Debian/Ubtunu. En effet, grâce aux utilitaires suivants :
— h5dump : afficher le contenu
— h5cc,h5c++ : compiler
— HDFView : visualisation

Il est également possible de les manipuler grâce à Python. Grâce à la librairie "h5py", on peut aisément ouvrir un fichier de format HDF, et le lire comme un dictionnaire. Il y a ,en effet, des clefs qui permettent d'accéder aux différents attributs. Dans le cas particu-

liers des cendres, chaque type de particule a un identifiant, et cet identifiant est la clef qui permet de trouver les données correspondantes. Voici la figure 3 qui montre ce principe : [3]

FIGURE 3 – Hiérarchie d'un fichier HDF

Nous voyons les clefs qui permettent d'accéder aux *datasets*.

Toujours avec cette même librairie, nous pouvons récupérer les méta-données dont nous avons parlé plus haut. Concernant les fichiers de cendres, il y a plusieurs données inscrites comme par exemple, le Δx entre deux points, le point d'origine, des valeurs d'atmosphères etc.

3. Ouvert grâce à "hdfview" sous Debian/Ubuntu

4 Logiciel de visualisation de dépôts

4.1 Introduction

Nous allons présenter les fonctionnalités et l'implémentation du logiciel de visualisation. Nous traiterons également des apis utilisées lors de l'implémentation.

Le but recherché est de pouvoir prendre un fichier de terrain (cf. la sous-section **Geotiff**), ainsi qu'un fichier de cendres (cf. la sous-section **HDF**), et d'afficher les cendres sur le terrain. Le choix du langage c'est porté sur Python car :

Premièrement, il dispose de librairies fortes utiles pour la réalisation de ce projet. Il s'agit de :

— numpy
— matplotlib
— scipy
— h5py
— gdal

La seconde raison est que le langage Python permet de s'abstraire des contraintes bas niveau tels que la gestion de la mémoire ou les pointeurs. Éléments présents dans le C++, par exemple. Cela offre donc une certaine facilité de codage. Il est également important de noter, qu'étant un langage répandu, il dispose d'une bonne documentation. Le fait que le reste des outils de visualisation du projet Tetras [10] sont écrits en Python est aussi une raison.

4.2 Librairies

Elles ont été mentionnées plus haut, détaillons les maintenant une à une. Afin d'expliquer, les raisons de leur présence.

4.2.1 Numpy

Numpy est la librairie de calcul scientifique la plus connue de Python. Elle offre un moyen de manipuler les données, comme du R ou du Matlab. On peut donc déclarer des matrices facilement :

```
myMatrix = numpy.matrix(([1,2],[3,4]))
```

Ce qui nous donne une matrice de cette forme : $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$

Là où *Numpy* prend tout son sens, c'est dans la manipulation ,rapide, de cette matrice. On prend :

```
# la transposer :
myMatrix.transpose()
# en obtenir le gradient :
numpy.gradient(myMatrix)
```

```
# en connaître la taille :
myMatrix.shape
# en prendre une sous-partie comme dans Matlab
myMatrix[1,0:1] #=> [3,4]
# addition des vecteurs entre deux
numpy.array([1,2,3]) + numpy.array([3,2,1]) # => array
    ([4,4,4])
# changer le type de la matrice :
myMatrix.astype(float)
# inverser les lignes avec
numpy.flipud(myMatrix)
#=> ([[3,4],
#=>  [1,2]])
```

Les éléments ci-dessus correspondent à toutes les fonctions utilisées dans le code du logiciel de visualisation. Nous y reviendrons mais les valeurs d'élévations ainsi que les densités de cendres sont représentées sous formes de matrices. Ces fonctions ont donc été nécessaires dans la manipulation de ces matrices.

4.2.2 Matplotlib

Cette librairie sert à tracer des graphiques de manières simples. Elle a pour avantage de fonctionner, presque, de la même manière que les *plot* de Matlab, et de donner des images propres et soignées. Comparé à la richesse de Matplotlib, le programme n'exploite qu'une infime partie Plus précisément :

```
# on crée une matrice avec numpy
myMatrix = numpy.random.rand(256,256)
# on trace avec matplotlib
matplotlib.pyplot.imshow(myMatrix)
# on trace un point en (50,50)
matplotlib.pyplot.plot(50,50)
# on affiche le tout
matplotlib.pyplot.show()
```

La fonction *imshow* du module *matplotlib.pyplot* sert à tracer des surfaces. On a donc une matrice, où les indices (x, y) correspondent aux composantes x, y d'un point en 3D, et que la valeur contenue en $myMatrix[x, y]$ correspond à la composante z du dit point. Nous avons donc un ensemble de points 3D, que cette librairie va nous représenter sous forme de surface pleine. Nous sommes exactement dans ce cas avec notre matrice de cendres/élévations. Les indices correspondent aux x, y et la valeur d'élévation ou du dépôt au z. Le *plot* ,quant à lui, permet de dessiner un point , en 2D. Ceci nous est utile pour représenter le centre du volcan. Et donc d'avoir un point de repère visuel. Le rendu n'est pas automatique, il faut appeler *matplotlib.pyplot.show()* pour pouvoir l'afficher.

Pour obtenir un rendu, un peu différent, tant au niveau des couleurs que des axes, il est possible de jouer avec les paramètres de *imshow*.

4.2.3 Scipy

Scipy est également une bibliothèque scientifique de calcul, pour Python. Elle est complémentaire de *Numpy*. Il n'y a qu'une seule fonction de cette librairie qui est appelée dans le logiciel. Il s'agit de *imresize* qui permet de redimensionner une image/matrice.

```
# on crée une matrice aléatoire
myMatrix = numpy.random.rand(100,100)
# on la redimensionne
myMatrix = scipy.imresize(myMatrix,(256,256),'cubic')
myMatrix.shape #=> (256,256)
```

Nous pouvons même choisir son ordre d'interpolation. En l'occurrence, il s'agit de l'ordre cubique, qui est le meilleur ratio calcul/rendu. Cela nous permet de redimensionner la matrice des cendres et donc faire en sorte qu'elle soit de même taille que la matrice d' élévations. Afin que le rendu soit optimal.

4.2.4 H5py

Pour pouvoir manipuler, les fichiers .h5 que Tetras [10] nous fournit, il est nécessaire d'utiliser cette librairie. Comme mentionné dans la partie sur le format **HDF**, elle permet d'ouvrir et de manipuler les méta-données. Voici un aperçu des méthodes qui utilisées dans le code Python :

```
# on ouvre le fichier
ash = h5py.File('file.h5')
# on peut récupérer un attribut:
terrain_position = ash.attrs.get('terrain_position')
# on peut récupérer les clefs correspondant aux données
key_list = ash.keys()
# prend les données correspondant à une clef
myMatrix = ash.get(key_list[0])
```

Dans l'exemple ci-dessus :

1. On ouvre le fichier
2. On récupère les méta-données utiles
3. On récupère la liste des clefs
4. On itère sur toutes les clefs pour avoir les données.

Cela permet donc *in fine* d'avoir une matrice avec l'addition de tous les types de cendres. Ainsi que les méta-données importantes comme la position du terrain ainsi que le Δx.

4.2.5 Gdal

À l'instar du code en C++, *Gdal* est un élément primordial dans la réalisation de ce projet. Elle permet d'ouvrir et surtout de manipuler des fichiers **Geotiff**. Nous pouvons donc accéder aux données et surtout aux méta-données. Comme dans les autres sous-sections, il va être question de montrer les fonctions utilisées ainsi que la manière de les appeler avant d'expliquer les raisons de ce choix.

```
# pour ouvrire une image
file = gdal.Open('File.tiff')
# on récupère la partie qui nous intéresse
band = file.GetRasterBand(1)
# on le lit comme une matrice
elevations = band.ReadAsArray()
```

C'est de cette manière que nous procédons pour obtenir la matrice d'élévations. Ensuite pour obtenir les méta-données, il faut :

```
geo_t = file.GetGeoTransform()
```

Cette structure contient ces éléments :

```
    geo_t[0] #=> top left x
    geo_t[1] #=> w-e pixel resolution
    geo_t[2] #=> rotation, 0 if it is "north up"
    geo_t[3] #=> top left y
    geo_t[4] #=> rotation, 0 if it is "north up"
    geo_t[5] #=> n-s pixel resolution
```

Il nous est donc possible ensuite d'obtenir les coordonnées des quatre coins de l'image. Et ainsi d'opérer les transformations nécessaires. Et ce de cette manière :

```
minx = geo_t[0]
miny = geo_t[3] + width_x * geo_t[4] + height_y * geo_t
    [5]
maxx = geo_t[0] + width_x * geo_t[1] + height_y * geo_t
    [2]
maxy = geo_t[3]
```

Grâce à ces éléments, il devient possible de convertir les distances utiles dans le code du logiciel de visualisation.

Voici donc les six librairies qui sont utilisées dans mon programme. Maintenant que j'ai expliqué les outils, je vais passer au coeur du programme.

4.3 Code Python

Nous allons dans un premier temps expliquer sous forme de pseudo-code, le déroulement du programme. Puis détailler les fonctions crées pour répondre à ce pseudo-code.

4.3.1 Pseudo-code

1. *récupération des arguments* : Il est nécessaire de récupérer les arguments comme l'image Geotiff, ainsi que le fichier de cendres, et la position du volcan. C'est sur ces éléments que nous allons travailler.
2. *application des transformations* : On transforme les éléments récupérés pour que les tailles soient les mêmes, ainsi que les référentiels, *i.e* être dans un système en longitude/latitude ou bien en longueur/largeur d'une matrice.
3. *traçage du graphe* : Une fois, les éléments transformés, il ne reste plus qu'à les tracer.

4.3.2 Fonctions

Voyons donc maintenant comment passer du pseudo-code à l'implémentation.

1. Afin de récupérer les arguments, nous avons :

```
parsing()
```

Cette fonction retourne un dictionnaire avec toutes les variables importantes du programme. C'est ce dictionnaire qu'on va passer en argument à toutes les autres fonctions, qui vont l'utiliser et le modifier. C'est ici que nous utilisons la librairie **Argparse**.

2. La partie sur les transformations concentre la majorité des fonctions. Il faut en premier lieu charger les cendres, si elles existent. Pour cela :

```
load_ash(returned_values)
```

Dans cette fonction, et à partir des valeurs contenues dans *returned_ values*, nous allons utiliser **H5py** et charger les cendres ainsi que les méta-données utiles :

— la position du terrain
— le Δx de la simulation

dans le dictionnaire, puis l'utiliser comme retour de fonction.

Ensuite, il faut adapter les référentiels. Nous désirons connaître les valeurs de longitudes et latitudes, ainsi que la longueur, la largeur, les valeurs maximales et minimales et la distance à laquelle correspond un élément de la matrice. Grâce aux opérations [4.2.5], nous obtenons les coins de l'images. Soit x_0,x_1,y_0,y_1, représentées en coordonnées. Nous pouvons ensuite obtenir la distance correspondante à une case de la matrice par cette formule :

$$step_x = |x_1 - x_0|/width$$
$$step_y = |y_1 - y_0|/height$$

Où *width* et *height* sont la longueur et la largeur de l'image. Enfin nous convertissons la distance de kilomètres à degrés grâce à cette formule [14] :

1 degré de longitude = 111.320*cos(latitude) km
1 degré de latitude = 110.574 km

Toutes ces opérations sont effectuées dans :

```
convert_information(returned_values)
```

Le retour consiste en notre dictionnaire qui a été mis à jour.

Nous devons également être capable de convertir un point latitude/longitude en un point x/y. Où ces dernières sont les indices de notre matrice. Avec les informations récupérées précédemment, il suffit de faire une règle de trois :

$$y = |lat - y_1]|/step_y$$
$$x = |lng - x_1]|/step_x$$

```
lat_long_2_x_y(lat,lng,returned_values)
```

Cela va nous donner une liste $[x, y]$.

Il faut que la matrice ait la taille voulue, avant d'être affichée. Nous allons donc la redimensionner :

```
x_0, y_0 = lat_long_2_x_y(x - width / 2, y - height /
   2, returned_values)
x_1, y_1 = lat_long_2_x_y(x + width / 2, y + height /
   2, returned_values)
```

Où x et y sont les coordonnées du point central. Puis il suffit de prendre les éléments compris entre ces bornes :

$$matrix_{new} = matrice_{old}[x_0 : x_1, y_0 : y_1]$$

```
rescale_matrix(elevation, returned_values)
```

On va recevoir une liste avec [elevation,returned_values].

3. Grâce à toutes ces fonctions, nous obtenons tout ce qui nous est nécessaire pour calculer et afficher le rendu. Afin d'avoir un rendu en simili-3D, on va calculer une ombre.

```
hillshade(array,azimuth, angle_altitude)
```

Cette fonction calcule une illumination en prenant en compte la position du soleil et son altitude [11], [12]. En premier lieu, le script calcule le gradient de l'image pour les deux directions : x,y. Ce qui nous donne deux matrices :

$$m_1, m_2 = \nabla(image)$$

Où m_1 et m_2 sont des matrices de même taille que *image*. Nous pouvons maintenant calculer l'aspect et la pente.L'aspect correspond à l'orientation des montagnes, lors de l'illumination à partir du *azimuth*. La pente ,quand à elle, changera le rendu suivant la valeur de l'angle. Cet angle représente la hauteur du "soleil".

Pour calculer le rendu final, il faudra :

$$rendu = sin(altitude) * sin(slope) + cos(altitude) * cos(slope) * cos(azimuth - aspect)$$

À partir de maintenant, nous pouvons simplement tracer les graphiques. Il y a deux fonctions, une pour tracer le terrain sans les cendres et une pour le terrain avec les cendres.

```
display_land_without_ash(returned_values)
display_land_with_ash(returned_values)
```

4.4 Exemple

Voici un exemple de code, afin d'afficher le terrain du Ruapehu ,en $km \times km$, ainsi que les cendres correspondantes à une éruption donnée :

```
./src/view.py --img file_exemple/map_ruapeu.tif --ash
   file_exemple/r90u40d1000da4500.h5 --posx 175.56 --posy
   -39.26 --alpha 0.5
```

Ce qui donne sur la figure 4 :

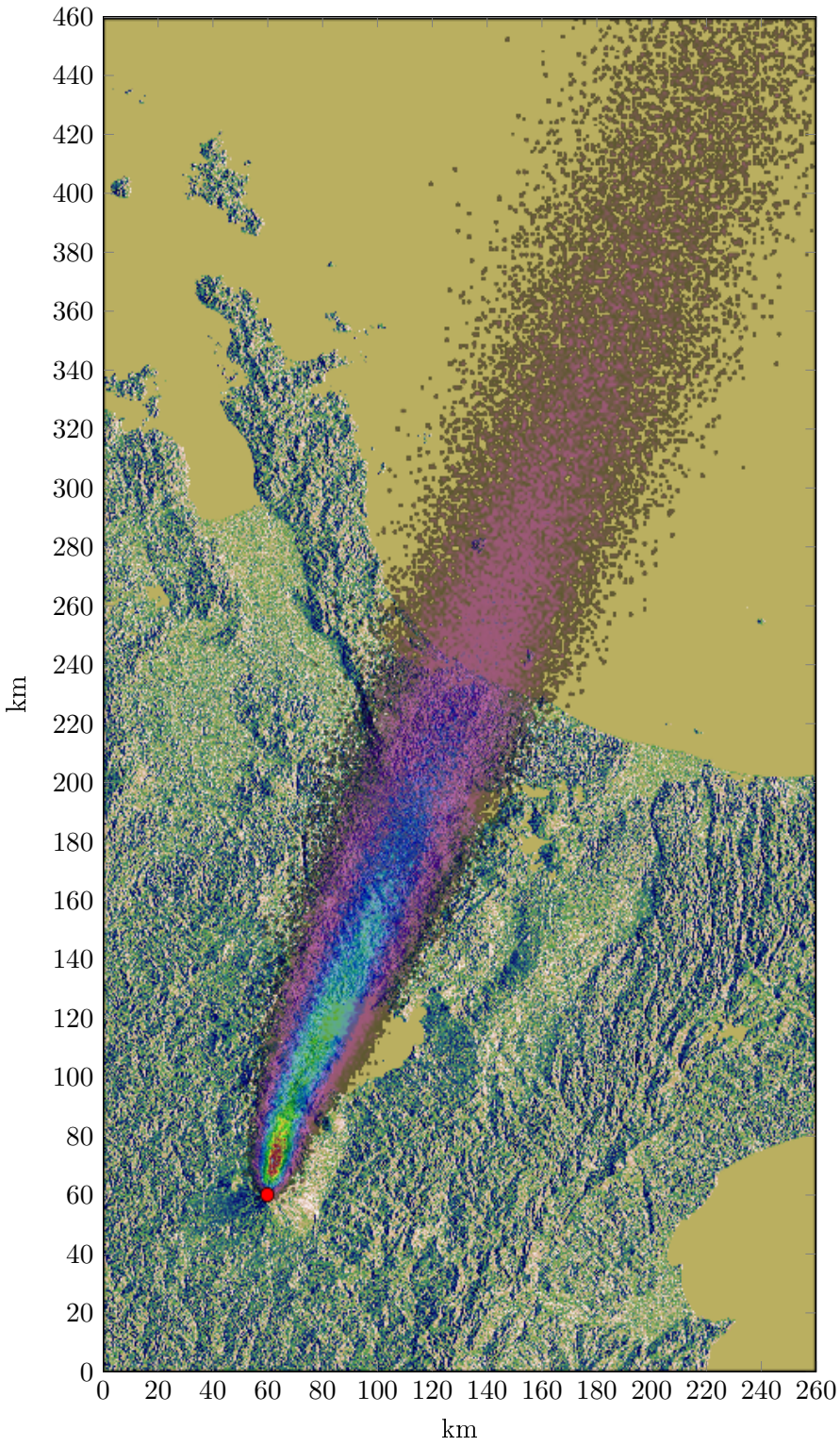

FIGURE 4 – Éruption du Ruapehu, graphique en $[km \times km]$

On remarque que les cendres sont déportées par du vent. Il a été introduit dans le Tetras pour une simulation d'une éruption du Ruapehu. Dans le fichier .h5 fournit, il a également d'autres paramètres comme :

- atmosphere_P0 = 101325.0 : la pression atmosphérique au niveau de la mer (Pa)
- atmosphere_T0 = 288.0 : la température atmosphérique au niveau de la mer (K)
- atmosphere_horizontal_diffusion = 4500.0 : le coefficient de diffusion des particules dans l'atmosphère (horizontal)
- atmosphere_tropopause = 16000.0 : l'altitude de la tropopause (m)
- atmosphere_vertical_diffusion = 4500.0 : le coefficient de diffusion dans l'atmosphère (vertical)
- column_horizontal_diffusion = 1000.0 : le coefficient de diffusion des particules dans la colonne volcanique (horizontal)
- column_vertical_diffusion = 1000.0 : coefficient de diffusion des particules dans la colonne volcanique (vertial)
- simulation_dt = 6.199999999999999 : le δt de la simulation
- simulation_dx = 500.0 : la taille en mètres lors de la simulation
- simulation_type = EXACT : le type de simulation
- terrain_dx = 500.0 : la taille en mètres d'une case de la matrice
- terrain_position = -30000.0,-30000.0 : la position du volcan dans l'espace
- tetras_version = simulation-2-43-gb35fca6bdde9 : la version du logiciel

5 Logiciel de visualisation des particules

5.1 Introduction

Il est apparu au cours du projet, qu'il était intéressant de pouvoir visualiser les particules rejetées par le volcan. Pour ce faire, nous avons utilisé ParaView [15]. C'est un logiciel opensource et multi-plateforme, de visualisation de données. Il est optimisé pour analyser et afficher des grandes quantités de données. Dans notre cas, il s'agit des particules volcaniques au court du temps. ParaView nous permet également de réaliser des vidéos à partir d'une suite d'image. Nous allons voir dans la prochaine sous-section comment transformer les données de Tetras [10], afin de les afficher en 3D dans le logiciel.

5.2 Transformations de données

ParaView ne peut lire et analyser que certains fichiers. Il faut donc transformer les fichiers *.h5* en un autre format compréhensible. Notre choix s'est porté sur le format *.vtk*. Un fichier est construit comme suit :

- un entête :
  "#vtk DataFile Version 3.0
  ASCII
  DATASET UNSTRUCTURED_GRID
  POINTS 432269"

Cet entête nous décrit le type de données (des points sur une grille). Ainsi que le nombre de points (ici 432269) et le format (ASCII).

— les données :
"-11143.05109166 -13784.89063692 2624.36392606"
Il y a donc 432269 lignes comme cela, avec les coordonnées x,y,z de chaque points/-particules.

Une fois, le fichier transformé de *.h5* à *.vtk*, il suffit de le charger dans ParaView pour visualiser les points. Pour faire la conversion d'un premier format vers le deuxième, nous avons écrit un script python nommé "h5ToVtk.py" qui peut s'appeler comme cela :

```
./h5ToVtk.py trackpart-21014.700000.h5 out.vtk 1
```

Où :

— *trackpart-21014.700000.h5* est le fichier de particule.
— *out.vtk* est notre fichier de sortie.
— *1* est l'index des données, s'il y a plusieurs conteneurs dans le *.h5*.

Il est possible de générer un ensemble de fichiers *.vtk* comme cela :

```
for e1 in `seq 1 90`:
do
  ./h5ToVtk.py trackpart-21014.700000.h5 out.vtk $e1
done
```

Nous obtiendrons donc une suite de fichier nommés : out[n].vtk, où n est l'index. Il est possible de charger directement cet ensemble de fichier en une fois dans Paraview. Il va les trier suivant leur index.

Pour construire un film à partir d'un fichier de particules, il faut

— transformer tout les index en *.vtk*.
— les charger ensemble dans ParaView.
— sauver l'animation.

Voir la figure 5 pour un exemple d'image générable à partir d'un fichier *.vtk* :

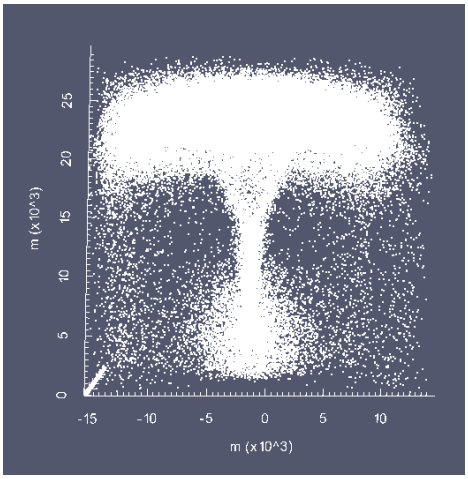

FIGURE 5 – Particules au temps 21014[s] après l'éruption du Volcan Pululagua.

6 Code Simulateur

6.1 Introduction

Le but du code en C++ est de pouvoir charger dans le simulateur, une partie d'un terrain donné afin de rendre la simulation plus proche de la réalité. Le choix du langage était donc obligatoirement le même que celui du simulateur afin d'obtenir la plus grande compatibilité possible. Les données sont stockées dans une structure en C++. On y retrouve les éléments nécessaires à la simulation comme ceux présentés à la section [4.4]. La structure de données pour le terrain est un tableau à deux dimensions de *doubles* représentant les élévations du terrain. Dont les dimensions correspondent à la taille du terrain passé en paramètre.

De part la structure du code déjà établie, le travail consistait à remplir le constructeur qui se trouvait :

```
piaf/src/Simulator/GridTerrain.cpp
```

Il y avait déjà un constructeur, cependant le terrain ,qui y était inséré, était plat.

```
for( int y = 0 ; y < discreteSize_.y_ ; y++ ){
   for( int x = 0 ; x < discreteSize_.x_; x++ ){
 terrain_[ index2d( { x , y } ) ] = 0.1 * dx_ ;
   }
}
```

Il fallu donc modifier cette partie. Il était également nécessaire d'ajouter certaines informations, d'où la modification de la structure de données. Nous y avons ajouté :

— la longitude : *lon_*
— la latitude : *lat_*

— le chemin jusqu'au fichier de terrain : *string path*

Détaillons maintenant les *apis* de la libraire GDAL [7], qui ont déjà été expliquées à la section [4.2.5]. Avant de passer sur l'implémentation du constructeur.

6.2 Implémentation

En premier lieu, il faut déclarer un pointeur du type de la structure GDAL, et d'initialiser :

```
GDALDataset* p;
GDALAllRegister();
```

Puis nous pouvons charger le fichier Geotiff :

```
p = (GDALDataset*) GDALOpen(path.c_str(), GA_ReadOnly);
```

Le *path* doit être un *string*. Grâce à cette structure, nous pouvons récupérer les valeurs maximales ainsi que les longueurs et largeurs :

```
GDALRasterBand *p_Band = p->GetRasterBand(1);
int nXsize = p_Band->GetXSize();
int nYsize = p_Band->GetYSize();
double width = p->GetRasterXSize();
double height = p->GetRasterYSize();
```

On peut aussi avoir accès à une structure semblable à celle de Python ; [4.2.5]. Elle nous sera, ici encore, très utile afin de transformer nos données pour obtenir la bonne partie du terrain.

```
double geo_t[6];
p->GetGeoTransform(geo_t);
```

Il ne nous manque plus que les valeurs d'élévations :

```
float *pafScanLine;
pafScanLine = (float*) CPLMalloc(sizeof(float)*nXsize*
   nYsize);
p->RasterIO( GF_Read, 0 , 0, nXsize-1, nYsize-1,
   pafScanLine, nXsize,nYsize, GDT_Float32,0,0);
```

Ces deux dernières lignes méritent des explications. Dans la première, on fait une allocation de la taille de la matrice entière, soit $nXsize$ éléments par $nYsize$ éléments.

Ensuite, on charge la matrice dans le tableau. On a l'index initial des deux composants qui vaut 0, ensuite ceux des derniers éléments en x et en y qui valent $nXsize-1, nYsize-1$. Après cela vient le tableau, puis le nombre d'éléments, soit $nXsize$ et $nYsize$, avec le type juste après. Les deux derniers 0 correspondent à l'espacement en terme de pixel. On laisse aux valeurs par défaut.

Nous avons donc tout les éléments nécessaires pour charger le terrain dans le constructeur puis dans la strcuture terrain.

6.3 Code C++

Nous allons aborder les étapes accomplies dans le constructeur.

1. Charger toutes les données depuis l'image. Il faut donc utiliser l'*api* Implémentation.

2. Traiter les informations afin d'obtenir les indices minimaux et maximaux de ma nouvelle matrice. Ce sont des conversions qui ressemblent très fortement à celles faites en Python, pour la simple raison que le but recherché était le même dans les deux cas.

3. Une fois ces indices obtenus, nous sous-échantillonnons la matrice (soit *pafScanLine*), avec un certain pas, afin que les données soient utilisables par le simulateur Tetras [10].

7 Conclusion

Le point de départ de ce travail a été la recherche de données topographiques. Afin de pouvoir visualiser et manipuler ces informations, il fallait un type de fichier qui disposaient d'un *api* répondant à nos besoins. Le format *geotiff* s'est imposé pour ces deux raisons.

Concernant l'implémentation de l'outil de visualisation le fait de disposer d'autant de librairies et de documentations a facilité l'implémentation. Les performances sont moyennes, ceci étant dû au fait qu'on a de grandes quantités des données et que Matplotlib et Numpy ne fonctionnent pas avec le même format de données. Du coup, cela prend du temps de passer des données brutes, à une image. Sans compter le fait que l'on applique le rendu ombré à toute la matrice d'élévations, ce qui réduit les performances. En dehors de ça, le logiciel est assez modulaire et donc pu être utilisé dans plusieurs cas. Il remplit le rôle voulu. C'est à dire de pouvoir visualiser des données topographiques avec ,ou sans, un dépôt de particules. On pourrait l'améliorer en le rendant modulable en temps réel ou bien en ajoutant une interface graphique.

Pour pouvoir visualiser les particules durant l'éruption, il a fallu écrire un outil de conversion de données. Une fois, cette tâche accomplie, il est possible de convertir les fichiers fournis par Tetras en des fichiers utilisables par Paraview et donc de représenter les particules. Paraview nous permet même de construire un film à partir de plusieurs images.

L'implémentation en C++ ayant été faite après le partie Python, il eut été possible de profiter des acquis concernant les transformations, ainsi que les données utiles et l'*api* GDAL [6.2]. La structure du simulateur était presque totalement adaptée pour contenir un "vrai" terrain, il n'a donc pas fallu modifier énormément de fichiers. Ce qui a facilité l'implémentation et surtout le déploiement. Il est donc désormais possible de donner un fichier de terrain *.geotiff*, et d'inclure ses données dans la simulation. Les éléments qui peuvent encore être implémentés afin de parfaire la simulation (du point de vue du terrain) sont :

— L'interpolation. En effet, il n'est actuellement pas possible de donner un fichier dont la taille est plus petite que le δx signifié lors de la simulation. Il peut être intéressant d'ajouter une interpolation afin d'obtenir les points manquants.
— L'intersection entre les cendres et le terrain. Pour l'instant elle est d'ordre 1. Le terrain est "en marches d'escaliers". Là aussi, afin d'accroître la précision du calcul, il faudrait augmenter l'ordre pour avoir une meilleur estimation des dépôts de cendres.

8 Bibliographie

1. Documentation officielle de Geotiff : http://www.remotesensing.org/geotiff/spec/geotiffhome.html
2. Site pour récupérer des données sur le monde entier : http://srtm.csi.cgiar.org/SELECTION/listImages.asp
3. Documentation HDF : "HDF5Intro.pdf" : https://www.hdfgroup.org/HDF5/Tutor/HDF5Intro.pdf
4. Spécifications du format HDF : https://www.hdfgroup.org/HDF5/doc/H5.format.html
5. Documentation Numpy & Scipy : http://docs.scipy.org/doc/
6. Documentation H5py : http://docs.h5py.org/en/latest/
7. Documentation GDAL : http://www.gdal.org/gdal_tutorial.html
8. Données élévations : http://viewfinderpanoramas.org/Coverage%20map%20viewfinderpanoramas_org3.htm
9. Document "WGS 84" : http://earth-info.nga.mil/GandG/publications/tr8350.2/tr8350_2.html
10. *Parallel simulation of particle transport in an advection field applied to volcanic explosive eruptions* : Pierre Künzli, Kae Tsunematsu, Paul Albuquerque, Jean-Luc Falcone, Bastien Chopard, Costanza Bonadonna
11. Relief avec python : http://rnovitsky.blogspot.ch/2010/04/using-hillshade-image-as-intensity.html
12. Relief avec gdal : http://geoexamples.blogspot.ch/2014/03/shaded-relief-images-using-gdal-python.html
13. CGIAR : http://www.cgiar.org/who-we-are/
14. Conversions kilomètres :degrés : https://fr.wikipedia.org/wiki/Coordonn%C3%A9es_g%C3%A9ographiques
15. Site officiel ParaView : http://www.paraview.org/
16. Interpolation et données CGIAR : http://srtm.csi.cgiar.org/SRTMdataProcessingMethodology.asp
17. Site SRTM : http://www2.jpl.nasa.gov/srtm/