

Projet Programmation

QUEL EST LE MEILLEUR ITINERAIRE POUR UN VOYAGE ?

LEMAGNEN ANTHIME - SEBASTIEN DORS

Table des matières :

I.	Contexte	2
1.	Complexité du problème	2
II.	Résolution.....	2
A.	Les premiers pas dans la résolution	2
B.	Monte-Carlo	3
III.	Programme.....	4
A.	Logigramme	4
B.	Pseudocode	5
C.	Code.....	6
D.	Explication des parties.....	6
1.	Les Villes	6
2.	Distance	6
3.	Trajet	6
4.	Ajouts.....	7
a)	Affichage.....	7
b)	Vitesse moyenne	7
c)	Emission de CO ₂	7
d)	Changement de ville initiale	8
e)	Optimisation	8
5.	Code supplémentaire	9
a)	Utilisation de fonctions	9
b)	Interface Graphique	9
6.	Analyse de résultat	10
a)	Kilométrage en fonction du nombre d'itérations.....	10
b)	Temps de voyage en fonction du nombre d'itérations :	11
c)	Temps d'exécution du programme en fonction des occurrences :	11
IV.	Conclusion	13
V.	Annexes	14
VI.	Bibliographie.....	16

I. Contexte

Notre entreprise a été démarchée afin de créer un programme. Ce programme pourra être implémenté dans un GPS, une compagnie de transports (ferroviaire, aérienne, ...).

Cet algorithme consiste à trouver le meilleur itinéraire pour un road trip. Les distances se feront à vol d'oiseau. Le trajet doit passer par toutes les villes avec une contrainte, partir de Paris et revenir à Paris.

L'objectif sera de créer un code python permettant de trouver le trajet le plus court parmi les 30 plus grandes villes de France.

1. Complexité du problème

Tout d'abord il faut savoir que pour passer dans les 30 villes en ne passant qu'une seule fois dans chacune d'elles. Il y a $29!$ soit $8.84 \cdot 10^{30}$ de possibilités. Ce nombre est trouvé par la relation factorielle ($1 \times 2 \times 3 \times 4 \times 5 \times 6 \times \dots \times 28 \times 29$).

II. Résolution

A. Les premiers pas dans la résolution

Afin de résoudre ce programme nous devons commencer par poser le problème.

Nous commençons par simplifier le problème. Nous avons réduit le nombre de villes à 5. En généralisant l'algorithme pour 5 villes, cela fonctionnera pour autant de villes que nous souhaitons, 30 y compris.

Pour démarrer le code et après s'être posé la question "Quelle est la partie la plus importante dans le programme ?", nous avons dressé une liste des 5 premières villes contenant leurs noms, longitudes et latitudes.

La suite en a découlé petit à petit et surtout dans la logique car toutes les parties étaient nécessaires à un gros puzzle (La distance entre les villes, la comparaison entre les meilleurs trajets...).

B. Monte-Carlo

Le squelette de l'algorithme se base sur la méthode Monte-Carlo. Ne connaissant pas ce procédé, nous devons tout d'abord apprendre à la connaître. Après quelques vidéos explicatives, programmes et pages internet nous en avons cerné la méthode, "Le hasard".

Cette stratégie assez crue consiste à miser sur le hasard et les probabilités.

Contextualisation :

L'algorithme essaiera toutes les combinaisons possibles de routes. Comme exemple prenons trois villes : Paris, Marseille, Lille, Montpellier. En simulant les 6 combinaisons possibles (Nous ne comptons pas Paris), le trajet le plus court sera : Paris -> Montpellier -> Marseille -> Lille -> Paris.

Nous en concluons que notre programme devra parcourir et tester toutes les possibilités pour trouver le meilleur itinéraire.

III. Programme

A. Logigramme

Après avoir mis ordonnées nos idées nous avons réalisé ce logigramme :

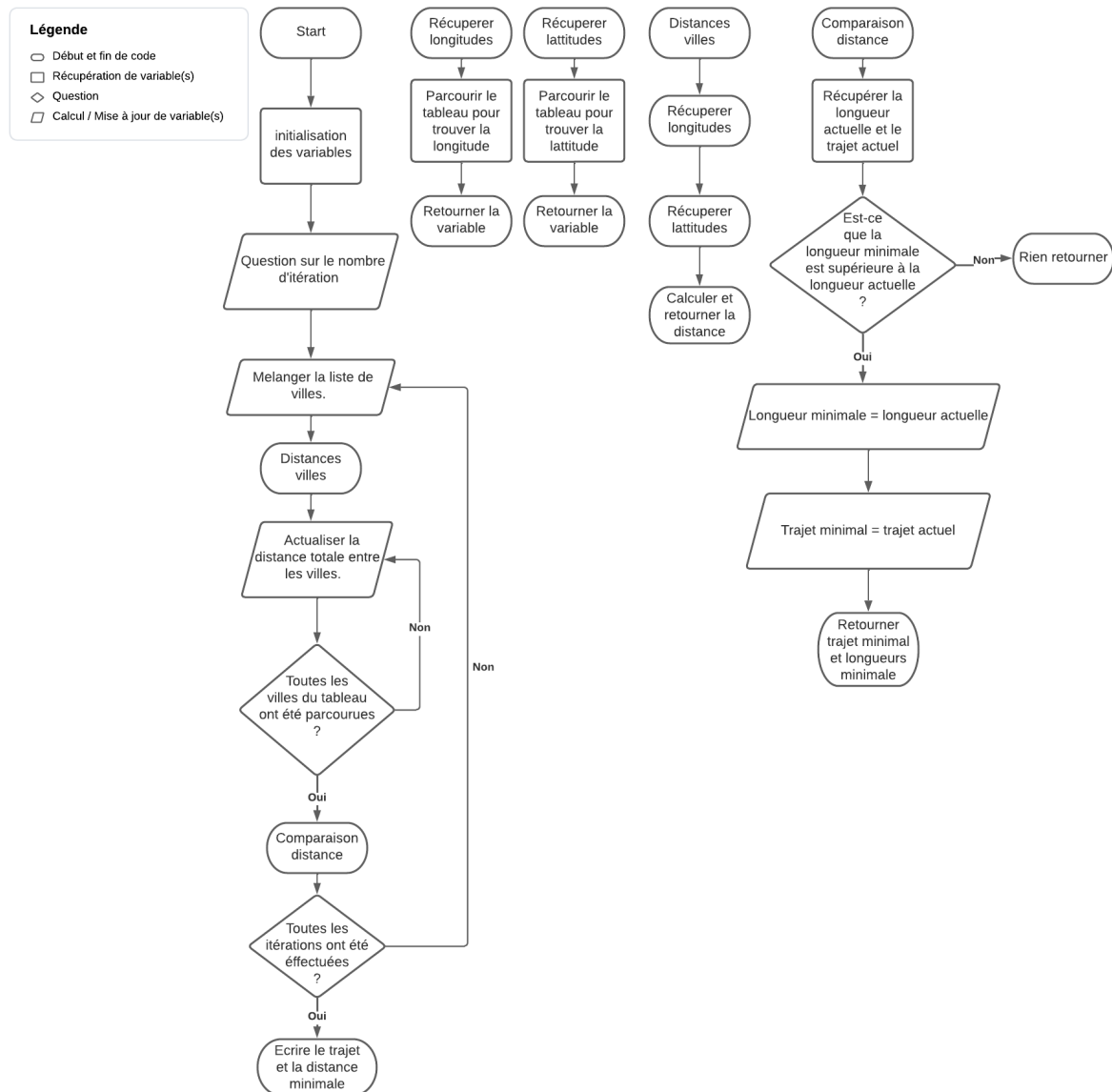


Figure 1 - Logigramme

Le logigramme comporte plusieurs parties de code qui mit ensemble contribue à répondre à la problématique.

B. Pseudocode

Le pseudo code ci-dessous s'en ai découlé du logigramme :

```

Début
Importer les bibliothèques nécessaires

Créer une liste contenant les villes et leurs coordonnées géographiques : Liste_Des_Villes

Définir une fonction pour obtenir la longitude d'une ville : LongVille(NomVille)
    Pour i allant de 0 à la longueur de la liste des villes
        Si le nom de la ville correspond à la ville i
            Renvoyer la longitude de la ville i

Définir une fonction pour obtenir la latitude d'une ville : LatVille(NomVille)
    Pour i allant de 0 à la longueur de la liste des villes
        Si le nom de la ville correspond à la ville i
            Renvoyer la latitude de la ville i

Définir une fonction pour calculer la distance entre deux villes : Distance_Villes(VilleA, VilleB)
    Définir le rayon de la terre en mètres : Rayon = 6_367_445
    Convertir la longitude et la latitude des villes VilleA et VilleB en radians
    Calculer la distance entre les villes en utilisant la formule haversine et renvoyer la distance en kilomètres
    arrondie

Initialiser les variables générales : LongueurMin, TrajetMin
Effacer le terminal
Demander le nombre d'itérations à effectuer : NmbIterations
Définir une fonction pour comparer la distance actuelle avec la distance minimale : ComparaisonDistance(x)
    Si la distance actuelle est plus courte que la distance minimale
        Mettre à jour la distance minimale et le trajet minimum

Récupérer le temps du début de la simulation
Début de la barre de progression
Pour i allant de 0 à NmbIterations
    Initialiser la distance actuelle à 0
    Définir la ville de départ à Paris
    Sélectionner aléatoirement les villes à visiter : TrajetVilles
    Pour j allant de 0 à la longueur de TrajetVilles
        Définir la ville étape : VilleEtape
        Ajouter la distance entre la ville de départ et la ville étape à la distance actuelle
        Ajouter la ville étape au trajet actuel
        Définir la ville de départ à la ville étape
    Ajouter la distance de la dernière ville à Paris à la distance actuelle
    Ajouter Paris au trajet actuel
    Appeler la fonction ComparaisonDistance avec la distance actuelle
    Mettre à jour la barre de progression
Calculer le temps mis pour parcourir le trajet minimum
Récupérer le temps final et calculer le delta temps
Ecrire le message avec marquées les différents trajets et temps trajets.
Fin

```

Figure 2 – Pseudo Code

C. Code

A partir de maintenant pour simplifier la compréhension et pour une utilisation optimale de notre algorithme. Un fichier est mis à disposition contenant le code final ainsi que les différents programmes additionnels. Vous n'aurez qu'à lancer le programme recommandé pour avoir le résultat souhaité.

Vous pouvez trouver le projet sur [GitHub](#).

D. Explication des parties

1. Les Villes

La première étape fût d'établir la liste des villes. Cet inventaire sera fixe car jamais modifié. Dans notre disposition nous avons spécifié le nom, longitude, latitude pour chacune des villes. Grâce à cela, le programme pourra en ressortir rapidement l'élément souhaité.

```
# Création d'un tableau contenant les villes et leurs coordonnées géographiques
Liste_Des_Villes = [

# [Nom de la ville, Latitude, Longitude]
["Paris", 48.8566, 2.3522],
["Marseille", 43.2965, 5.3698],
["Lyon", 45.764, 4.8357],
["Toulouse", 43.6047, 1.4442],
["Nice", 43.7101, 7.262],
["Nantes", 47.2184, -1.5536],
["Strasbourg", 48.5734, 7.7521],
["Montpellier", 43.6107, 3.8767],
["Bordeaux", 44.8379, -0.5795],
["Lille", 50.6329, 3.0583],
```

Figure 3 - Liste des villes

2. Distance

Le calcul d'une distance entre deux villes est assez complexe à trouver. Grâce à internet nous avons découvert une fonction impliquant les longitudes et latitudes des villes A et B :

$$S_{A-B} = R \times \cos^{-1}(\sin \varphi_A \times \sin \varphi_B + \cos \varphi_A \times \cos \varphi_B \times \cos(\lambda_B - \lambda_A))$$

Avec A point de départ, B point d'arrivée, φ la latitude, λ la longitude et R le rayon de la terre.

(Voir figure 8)

3. Trajet

Concernant cette partie de code nous avons longtemps divagué avant de trouver la bonne méthode.

L'algorithme récolte toutes les villes passables (excluant donc la ville de départ) dans un tableau pour ensuite les mélanger grâce à la fonction *shuffle* dans la librairie python "random".

Il parcourra ensuite toutes les villes de la première à la dernière dans le nouveau tableau. (Voir figure 9)

4. Ajouts

a) Affichage

Lorsque le code s'exécutait, nous avions souvent beaucoup de mal à savoir le temps restant. Le nombre de simulation est trop grand ? La simulation est en exécution ?

Afin de répondre à cela nous avons ajouté une barre de progression. Cet ajout permet de voir le pourcentage d'avancement, le temps restant et le temps écoulé. Ceci fût possible avec l'utilisation de la librairie python *progressbar*. (Voir figure 10)

Finalement, pour connaître le temps entre le début et la fin de simulation nous avons récolté l'heure exacte. Une petite conversion en minutes et tout était prêt. (Voir figure 11)

b) Vitesse moyenne

(Code "*VitesseMoyenne.py*")

Nous voulons connaître le temps passé pour le trajet, pour cela nous avons ajouté une vitesse moyenne,

- Si la distance est inférieure à 70km nous considérons qu'il prendra les petites routes. Alors La vitesse moyenne est de 80km/h.
- Si la distance est supérieure à 70km nous considérons qu'il prendra l'autoroute. La vitesse moyenne est de 120km/h

Nous avons décidé d'utiliser une fonction permettant de prendre la décision entre les 2 vitesses moyennes prenant en compte la distance entre chaque ville. (Voir figure 12)

Ensuite, nous utilisons la relation entre le temps, la vitesse et la distance :

$$t = \frac{D}{v}$$

Pour connaître la durée du voyage, nous divisons donc la distance avec la vitesse moyenne calculée précédemment.

c) Emission de CO₂

(Code "*EmissionCO2.py*")

Du côté des émissions de CO₂ nous avons 2 contraintes.

La voiture étant hybride, son autonomie électrique est de 400km en émettant une quantité de 6kg/km.

Au-delà de cette distance, la voiture passe en thermique en relâchant 120kg/km de CO₂.

Afin de calculer l'émission nous utilisons cette relation :

$$Emission = Emission_{/km} \times Distance(km)$$

Nous adaptons la relation mathématique et nos contraintes pour obtenir cela :

```
def EmissionCO2Calcul(km):
    """
    Fonction calculant l'émission de CO2 de la voiture pour chaque trajet entre les villes
    Electrique 6g/km Autonomie 400km
    Thermique 120g/km Autonomie infinie

    Input : km 'int'
    Output : None
    """
    global EmissionCO2Actuel
    if km <= 400:
        return 6*km
    else:
        return (6*400 + (km-400)*120)
```

Figure 4 - Calcul des émissions de CO2

d) *Changement de ville initiale*
(Code "VilleDepartChanged.py")

Une petite question nous est venue à l'esprit :

Paris est-elle la meilleure ville pour commencer le trajet ? Si non quelle est la meilleure ville ?

Pour répondre à cette question nous avons modifié notre code permettant de choisir la ville de départ. La petite subtilité était la suppression de la ville dans notre liste pour ne pas repasser par celle-ci. La commande *pop* intégrée nativement dans python était parfaite pour nous, elle permet de supprimer une ligne dans un tableau.

Pour simplifier le temps de simulation, les calculs et avoir la certitude d'obtenir le meilleur résultat nous avons réduit le nombre de villes à 8. En faisant cela, 100 000 itérations recouvraient totalement le nombre de tests pour trouver le bon résultat (40 320).

Après plusieurs essais en partant des 8 villes différentes les résultats étaient identiques.

Avec de la logique cela est cohérent. Si le trajet est un cercle, qu'importe le point où nous démarrons, la distance à parcourir (le périmètre) reste la même.

e) *Optimisation*

Pour des petites simulations allant de 1 à 1million, le temps mis par l'ordinateur pour effectuer le calcul était correct. Mais lorsque nous voulions les meilleurs résultats, il fallait optimiser le code pour le rendre plus rapide et améliorer sa lisibilité.

Pour cela nous avons commencé par résumer notre programme.

- Avant d'utiliser la commande *shuffle* nous avons employé *sample*. Afin d'utiliser cette commande nous devons faire une transposée de notre tableau pour ne garder finalement que la première ligne, et cela pour chaque itération. Actuellement, nous utilisons une boucle permettant de récupérer la première colonne, et cela, qu'une fois en début de code.
- Pour la lisibilité, en début de projet la disposition des villes était comme cela :

```
[["VilleA","VilleB","VilleC"],  
[LongitudeA,LongitudeB,LongitudeC],  
[LatitudeA,LatitudeB,LatitudeC]]
```

Pour 3 villes cela ne paraît pas si illisible mais à 30, la liste est très longue à l'horizontale. Nous avons donc décidé de changer pour cet arrangement :

```
[["VilleA",LongitudeA ,LatitudeA],  
["VilleB",LongitudeB ,LatitudeB],  
["VilleC",LongitudeC ,LatitudeC]]
```

Cela est devenu beaucoup plus lisible mais tout aussi long verticalement.

Grâce à cela nous avons pu gagner en temps de résolution et en lisibilité.

5. Code supplémentaire

Voici une liste d'ajout que nous avons réalisée. Certains simplifient le programme, d'autres ajoutent des compléments.

a) *Utilisation de fonctions*

Afin de simplifier le code, nous avons généré un dictionnaire avec les différentes définitions. Dans notre code nous n'avons qu'à appeler chaque définition et de l'utiliser.

Ce code a des avantages :

- Le gain en lisibilité car l'utilisation d'une définition est d'une seule ligne.
- En compréhension, il est plus facile de comprendre car chaque définition définit un calcul.
- En simplification, ceci nous permet de diviser par 2 le nombre de ligne.

b) *Interface Graphique*

(Code "*Programme_Interface.py*")

Ce code fit beaucoup plus compliquer à mettre en place car cela était une découverte, nous avons utilisé une nouvelle librairie se nommant *tkinter* (voir bibliographie 3/4)

Mis bout à bout, nous avons passé 4 jours à temps plein sur ce code afin qu'il fonctionne.

Pour réaliser ce code, des livres, chat GPT et google ont fortement contribué à la création de ce code.

Lors de la création de ce programme, nous sommes passées d'étape en étape :

- Création de l'interface affichant le trajet à effectuer.
- Sélection du nombre de simulation à l'aide de boutons.
- Affichage d'une carte ainsi que les différentes villes pour la visualisation.

Le principe de l'interface est simple. On rentre dans l'interface, on crée un bouton, on lui dit ce qu'il fait à l'aide d'une définition et enfin on place le bouton sur l'interface.

Le plus difficile fût la mise en place de relation entre les différentes définitions. Certaines définitions ont besoin de variable, qui n'est pas forcément disponible à un instant $t=0$.

6. Analyse de résultat

Nous avons généré des graphiques afin de visualiser et comprendre nos résultats :

a) Kilométrage en fonction du nombre d'itérations

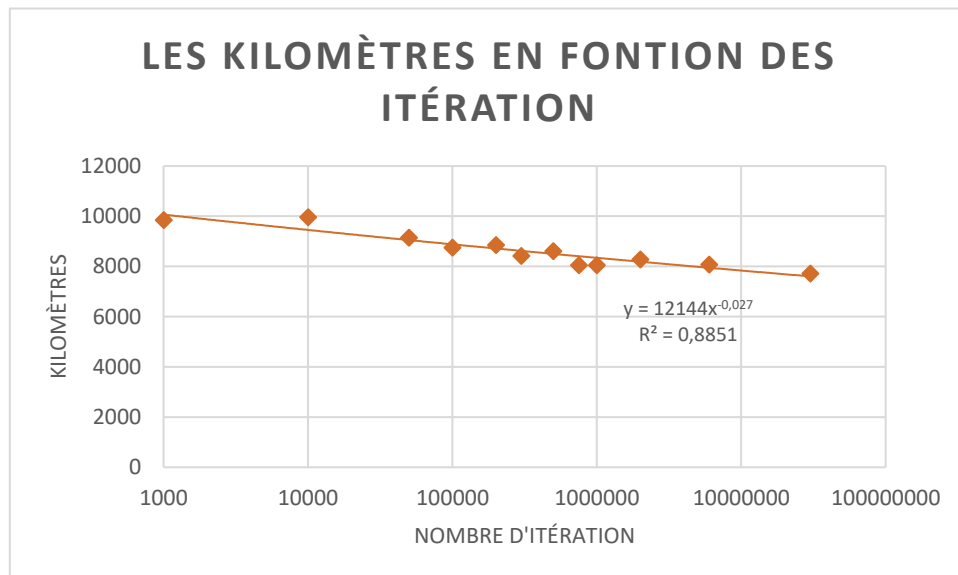


Figure 5

La courbe représente la distance la plus courte en fonction du nombre d'itérations.

On remarque qu'elle est décroissante donc elle aura une limite finie. Nous pouvons également constater que sa limite finie sera atteinte avec une occurrence très grande voire infinie.

b) Temps de voyage en fonction du nombre d'itérations :

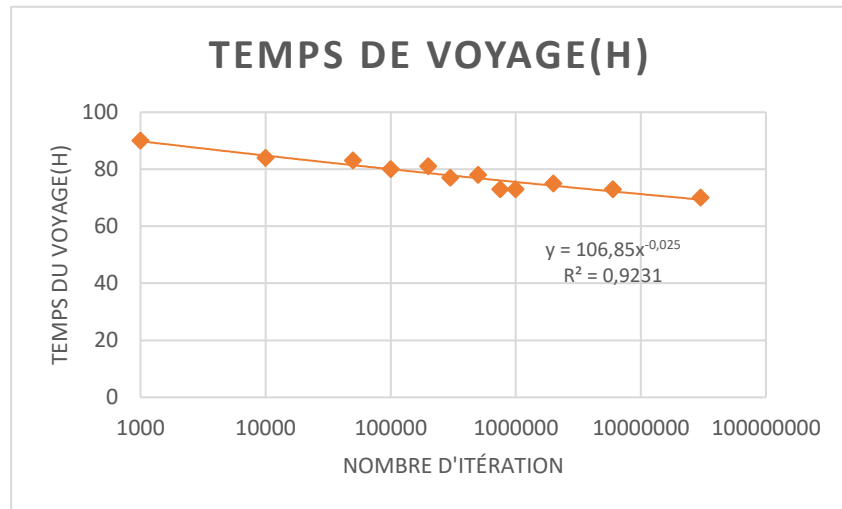


Figure 6

Comme la précédente courbe, elle admet une limite finie. De même, le nombre d'occurrence pour obtenir cette limite est infini.

c) Temps d'exécution du programme en fonction des occurrences :

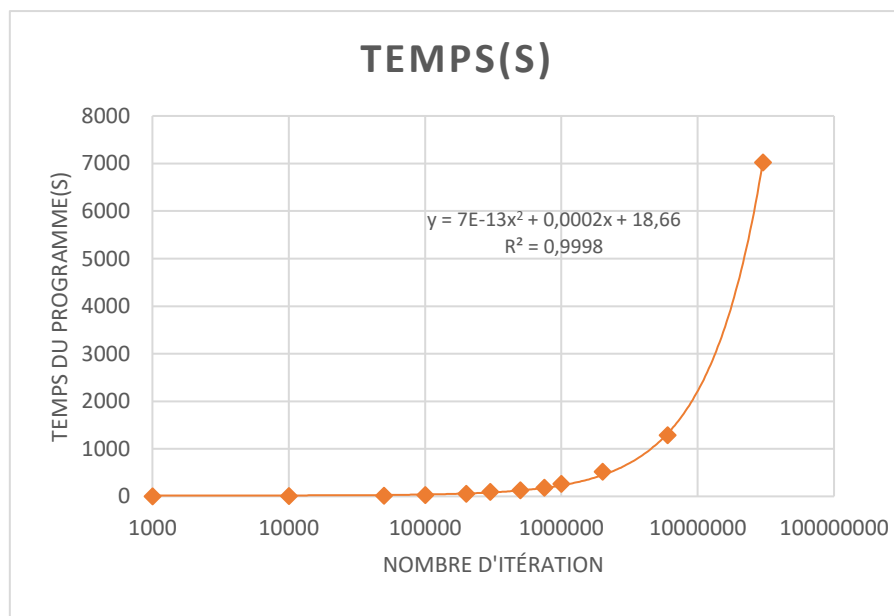


Figure 7

Cette représentation nous informe de la difficulté d'obtenir le trajet le plus court. Pour des millions d'occurrences, le temps en secondes devient colossal.

Par comparaison nous avons une probabilité de 8.84×10^{30} de tomber sur le même trajet. Nous savons aujourd'hui que 1×10^9 d'occurrences sont presque déjà impossibles puisque c'est une fonction exponentielle. Nous avons parcouru sur les 29 ! de possibilité 30 000 000 c'est-à-dire $3,39 \times 10^{-24}\%$.

IV. Conclusion

Nous voulions commencer par remercier le client de nous avoir commandés ce programme. Cela nous a permis de découvrir de nouvelles fonctionnalités sur Python. Le programme en lui-même fonctionne et nous avons pu rajouter des fonctionnalités intéressantes et qui j'espère intéressera le client.

Notre meilleur résultat pour 30 millions d'itérations et 2h de simulation expose une distance de 7 718km pour une durée de voyage de 70h.

Nous avons découvert que la méthode de Monte-Carlo n'est peut-être pas la meilleur afin de trouver le meilleur trajet puisque nous avons parcouru en 2h seulement $3,39 \times 10^{-24}\%$. Nous pouvons imaginer un programme recherchant la ville la plus proche de chaque ville pour pouvoir rapidement choisir la prochaine étape et ainsi optimiser l'algorithme pour espérer se rapprocher des 50%.

Du point de vue personnel, nous avons apprécié ce projet nous permettant une très grosse réflexion sur chacune de nos prises de décision. Comment arranger les villes ? Réaliser un seul code ou plusieurs fonctions que nous utiliserons ? Quelle librairie et / ou commandes utiliser pour optimiser le programme ?

Le programme étant réalisé, un simple changement de ville contenant sa longitude et latitude permet de changer son utilisation. L'implémentation des capitales d'Europe permet de faire un Road-Trip.

Prévoir ses vacances connaissant les villes à visiter devient maintenant plus simple.

V. Annexes

```
def Distance_Villes(VilleA, VilleB):
    """
    Fonction permettant de calculer la distance entre deux villes

    Entree : VilleA, VilleB 'string'
    Return : Distance 'int'
    """
    # Rayon de la Terre en mètre
    Rayon = 6_367_445
    LongA, LongB = math.radians(LongVille(VilleA)), math.radians(LongVille(VilleB))
    LatA, LatB = math.radians(LatVille(VilleA)), math.radians(LatVille(VilleB))
    return round((Rayon*(np.arccos(np.sin(LatA)*np.sin(LatB)+np.cos(LatA)*np.cos(LatB)*np.cos(LongB-LongA))))/1000)
```

Figure 8 - Distance entre deux villes

```
shuffle(TrajetVilles)
```

Figure 9 - La commande shuffle

```
from progressbar import ProgressBar, Percentage, Timer, Bar, ETA

bar = ProgressBar(widgets=[Percentage(), Timer(), Bar(), ETA()], maxval=NmbIterations)
bar.start()
#Code Monte Carlo s'executant
bar.update(i)
```

Figure 100 - Ajout et utilisation barre de progression

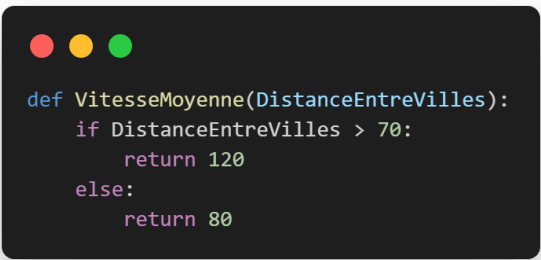
```
from datetime import datetime

# Récupérer le temps du début de la simulation
StartTime = datetime.strptime(datetime.now().strftime('%H:%M:%S'), "%H:%M:%S")

#Code Monte Carlo s'executant

# Récupérer le temps final et trouver le delta
EndTime = datetime.strptime(datetime.now().strftime('%H:%M:%S'), "%H:%M:%S")
DeltaTime = EndTime - StartTime
```

Figure 11 - Ajout et utilisation du temps de la simulation



```
def VitesseMoyenne(DistanceEntreVilles):  
    if DistanceEntreVilles > 70:  
        return 120  
    else:  
        return 80
```

Figure 11-Vitesse moyenne

VI. Bibliographie

1. **Lenka, Chinmoy.** *geeksforgeeks*. [En ligne] 26 juillet 2022. <https://www.geeksforgeeks.org/float-in-python/>.
2. **ankthon.** *geeksforgeeks*. [En ligne] 29 Août 2018. <https://www.geeksforgeeks.org/python-random-sample-function/>.
3. **freeCodeCamp.org.** Tkinter Course - Create Graphic User Interfaces in Python Tutorial. *YouTube*. [En ligne] 19 Novembre 2019. Source d'information conséquentes, elle permet une approche simpliste et complète de cette façon de coder.. <https://www.youtube.com/watch?v=YXPYB4XeYLA&t=8110s>.
4. **OpenIA.** *Chat GPT*. [En ligne] Ce site nous à permis de régler quelques problèmes liés à l'interface.. <https://chat.openai.com/chat>.