Université d'Évry Val d'Essonne, 91000
Évry, France

Projet Optimisation
M1 Mathématiques et Interactions

Points de Torricelli Généralisés.

QRIOUET Saâd
N° étudiant : 20171683

2020-2021

Table des matières

Introduction

Objectif

L'objectif de ce projet est de mettre en oeuvre plusieurs algorithmes d'optimisation avec ou sans contraintes dont le but est de trouver le point X^* qui minimise la somme des distances à N points non alignés $(X_i)_{i\in[\![1;N]\!]} \in \mathbb{R}^2$. Pour ce faire, on définit la fonction suivante :

$$J_p : \left| \begin{array}{ccc} \mathbb{R}^2 & \longrightarrow & \mathbb{R} \\ X & \longmapsto & \sum_{i=1}^{n} \|X - X_i\|_p \end{array} \right.$$

où pour tout entier $p > 1$, $\|.\|$p désigne la norme p de $\mathbb{R}^2$. Alors, sous réserve d'existence, le point X^* recherché est un minimum de la fonction J_p.

Déroulement

Ce projet est divisé en 3 parties :

La première partie dans laquelle nous allons nous intéresser d'un point de vue théorique à l'existence, l'unicité du minimum du problème, comment choisir un point de départ stable pour les algorithmes, algorithmes d'optimisation sans contraintes que nous allons implémenter afin de minimiser la fonction J_p.

Ensuite, nous allons nous intéresser à un système de projection de la Terre sur le plan. Donc, grâce aux coordonnées des villles que l'on précisera au cours du projet, nous allons nous ramener à une optimisation dans le plan, et ensuite modifier le problème de sorte à tenir compte de la déformation des géodésiques de la sphère par la projection.

Pour finir, la dernière partie du projet aura pour but de considérer dans le cas du plan, des contraintes et mettre en oeuvre un algorithme d'optimisation avec contraintes tel que l'algorithme d'Arrow-Hurwicz. Cette partie aura pour but de determiner des ensembles qui vérifieront les conditions de qualification vus en cours, mais qui ne contiendront plus l'optimum non contraint afin d'étudier l'influence du choix du domaine contraint sur l'optimum associé.

Python

Nous répondrons au problème en implémentant des algorithmes sur le langage de programmation *Python*, nous devions installer au préalable et charger des packages nécessaires tels que *Numpy*, *Matplotlib.pyplot*, *Scipy*, *Pandas* ou encore *Geopandas*.

Pour réaliser des premiers tests sur les fonctions et algorithmes (ici, nous n'évoquerons que les tests réalisés sur le jeu de données contenant les villes), nous avons créés un nuage de points E qui sera utilisé par défaut dans l'implémentation de différentes fonctions. Pour avoir une idée du point minimum et de vérifier nos algorithmes d'optimisation, nous avons aussi réalisé un graphe de courbes de niveaux. Afin de ne pas avoir de problèmes de compilation lors d'utilisations des fonctions, nous allons mettre par défaut certains paramètres des fonctions que l'on implémentera tout au long du projet.

Nous avons tout d'abord implémenté la fonction J_p, son Gradient et sa matrice Hessienne, voici l'implémentation de ces fonctions :

```python
def my_Jp(x, E=E, ord=2):
    X = x.reshape(2,-1)
    return np.sum(np.linalg.norm(X-E, axis=0, ord=ord))

def my_grad_Jp(x, E=E, ord=2, eps=1e-18):
    X = x.reshape(2,-1)
    return (np.sum(np.sign(X-E) * np.abs(X-E)**(ord-1) * (np.linalg.norm(X-E, axis=0, ord=ord) + eps)**(1-ord), axis=1)).reshape(2,-1)

def my_Hess_Jp(x, E=E, ord=2, eps=1e-18):
    X = x.reshape(2,-1)
    L = (ord-1) * (np.linalg.norm(X - E, ord=ord) + eps)**(1-ord) * np.abs(X-E)**(ord-2) * (1 - (np.linalg.norm(X-E, ord=ord) + eps)**(-ord) * np.abs(X-E)**or
    dxdy_ = np.sign(X-E) * (np.abs(X-E) ** (ord-1))
    dxdy = dxdy_[0] * dxdy_[1] * (1-ord) * ((np.linalg.norm(X-E, ord=ord) + eps)** (1 - 2 * ord))
    return np.array([[np.sum(L[0]), np.sum(dxdy)], [np.sum(dxdy), np.sum(L[1])]])
```

Ici, chacunes des fonctions prennent en argument le point x dans laquelle elle est évaluée, l'ensemble de points, l'ord qui correspond à la norme p associée à la fonction, seulement pour le Gradient et la matrice Hessienne un espilon très petit afin d'éviter de diviser par zéro lorsque l'on évalue la fonction en un point de l'ensemble de points utilisé.

Chapitre 1

Optimisation sans contraintes dans le plan

Dans cette partie, nous allons tout d'abord etudier la fonction J_p, notamment sa coercivité, sa stricte convexité, et donc qu'elle admet un unique minimum global ; qui sera un élément central dans notre projet étant donné que l'on le determinera à travers différents algorithmes, avec ou sans différentes contraintes, et selon différentes projections.

Les différentes démonstrations ont été faites à l'écrit et sont dans la partie "Annexe".

Avant d'implémenter les algorithmes qui vont nous permettre de determiner le minimum x* de la fonction J_p, nous allons implémenter un algorithme de construction d'enveloppe convexe de points donnés.

1.1 Algorithme de construction de l'enveloppe convexe

Pour construire l'enveloppe convexe de points donnés, nous avons optés pour l'algorithme de Jarvis.

Nous avons créés cet algorithme à l'aide de plusieurs fonctions :
Tout d'abord, nous avons créés la fonction "point_abs_min" qui prend en paramètre un nuage de points L et qui nous donne le point avec la plus petite abscisse, et si besoin (en cas de plusieurs points ayant la même abscisse minimale), le point possédant aussi la plus petite ordonnée.

```
def point_abs_min(L):
    min = L[0]
    index = 0
    for i, p in enumerate(L[1:], 1):
        if p[0] < min[0]:
            min = p
            indice = i
        elif p[0] == min[0]:
            if p[1] < min[1]:
                min = p
                index = i
    return index
```

Ensuite, nous avons créés la fonction "orientation" qui prend en paramètre 3 points du nuage de points et qui renvoie une valeurs parmis -1 ;0 ;1 selon si le triplet est en sens indirect, alignement ou direct.

```
def orientation(a, b ,c):
    v_ab = (b[0] - a[0], b[1] - a[1])
    v_ac = (c[0] - a[0], c[1] - a[1])
    det = v_ab[0]*v_ac[1] - v_ab[1]*v_ac[0]
    if det > 0:
        return 1
    if det < 0:
        return -1
    return 0
```

Puis, nous avons créés la fonction "prochain_sommet" qui prend en paramètre un nuage de points L et un sommet i, et qui nous renvoie l'indice du prochain sommmet qui formera l'enveloppe convexe.

```
def prochain_sommet(L, i, debug=False):
    val = []
    for j in range(len(L)):
        if j == i:
            continue
        v = 0
        for k in range(len(L)):
            if k == i or k == j:
                continue
            v += orientation(L[i], L[j], L[k])
        val.append((j, v))
        if debug:
            print(val)
    val.sort(key=lambda x: x[1])
    if debug :
      print(val)
    return val[-1][0]
```

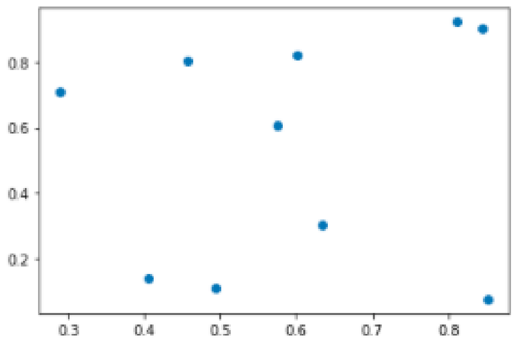
Grâce a ces 3 fonctions implémentées, nous avons pu créer la fonction "Jarvis" qui prend en argument un nuage de points et qui renvoie l'enveloppe convexe associé a ce dernier.

```
def Jarvis(L):
    i = point_abs_min(L)
    prochain = prochain_sommet(L, i)
    Enveloppe = [i, prochain]
    while prochain_sommet(L, prochain) != i:
        prochain = prochain_sommet(L, prochain)
        Enveloppe.append(prochain)
    return Enveloppe
```

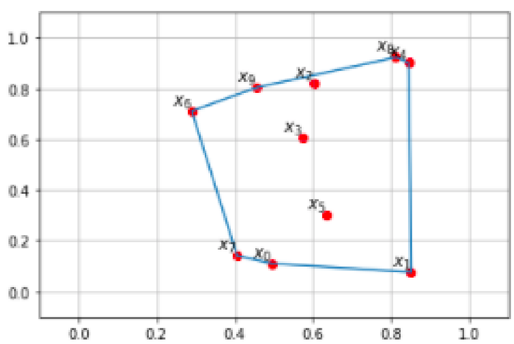
Pour finir, nous avons créés la fonction "affiche_enveloppe" qui prend en argument le nuage de points, l'algorithme utilisé (ici on utilisera Jarvis, mais on peut utiliser d'autres algorithmes de construction d'enveloppe convexe tel que le parcours de Graham par exemple), et deux coordonnées qui nous permettent de redimensionner la fenêtre d'affichage.

```python
def affiche_enveloppe(L, algorithme, xlim, ylim):
    Enveloppe = algorithme(L)

    X = [L[i][0] for i in Enveloppe]
    X.append(L[Enveloppe[0]][0])
    Y = [L[i][1] for i in Enveloppe]
    Y.append(L[Enveloppe[0]][1])

    plt.xlim(xlim)
    plt.ylim(ylim)

    plt.plot(X, Y)

    labels = ['$x_{0}$'.format(i) for i in range(len(L))]
    for label, x, y in zip(labels, [p[0] for p in L], [p[1] for p in L]):
        plt.annotate(label, xy=(x, y),  va='bottom', ha='right', size='large', color='black')
    plt.grid(True)
    plt.scatter(L[:,0], L[:,1], c='r')
    plt.show()
```

Afin de tester l'algorithme de Jarvis et son affichage, nous avons générés un nuage de points afin de creer l'enveloppe convexe de ces points donnés :



Ainsi, nous avons l'enveloppe convexe suivant :



1.2 Algorithmes d'optimisation

Dans cette partie, nous allons implémenter 4 algorithmes d'optimisation sans contraintes qui nous permettront de minimiser J_p. Les 4 algorithmes sont l'algorithme de gradient à pas fixe, à pas optimal, la méthode gradient conjugué, et l'algorithme de Newton d'ordre 2.

Afin de comparer les différents algorithmes (et donc décider lequel semble être le plus performant), nous allons directement les tester dans la partie "Projection et mise en oeuvre pratique", en utilisant les villes du problème de cette partie en question.

Avant d'implémenter les différents algorithmes, nous avons créés une fonction d'affichage appelée "affichage_resultats", qui prend en argument un point de départ, un point d'arrivée (ie : la solution), le nombre d'itérations de l'algorithme utilisé, un tableau représentant la trajectoire des points donnés par l'algorithme, un tableau représentant l'évolution de Jp, celui du gradient de Jp, et le nuage de points utilisé dans l'algorithme.
Cette fonction nous donne un rendu clair, et différentes informations concernant les performances de l'algorithme utilisé et donc nous permettra de mieux conclure sur l'algorithme le plus performant parmi les 4.

```python
def affichage_resultats(X0, X, it, trajectoire, valeurs_J, gradients_J, E):

    print('Départ :', X0.reshape(2,))
    print('Solution :', X.reshape(2,))
    print('Nombre d\'iterations :', it)
    print('Norme du gradient à la dernière itération :', gradients_J[-1], '\n')


    fig, axs = plt.subplots(nrows=2, ncols=2, constrained_layout=True, figsize=(10,5))
    #params = {'mathtext.default': 'regular' }
    #plt.rcParams.update(params)

    axs[0, 0].scatter(E[0], E[1], c='r')        # nuage de points en rouge
    axs[0, 0].scatter(X0[0], X0[1], c='g')      # point de départ en vert
    axs[0, 0].scatter(X[0], X[1], c='b')        # point d'arrivée en bleu
    axs[0, 0].set_title('Départ : point vert - Arrivée : point bleu')
    axs[0, 0].set_xlabel('x')
    axs[0, 0].set_ylabel('y')

    axs[0, 1].scatter(E[0], E[1], c='r')
    axs[0, 1].scatter(trajectoire[0], trajectoire[1], c='b')
    axs[0, 1].set_title('Trajectoire des points Xk donnés par l\'algorithme')
    axs[0, 1].set_xlabel('x')
    axs[0, 1].set_ylabel('y')

    axs[1, 0].plot(valeurs_J)
    axs[1, 0].set_title('Evolution de $J_{p}(X^{k})$')
    axs[1, 0].set_xlabel('k')
    axs[1, 0].set_ylabel('$J_{p}(X^{k})$')

    axs[1, 1].plot(np.log(gradients_J))
    axs[1, 1].set_title('Evolution du gradient de $J_{p}(X^{k})$')
    axs[1, 1].set_xlabel('k')
    axs[1, 1].set_ylabel('log(|| grad $J_{p}(X^{k})$ ||)')

    return axs
```

1.2.1 Descente de gradient à pas fixe

Voici l'algorithme de gradient à pas fixe, il prend en argument un point de départ, un pas, une tolérance, un nombre maximum d'itération, un nuage de point, et l'ord :

```
# Descente de gradient à pas fixe :

def desc_pf_Jp(X0, h=0.01, tol=1e-6, it_max=1000, E=villes, ord=2):
    X = X0.reshape(2,-1)
    it = 0
    stop = False
    valeurs_J = []
    gradients_J = []
    trajectoire = [X]
    while not stop:
        X = X - h * my_grad_Jp(X, E, ord)
        it += 1
        stop = (it >= it_max) or (np.linalg.norm(my_grad_Jp(X, E, ord)) <= tol)
        valeurs_J.append(my_Jp(X, E, ord))
        gradients_J.append(np.linalg.norm(my_grad_Jp(X, E, ord), ord=ord))
        trajectoire.append(X)
    trajectoire = (np.array(trajectoire).reshape(-1, 2)).T
    return X, it, trajectoire, valeurs_J, gradients_J
```

1.2.2 Descente de gradient à pas optimal

Avant d'implémenter d'algorithme de gradient à pas optimal, nous devions créer une fonction qui optimise le pas de l'algorithme. Pour cela, nous avons créés deux fonctions de la sorte : une qui recharche le pas optimal à l'aide de la méthode de Newton, et une par dichotomie.

Notons que ces deux méthodes nous serviront aussi pour l'algorithme de gradient conjugué.

Voici la fonction "recherche_pas_optimal_newton" qui optimise le pas à l'aide de la méthode de Newton :

```
# Recherche du pas optimal à l'aide de la méthode de Newton
def recherche_pas_optimal_newton(Xk, Vk, h_, tol=1e-6, it_max=50, E=villes, ord=2):
    X = Xk.reshape(2,-1)
    V = Vk.reshape(2,-1)

    list_h = []
    values = []
    grads = []

    h = h_
    it = 0
    stop = False
    while not stop:
        H = my_Hess_Jp(X - h * V, E, ord)
        h = h + np.vdot(V, my_grad_Jp(X - h * V, E, ord)) / (np.vdot(V, np.dot(H, V))) / 10
        it +=1
        list_h.append(h)
        values.append(my_Jp(X - h * V, E, ord))
        grads.append(np.vdot(V, my_grad_Jp(X - h * V, E, ord)))
        stop = (np.abs(np.vdot(V, my_grad_Jp(X - h * V, E, ord))) <= tol) or (it >= it_max)
    return h, it, list_h, values, grads
```

Voici la fonction "recherche_pas_optimal_dichotomie" qui optimise le pas par dichotomie :

```python
# Recherche du pas optimal par dichotomie
def recherche_pas_optimal_dichotomie(Xk, Vk, tol=1e-6, it_max=100, E=villes, ord=2):
    X = Xk.reshape(2,-1)
    V = Vk.reshape(2,-1)

    list_h = []
    values = []
    grads = []

    it = 0
    a = 0
    b = 10
    if -np.vdot(V, my_grad_Jp(X - b * V, E, ord)) < 0:
        stop_cherche_b = False
        while not stop_cherche_b:
            b = np.random.rand() * 10 ** (np.random.randint(-3,5))
            stop_cherche_b = (-np.vdot(V, my_grad_Jp(X - b * V, E, ord)) > 0)

    h = (a+b)/2
    val = -np.vdot(V, my_grad_Jp(X - h * V, E, ord))

    stop = False
    while not stop:
        if val < 0:
            a = h
        else:
            b = h
        h = (a+b) / 2
        val = -np.vdot(V, my_grad_Jp(X - h * V, E, ord))
        it +=1
        list_h.append(h)
        values.append(my_Jp(X - h * V, E, ord))
        grads.append(np.vdot(V, my_grad_Jp(X - h * V, E, ord)))
        stop = (np.abs(val) <= tol) or (it >= it_max)
    return h, it, list_h, values, grads
```

Voici l'algorithme de gradient à pas optimal, il prend en argument un point de départ, une méthode pour optimiser le pas (par défaut est la méthode de dichotomie), une tolérance, un nombre maximum ditération, un nuage de points, et l'ord :

```python
# Descente de gradient à pas optimal
def desc_po_Jp(X0, methode_recherche_pas_optimal='dichotomie', tol=1e-6, it_max=1000, E=villes, ord=2) :
    X = X0.reshape(2,-1)
    it = 0
    stop = False
    valeurs_J = []
    gradients_J = []
    trajectoire = [X]
    h = 200
    steps = []
    while not stop:
        grad_X = my_grad_Jp(X,E,ord)
        if methode_recherche_pas_optimal == 'newton':
            h, _, _, _, _ = recherche_pas_optimal_newton(X, grad_X, h, E=E, ord=ord)
        if methode_recherche_pas_optimal == 'dichotomie':
            h, _, _, _, _ = recherche_pas_optimal_dichotomie(X, grad_X, E=E, ord=ord)
        X = X - h * grad_X
        norm_grad_ = np.linalg.norm(my_grad_Jp(X,E,ord))
        it += 1
        stop = (it >= it_max) or (norm_grad_ <= tol)
        valeurs_J.append(my_Jp(X, E, ord))
        gradients_J.append(norm_grad_)
        trajectoire.append(X)
        steps.append(h)
    trajectoire = (np.array(trajectoire).reshape(-1, 2)).T
    return X, it, trajectoire, valeurs_J, gradients_J, steps
```

1.2.3 Méthode du gradient conjugué

Voici l'algorithme de gradient conjugué, il prend les mêmes arguments que l'algorithme de gradient à pas optimal :

```
# Algorithme du gradient conjugué :
def grad_conj_Jp(X0, methode_recherche_pas_optimal='dichotomie', tol=1e-6, it_max=1000, E=villes, ord=2) :
    X = X0.reshape(2,-1)
    it = 0
    stop = False
    valeurs_J = []
    gradients_J = []
    trajectoire = [X]
    steps = []
    d = np.zeros_like(X)
    h = 200

    while not stop:
        grad_X = my_grad_Jp(X,E,ord)
        hess_X = my_Hess_Jp(X,E,ord)
        d = grad_X - np.vdot(grad_X, hess_X.dot(d)) / (np.vdot(d, hess_X.dot(d)) + 1e-9) * d
        if methode_recherche_pas_optimal == 'newton':
            h, _, _, _, _ = recherche_pas_optimal_newton(X, d, h, E=E, ord=ord)
        if methode_recherche_pas_optimal == 'dichotomie':
            h, _, _, _, _ = recherche_pas_optimal_dichotomie(X, d, E=E, ord=ord)
        X = X - h * d
        norm_grad_ = np.linalg.norm(my_grad_Jp(X,E,ord))
        it += 1
        stop = (it >= it_max) or (norm_grad_ <= tol)
        valeurs_J.append(my_Jp(X, E, ord))
        gradients_J.append(norm_grad_)
        trajectoire.append(X)
        steps.append(h)
    trajectoire = (np.array(trajectoire).reshape(-1, 2)).T
    return X, it, trajectoire, valeurs_J, gradients_J, steps
```

1.2.4 Méthode de Newton

Voici l'algorithme de Newton d'ordre 2, il prend en arguments une tolérance, un nombre maximum d'itération, un nuage de point, et l'ord :

```
def my_Newton_Jp(X0, tol=1e-6, it_max=100, E=villes, ord=2):
    X = X0.reshape(2,-1)
    it = 0
    stop = False
    valeurs_J = []
    gradients_J = []
    trajectoire = [X]
    while not stop:
        X = X - np.linalg.solve(my_Hess_Jp(X, E, ord), my_grad_Jp(X, E, ord)) / 5   # H^-1 x grad vérifie H X = grad
#        X = X - np.dot(np.linalg.inv(my_Hess_Jp(X, E, ord)),  my_grad_Jp(X, E, ord))
        norm_grad_ = np.linalg.norm(my_grad_Jp(X,E,ord))
        it += 1
        stop = (it >= it_max) or (norm_grad_ <= tol)
        valeurs_J.append(my_Jp(X, E, ord))
        gradients_J.append(np.linalg.norm(my_grad_Jp(X, E, ord), ord=ord))
        trajectoire.append(X)
    trajectoire = (np.array(trajectoire).reshape(-1, 2)).T
    return X, it, trajectoire, valeurs_J, gradients_J
```

Chapitre 2

Projection et mise en oeuvre pratique

Dans cette partie, nous allons determiner le point optimal du problème d'optimisation suivant :
Nous voulons minimiser la distance qui sépare les différentes villes que l'on aura dans notre jeu de données.
En plus de résoudre ce problème, nous allons afficher l'optimal sur une carte avec toutes les villes que l'on aura considéré dans notre mise en pratique à l'aide du module *Pandas* et plus particulièrement *Geopandas*.

Nous allons en premier temps, résoudre ce problème de deux manières : d'abord en nous référant sur un système de projection choisi, et ensuite en prenant en compte la déformation des géodésiques de la sphère causé par la projection choisie.
Pour finir, nous allons comparer les résultats obtenus par ces deux différentes méthodes.

2.1 Villes considérées

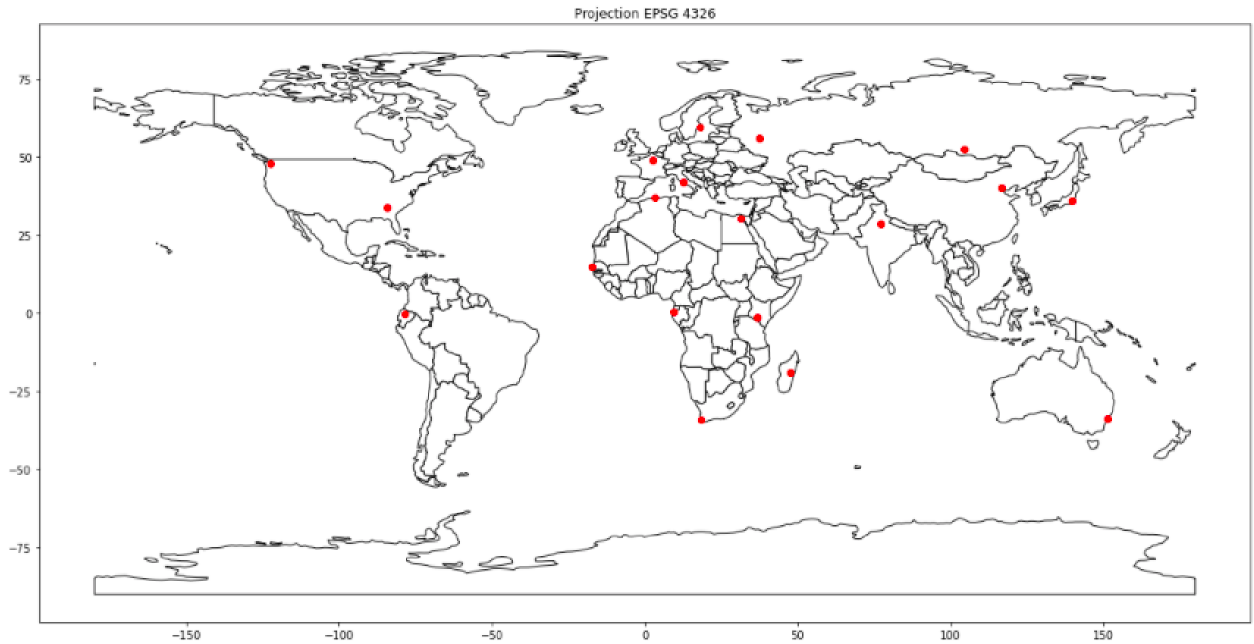
Grâce a *Pandas* nous avons crées un jeu de données comportant les villes avec différentes caractéristiques : "Latitiude" et "Longitude". Ce premier jeu de données est appelé "cities". Ensuite grâce à *Geopandas*, nous avons utilisés la fonction "geopandas.GeoDataFrame" afin d'obtenir un nouveau jeu de données, que l'on transformera grâce à la fonction "set.crs" afin d'obtenir les coordonnées de ces villes selon **le système de projection WGS84** (EPSG :4326 sur le code) et donc d'obtenir le jeu de données "gps_cities".
Pour finir, nous avons stockés les villes dans un tableau numpy afin que cela soit plus facile à manipuler (que l'on va appeler "villes").
Le tableau contenant différentes informations sur les villes que nous allons utiliser sur cette partie du projet est le suivant :

	City	Longitude	Latitude	geometry
0	Alger	3.060188	36.775361	POINT (3.06019 36.77536)
1	Antananarivo	47.525581	-18.910012	POINT (47.52558 -18.91001)
2	Atlanta	-84.390264	33.748992	POINT (-84.39026 33.74899)
3	Cape Town	18.417396	-33.928992	POINT (18.41740 -33.92899)
4	Dakar	-17.447938	14.693425	POINT (-17.44794 14.69342)
5	Irkoutsk	104.280586	52.289597	POINT (104.28059 52.28960)
6	Le Caire	31.235726	30.044388	POINT (31.23573 30.04439)
7	Libreville	9.454001	0.390002	POINT (9.45400 0.39000)
8	Moscou	37.617494	55.750446	POINT (37.61749 55.75045)
9	Nairobi	36.817245	-1.283253	POINT (36.81724 -1.28325)
10	New Delhi	77.209006	28.613895	POINT (77.20901 28.61390)
11	Paris	2.351462	48.856697	POINT (2.35146 48.85670)
12	Pékin	116.718583	39.902067	POINT (116.71858 39.90207)
13	Quito	-78.512327	-0.220164	POINT (-78.51233 -0.22016)
14	Rome	12.482932	41.893320	POINT (12.48293 41.89332)
15	Seattle	-122.330062	47.603832	POINT (-122.33006 47.60383)
16	Sydney	151.216454	-33.854816	POINT (151.21645 -33.85482)
17	Stockholm	18.071093	59.325117	POINT (18.07109 59.32512)
18	Tokyo	139.759455	35.682839	POINT (139.75945 35.68284)

Nous avons crées une carte du monde en chargeant à l'aide d'une fonction de *Geopandas* une carte appelée "naturalearth_lowres" , et y avons affichés les villes issus de "gpd_cities" à l'aide des lignes de codes suivant :

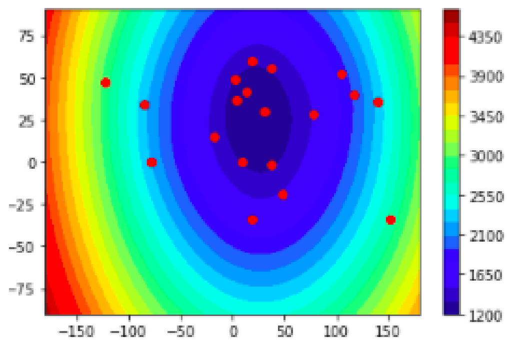
```
# Affichage des villes :


world = geopandas.read_file(geopandas.datasets.get_path('naturalearth_lowres'))
ax = world.plot(color='white', edgecolor='black', figsize=(20, 10))
gpd_cities.plot(ax=ax, color='red')
ax.set_title('Projection EPSG 4326')
plt.show()
```

Ainsi, nous avons la carte du monde suivante :



Nous trouvons les 19 villes de notre jeu de données affichées en rouge.

Afin, d'avoir une idée de la position du point optimmum du problème, nous allons afficher les courbes de niveaux :



Le point optimum semble se trouver en Europe ou en Afrique.

Nous pourrions aussi aussi déterminer le point barycentrique de l'ensemble des villes. En effet, ce point est une bonne première approximation de l'optimum.

2.2 Résolution du problème d'optimisation

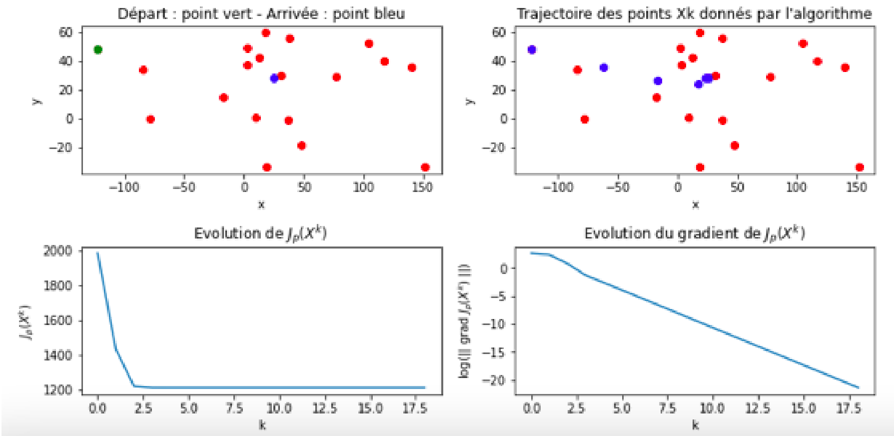
Comme précisé dans la partie 1. "Optimisation sans contraintes dans le plan", nous allons tester les différents algorithmes sur notre jeu de données afin de déterminer l'optimum, et par la même occasion, determiner lequel de ces algorithmes est le meilleur.

2.2.1 Résolution du problème dans le plan

Nous avons décidés de partir de Seattle (USA) pour determiner le point optimum a ce problème. Nous aurions pu prendre comme point de départ un point sur la carte comme le point (-150,75), une autre ville comme Alger (Algerie), ou encore le point barycentrique de l'ensemble des villes.
Avec Seattle comme point de départ, nous avons donc les résultats suivants :

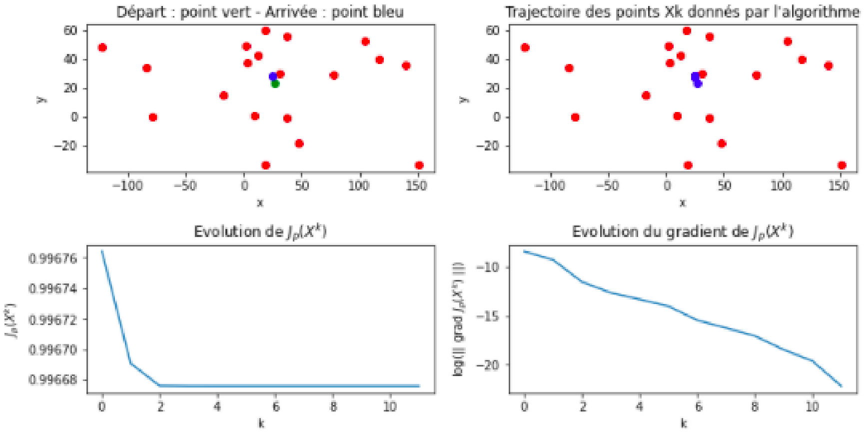
Descente de gradient à pas fixe

```
Départ : [-122.3300624   47.6038321]
Solution : [24.81036023 27.934105   ]
Valeur : 1212.9601356744515
Nombre d'iterations : 19
Norme du gradient à la dernière itération : 5.182949713477032e-10

array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7fa0625967b8>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7fa06253fb70>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7fa0624f65f8>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7fa06252b780>]],
      dtype=object)
```



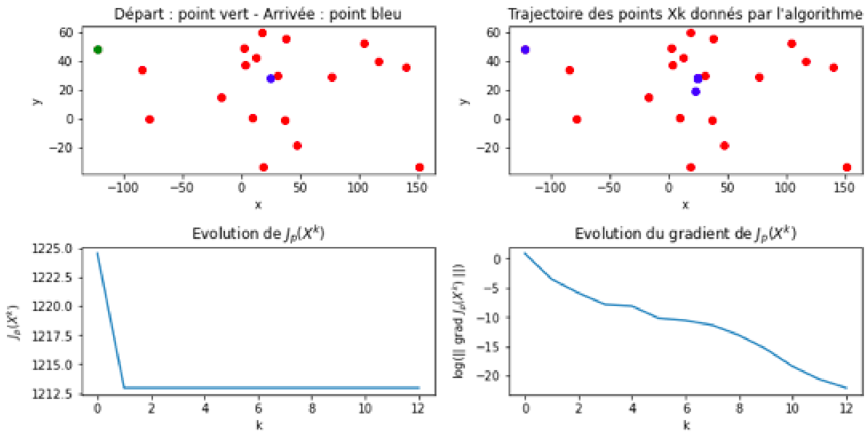
Descente de gradient à pas optimal

```
Départ : [26.50192683 23.01961796]
Solution : [24.81035958 27.93410573]
Nombre d'iterations : 12
Norme du gradient à la dernière itération : 2.294216977593709e-10

array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7fe65297a518>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7fe6528bc828>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7fe65285b860>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7fe65288b898>]],
      dtype=object)
```



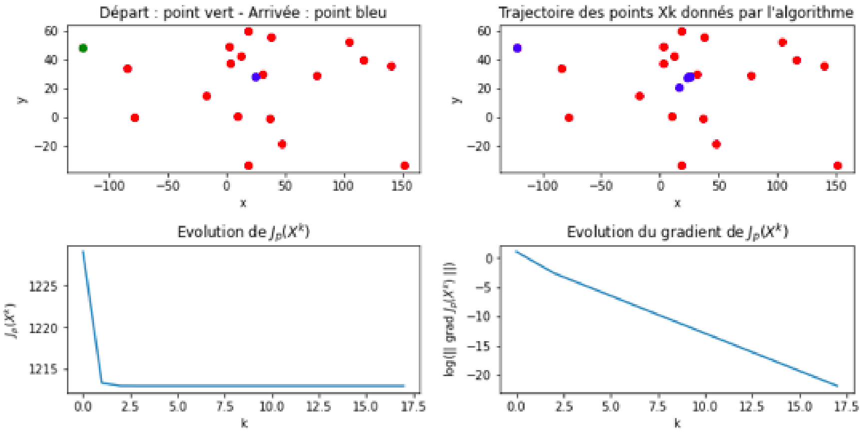
Méthode du gradient conjugué

```
Départ : [-122.3300624    47.6038321]
Solution : [24.81036023 27.934105   ]
Valeur : 1212.9601356744515
Nombre d'iterations : 13
Norme du gradient à la dernière itération : 2.6029935440922046e-10

array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7fa06176de10>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7fa061690c50>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7fa0616434e0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7fa0615f3668>]],
      dtype=object)
```



Méthode de Newton

```
Départ : [-122.3300624   47.6038321]
Solution : [24.81036023 27.934105   ]
Valeur : 1212.9601356744515
Nombre d'iterations : 18
Norme du gradient à la dernière itération : 3.1077750867682727e-10

array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7fa0614cce80>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7fa0614735f8>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7fa0614a2780>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7fa061453908>]],
      dtype=object)
```



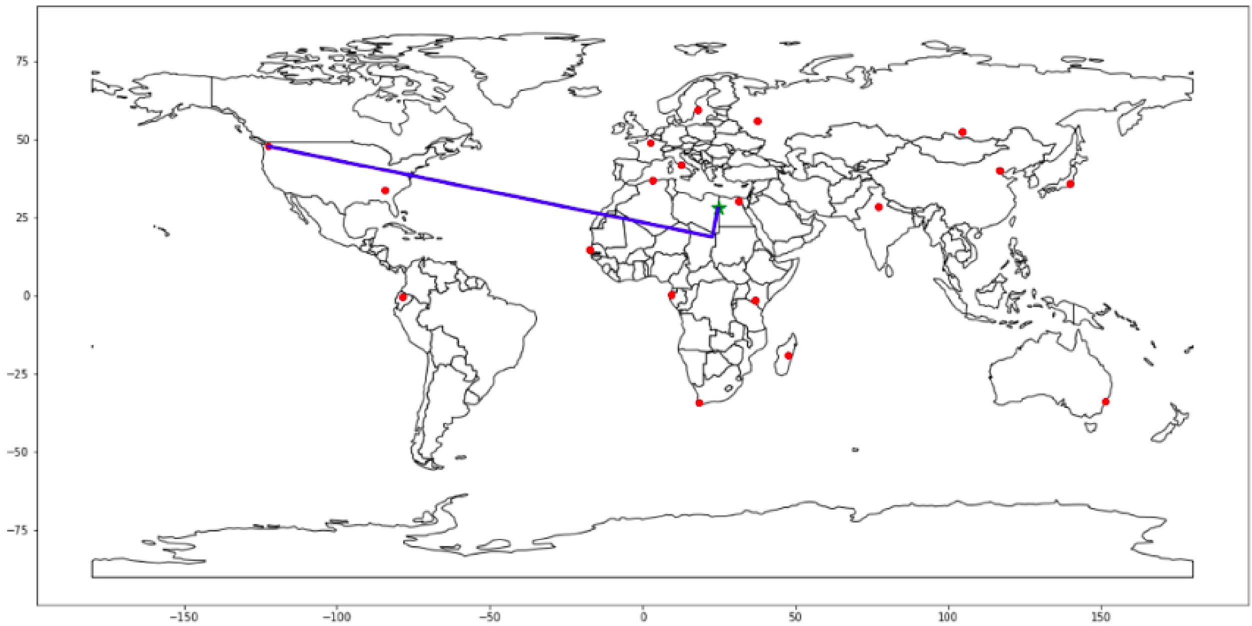
Nous remarquons que la méthode de Newton est très efficace, mais la convergence est beaucoup trop dépendante du choix du point de départ. Il suffit de prendre un nuage de points "compliqué" pour s'en convaincre.

Choix de l'algorithme et affichage

On voit que les 4 algorithmes marchent marchent très bien, et sont tres efficaces. Notons que les algorithmes de gradient à pas optimal et gradient conjugué convergent en moins d'itérations, et ont une meilleure précision que les deux autres. Nous allons opter pour l'un des deux (même si l'on pourrait dans notre cas prendre un des 4). Nous pourrions compter le nombre d'itérations dans la méthode de recherche du pas optimal afin d'ajouter un argument dans le choix du meilleur algorithme parmis les quatres.

Nous choisissons l'algorithme de gradient à pas optimal.

Voici la trajectoire avec l'algorithme de gradient à pas optimal et l'affichage du point optimum sur la carte :



On est parti de Seattle et on arrive à l'optimum situé à la frontière de la Lybie et l'Egypte (point désigné par une étoile verte). On remarque que les premières directions de descente sont bien orthogonales. Les suivantes sont trop petites pour être visibles.

2.2.2 Prise en compte de la déformation des géodésiques

Pour tenir compte de la déformation des géodésiques, il ne faut plus calculer la distance associée à la norme p entre deux points A et B du plan mais calculer la trajectoire sur la sphère entre A et B.

Nous avons créer une fonction "dist_sphere" (qui prend en arguments deux points x et y, et un rayon) qui calcule la distance entre deux points donnés d'une sphère de rayon donnée. Nous avons aussi réalisés un test qui vérifie que la distance entre deux points opposés d'une sphère de rayon 1 vaut Pi.

Voici les lignes de codes correspondants :

```
def dist_sphere(x, y, rayon=6400):  # rayon en km, A: point, B: point ou nuage de points écrits en colonnes
# x, y : longitudes et latitudes des points x et y en degrés
# A, B : longitudes et latitudes des points x et y en radians
    A = x.reshape(2,1) * np.pi / 180
    B = y.reshape(2,-1) * np.pi / 180
    res = np.arccos(np.sin(A[1]) * np.sin(B[1]) + np.cos(A[1]) * np.cos(B[1]) * np.cos(B[0] - A[0]))
    return np.sum(res) * rayon

# on vérifie que la distance entre deux points opposés de la sphère de rayon 1 vaut pi
print(dist_sphere(np.array([90, 0]), np.array([-90, 0]), rayon=1))
```

```
3.141592653589793
```

Ainsi, il faudrait donc définir une nouvelle fonction coût ainsi que ces dérivées.

2.3 Comparaison des optimaux obtenus

Les optimaux obtenus suivant ces deux méthodes peuvent être très différents.
Par exemple, si on cherche le point qui minimise la somme des distances à Tokyo, Seattle et Quito sur la sphère, on obtient un point à l'intérieur du triangle formé par ces trois villes, au milieu du Pacifique. Si on cherche le point qui minimise les distances à ces villes sur le plan, on obtiendrait un point situé approximativement en Europe ou en Afrique du Nord !

Chapitre 3

Cas contraint

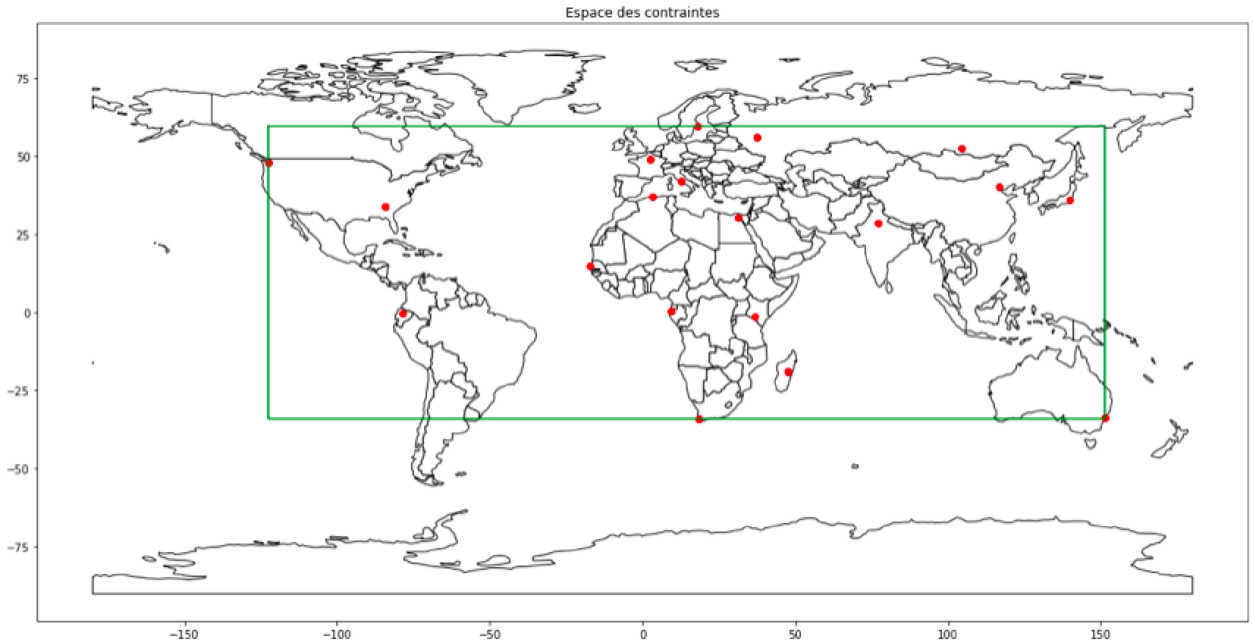
Dans cette dernière partie, nous allons mettre en oeuvre l'algortihme d'Arrow-Hurwicz afin de déterminer le minimum de la fonction J_p sous différentes contraintes tels que les points soient sous les conditions de qualification vus en cours (Karush Kuhn et Tucker, ou Slater).

3.1 Algorithme d'Arrow-Hurwicz

Dans le cadre de la partie 1, nous devons considérer un domaine de contraintes qui contient l'optimum sans contraintes

Avant d'implémenter l'algorithme d'Arrow-Hurwicz, nous devons définir l'espace des contraintes On choisit pour espace des contraintes un rectangle qui est convexe, et contenant l'ensemble des villes.

Affichons sur la carte le rectangle en question :



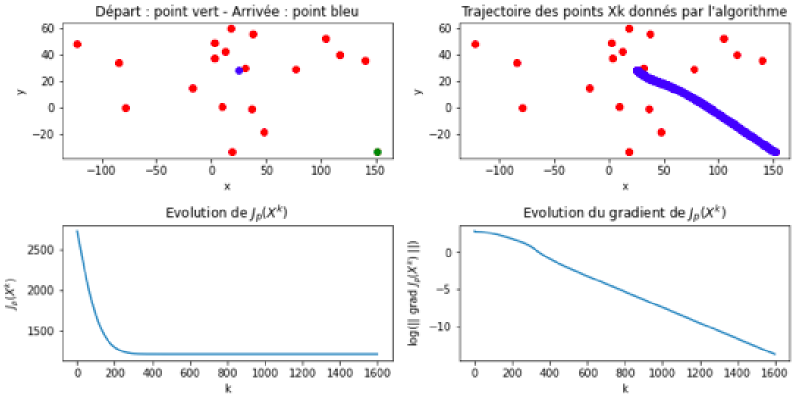
Voici l'algorithme d'Arrow-Hurwicz implémenté pour obtenir l'optimum dans cet espace de contraintes :

Algorithme d'Arrow Hurwicz :

```
[ ] X0 = np.array([xmax, ymin])
    Y0 = np.zeros((4,1))
    sigma, rho = 0.05, 0.05
    tol = 1e-6

    stop = False
    X = X0.reshape(2,-1)
    Y = Y0.reshape(4,-1)
    it = 0
    itmax = 5000
    trajectoire = [X]
    valeurs_J = [my_Jp(X, villes)]
    gradients_J = [np.linalg.norm(my_grad_Jp(X, villes))]

    while not stop:
      X = X - sigma * (my_grad_Jp(X, villes) + np.array([[-Y[0,0] + Y[1,0]], [-Y[2,0] + Y[3,0]]])) #np.array([[-1, 1, 0, 0], [0, 0, -1, 1]]).dot(Y)
      Y = Y + rho * np.array([[-X[0,0] + xmin], [X[0,0] - xmax], [-X[1,0] + ymin], [X[1,0] - ymax]])
      Y = Y * (Y > 0)

      val = my_Jp(X, villes)
      grad = my_grad_Jp(X, villes)
      norm_grad = np.linalg.norm(grad)

      it += 1
      stop = (norm_grad < tol) or (it >= itmax)

      trajectoire.append(X)
      valeurs_J.append(val)
      gradients_J.append(norm_grad)

    trajectoire = (np.array(trajectoire).reshape(-1, 2)).T

    X_opti_arrow_hurwicz = X
    trajectoire_opti_arrow_hurwicz = trajectoire

    affichage_resultats(X0, X, it, trajectoire, valeurs_J, gradients_J, villes)
```

Appliqué à notre ensemble de villes, nous obtenons les résultats suivants :

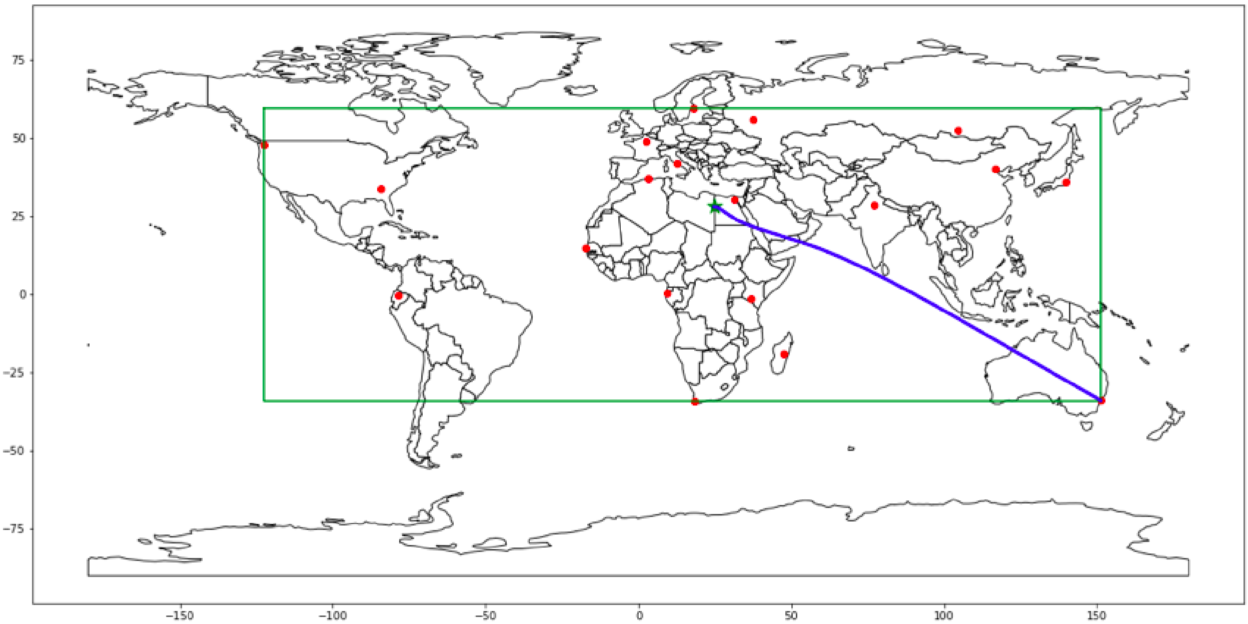
```
Départ : [151.2164539 -33.928992 ]
Solution : [24.81036494 27.93410495]
Nombre d'iterations : 1598
Norme du gradient à la dernière itération : 9.911057838731184e-07

array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7f1fe79ebb00>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f1fe7efd898>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f1fe7f09b70>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f1fe8110518>]],
      dtype=object)
```



Nous remarquons que l'algorithme converge en 1598 itérations, avec une précision de l'ordre de 10^{-7}.

Nous sommes parti d'un point assez éloigné de la solution, afin de vérifier la stabilité de l'algorithme concernant le point de départ dans l'espace de contraintes.

Voici l'affichage sur la carte du point optimum, ainsi que de la trajectoire :



Nous remarquons que la solution est la même que la première trouvée en partie 2.

3.2 Minimisation de la fonction coût sous des contraintes excluant l'optimum global

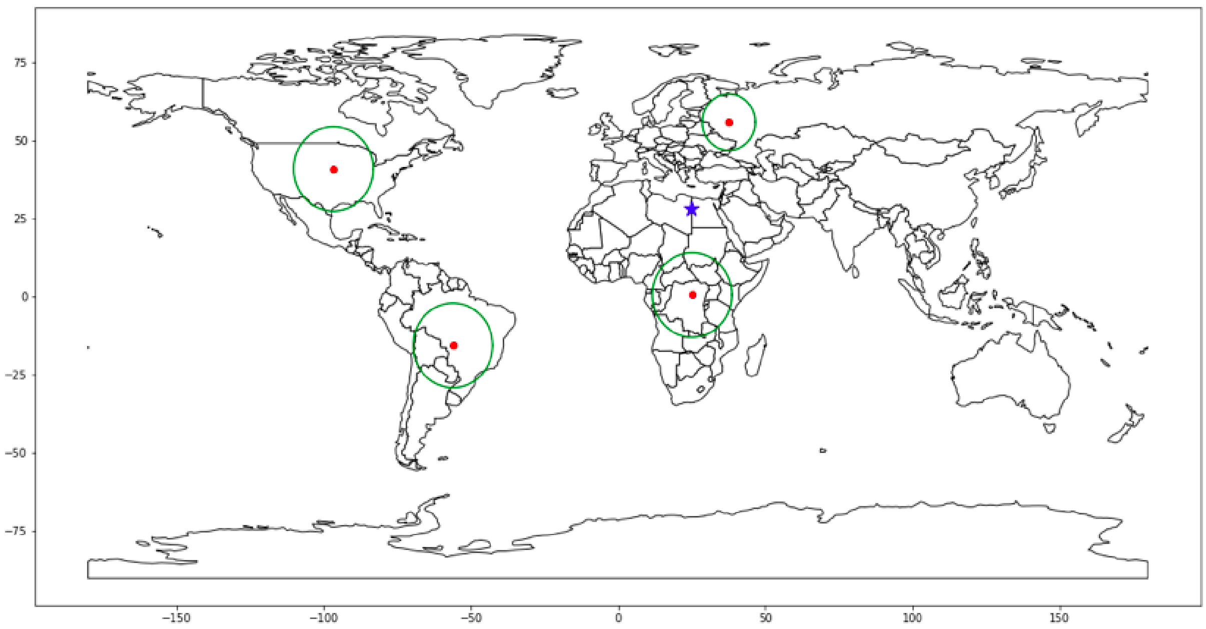
Dans le cadre de l'optimisation planaire, les boules données dans le point suivant sont des exemples d'ensembles qui ne contiennent pas l'optimum et chacune d'entre elles vérifie les conditions de qualification de KKT car elle est définie par une seule contrainte dont le gradient n'est jamais nul quand la contrainte est saturée.
Toujours dans le même cadre, nous allons essayer déterminer un optimum avec les contraintes suivantes :
Nous allons considérer des boules de différents rayons autour de 4 villes : Moscou (Russie), Lincoln Nebraska (USA), Kisangani (RDC) et Cuiaba (Brésil).

Tout d'abord, nous avons créés une fonction "arrow_hurwicz_2" pour éviter de réaliser 4 fois la même chose.
Voici l'algorithme de Arrow-Hurwicz :

```python
def arrow_hurwicz_2(X0, Y0, r, Z0, sigma=0.05, rho=0.05, tol=1e-6, it_max=1000, villes=villes, ord=2):

  stop = False
  X = X0.reshape(2,-1)
  Z = Z0.reshape(2,-1)
  Y = Y0
  it = 0

  trajectoire = [X]
  valeurs_J = [my_Jp(X, villes)]
  gradients_J = [np.linalg.norm(my_grad_Jp(X, villes))]

  while not stop:
    X = X - sigma * (my_grad_Jp(X, villes) + 2 * (X - Z))
    Y = Y + rho * (np.linalg.norm(X-Z)**2 - r**2)
    Y = Y * (Y > 0)

    val = my_Jp(X, villes)
    grad = my_grad_Jp(X, villes)
    norm_grad = np.linalg.norm(grad)

    it += 1
    stop = (norm_grad < tol) or (it >= itmax)

    trajectoire.append(X)
    valeurs_J.append(val)
    gradients_J.append(norm_grad)

  trajectoire = (np.array(trajectoire).reshape(-1, 2)).T

  return X, Y, it, trajectoire, valeurs_J, gradients_J
```

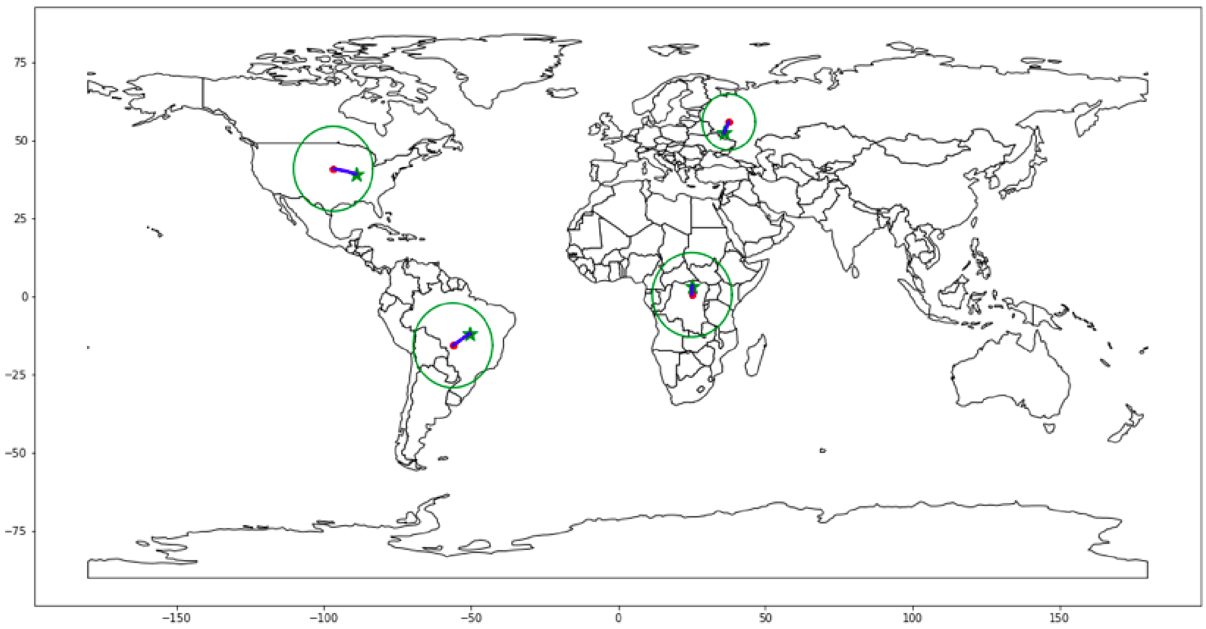
Concernant les 4 villes du problème, nous allons créer un jeu de données les contenant comme réalisé en partie 2, et en y rajoutant le rayon associé à la boule autour de la ville en question. Après avoir réalisé ce jeu de données, nous allons former ces boules et les afficher sur notre carte avec le point optimum de la partie 2.

Voici le résultat suivant :



On voit bien sur la carte que l'optimum n'est dans aucune des quatre boules, nous allons donc pas trouver l'optimum global.

En restant dans ces domaines de contraintes, nous trouvons donc les optima suivants :



Conclusion

Ainsi, dans ce projet nous avons d'abord vérifié des propriétés pour minimiser la fonction telles que l'existence, l'unicité du minimum, ou encore le choix du point de départ adéquat pour différents algorithmes d'optimiisation.
Puis, nous avons implémentés l'algorithme de Jarvis qui est un algorithme de construction d'enveloppe convexe.
Ensuite, nous avons mis en oeuvre différents algorithmes d'optimisations sans, puis avec contraintes qui nous ont permit de répondre au problème de minimisation initial. En effet, a l'aide d'algorithmes d'optimisation sans contraintes tel que le gradient à pas optimal nous avons determinés, d'abord dans le cadre de l'optimisation planaire, le point minimisant la somme de la distance à chacune des villes ; puis, en prenant en compte de la déformation des géodésiques un optimum qui peuvent êtres différents comme expliqué dans la partie "Projection et mise en oeuvre pratique".
Aussi, toujours dans le cadre de l'optimisation planaire, à l'aide d'un algorithme d'optimisation avec contraintes tel que l'algorithme d'Arrow-Hurwicz et en prenant des domaines de contraintes tels que les points soient qualifiés, nous avons considéré des ensembles adéquats qui ne contiennent plus le point optimal non contraint, donc des ensembles qui ne contiennent pas l'optimum global.

Bibliographie / Webographie

Algorithmes d'optimisation, de Pierre Gilles LEMARIE-RIEUSSET
https://fr.wikipedia.org/wiki/Marche_de_Jarvis
https://fr.wikipedia.org/wiki/Calcul_de_l%27enveloppe_convexe
https://www-sop.inria.fr/geometrica/courses/slides/enveloppe-convexe-od.pdf
https://transp-or.epfl.ch/optimization/slides/03-optimalite.pdf
https://www.i2m.univ-amu.fr/perso/thierry.gallouet/licence.d/anum.d/anum-c10.pdf
https://pandas.pydata.org
http://www.python-simple.com/python-pandas/panda-intro.php
https://geopandas.org
https://portailsig.org/content/python-geopandas-ou-le-pandas-spatial.html
https://www.coordonnees-gps.fr
https://learnosm.org/fr/advanced/projections-and-files-format/
https://epsg.io/4326

Annexes

Etablir que la fonction Jp est coercive :

$\|x - x_i\|_p \geq \|x\|_p - \|x_i\|_p \xrightarrow[\|x\|_p \to +\infty]{} +\infty$

$\sum_{i=1}^{m} \left(\|x\|_p - \|x_i\|_p\right) \xrightarrow[\|x\|_p \to +\infty]{} +\infty$

Donc $J_p(x) = \sum_{n=1}^{m} \|x - x_i\|_p \geq \sum_{i=1}^{m} \left(\|x\|_p - \|x_i\|_p\right) \xrightarrow[\|x\|_p \to +\infty]{} +\infty$

Donc J_p est coercive.

Montrer qu'elle est strictement convexe sur $\mathbb{R}^d$:

Soient $X, Y \in \mathbb{R}^2$, $\lambda \in]0,1[$, N points $(X_i)_{1 \leq i \leq N}$ non alignés

$J_p(\lambda X + (1-\lambda)Y) = \sum_{i=1}^{N} \|\lambda X + (1-\lambda)Y - X_i\|_p$

$= \sum_{i=1}^{N} \|\lambda X + (1-\lambda)Y - \lambda X_i - (1-\lambda)X_i\|_p$

$= \sum_{i=1}^{N} \|\lambda(X - X_i) + (1-\lambda)(Y - X_i)\|_p$

(*) $\leq \sum_{i=1}^{N} \lambda \|X - X_i\|_p + (1-\lambda)\|Y - X_i\|_p = \lambda J_p(X) + (1-\lambda) J_p(Y)$

À i fixé, $\|\lambda(X - X_i) + (1-\lambda)(Y - X_i)\|_p \leq \|\lambda(X - X_i)\|_p + \|(1-\lambda)(Y - X_i)\|_p$.
Cette inégalité est une égalité ssi $\lambda(X - X_i)$ et $(1-\lambda)(Y - X_i)$ colinéaires
ie : $(X - X_i)$ et $(Y - X_i)$ colinéaires, ie : X, Y et X_i alignés.

$J(\lambda X + (1-\lambda)Y) \overset{(*)}{=} \lambda J(X) + (1-\lambda)J(Y)$ ssi $\forall i \in [\![1, N]\!]$ X, Y et X_i alignés
ie : $\forall i$ $X_i \in$ à la droite (XY) : pas possible car les X_i pas alignés (par hypothèse)
Ainsi, (*) est une inégalité stricte : $J(\lambda X + (1-\lambda)Y) < \lambda J(X) + (1-\lambda)J(Y)$
Donc J est strictement convexe.

En déduire qu'il existe un minimum global x* tel que Jp(x*) = $\min x \in \mathbf{R}^2 Jp(x)$:

J est coercive, donc $\exists x* \in$R+^{2} tel que J(x*) = $\min x \in$R^2 Jp(x) :
J est strictement convexe, donc ce minimum est unique.

Montrer ensuite que ce point x* appartient à l'enveloppe convexe des points (xi) i $\in [\![1; n]\!]$:

• Soit $p = 2$, $J_2(X) = \sum_{i=1}^{N} \|X - X_i\|_2 = \sum_{i=1}^{N} \left((x-x_i)^2 + (y-y_i)^2\right)^{1/2}$

$$\frac{d J_2(X)}{dx} = \sum_{i=1}^{N} \frac{1}{2} \times 2(x-x_i)\left((x-x_i)^2 + (y-y_i)^2\right)^{\frac{1}{2}-1}$$

$$= \sum_{i=1}^{N} \frac{(x-x_i)}{\|X-X_i\|_2}$$

$$\frac{d J_2(X)}{dy} = \sum_{i=1}^{N} \frac{(y-y_i)}{\|X-X_i\|_2}$$

$$\nabla J_2(X) = \begin{cases} \displaystyle\sum_{i=1}^{N} \frac{(X-X_i)}{\|X-X_i\|_2} & \text{si } X \notin \{X_i\}_{1 \leq i \leq N} \\ \displaystyle\sum_{\substack{i=1 \\ i \neq k}}^{N} \frac{(X-X_i)}{\|X-X_i\|^2} & \text{si } \exists\, k \in [\![1, N]\!] : X = X_k \end{cases}$$

$\forall p \geq 1$, J_p coercive et strictement convexe, donc il existe un unique minimum global $X^* \in \mathbb{R}^2$ de J_p.

Alors on a $\nabla J_2(X^*) = 0$, i.e. : $\displaystyle\sum_{i=1}^{N} \frac{X^* - X_i}{\|X^* - X_i\|_2} = 0$

On définit $\alpha_i = \dfrac{1}{\|X^* - X_i\|_2}$ et $S = \displaystyle\sum_{i=1}^{N} \alpha_i$

On a $S X^* = \displaystyle\sum_{i=1}^{N} \alpha_i X_i$, i.e. : $X^* = \displaystyle\sum_{i=1}^{N} \frac{\alpha_i}{S} X_i$

$\forall i \quad 0 \leq \dfrac{\alpha_i}{S} \leq 1$ et $\sum \dfrac{\alpha_i}{S} = \dfrac{S}{S} = 1$

X^* est donc une combinaison convexe des points $\{X_i\}_{1 \leq i \leq N}$

$X^* \in \mathrm{Conv}(\{X_i\}_{1 \leq i \leq N})$

Nous avons montré le résultat lorsque p = 2, le résultat se généralise pour $\forall p \geqslant 1$.