```
# traitement_BDD.py

0001| import sqlite3 as sql
0002| import numpy as np
0003| import time
0004| import matplotlib.pyplot as pl
0005| from algorithmes_de_minimisation import minimiser, minimum
0006|
0007| #nous utiliserons deux connections pour pouvoir effectuer
des actions à l'intérieur d'une boucle générée par la première
connection
0008| #(écrire avec c2 à l'intérieur de c provoque des erreurs)
0009| conn = sql.connect(r"F:/informatique/TIPE/database/produit
exploitable/GTFS.db")
0010| c = conn.cursor()
0011| c2 = conn.cursor()
0012|
0013| ##outils
0014| def traitement_tuple(L):
0015|     rep=[]
0016|     for tupl in L:
0017|         rep.append(tupl[0])
0018|     return rep
0019|
0020| def somme(L, ind):
0021|     s=0
0022|     for l in L:
0023|         s+=l[ind]
0024|     return s
0025|
0026| def reverse(liste):
0027|     N = len(liste)
0028|     reverse_liste=[]
0029|     for num in range(N-1,-1,-1):
0030|         reverse_liste.append(liste[num])
0031|     return reverse_liste
0032|
0033| def recuperer_id_groupe(x, y):
0034|     c.execute('''
0035|     select id_groupe
0036|     from stops_groupe
0037|     where (stops_groupe.x between {a}- 500 and {a}+ 500)
and (stops_groupe.y between {b}- 500 and {b}+ 500)
0038|     '''.format(a=x, b=y))
0039|     return c.fetchall()
0040|
0041| def recuperer_liaisons(x, y):
0042|     c.execute('''
0043|     select stops_groupe.stop_name, stops_groupe.id_groupe
0044|     from stops_groupe
```

```
0045|     join graphe
0046|         on graphe.to_id_groupe=stops_groupe.id_groupe
0047|     where graphe.from_id_groupe in(
0048|         select id_groupe
0049|         from stops_groupe
0050|             where (stops_groupe.x between {a}- 300 and {a}
+ 300) and (stops_groupe.y between {b}- 300 and {b}+ 300))
0051|     '''.format(a=x, b=y))
0052|     return c.fetchall()
0053|
0054| def recuperer_x_y(id_groupe):
0055|     c2.execute('select x,y from stops_groupe where
id_groupe={}'.format(id_groupe))
0056|     return c2.fetchone()
0057|
0058| def calcul_distance(dernier_id_groupe, id_groupe):
0059|     XY=[]
0060|     for id_g in [dernier_id_groupe, id_groupe]:
0061|         c.execute('select x, y from stops_groupe where
id_groupe={}'.format(id_g))
0062|         XY.append(c.fetchone())
0063|     [(x1,y1), (x2,y2)]=XY
0064|     d=dist(x1, y1, x2, y2)
0065|     return d
0066|
0067| def dist(x1, y1, x2, y2):
0068|     return np.sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2))
0069|
0070| def random_color():
0071|     return [np.random.rand()*0.8 for k in range(3)]
0072|
0073| ##paramètres du système lambert93
0074| def conversion_rad(angle_degre):
0075|     return angle_degre*np.pi/180
0076|
0077| def latitude_iso(lat_rad):  #reparamétrage symétrique de
la latitude
0078|     sin=np.sin(lat_rad)
0079|     p1=np.arctanh(sin)
0080|     p2=np.arctanh(e*sin)
0081|     result=p1-e*p2
0082|     return result
0083|
0084| xs=700000; ys=12655612.05 #coordonnées du pole nord dans
le système lambert93
0085| phi1=conversion_rad(44); ss1=np.sin(phi1);
cs1=np.cos(phi1) # Premier parallèle automécoïque ie lattitude
de sécance
0086| phi2=conversion_rad(49); ss2=np.sin(phi2);
cs2=np.cos(phi2) # Deuxième parallèle automécoïque
```

```
0087| phi0=(phi1+phi2)/2
0088| a=6378388 #grand axe de l'ellipse en m de la Terre ie
rayon à l'équateur
0089| e=0.08248325676 #excentricité de l'ellipse de la Terre
0090| gN1=a/np.sqrt(1-(e*ss1)**2); gl1=latitude_iso(phi1)
#grande normales aux sécantes (meilleur rayon de courbure)
0091| gN2=a/np.sqrt(1-(e*ss2)**2); gl2=latitude_iso(phi2)
0092| lon0=conversion_rad(3)  #méridien de greenwitch par
rapport à celui de Paris en degrés
0093| expo=np.log(gN2/gN1*cs2/cs1) / (gl1-gl2) #exposant de la
projection
0094| C=((gN1 * cs1) / expo) * np.exp(expo * gl1)
0095|
0096| ##conversion WGS84_to_lambert93
0097| def WGS84_to_lambert93(lat, lon):
0098|     lat_rad, lon_rad = conversion_rad(lat),
conversion_rad(lon)
0099|     g1=latitude_iso(lat_rad)
0100|     R = C*np.exp(-expo*g1)
0101|     teta = expo*(lon_rad-lon0)
0102|     X = xs + R*np.sin(teta)
0103|     Y = ys - R*np.cos(teta)
0104|     return X, Y
0105|
0106| ##fonction commune aux flux
0107| #il faut avoir au préalable rempli le cursor
0108| def representation_flux(nb_title, seuil, xn=600, xm=715,
yn=6800, ym=6905):
0109|     fig_mob=pl.figure(figsize=[23, 16])
0110|     mob_ax=pl.axes(title='flux de personnes émis par la
ville par jour pour motif professionnel(Lambert 93 en km)')
0111|     mob_ax.axis('equal')
0112|     X, Y, S = [], [], []
0113|     compt=0; compt2=0
0114|     for x, y, flux, name in c:
0115|         x_k, y_k = x/1000, y/1000
0116|         if x_k>xn and x_k<xm and y_k>yn and y_k<ym:
0117|             S.append(flux/1000)
0118|             X.append(x/1000)
0119|             Y.append(y/1000)
0120|             compt2+=1
0121|             if flux/1000>seuil:
0122|                 compt+=1
0123|                 if compt%4==0:
0124|                     mob_ax.text(x=x_k, y=y_k, s=name)
0125|             else:
0126|                 if compt2%nb_title==0:
0127|                     mob_ax.text(x=x_k, y=y_k, s=name)
0128|     scat = mob_ax.scatter(X, Y, s=S)
0129|     grid(mob_ax)
```

```
0130|     handles, labels = scat.legend_elements(prop="sizes")
0131|     mob_ax.legend(handles, labels, loc='lower right',
title="nombre d'habitants \n(en milliers)")
0132|
0133|
0134|
0135| ##mobilites pro -flux emis par VILLE
0136| def conversion_mobilites():
0137|     c.execute('select cle, lat, lon from mobilites_pro')
0138|     compt=0
0139|     for cle, lat, lon in c:
0140|         x, y = WGS84_to_lambert93(lat, lon)
0141|         c2.execute('update mobilites_pro set x={x}, y={y}
where cle={c}'.format(x=x, y=y, c=cle))
0142|         compt+=1
0143|         if compt%100==0:
0144|             print(compt)
0145|     conn.commit()
0146|
0147| def representation_mobilites():
0148|     c.execute('select x, y, flux_emis_jour, name from
mobilites_pro')
0149|     representation_flux(45, 20)
0150|     pl.savefig('flux émis par ville (mobilités pro)',
dpi=300, bbox_inches='tight')
0151|     pl.show()
0152|
0153| ##flux emis par POLE-ROUTE_ID (frequentation_m)
0154|
0155| '''on connait les flux émis par chaque ville, on répartit
ce flux entre les différents poles (frequentation_m)
0156| puis on suppose que le flux dans les transports en commun
est proportionnel a ce flux tous transports confondus
0157| on va donc faire en sorte que ce facteur de prop donne un
flux émis par jour dans chaque gare le plus proche de la
fréquentation de référence donnée par RATP'''
0158|
0159| def remplir_nb_trips():
0160|     c.execute('''
0161|     select trips.route_id, count(trips.trip_id)
0162|     from trips
0163|     join calendar
0164|         on calendar.service_id=trips.service_id
0165|         where calendar.start_date<20200522 and
calendar.stop_date>20200522
0166|     group by trips.route_id''')
0167|     for route_id, nb_trips in c:
0168|         c2.execute('update routes set nb_trips={n} where
route_id="{r}"'.format(n=nb_trips, r=route_id))
0169|     conn.commit()
```

```
0170|
0171| modes=[0.8, 4.2, 2.7, 3.9] #en millions de déplacements
par jour selon les routes_types
0172| # 0  tramway; 1 métro; 2   RER; 3 bus
0173| #on va donc comparer les contenances relatives
0174| remplissage_contenance=[0]*4
0175| def contenances():
0176|     c.execute('select sum(nb_trips) from routes group by
route_type')
0177|     repart = c.fetchall()
0178|     for k in range(4):
0179|         remplissage_contenance[k] = modes[k]*10**6/
repart[k][0]
0180| #les contenances trouvés sont nettement inférieures à la
réalité mais elles donnent un poids relatif
0181| #certaines courses ont propablement été comptées en
multiple
0182|
0183| def remplir_flux_emis():
0184|     c.execute('update flux_emis set frequentation_m=0')
0185|     contenances()
0186|     c.execute('''
0187|     select mobilites_pro.code_commune,
mobilites_pro.flux_emis_jour
0188|     from stops_groupe
0189|     join mobilites_pro
0190|         on
mobilites_pro.code_commune=stops_groupe.code_commune
0191|     group by mobilites_pro.code_commune
0192|     ''')
0193|     #on ne sélectionne que les codes communes qui ont des
pôles
0194|     compt=0
0195|     for code_commune, flux_ville in c:
0196|         execute_repartition_flux(code_commune, flux_ville)
0197|         compt+=1
0198|         if compt%50==0:
0199|             print(compt)
0200|             conn.commit()
0201|     conn.commit()
0202|
0203| def execute_repartition_flux(code_commune, flux_ville,
affichage=False, ville='à remplir'):
0204|     if affichage:
0205|         contenances()
0206|     c2.execute('''
0207|     select s.id_groupe, graphe.route_id,
graphe.route_type, nb_trips, routes.route_short_name
0208|         from graphe
0209|         join (select id_groupe
```

```
0210|                 from stops_groupe
0211|                 where code_commune={}) as s
0212|             on s.id_groupe=graphe.from_id_groupe
0213|         join routes
0214|             on routes.route_id=graphe.route_id
0215|             where graphe.route_type>=0 and
graphe.route_type<=3
0216|         order by graphe.route_type,
graphe.route_id'''.format(code_commune))
0217|     #on récupère, toutes les liaisons dont le departure_id
est dans la ville de code_commune donné
0218|     weight=[]
0219|     dernier_route_id='init'
0220|     #on répartit le flux_ville entre les différentes
lignes de la ville
0221|     #le poids de chaque ligne est nb_trips*contenance
0222|     #fact est donc le nombre de voyageurs qui transitent
par la ligne route_id
0223|     for id_groupe, route_id, route_type, nb_trips, name in
c2:
0224|         if route_id==dernier_route_id:
0225|             weight[-1][-1]+=(id_groupe,)
0226|         else:
0227|             dernier_route_id=route_id
0228|
fact=remplissage_contenance[int(route_type)]*nb_trips
0229|             weight.append([fact, route_type, name,
route_id, (id_groupe,)])
0230|         print(id_groupe, route_id, nb_trips, name, fact)
0231|     tot_modes=somme(weight, 0)
0232|     if affichage:
0233|         print(weight)
0234|         repr_distrib_ville(weight, flux_ville, ville,
tot_modes)
0235|     #puis pour chaque ligne, on répartit ce flux entre les
poles (de manière équitable puisqu'on suppose que l'attractivité
se joue à l'echelle d'une ville)
0236|     for ligne in weight:
0237|         fact, route_type, name, route_id, poles = ligne
0238|         freq_ligne = fact/tot_modes*flux_ville
0239|         freq_pole = freq_ligne/len(poles)
0240|         # print('\n'+name)
0241|         for pol in poles:
0242|             update_freq_m(pol, freq_pole, route_id)
0243|
0244| def update_freq_m(pol, freq_pole, route_id):
0245|     c2.execute('select frequentation_m from flux_emis
where id_groupe={i} and route_id="{rid}"'.format(i=pol,
rid=route_id))
0246|     freq=c2.fetchone()
```

```
0247|     if freq==None:
0248|         c2.execute('insert into flux_emis (id_groupe,
route_id, frequentation_m, frequentation_c) values (?,?,?,?)',
(pol, route_id, freq_pole, 0))
0249|     else:
0250|         freq=freq[0]
0251|         c2.execute('''
0252|         update flux_emis
0253|         set frequentation_m={f}
0254|             where id_groupe={p} and route_id="{rid}"
0255|         '''.format(p=pol, f=freq_pole+freq, rid=route_id))
0256|
0257|
0258| def repr_distrib_ville(weight, flux_ville, ville,
tot_modes):
0259|     sizes=[[] for k in range(4)]
0260|     labels=[[] for k in range(4)]
0261|     labels_globaux=['tramway', 'metro', 'RER', 'bus']
0262|     fig_rep = pl.figure(figsize=[23, 16])
0263|     X = [(1, 3, 1), (2, 3, 2), (2, 3, 3), (2, 3, 5), (2,
3, 6)]
0264|     axes=[]
0265|     for nrows, ncols, plot_number in X:
0266|         sb=fig_rep.add_subplot(nrows, ncols, plot_number)
0267|         axes.append(sb)
0268|     rep_ax=axes[0]
0269|     rep_ax.set_title('répartition des flux émis
('+str(flux_ville)+' voyageurs) en fonctions des différents
modes à '+ville)
0270|     for ligne in weight:
0271|         fact, route_type, name, route_id, poles = ligne
0272|         freq_ligne = fact/tot_modes*flux_ville
0273|         print(freq_ligne)
0274|         if route_type<4 and freq_ligne>1:
0275|             sizes[route_type].append(freq_ligne)
0276|             labels[route_type].append(name)
0277|     t=[sum(sizes[k]) for k in range(4)]
0278|     print(sizes, sum(t))
0279|     sizes_globaux=[sum(sizes[k]) for k in range(4)]
0280|     for k in range(3, -1, -1):
0281|         axes[k+1].pie(sizes[k], labels=labels[k], autopct
= lambda x: str(int(x)) + '%')
0282|         if len(sizes[k])>0:
0283|             axes[k+1].set_title('répartition par lignes de
'+labels_globaux[k])
0284|         if sizes_globaux[k]==0:
0285|             sizes_globaux=sizes_globaux[:k]
+sizes_globaux[k+1:]
0286|             labels_globaux=labels_globaux[:k]
+labels_globaux[k+1:]
```

```
0287|     rep_ax.pie(sizes_globaux, labels=labels_globaux,
autopct= lambda x : int(x*flux_ville/100))
0288|     pl.savefig('répartition modale test', dpi=300,
bbox_inches='tight')
0289|     pl.show()
0290| #execute_repartition_flux(75105, 16842.0, affichage=True,
ville='Paris 05')
0291|
0292| def representation_flux_emis():
0293|     c.execute('''
0294|     select x, y, sum(frequentation_m), ""
0295|     from flux_emis
0296|     join stops_groupe
0297|         on stops_groupe.id_groupe=flux_emis.id_groupe
0298|     group by flux_emis.id_groupe''')
0299|     representation_flux(500, 10)
0300|     pl.savefig('flux émis par pôle', dpi=300,
bbox_inches='tight')
0301|     pl.show()
0302|
0303|
0304|
0305|
0306| ##normalisation du flux emis
0307| '''deux approches:
0308| 1) on veut minimiser les écarts au carré de flux émis en
choisissant le facteur multiplicatuif
0309| 2) on veut que la totalité des flux émis par les gares de
la RATP corresponde aux données
0310| '''
0311| #valeur de référence donnée par RATP (frequentation_c)
0312|
0313| def calcul_ecart(facteur_population):
0314|     c.execute('''
0315|     select sum(frequentation_m), sum(frequentation_c)
0316|     from flux_emis
0317|     group by id_groupe''')
0318|     ecart=0; ecart_flat=0
0319|     for frequentation_m, frequentation_c in c:
0320|         if frequentation_c!=0:
0321|             #
ecart_flat+=abs(frequentation_m*facteur_population-
frequentation_c)
0322|
ecart+=np.sqrt((frequentation_m*facteur_population-
frequentation_c)**2)
0323|     return np.sqrt(ecart)
0324|
0325| def calcul_ecart_par_ligne(facteur_population):
0326|     c.execute('''
```

```
0327|     select frequentation_m, frequentation_c
0328|     from flux_emis
0329|     where frequentation_c!=0''')
0330|     ecart=0; ecart_flat=0
0331|     for frequentation_m, frequentation_c in c:
0332|         if frequentation_c!=0:
0333|             #
ecart_flat+=abs(frequentation_m*facteur_population-
frequentation_c)
0334|
ecart+=np.sqrt((frequentation_m*facteur_population-
frequentation_c)**2)
0335|     return np.sqrt(ecart_flat)
0336|
0337| def calcul_facteur_normalisation():
0338|     c.execute('''
0339|     select sum(frequentation_m), sum(frequentation_c)
0340|     from flux_emis
0341|     group by id_groupe''')
0342|     sum_c=0; sum_m=0
0343|     for frequentation_m, frequentation_c in c:
0344|         if frequentation_c!=0:
0345|             sum_c+=frequentation_c
0346|             sum_m+=frequentation_m
0347|     K = sum_c/sum_m
0348|     return K
0349|
0350| # calcul_facteur_normalisation()
0351| # >> 4.169361260527865
0352|
0353| #fonction vectorielle pour le gradient descent
0354| def calcul_ecart_vecto(facteur_population):
0355|     f=facteur_population[0]
0356|     return calcul_ecart(f)
0357|
0358| def repr_ecarts():
0359|     X=np.arange(0, 5, 0.05)
0360|     Y=[]
0361|     for x in X:
0362|         Y.append(calcul_ecart_par_ligne(x))
0363|     fig, ax = pl.subplots(figsize=[24, 12])
0364|     title='valeur absolue des écarts entre les flux émis
selon les données RATP et selon le modèle 1'
0365|     ax.set_title(title)
0366|     ax.plot(X, Y)
0367|     pl.xlabel('facteur_population')
0368|     pl.savefig(title, dpi=300, bbox_inches='tight')
0369|     pl.show()
0370|
0371| '''
```

```
0372| minimiser(0, 5, 0.0001, calcul_ecart)=1.9097520275672006
par méthode du nombre d'or
0373| minimum(calcul_ecart_vecto, [1], [[0, 5]], [0.001], 0.001,
0.01, False) par descente de gradient
0374| '''
0375|
0376| facteur_population=1.9
0377| # facteur_population=2.1 #par ligne carré
0378| # facteur_population=3.34 #carré
0379| # facteur_population=1.9 #flat
0380|
0381|
0382| def scalairisation_flux_emis():
0383|     c.execute('select id_groupe, route_id, frequentation_m
from flux_emis')
0384|     for id_groupe, route_id, frequentation_m in c:
0385|         c2.execute('update flux_emis set
frequentation_m={}'.format(facteur_population*frequentation_m))
0386|     conn.commit()
0387|
0388| def repr_ecarts_ligne(par_ligne=False, xn=630000,
xm=670000, yn=6845000, ym=6875000):
0389|     fig_emis, axes = pl.subplots(1,2, figsize=[24, 12])
0390|     ax_m, ax_c = axes
0391|     ax_m.axis('equal'); ax_c.axis('equal'); grid(ax_m);
grid(ax_c)
0392|     ax_c.set_title("flux emis par jour d'après les données
RATP \n soit 289 pôles (Lambert 93 en km)")
0393|     ax_m.set_title("flux modélisés sur ces mêmes pôles")
0394|     #1=ax_m (modèle) ; 2=ax_c (théorie)
0395|     if par_ligne:
0396|         c.execute('''
0397|         select stops_groupe.x, stops_groupe.y,
frequentation_m, frequentation_c, stop_name
0398|         from stops_groupe
0399|         join flux_emis
0400|             on flux_emis.id_groupe=stops_groupe.id_groupe
0401|         where (stops_groupe.x between {xn} and {xm} ) and
(stops_groupe.y between {yn} and {ym} )
0402|         '''.format(xn=xn, xm=xm, yn=yn, ym=ym))
0403|     else:
0404|         c.execute('''
0405|         select stops_groupe.x, stops_groupe.y,
sum(frequentation_m), sum(frequentation_c), stop_name
0406|         from stops_groupe
0407|         join flux_emis
0408|             on flux_emis.id_groupe=stops_groupe.id_groupe
0409|         where (stops_groupe.x between {xn} and {xm} ) and
(stops_groupe.y between {yn} and {ym} )
0410|         group by stops_groupe.id_groupe
```

```
0411|         '''.format(xn=xn, xm=xm, yn=yn, ym=ym))
0412|     X, Y, S1, S2 = [], [], [], []
0413|     compt=0
0414|     for x, y, f1, f2, stop_name in c:
0415|         if f2!=0:
0416|             compt+=1
0417|             xs, ys = x/1000, y/1000
0418|             X.append(xs)
0419|             Y.append(ys)
0420|             s1, s2 = f1*facteur_population/1000, f2/1000
0421|             # print(s1, s2)
0422|             S1.append(s1)
0423|             S2.append(s2)
0424|             if s1==0:
0425|                 ax_m.scatter(xs, ys, s=4, color='grey')
0426|             if s1>500 or s2>300:
0427|                 ax_m.text(xs, ys, s="")
0428|                 ax_c.text(xs, ys, s="")
0429|     scat1 = ax_m.scatter(X, Y, s=S1)
0430|     scat2 = ax_c.scatter(X, Y, s=S2)
0431|     handles1, labels1 =
scat1.legend_elements(prop="sizes", num=np.arange(10, 60, 10))
0432|     handles2, labels2 =
scat2.legend_elements(prop="sizes", num=np.arange(10, 60, 10))
0433|     ax_m.legend(handles1, labels1, loc='lower right',
title='flux emis par jour (milliers de déplacements)')
0434|     ax_c.legend(handles2, labels2, loc='lower right',
title='flux emis par jour (milliers de déplacements)')
0435|     pl.savefig('flux emis sur la ligne B
'.format(facteur_population), dpi=300, bbox_inches='tight')
0436|     pl.show()
0437|
0438|
0439| ## suppression des exceptions
0440| #on prend les trains rajoutés n'assurant pas un trajet
régulier et on les supprime
0441| def exceptions():
0442|     c.execute('''
0443|     delete from stop_times
0444|     where stop_times.trip_id in(
0445|     select trips.trip_id
0446|     from trips
0447|     join calendar_dates
0448|         on calendar_dates.service_id=trips.service_id
0449|     where exception_type=1 and calendar_dates.service_id
not in(
0450|             select service_id
0451|                 from calendar))''')
0452|     c.execute('''
0453|     delete from trips
```

```
0454|     where trip_id in(
0455|     select trips.trip_id
0456|     from trips
0457|     join calendar_dates
0458|         on calendar_dates.service_id=trips.service_id
0459|     where exception_type=1 and calendar_dates.service_id
not in(
0460|             select service_id
0461|                 from calendar))''')
0462|     conn.commit()
0463|
0464| ##creation de groupe
0465| #dans la DB, plusieurs stop_id designent la même gare (un
stop_id par ligne de la gare + un stop_id par stop_area (=entrée
de la gare) )
0466| def enlever_proche(liste):
0467|     compt = 0
0468|     non_traite = liste
0469|     stopid_groupeid=[]
0470|     while len(non_traite)>1:
0471|         stop_id1, lat1, lon1 = non_traite[0]; stop_id2,
lat2, lon2 = non_traite[1]
0472|         non_traite.remove(non_traite[0])
0473|         stopid_groupeid.append([compt, stop_id1])
0474|         indice = 0
0475|         while  coordonnee_proche(lat1, lat2, eps_lat) and
indice<len(non_traite)-1:  #la liste est classée par latitude
0476|             if coordonnee_proche(lon1, lon2, eps_lon):
#je ne compare longitude que pour deux latitude proches
0477|                 non_traite.remove((stop_id2, lat2, lon2))
0478|                 stopid_groupeid.append([compt, stop_id2])
#meme id_groupe (=compt) si proches
0479|             else:
0480|                 indice+=1
0481|             stop_id2, lat2, lon2 = non_traite[indice]
0482|         compt+=1
0483|         if compt%1000==0:
0484|             print(compt)
0485|     stopid_groupeid.append([compt, non_traite[0][0]])
0486|     return stopid_groupeid, compt
0487|
0488|
0489| def coordonnee_proche(lat_long1, lat_long2, eps):
0490|     if abs(lat_long1-lat_long2)<eps:
0491|         return True
0492|     return False
0493|
0494| lat_ref, lon_ref = 48.8, 2.3
0495| rayon_reel=100 #distance en m
0496| def choix_eps(distance):
```

```
0497|     x_ref, y_ref = WGS84_to_lambert93(lat_ref, lon_ref)
0498|     x, y = x_ref, y_ref
0499|     lat, lon = lat_ref, lon_ref
0500|     while dist(x, y, x_ref, y_ref)<distance:
0501|         lat+=0.00001
0502|         x, y = WGS84_to_lambert93(lat, lon_ref)
0503|     eps_lat=lat-lat_ref
0504|
0505|     x, y = x_ref, y_ref
0506|     lat, lon = lat_ref, lon_ref
0507|     while dist(x, y, x_ref, y_ref)<distance:
0508|         lon+=0.00001
0509|         x, y = WGS84_to_lambert93(lat_ref, lon)
0510|     eps_lon=lon-lon_ref
0511|     return eps_lat, eps_lon
0512|
0513| #on trouve
0514| eps_lat, eps_lon = choix_eps(rayon_reel)
0515|
0516| def regroupement():
0517|     c.execute('delete from groupe')
0518|     conn.commit()
0519|     c.execute('select stop_id, stop_lat, stop_lon from
stops order by stop_lat')
0520|     liste = c.fetchall()
0521|     stopid_groupeid, compt = enlever_proche(liste)
0522|     print('il y a', compt, 'id_groupe')
0523|     compt = 0
0524|     for id_groupe, stop_id in stopid_groupeid:
0525|         c.execute('insert into groupe(id_groupe, stop_id)
values (?,?)', (id_groupe, stop_id))
0526|         if compt%3000==0:
0527|             print(compt)
0528|         compt+=1
0529|     conn.commit()
0530|     print('done\n')
0531|     remplissage_stops_groupe()
0532|
0533|
0534| ## validité du regroupement
0535| #on regarde le nombre de stop_id regroupés et à quelle
gare ça correspond
0536| def verification_nombre_par_regroupement(condition):
0537|     c.execute('select count(stop_id) as compt, id_groupe
from groupe group by id_groupe having compt>3 order by compt
desc limit 5')
0538|     liste=c.fetchall()
0539|     print(liste)
0540|     time.sleep(3)
0541|     if condition:
```

```
0542|         for count, id_groupe in liste:
0543|             c.execute('''
0544|             select stop_name
0545|             from stops
0546|             join groupe
0547|                 on stops.stop_id = groupe.stop_id
0548|             where
groupe.id_groupe={}'''.format(id_groupe))
0549|             print(id_groupe,
traitement_tuple(c.fetchall()), '\n')
0550|
0551| # on regarde s'il y a des transfers (=correspondance) de
moins d'une minute entre deux stop_id non regroupés
0552| def verification_regroupements_transfers():
0553|     c.execute('''select g1.id_groupe, t1.stop_name,
g2.id_groupe, t2.stop_name, transfers.transfer_time
0554|     from transfers
0555|     join groupe as g1
0556|         on g1.stop_id=transfers.from_stop_id
0557|     join groupe as g2
0558|         on g2.stop_id=transfers.to_stop_id
0559|
0560|     join stops_groupe as t1
0561|         on t1.id_groupe=g1.id_groupe
0562|     join stops_groupe as t2
0563|         on t2.id_groupe=g2.id_groupe
0564|
0565|     where g1.id_groupe!=g2.id_groupe and
transfers.transfer_time<=60
0566|     group by g1.id_groupe, g2.id_groupe
0567|     order by transfer_time
0568|     limit 50''')
0569|     return c.fetchall()
0570|
0571|
0572|
0573| def validation_regroupement(xmin=660000, xmax=662000,
ymin=6870000, ymax=6872000):
0574|     fig_GTFS, axes = pl.subplots(1,2, figsize=[24, 12])
0575|     ax2, ax3 = axes
0576|     ax2.axis('equal'); ax3.axis('equal')
0577|     c.execute('select count(stop_id) from stops')
0578|     N1 = c.fetchone()[0]
0579|     c.execute('select count(id_groupe) from stops_groupe')
0580|     N2 = c.fetchone()[0]
0581|     c.execute('select stop_lat, stop_lon from stops')
0582|     n1=0
0583|     for lat, lon in c:
0584|         x, y = WGS84_to_lambert93(lat, lon)
0585|         if x<xmax and x>xmin and y<ymax and y>ymin:
```

```
0586|             ax2.plot(x/1000, y/1000, marker='o')
0587|             n1+=1
0588|     c.execute('''
0589|     select x, y
0590|     from stops_groupe
0591|     where x<{xmax} and x>{xmin} and y<{ymax} and
y>{ymin}'''.format(xmax=xmax, xmin=xmin, ymax=ymax, ymin=ymin))
0592|     n2=0
0593|     for x,y in c:
0594|         n2+=1
0595|         ax2.plot(x/1000, y/1000, marker='+')
0596|         ax3.plot(x/1000, y/1000, marker='o', markersize=8)
0597|         centre=x/1000, y/1000
0598|         creation_cercle(centre, (rayon_reel+10)/1000, ax2)
0599|     grid(ax3)
0600|     ax2.set_title('position de ' + str(n1) + ' pôles avant
regroupement dans une zone de 2 km \n (lambert93 en km)\n total:
' + str(N1))
0601|     ax3.set_title('position de ' + str(n2) + ' pôles après
regroupement dans cette même zone \n total: ' + str(N2))
0602|     pl.savefig('efficacité du regroupement spatial',
dpi=300, bbox_inches='tight')
0603|     pl.show()
0604|
0605| def creation_cercle(centre, rayon, ax):
0606|     x, y = centre
0607|     X=[x+rayon*np.cos(teta) for teta in np.linspace(0,
2*np.pi, 30)]
0608|     Y=[y+rayon*np.sin(teta) for teta in np.linspace(0,
2*np.pi, 30)]
0609|     ax.plot(X, Y)
0610|
0611| ##Stops_groupe
0612| #on utilise les coordonnes de lambert (projection conique)
pour pouvoir représenter le réseau
0613| def remplissage_stops_groupe():
0614|     c.execute('delete from stops_groupe')
0615|     c.execute('''
0616|     select stops.stop_lat, stops.stop_lon,
groupe.id_groupe, stop_name, sum(frequentation_c)
0617|         from stops
0618|         join groupe
0619|             on stops.stop_id = groupe.stop_id
0620|         group by groupe.id_groupe''')
0621|     for lat, lon, id_groupe, stop_name, freq in c:
0622|         x, y = WGS84_to_lambert93(lat, lon)
0623|         c2.execute('insert into stops_groupe(id_groupe,
stop_name, x, y, frequentation_c) values (?,?,?,?,?)',
(id_groupe, stop_name, x, y, freq))
0624|     conn.commit()
```

```
0625|     print('done \n')
0626|
0627| #on se donne un carré n*n que l'on fait grandir jusqu'a
trouver une ville correspondant à la gare
0628| def remplissage_code_commune():
0629|     c.execute('select id_groupe, x, y, n_maillage from
stops_groupe')
0630|     compt=0
0631|     for id_groupe, x, y, n_maillage in c:
0632|         data=trouver_ville(n_maillage)
0633|         code_commune=plus_proche(data, x, y)
0634|         c2.execute('update stops_groupe set
code_commune={c} where id_groupe={i}'.format(c=code_commune,
i=id_groupe))
0635|         compt+=1
0636|         if compt%1000==0:
0637|             print(compt)
0638|     conn.commit()
0639|
0640|
0641| def trouver_ville(n_maillage):
0642|     carre=carre_n_maillage(n_maillage-3*m-3, 7)
0643|     c2.execute('select x, y, code_commune, n_maillage from
mobilites_pro where n_maillage in {}'.format(carre))
0644|     data=c2.fetchall()
0645|     if len(data)==0:
0646|         print('non trouvé') #on agrandit suffisement la
séléction
0647|         carre=carre_n_maillage(n_maillage-4*m-4, 9)
0648|         c2.execute('select x, y, code_commune, n_maillage
from mobilites_pro where n_maillage in {}'.format(carre))
0649|         data=c2.fetchall()
0650|     return data
0651|
0652| def plus_proche(data, x, y):
0653|     d=np.inf
0654|     for x1, y1, code_commune, n_maillage in data:
0655|         d1=dist(x, y, x1, y1)
0656|         if d1<d:
0657|             d=d1
0658|             code=code_commune
0659|     return code
0660|
0661| def repr_attribution_ville(xmin=660000, xmax=675000,
ymin=6870000, ymax=6885000):
0662|     fig_comm=pl.figure(figsize=[22, 15])
0663|     comm_ax=pl.axes(title='attribution des pôles aux
villes adjacentes dans un rayon de 15 km (Lambert 93 en km)')
0664|     comm_ax.axis('equal')
0665|     c.execute('''
```

```
0666|     select stops_groupe.x, stops_groupe.y,
mobilites_pro.x, mobilites_pro.y, mobilites_pro.name
0667|     from stops_groupe
0668|     join mobilites_pro
0669|         on
mobilites_pro.code_commune=stops_groupe.code_commune
0670|     where stops_groupe.x<{xmax} and stops_groupe.x>{xmin}
and stops_groupe.y<{ymax} and stops_groupe.y>{ymin}
0671|     '''.format(xmax=xmax, xmin=xmin, ymax=ymax,
ymin=ymin))
0672|     for x1, y1, x2, y2, ville_name in c:
0673|         x1k, y1k, x2k, y2k = x1/1000, y1/1000, x2/1000,
y2/1000
0674|         comm_ax.plot(x1k, y1k, marker='o')
0675|         comm_ax.plot(x2k, y2k, marker='s', markersize=10,
color='red')
0676|         comm_ax.text(x=x2k, y=y2k, s=str(ville_name))
0677|         comm_ax.annotate(s='', xy=(x2k, y2k),
xytext=(x1k,y1k),  arrowprops=dict(arrowstyle="->", lw=0.5,
mutation_scale=1))
0678|     grid(comm_ax)
0679|     pl.savefig('attribution_ville', dpi=300,
bbox_inches='tight')
0680|     pl.show()
0681|
0682| ##maillage
0683| #pour limiter le coût des calculs de gares proches, on
associe à chaque id_groupe un numero correspondant à un
cadrillage n*m
0684| #chaque case est de coté 1000m
0685| # n:lignes, m:colonnes --> numerotation par ligne en
partant de bas à gauche
0686|
0687| def limites():
0688|     global minx, miny, maxx, maxy, n, m
0689|     c.execute('select min(x), min(y), max(x), max(y) from
mobilites_pro')
0690|     [(minx1, miny1, maxx1, maxy1)] = c.fetchall()
0691|     c.execute('select min(x), min(y), max(x), max(y) from
stops_groupe')
0692|     [(minx2, miny2, maxx2, maxy2)] = c.fetchall()
0693|     minx=min(minx1, minx2); miny=min(miny1, miny2);
maxx=max(maxx1, maxx2); maxy=max(maxy1, maxy2)
0694|     n=(maxy-miny)//1000+2; m=(maxx-minx)//1000+2
0695|     n, m = int(n), int(m)
0696|     c.execute('delete from global_data')
0697|     c.execute('insert into global_data (n_ligne,
m_colonne, minx, miny, maxx, maxy) values (?,?,?,?,?,?)',(n,m,
minx, miny, maxx, maxy ))
0698|     conn.commit()
```

```
0699|
0700|
0701| def maillage():
0702|     limites()
0703|     compt=0
0704|     print("maillage jusqu'à", n*m)
0705|     for lig in range(n):
0706|         for col in range(m): # case de 0 --> n*m-1
0707|             x=(minx//1000)*1000+col*1000
0708|             y=(miny//1000)*1000+lig*1000
0709|             c.execute('''update stops_groupe
0710|             set n_maillage={num}
0711|             where x between {x0} and {x0}+1000
0712|             and y between {y0} and {y0}+1000
0713|             '''.format(x0=x, y0=y, num=compt))
0714|             c.execute('''update mobilites_pro
0715|             set n_maillage={num}
0716|             where x between {x0} and {x0}+1000
0717|             and y between {y0} and {y0}+1000
0718|             '''.format(x0=x, y0=y, num=compt))
0719|             compt+=1
0720|             if compt%1000==0:
0721|                 print(compt)
0722|     conn.commit()
0723|
0724|
0725| ##maj de l'importance
0726| #on prend le carré 3*3 pour etre plus exact
0727| #le centre est donc n_maillage+m+1
0728| def importance(liste, n_maillage):
0729|     c.execute('insert into importance(n_maillage, n_gares)
values(?,?)', (n_maillage+m+1, len(liste)))
0730|
0731| #on veut aussi remplir les bordures
0732| def completer_importance():
0733|     for col in range(m):
0734|         c.execute('insert into importance(n_maillage,
n_gares) values(?,?)', (col, 0))
0735|         c.execute('insert into importance(n_maillage,
n_gares) values(?,?)', (col+(n-1)*m, 0))
0736|     for lig in range(1, n-1):
0737|         c.execute('insert into importance(n_maillage,
n_gares) values(?,?)', (lig*m, 0))
0738|         c.execute('insert into importance(n_maillage,
n_gares) values(?,?)', (lig*m-1, 0))
0739|
0740| ##graphe du reseau
0741| "il est plus rapide de tout récuperer depuis stop_times et
de sélectionner progressivement les infos intéressantes. La
requête SQL trop complexe n'aboutissait pas"
```

```
0742|
0743| #on retrouve le graphe du réseau à partir de stop_times
(=horaires de chaque train)
0744| #on traite par route_id pour supprimer les trip directs
0745| #calcul des distances seulement si il y a une liaison
(=voie) ou si les gares sont proches (à pied)
0746| def selection_graphe():
0747|     print('recuperation de stop_times')
0748|     c.execute('''
0749|     select routes.route_type, selection_trip.route_id,
groupe.id_groupe, stop_times.stop_sequence
0750|     from stop_times
0751|     join (
0752|             select trips.trip_id, trips.route_id,
trips.service_id
0753|             from (
0754|                     select service_id, count(service_id)
as nombre
0755|                     from trips
0756|                     group by service_id) as selection_ser
0757|             join trips
0758|                 on
trips.service_id=selection_ser.service_id
0759|             where selection_ser.nombre>5) as
selection_trip
0760|         on selection_trip.trip_id = stop_times.trip_id
0761|     join groupe
0762|         on groupe.stop_id=stop_times.stop_id
0763|     join calendar
0764|         on calendar.service_id=selection_trip.service_id
0765|     join routes
0766|         on routes.route_id=selection_trip.route_id
0767|     where selection_trip.route_id!="800:TER" and
calendar.start_date<20200522 and calendar.stop_date>20200522
0768|     order by selection_trip.route_id,
selection_trip.trip_id, stop_times.stop_sequence
0769|     ''')
0770|     print('liste recupere')
0771|
0772|
0773| # regroupe est ordonne selon cette forme
0774| # liste_dest=[[route_id, dest], [route_id2, dest2]]
0775| # dest=[[id_groupe1, id_groupe2, id_groupe3], [id_groupe7,
id_groupe8]]
0776| #c est params car cette fonction est réutilisée dans
données_traffic
0777| def creation_liste_dest(c):
0778|     route_id_en_cours="init"
0779|     route_type_en_cours="init"
0780|     dernier_stop_seq="init"
```

```
0781|     compt = 0
0782|     liste_dest=[]
0783|     dest=[]
0784|     for route_type, route_id, id_groupe, stop_sequence in
c:
0785|         if route_id==route_id_en_cours:
0786|             if stop_sequence!=dernier_stop_seq+1:
#nouveau trip
0787|                 dest.append([id_groupe])
0788|                 dernier_stop_seq=stop_sequence
0789|             else: #je continue un trip_id
0790|                 dest[-1].append(id_groupe)
0791|                 dernier_stop_seq+=1
0792|         else: #fin de la ligne (route_id)
0793|             if compt!=0:
0794|                 liste_dest.append([route_type_en_cours,
route_id_en_cours, dest])
0795|             dest=[[id_groupe]] #on réinitialise et on
ajoute le depart
0796|             route_id_en_cours=route_id
0797|             route_type_en_cours=route_type
0798|             dernier_stop_seq=stop_sequence
0799|         compt+=1
0800|         if compt%100000==0:
0801|             print(compt)
0802|     liste_dest.append([route_type_en_cours,
route_id_en_cours, dest]) #terminer
0803|     print('liste_dest crée')
0804|     return liste_dest
0805|
0806|
0807| def remplissage_graphe():
0808|     c.execute('delete from graphe')
0809|     conn.commit()
0810|     selection_graphe()  #je recupere les stop_times
0811|     liste_dest=creation_liste_dest(c)
0812|     liste_dest=prep_suppr_double(liste_dest)
0813|     liste_dest=traitement_directs(liste_dest)
0814|     print('directs traités', len(liste_dest), 'lignes')
0815|     for route_type, route_id, dest in liste_dest:
0816|         print(route_id)
0817|         for destination in dest:
0818|             insertion_une_destination(destination,
route_type, route_id)
0819|     conn.commit()
0820|     suppression_doublons_graphe()
0821|     print('graphe recopie \n')
0822|
0823| def insertion_une_destination(destination, route_type,
route_id):
```

```
0824|     compt=0
0825|     dernier_id_groupe='init'
0826|     for id_groupe in destination:
0827|         if compt!=0:
0828|             dist=calcul_distance(dernier_id_groupe,
id_groupe)
0829|             c.execute('insert into graphe(from_id_groupe,
to_id_groupe, route_type, route_id, distance) values
(?,?,?,?,?)', (dernier_id_groupe, id_groupe, route_type,
route_id, dist))
0830|         dernier_id_groupe = id_groupe
0831|         compt+=1
0832|
0833| #on veut que le graphe soit symetrique, si les gares sont
reliées elles le sont dans les deux sens
0834| #redondance des informations dans la DB mais requetes SQL
plus faciles
0835| def retablir_symetrie():
0836|     print('symetrisation')
0837|     c.execute('''
0838|     insert into graphe (from_id_groupe, to_id_groupe,
route_type, route_id, distance)
0839|         select graphe.to_id_groupe, graphe.from_id_groupe,
route_type, route_id, distance from graphe
0840|         join (select from_id_groupe, to_id_groupe from
graphe
0841|                 except
0842|             select to_id_groupe, from_id_groupe from
graphe) as selection
0843|             on
selection.from_id_groupe=graphe.from_id_groupe and
selection.to_id_groupe=graphe.to_id_groupe
0844|             ''')
0845|     conn.commit()
0846|
0847| def prep_suppr_double(liste_dest):
0848|     liste_dest_corrige=[]
0849|     for route_type, route_id, dest in liste_dest:
0850|         liste_dest_corrige.append([route_type, route_id,
suppr_double(dest)])
0851|     return liste_dest_corrige
0852|
0853| def suppr_double(dest):
0854|     new_dest=[]
0855|     for destination in dest:
0856|         if destination not in new_dest:
0857|             new_dest.append(destination)
0858|     return new_dest
0859|
0860| ##traitement directs
```

```
0861| '''je ne veut pas dans le graphe de liaisons virtuelles
créées par un train direct
0862| je regarde pour chaque liaison si il n'y en a pas une plus
longue'''
0863| #j'applique correction_directs_ligne à toutes les lignes
0864| #parfois un direct est emboité dans un plus direct encore
et il faut plusieurs itérations de correction_directs_ligne
0865| def traitement_directs(liste_dest):
0866|     liste_dest_corrige=[]
0867|     for route_type, route_id, dest in liste_dest:
0868|         nombre_chgmts=1
0869|         if route_type!=3: #les bus n'ont pas ce problème
0870|             while nombre_chgmts!=0:
0871|                 dest, nombre_chgmts =
correction_directs_ligne(dest, False)
0872|         liste_dest_corrige.append([route_type, route_id,
dest])
0873|     return liste_dest_corrige
0874|
0875| #remplace dans dest (pour un route_id donc) les trajets
directs par les omnibus
0876| #ce programme est utilisé dans donnees traffic pour
recuperer directs, dans ce cas cond_creation_directs=True et on
peut renseigner directs
0877| def correction_directs_ligne(dest, cond_creation_directs,
directs=[]):
0878|     nombre_chgmts=0
0879|     N=len(dest)
0880|     for numdest in range(N):
0881|         feuille_route=dest[numdest]
0882|         for k in range(len(feuille_route)-2, -1, -1):
0883|             gare1=feuille_route[k];
gare2=feuille_route[k+1]
0884|             liste_recherche=dest[:numdest]
+dest[numdest+1:]
0885|
intermediaires=recherche_trajet_equivalent(gare1, gare2,
liste_recherche)
0886|             if verification_boucle(intermediaires,
feuille_route):
0887|                 feuille_route=feuille_route[:k+1]
+intermediaires+feuille_route[k+1:]  #compteur descendant
0888|                 if cond_creation_directs:
0889|
directs[numdest].extend(intermediaires)
0890|                 nombre_chgmts+=len(intermediaires)
0891|         dest[numdest] = feuille_route
0892|     if cond_creation_directs:
0893|         return dest, nombre_chgmts, directs
0894|     else:
```

```
0895|         return dest, nombre_chgmts
0896|
0897| #il ne faut pas créer de boucle --> infini sinon
0898| #cette situation arrive sur les lignes ou il y a des
boucles (RER C par ex)
0899| def verification_boucle(intermediaires, destination):
0900|     for id_groupe in intermediaires:
0901|         if id_groupe in destination:
0902|             return False
0903|     return True
0904|
0905| #je recherche s'il y a un train non-direct entre gare1,
gare2
0906| def recherche_trajet_equivalent(gare1, gare2, liste_dest):
0907|     parties_changer=[]
0908|     for k in range(len(liste_dest)):
0909|         destination = liste_dest[k]
0910|         resultats=[-1, -1]
0911|         for index in range(len(destination)):
0912|             if destination[index]==gare1:
0913|                 resultats[0]=index
0914|             elif destination[index]==gare2:
0915|                 resultats[1]=index
0916|         if resultats[0]!=-1 and resultats[1]!=-1 and
abs(resultats[0]-resultats[1])>1: #il existe un trajet moins
direct
0917|             # print(gare1, gare2, k)
0918|             parties_changer = remetre_ordre(destination,
resultats, parties_changer)
0919|     if len(parties_changer)==0:
0920|         return []
0921|     else:
0922|         plus_long = selection_plus_long(gare1, gare2,
parties_changer)
0923|     return plus_long
0924|
0925| #on veut le plus long (=le moins direct) et pas de boucle
0926| def selection_plus_long(gare1, gare2, parties_changer):
0927|     plus_long=[]
0928|     for liste in parties_changer:
0929|         if len(liste)>len(plus_long) and gare1 not in
liste and gare2 not in liste:
0930|             plus_long = liste
0931|     return plus_long
0932|
0933| #le trajet intermediaire trouvé peut etre de gare2-->gare1
et pas dans le bon sens
0934| def remetre_ordre(destination, resultats,
parties_changer):
0935|     if resultats[0]<resultats[1]: #bon ordre
```

```
0936|         parties_changer.append(destination[min(resultats)
+1:max(resultats)])
0937|     else:
0938|
parties_changer.append(reverse(destination[min(resultats)
+1:max(resultats)]))  #ordre inverse
0939|     return parties_changer
0940|
0941| #après remplacement des directs par omnibus on a des
redondances
0942| def suppression_doublons_graphe():
0943|     print('suppression doublons')
0944|     c.execute('''
0945|     delete from graphe
0946|     where cle not in (
0947|     select  cle
0948|         from graphe
0949|         where from_id_groupe!=to_id_groupe
0950|         group by from_id_groupe, to_id_groupe, route_type,
route_id )''')
0951|     conn.commit()
0952|     print('sans doublons graphe \n')
0953|
0954| ##gares accessibles à pied
0955| '''route_id=-1 si à moins de 500m
0956| route_id=-2 si à moins de 1km
0957| le maillage permet de reduire le cout n*n en séquencant
l'espace
0958| on en profite pour calculer importance'''
0959| def marche():
0960|     print('ligne', n, 'colonne', m, 'total', n*m)
0961|     for lig in range(n-2):
0962|         for col in range(m-2):
0963|             n_maillage=lig*m + col
0964|             carre=carre_n_maillage(n_maillage, 3)
0965|             c.execute('select id_groupe, x, y from
stops_groupe where n_maillage in {}'.format(carre))
0966|             liste=c.fetchall()
0967|             traitement_graphe_marche(liste)
0968|             importance(liste, n_maillage)
0969|             if n_maillage%1000==0:
0970|                 print(n_maillage)
0971|     suppression_doublons_graphe()  #plus efficace de le
faire une fois à la fin
0972|     completer_importance()
0973|     conn.commit()
0974|     print('marche rempli\n')
0975|
0976| #on renseigne en bas à gauche et renvoie le carrée de k*k
0977| def carre_n_maillage(n_maillage, k):
```

```
0978|     rep=()
0979|     for ligne in range(k):
0980|         for col in range(k):
0981|             rep+=(n_maillage+ligne*m+col,)
0982|     return rep
0983|
0984| #il faudrait ne remplir qu'un demi et recopier pour gagner
du temps
0985| def traitement_graphe_marche(liste):
0986|     for id_groupe1, x1, y1 in liste:
0987|         for id_groupe2, x2, y2 in liste:
0988|             execute_marche(id_groupe1, x1, y1, id_groupe2,
x2, y2)
0989|
0990| def execute_marche(id_groupe1, x1, y1, id_groupe2, x2,
y2):
0991|     deltx=abs(x1-x2); delty=abs(x1-x2)
0992|     distance=np.sqrt(deltx**2+delty**2)
0993|     if distance<rayon_reel*3:
0994|         if distance<rayon_reel+10:
0995|             route_type=-1
0996|         else:
0997|             route_type=-2
0998|         c.execute('insert into graphe (from_id_groupe,
to_id_groupe, route_type, route_id, distance) values
(?,?,?,?,?)', (id_groupe1, id_groupe2, route_type, "marche",
distance))
0999|
1000|
1001| ##représentation graphe
1002| def grid(ax):
1003|     xmin, xmax, ymin, ymax = ax.axis()
1004|     x_ticks=np.arange(int(xmin), xmax+1, 1);
y_ticks=np.arange(int(ymin), ymax+1, 1)
1005|     ax.set_xticks(x_ticks, minor=True);
ax.set_yticks(y_ticks, minor=True)
1006|     ax.grid(which='both', alpha=0.2)
1007|
1008| titles=['tramway', 'métro', 'RER', 'bus']
1009| def representation(route_type, condition_global):  #on
choisis de tracer ligne par ligne ou tout d'un coup
1010|     fig_maill=pl.figure(figsize=[23, 16])
1011|     maill_ax=pl.axes()
1012|     maill_ax.axis('equal')
1013|     c.execute('select count(route_id) from routes where
route_type={}'.format(route_type))
1014|     N=c.fetchone()[0]
1015|     title = str(N) + ' lignes de ' + titles[route_type] +
' positionnés en km dans le système de coordonnées Lambert'
1016|     maill_ax.set_title(title)
```

```
1017|     c.execute('''
1018|     select from_id_groupe, to_id_groupe,
graphe.route_type, graphe.route_id, routes.route_long_name,
s1.x, s1.y, s2.x, s2.y
1019|     from graphe
1020|     join routes
1021|         on routes.route_id=graphe.route_id
1022|     join stops_groupe as s1
1023|         on s1.id_groupe=from_id_groupe
1024|     join stops_groupe as s2
1025|         on s2.id_groupe=to_id_groupe
1026|     where graphe.route_type={} and
from_id_groupe<to_id_groupe
1027|     order by graphe.route_id, s1.x,
s1.y'''.format(route_type))
1028|     dernier_route_id='init'
1029|     compt=0
1030|     for id1, id2, route_type, route_id, route_long_name,
x1, y1, x2, y2 in c:
1031|         if route_id!=dernier_route_id:
1032|             compt+=1
1033|             if not condition_global:
1034|                 pl.show()
1035|             if route_type!=3:
1036|                 color=random_color()
1037|                 maill_ax.plot([x1/1000, x2/1000],
[y1/1000, y2/1000], color=color, linewidth=3,
label=route_long_name)
1038|                 print(dernier_route_id)
1039|             else:
1040|                 color=[0, 0, 1]
1041|                 if compt%20==0:
1042|                     print(compt)
1043|         else:
1044|             maill_ax.plot([x1/1000, x2/1000], [y1/1000,
y2/1000], color=color, linewidth=3)
1045|         dernier_route_id = route_id
1046|     if route_type!=3:
1047|         maill_ax.legend(loc='lower right', title='nom des
lignes')
1048|     grid(maill_ax)
1049|     print(dernier_route_id)
1050|     pl.savefig('maillage ' + titles[route_type], dpi=400,
bbox_inches='tight')
1051|     pl.show()
1052|
1053| def representation_une_ligne(route_id, ax):
1054|     c.execute('''select s1.x, s1.y, s2.x, s2.y,
s1.stop_name, s1.id_groupe
1055|     from graphe
```

```
1056|     join stops_groupe as s1
1057|         on s1.id_groupe=graphe.from_id_groupe
1058|     join stops_groupe as s2
1059|         on s2.id_groupe=graphe.to_id_groupe
1060|     where graphe.route_id="{}" and
s1.id_groupe>s2.id_groupe
1061|     group by s1.stop_name, s2.stop_name
1062|     '''.format(route_id))
1063|     ax.axis('equal')
1064|     for x1, y1, x2, y2, name, id_groupe in c:
1065|         ax.plot([x1/1000, x2/1000], [y1/1000, y2/1000],
color='black')
1066|         # ax.annotate(name+' '+str(id_groupe), xy=(x1,
y1), size=6)
1067|
1068| ## outils de debug
1069| def liaisons(route_id):
1070|     c.execute('''
1071|     select s1.stop_name, s1.id_groupe, s2.stop_name,
s2.id_groupe
1072|     from graphe
1073|     join stops_groupe as s1
1074|         on s1.id_groupe=graphe.from_id_groupe
1075|     join stops_groupe as s2
1076|         on s2.id_groupe=graphe.to_id_groupe
1077|     where graphe.route_id="{}"
1078|     group by s1.stop_name, s2.stop_name
1079|     order by s1.stop_name'''.format(route_id))
1080|     return c
1081|
1082| def trip_id_passant_par(id_groupe):
1083|     c.execute('''
1084|     select trips.route_id, trips.trip_id, trips.service_id
1085|     from stop_times
1086|     join trips
1087|         on trips.trip_id=stop_times.trip_id
1088|     where stop_times.stop_id
1089|     in (select stops.stop_id
1090|             from stops
1091|             join groupe
1092|                 on groupe.stop_id=stops.stop_id
1093|             where groupe.id_groupe="{}")
1094|     group by trips.route_id'''.format(id_groupe))
1095|     return c
1096|
1097|
1098| def representation_maillage():
1099|     c.execute('select stop_lat, stop_lon from stops')
1100|     for lat, lon in c:
1101|         x,y = WGS84_to_lambert93(lat, lon)
```

```
1102|         pl.plot(x, y, marker='o')
1103|     pl.show()
1104|     c.execute('select x,y from stops_groupe')
1105|     for x,y in c:
1106|         pl.plot(x, y, marker='o')
1107|     pl.show()
1108|
1109| ##global
1110| def reseau():
1111|     print('remplissage de nb_trips\n')
1112|     remplir_nb_trips()
1113|     print('conversion Lambert93 mobilites\n')
1114|     conversion_mobilites()
1115|     print('suppression des exceptions\n')
1116|     exceptions()
1117|     print('regroupement\n')
1118|     regroupement()
1119|     maillage()
1120|     print('remplissage_graphe\n')
1121|     remplissage_graphe()
1122|     print('symétrisation du graphe\n')
1123|     retablir_symetrie()
1124|     print('attribution des villes\n')
1125|     remplissage_code_commune()
1126|     print('frequentations modèlisés\n')
1127|     remplir_flux_emis()
1128|     print('liaisons marche\n')
1129|     marche()
1130|     for route_type in range(3, -1, -1):
1131|         representation(route_type, True)
1132|     #il faut aussi actualiser les flux du RERB pour le
modèle
```