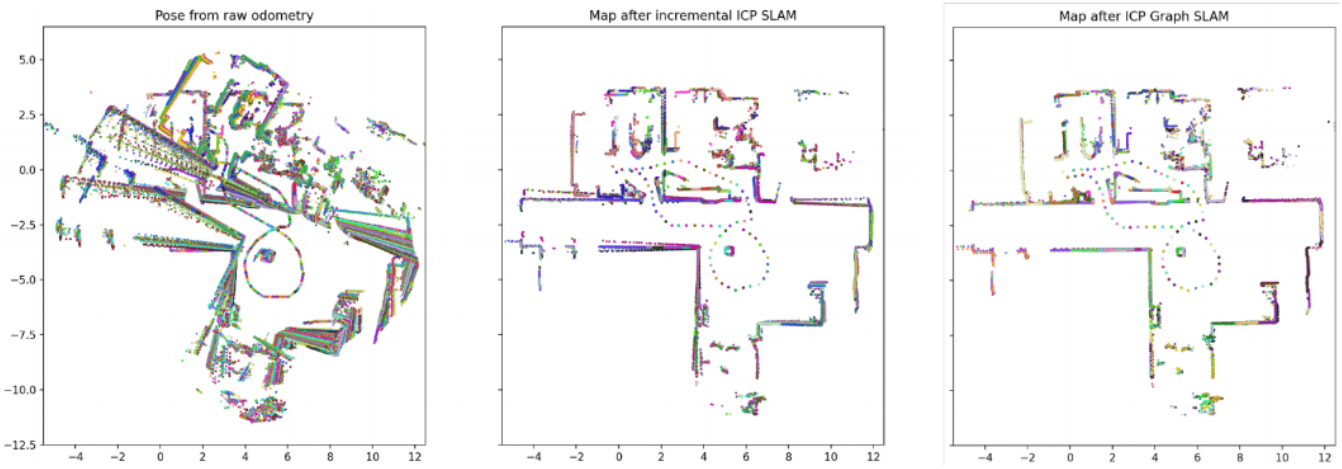
TP5 : GraphSLAM avec ICP

ROB312 - Navigation pour les systèmes autonomes

Bastien HUBERT



ENSTA Paris - novembre 2022

1 Q1

Le code ci-dessous correspond au code Python d'un SLAM (Simultaneous Localisation And Mapping) incrémental utilisant ICP (Iterative Closest Point) pour effectuer l'appariement entre la carte calculée à une itération donnée et le nuage de point obtenu par le lidar du robot à cette même itération. Ce code est appliqué au déplacement d'un robot et à la cartographie de son milieu. Il se décompose en différentes parties :

1.1 Initialisation

Cette partie correspond à l'initialisation de l'algorithme, durant laquelle on définit les paramètres de l'algorithme, notamment `dist_threshold_add` qui correspond à la distance à partir de laquelle un scan est ajouté à la carte globale, et on effectue une copie des valeurs d'odométrie pour comparer les cartes avec et sans correction par l'algorithme.

```
"""
 Incremental ICP SLAM - Basic implementation for teaching purpose only...
 Computes position of each scan with respect to closest one in the
 current map and add scan to the map if it is far enough from all the
 existing ones
 author: David Filliat
"""

import readDatasets as datasets
import matplotlib.pyplot as plt
import icp
import numpy as np
import copy


# Reading data
#scan_list = datasets.read_fr079(0)
#axis_limit = [-20, 25, -10, 30]
scan_list = datasets.read_u2is(0)
axis_limit = [-5.5, 12.5, -12.5, 6.5]

# Parameters for scan processing
min_scan = 0
step = 3
max_scan = len(scan_list) - step

# Parameters for map building
dist_threshold_add = 0.3
```

```
# Copy for reference display and map init
odom_scan_list = copy.deepcopy(scan_list)
map = [scan_list[min_scan]]

# Init displays
f, (ax1, ax2) = plt.subplots(1, 2, sharey=True, figsize=(14, 7))
c = np.random.rand(3,)
ax1.scatter(odom_scan_list[min_scan]["x"],
            odom_scan_list[min_scan]["y"], color=c, s=1)
ax1.axis(axis_limit)
ax1.set_title('Pose from raw odometry')
ax2.scatter(map[0]["x"], map[0]["y"], color=c, s=1)
ax2.axis(axis_limit)
ax2.set_title('Map after incremental ICP SLAM')
plt.pause(0.1)
```

1.2 Boucle du SLAM

Cette partie correspond à l'algorithme à proprement parler, et est responsable de la création de la carte ainsi que de la correction de la position du robot.

```
# Perform incremental SLAM
for i in range(min_scan + step, min_scan + step, step):
```

Dans un premier temps, on récupère le scan dans `map` qui est le plus proche au sens de la somme des distances entre chaque pair de point des deux scans grâce à la fonction `datasets.find_closest_scan` (il s'agit d'un algorithme glouton qui calcule la distance entre le scan courant et tous les scans contenus dans `map`) :

```
    # get list of map scan sorted by distance
    sorted_dist, sorted_id = datasets.find_closest_scan(map, scan_list[i])
    ref_scan_id = sorted_id[0]
    print('Matching new scan to reference scan ' + str(ref_scan_id))
```

On effectue un ICP entre le scan courant et le scan le plus proche contenu dans `map`, appelé scan de référence, afin d'obtenir les matrices de rotation et de translation correspondant à la transformation affine entre les deux scans :

```
    # perform ICP with closest scan
    R, t, error = icp.icp(map[ref_scan_id], scan_list[i], 200, 1e-7)
```

Le scan courant et tous les scans ultérieurs sont ensuite modifiés à partir des matrices obtenues afin de corriger les erreurs incrémentales de position et d'orientation induites par l'odométrie :

```
    # Correct all future scans odometry pose
    for j in range(i, max_scan, step):
        scan_list[j] = datasets.transform_scan(scan_list[j], R, t)
```

Si le scan corrigé est suffisamment loin du scan de référence, on l'ajoute à la carte à la fois dans `map` et dans l'affichage de `ax2`

```
# Add scan to map if it is far enough
if np.linalg.norm(scan_list[i]["pose"][0:2] -
                  map[ref_scan_id]["pose"][0:2]) > dist_threshold_add:
    map.append(scan_list[i])
    print('Added to map, new size : ' + str(len(map)))

    # Map display
    ax2.cla()
    for scan in map:
        c = np.random.rand(3,)
        ax2.scatter(scan["x"], scan["y"], color=c, s=1)
        ax2.scatter(scan["pose"][0],
                    scan["pose"][1], color=c, s=3)
    ax2.axis(axis_limit)
    ax2.set_title('Map after incremental ICP SLAM')
```

Dans tous les cas, on met à jour l'affichage de la "carte" obtenue en superposant les scans replacés à la position et l'orientation correspondant à l'odométrie pure du robot :

```
# Scan display
c = np.random.rand(3,)
ax1.scatter(odom_scan_list[i]["x"],
            odom_scan_list[i]["y"], color=c, s=1)
ax1.scatter(odom_scan_list[i]["pose"][0],
            odom_scan_list[i]["pose"][1], color=c, s=3)
plt.pause(0.1)
```

Cette opération est répétée tant qu'on n'a pas atteint le dernier scan `scan_list[min_scan]`. Une fois le dernier scan traité, il ne reste plus qu'à sauvegarder le résultat sous la forme d'une image et à l'afficher :

```
#plt.savefig('ICPIncrementalSLAM.png')
print("Press Q in figure to finish...")
plt.show()
```

L'exécution de ce code conduit à la figure 1 :

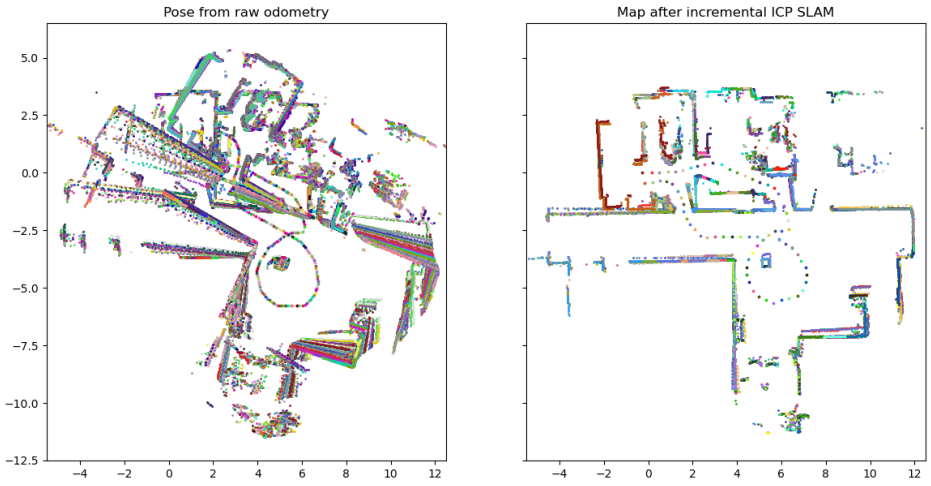


Figure 1: ICPincrementalSLAM avec les paramètres par défaut

2 Q2

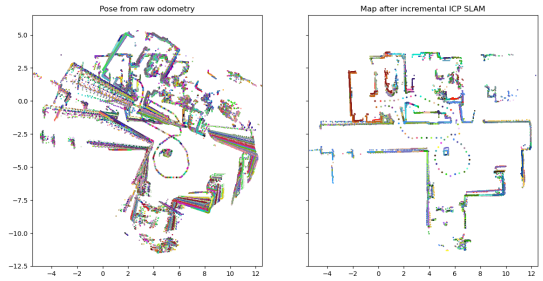
L'algorithme `ICPincrementalSLAM` ne détecte pas de fermeture de boucle, et ne corrige donc pas la carte rétrospectivement dans la cas où le robot retourne à une position qu'il avait déjà atteinte.

3 Q3

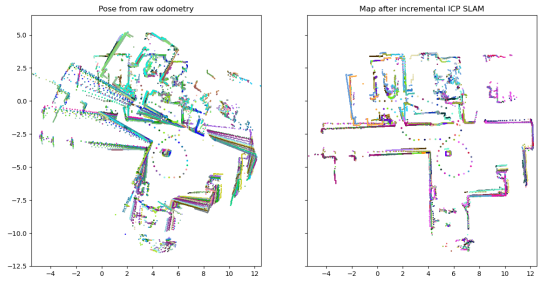
`step` est le paramètre qui détermine la fréquence de la boucle, c'est-à-dire tous les combien de scans l'algorithme va mettre à jour la carte et la position du robot.

Augmenter légèrement ce paramètre a pour effet de réduire les calculs car la fonction `datasets.find_closest_scan`, très coûteuse en temps, sera appelée moins de fois. De plus, cela diminuera localement la qualité de l'estimation de la position du robot, car une correction sera effectuée moins souvent. De même, la carte sera légèrement moins précise car le scan qui sera ajouté à la carte sera plus éloigné du scan de référence, réduisant ainsi la densité des scans constituant `map`. À l'inverse, le réduire augmente la durée des calculs mais améliore l'estimation de la position du robot et rend la carte plus dense.

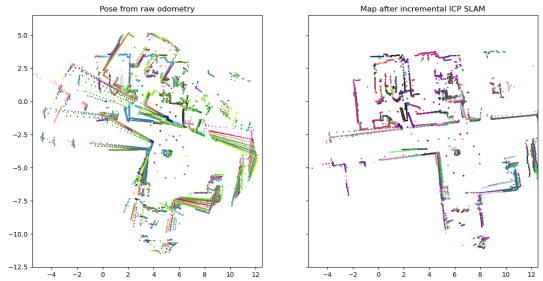
Si `step` devient trop important, des erreurs importantes peuvent apparaître lors du calcul des matrices de transformation avec ICP, particulièrement si de grands pans du scan de référence ne sont pas visibles sur le scan courant, ou si celui-ci présente des obstacles inconnus par le scan de référence. Dans ce cas, un décalage non corrigible par l'algorithme peut apparaître. Les figures 2a à 2d montrent l'impact du paramètre `step` sur la qualité de la carte :



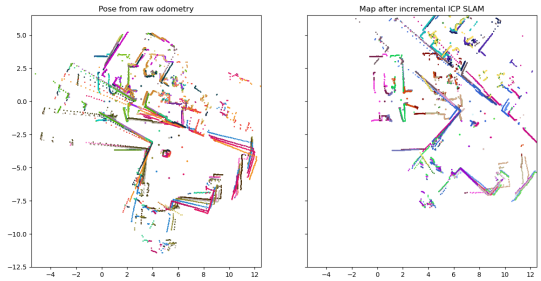
(a) $step = 3$



(b) $step = 10$



(c) $step = 20$



(d) $step = 30$

Figure 2: ICPincrementalSLAM pour différentes valeurs de *step*

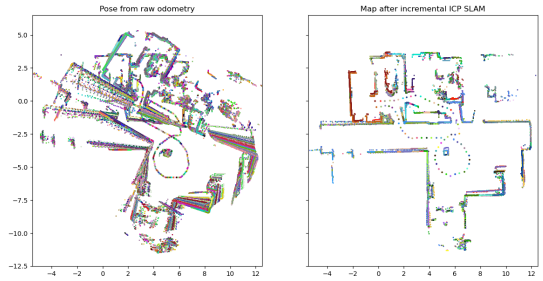
`dist_threshold_add` est le paramètre qui contrôle la distance minimale à partir de laquelle un nouveau scan sera ajouté dans `map`.

Augmenter légèrement ce paramètre réduit la densité de la carte en demandant une plus grande distance entre le scan courant et le scan de référence le plus proche. Ceci a également pour effet de réduire le temps de calcul de la fonction `datasets.find_closest_scan`, car vraisemblablement moins de scans seront ajoutés dans `map`. À l'inverse, réduire `dist_threshold_add` rend la carte plus dense mais augmente le temps de calcul nécessaire à une itération de l'algorithme.

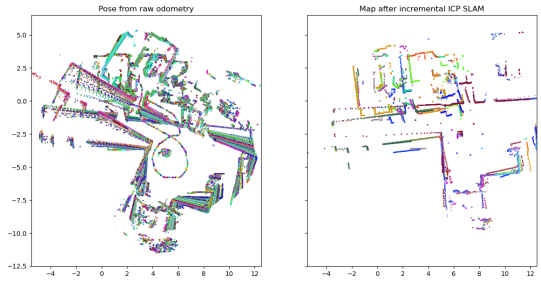
De même que pour `step`, trop augmenter `dist_threshold_add` risuqe de conduire à des erreurs non corrigibles dans ICP en raison de la trop grande distance entre le scan courant et le scan de référence (cette fois-ci car il est plus dur de devenir un scan de référence, et que donc celui qui sera choisi pour la comparaison pourra être bien trop différent du scan courant). Les figures 3a à 3d montrent l'impact du paramètre `dist_threshold_add` sur la qualité de la carte :

4 Q4

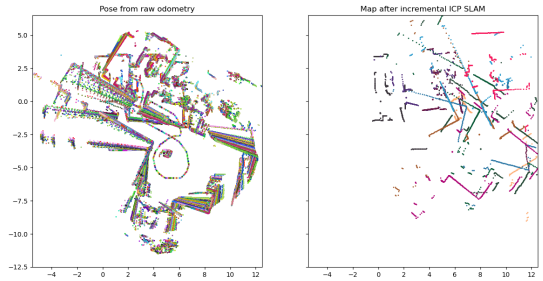
Le code ci-dessous correspond au code Python de l'algorithme GraphSLAM utilisant ICP pour effectuer l'appariement entre la carte calculée à une itération donnée et le nuage de point obtenu par le lidar du robot à cette même itération. Ce code est appliqué au déplacement d'un robot et à la cartographie de son milieu. Il se décompose en différentes parties :



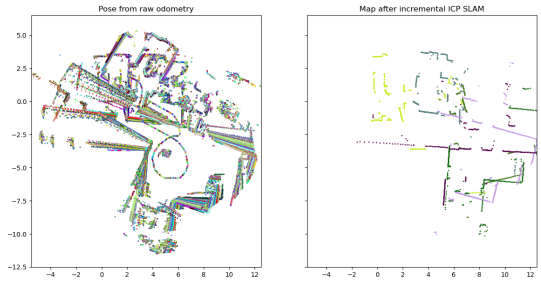
(a) *dist_threshold_add* = 0.3



(b) *dist_threshold_add* = 1



(c) *dist_threshold_add* = 2



(d) *dist_threshold_add* = 3

Figure 3: ICPincrementalSLAM pour différentes valeurs de *dist_threshold_add*

4.1 Initialisation

Cette partie correspond à l'initialisation de l'algorithme, durant laquelle on définit les paramètres de l'algorithme, notamment `dist_threshold_add` qui correspond à la distance à partir de laquelle un scan est ajouté à la carte globale, `dist_threshold_add` qui représente le seuil à partir duquel un scan de la carte est considéré comme suffisamment proche du scan courant pour pouvoir comparer les deux, et on effectue une copie des valeurs d'odométrie pour comparer les cartes avec et sans correction par l'algorithme. On définit également le paramètre `max_ICP_error` et les graphes `graph_theta`, `graph_d` et `graph_phi` qui représentent différents aspects de la topologie de la carte à construire.

```
import readDatasets as datasets
import matplotlib.pyplot as plt
import icp
import numpy as np
import copy
import math


# Reading data
#scan_list = datasets.read_fr079(0)
#axis_limit = [-20, 25, -10, 30]
scan_list = datasets.read_u2is(0)
axis_limit = [-5.5, 12.5, -12.5, 6.5]
```

```
# Parameters for scan processing
min_scan = 0
step = 3
max_scan = len(scan_list) - step

# Parameters for map building
dist_threshold_add = 0.3
dist_threshold_match = 0.8
max_ICP_error = 0.4

# Copy for reference display and map init
odom_scan_list = copy.deepcopy(scan_list)
map = [scan_list[min_scan]]

# Initialize graph of relative positions
# simple representation using matrixes
max_size = int(np.around((max_scan - min_scan)/step))
graph_theta = np.zeros((max_size, max_size)) # Change in robot direction between two nodes
graph_d = np.zeros((max_size, max_size)) # distance between two nodes
graph_phi = np.zeros((max_size, max_size)) # direction of motion between two nodes, relative



# Init displays
f, (ax1, ax2) = plt.subplots(1, 2, sharey=True, figsize=(14, 7))
c = np.random.rand(3,)
ax1.scatter(odom_scan_list[min_scan]["x"],
            odom_scan_list[min_scan]["y"], color=c, s=1)
ax1.scatter(odom_scan_list[min_scan]["pose"][0],
            odom_scan_list[min_scan]["pose"][1], color=c, s=3)
ax1.axis(axis_limit)
ax1.set_title('Poses from raw odometry')
ax2.scatter(map[0]["x"], map[0]["y"], color=c, s=1)
ax2.scatter(map[0]["pose"][0],
            map[0]["pose"][1], color=c, s=3)
ax2.axis(axis_limit)
ax2.set_title('Map after ICP Graph SLAM')
plt.pause(0.1)
```

4.2 Boucle du SLAM

Cette partie correspond à l'algorithme à proprement parler, et est responsable de la création de la carte ainsi que de la correction de la position du robot.

```
# Process scans
for i in range(min_scan + step, max_scan, step):

    print('Processing scan ' + str(i))
```

Dans un premier temps, on récupère la liste ordonnée des scans dans `map` dont la distance au sens de `datasets.find_closest_scan` est inférieure au seuil `dist_threshold_match` :

```
    # get list of map scan sorted by distance
    sorted_dist, sorted_id = datasets.find_closest_scan(map, scan_list[i])

    # Keep only the ones below the distance threshold, or the closest one
    close_scans = sorted_id[sorted_dist < dist_threshold_match]
    if len(close_scans) == 0:
        close_scans = [sorted_id[0]]
```

On effectue un ICP entre le scan courant et le scan le plus proche contenu dans `map` afin d'obtenir les matrices de rotation et de translation correspondant à la transformation affine entre les deux scans :

```
    # perform ICP with closest scan to correct future odometry
    #R, t, error, iter = icp.icp(map[close_scans[0]], scan_list[i], 200, 1e-7)
    R, t, error = icp.icp(map[close_scans[0]], scan_list[i], 200, 1e-7)
```

Le scan courant et tous les scans ultérieurs sont ensuite modifiés à partir des matrices obtenues afin de corriger les erreurs incrémentales de position et d'orientation induites par l'odométrie :

```
    # Correct all future scans odometry pose
    for j in range(i, max_scan, step):
        scan_list[j] = datasets.transform_scan(scan_list[j], R, t)
```

Le scan courant est ensuite ajouté dans la carte, un nouveau point est ajouté dans les graphes `graph_theta`, `graph_d` et `graph_phi`, et des valeurs initiales d'arêtes sont générées à partir des positions relatives du scan courant par rapport à *tous* les scans proches, servant donc tous de scans de référence ici :

```
    # --- Add scan to map and update graph if needed
    if np.linalg.norm(scan_list[i]["pose"][0:2] -
                      map[close_scans[0]]["pose"][0:2]) > dist_threshold_add:

        map.append(scan_list[i])
        print('Adding new scan with links to : ' + str(close_scans))

        # Get ref to last scan in map (i.e. new scan)
        id_new = len(map) - 1
        s_new = map[-1]

        # ---- Build graph
```

```
edge_NB = 0
for id_ref in close_scans:
    print("Adding edge")
    # take the reference scan among the closest map scan
    s_ref = map[id_ref]

    # compute position of new scan wrt the ref scan
    #Ri, ti, error, iter = icp.icp(s_ref, s_new, 200, 1e-7)
    Ri, ti, error = icp.icp(s_ref, s_new, 200, 1e-7)

    if error < max_ICP_error or edge_NB == 0:
        edge_NB += 1
        # compute absolute position of new scan
        s_new_corrected = datasets.transform_scan(s_new, Ri, ti)

        # compute relative pose with ref scan
        delta_theta = icp.angle_wrap(s_new_corrected["pose"][2] - s_ref["pose"][2])
        delta_t = s_new_corrected["pose"][0:2] - s_ref["pose"][0:2]

        # Add relative position of new scan wrt. ref scan pose in the graph
        graph_theta[id_ref, id_new] = delta_theta
        graph_d[id_ref,id_new] = np.linalg.norm(delta_t)
        graph_phi[id_ref,id_new] = np.arctan2(delta_t[1],delta_t[0]) - s_ref["pose"][

        # Fill the reverse direction in the graph
        graph_theta[id_new, id_ref] = - graph_theta[id_ref, id_new]
        graph_d[id_new,id_ref] = graph_d[id_ref,id_new]
        graph_phi[id_new,id_ref] = math.pi + np.arctan2(delta_t[1],delta_t[0]) - s_ne
```

Les arêtes du graphes sont ensuite modifiées à la manière de la détente d'un réseau de ressorts de sorte que la correction de la position relative d'un scan par rapport à un autre se propage de proche en proche dans toute la carte. Cette détente est répétée jusqu'à ce qu'un équilibre soit atteint dans le réseau de scans géré par les graphes, ou jusqu'à ce qu'un certain nombre (500 ici) d'itérations soient effectuées sans qu'un équilibre n'apparaisse :

```
# --- Optimize graph until updates fall below threshold
update_max = 1
update_NB = 0
while ( update_max > 1e-5 and update_NB < 500):
    update_max = 0
    update_NB += 1

    # Recompute each scan pose from its neighbors
    for k in range(len(map)):
        # Create a list of scan pose computed through neighbor pose and
```

```
                # relative position
                new_pose_list = []
                for l in range(len(map)):
                    if graph_d[l, k] != 0:
                        newAngle = icp.angle_wrap(map[l]["pose"][2] + graph_theta[l, k])
                        angle_link = map[l]["pose"][2] + graph_phi[l, k]
                        newX = map[l]["pose"][0] + graph_d[l, k] * math.cos(angle_link)
                        newY = map[l]["pose"][1] + graph_d[l, k] * math.sin(angle_link)

                        new_pose_list.append(newX)
                        new_pose_list.append(newY)
                        new_pose_list.append(newAngle)

                # Compute new pose as mean of positions from neighbors and update map
                new_pose = np.array([np.mean(new_pose_list[0::3]),
                                     np.mean(new_pose_list[1::3]),
                                     icp.mean_angle(new_pose_list[2::3])])
                update = abs(new_pose.reshape(-1) - map[k]["pose"].reshape(-1))
                map[k] = datasets.update_scan_pose(map[k], new_pose)
                update_max = max(max(update), update_max)

        print('New map size : ' + str(len(map)) + ', Graph Updates : ' + str(update_NB))
```

Finalement, la carte est re-générée en entier afin de prendre en compte la nouvelle structure du graphe des scans :

```
# Display
c = np.random.rand(3,)
ax1.scatter(odom_scan_list[i]["x"],
            odom_scan_list[i]["y"], color=c, s=1)
ax1.scatter(odom_scan_list[i]["pose"][0],
            odom_scan_list[i]["pose"][1], color=c, s=3)
ax2.cla()
for scan in map:
    c = np.random.rand(3,)
    ax2.scatter(scan["x"], scan["y"], color=c, s=1)
    ax2.scatter(scan["pose"][0],
                scan["pose"][1], color=c, s=3)
ax2.axis(axis_limit)
ax2.set_title('Map after ICP Graph SLAM')
plt.pause(0.1)
```

Cette opération est répétée tant qu'on n'a pas atteint le dernier scan `scan_list[min_scan]`. Une fois le dernier scan traité, il ne reste plus qu'à sauvegarder le résultat sous la forme d'une image et à l'afficher :

```
plt.savefig('ICPGraphSLAM.png')
#print("Press Q in figure to finish...")
#plt.show()
```

L'exécution de ce code conduit à la figure 4 :

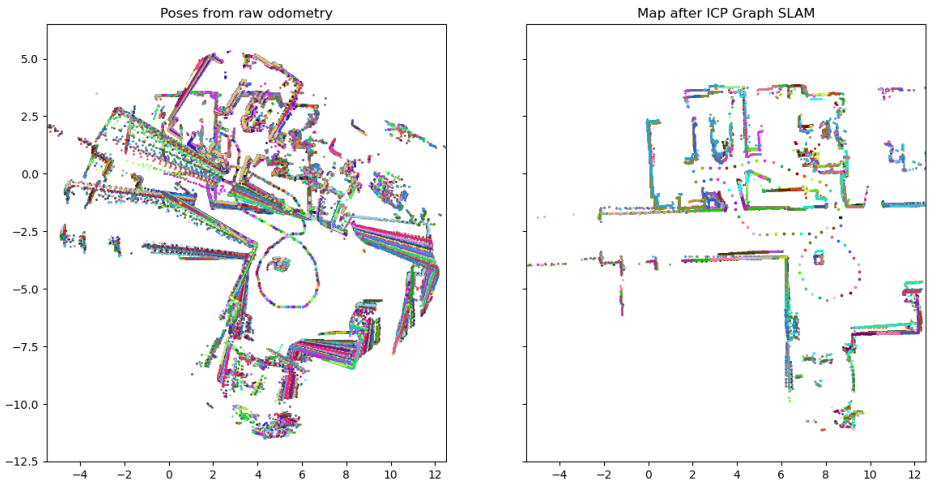


Figure 4: ICPGraphSLAM avec les paramètres par défaut

5 Q5

Quand le robot ferme une boucle, l'algorithme va détecter une proximité entre le scan courant et un précédant scan contenu dans la carte, et ainsi générer une arête entre ces deux scans, correspondant à la fermeture de la boucle. Il est vraisemblable que la topologie du graphe obtenu ne corresponde pas à un état détendu, notamment au niveau de l'arête entre le scan courant et le scan issu du robot lorsqu'il était dans la position actuelle pour la première fois. Ceci est dû à la dérive de l'odométrie.

Dans le cas de ICPGraphSLAM, le processus itératif de relaxation du graphe va venir corriger de proche en proche les positions et orientations des scans contenus dans la boucle (et les autres) jusqu'à ce que la boucle soit correctement fermée. À la fin de l'itération conduisant à la fermeture de la boucle, la carte va brutalement changer pour prendre en compte la nouvelle topologie des graphes, et les erreurs qui avaient potentiellement été induites avant le retour du robot à une position connue seront corrigées.

6 Q6

Le rôle de `dist_threshold_add` est le même que pour ICPincrementalSLAM, de même que son influence sur la qualité de la carte et les performances de l'algorithme.

`dist_threshold_match` est le paramètre qui contrôle la distance maximale en-dessous de laquelle une arête sera créée entre un scan et le scan courant dans le graphe de la carte.

Augmenter légèrement ce paramètre a pour effet de sur-contraindre le graphe en ajoutant des arêtes entre des scans qui ne devraient pas être liés. Ceci augmente sensiblement le temps de calcul de chaque itération de la boucle. À l'inverse, le réduire diminue la connexité du graphe en enlevant des arêtes entre des scans qui devraient être connectés, ce qui réduit la précision de la carte en empêchant la correction de la position et de l'orientation d'un scan en fonction de celles de plusieurs autres.

Une trop grande augmentation de ce paramètre ne semble pas conduire à une carte visiblement fausse comme dans les cas précédents (une carte obtenue pour une valeur de `dist_threshold_match` de 5 reste similaire à celle pour une valeur par défaut de 0.3).

Quand `dist_threshold_add` est plus grand que `dist_threshold_match`, les scans nouvellement rajoutés dans `map` ne sont plus automatiquement connectés au scan précédemment rajouté dans `map` comme c'est le cas sinon. Cela signifie que le graphe des scans peut être non connexe, empêchant ainsi les fermetures de boucle, ou de façon plus générale que le trajet effectif du robot peut ne pas être traduit par l'algorithme comme un chemin possible sur le graphe. En pratique, cela nuit grandement à la qualité de la carte, qui apparaît comme une superposition de scans indépendants, simplement décalés en raison de l'odométrie du robot. La figure 5

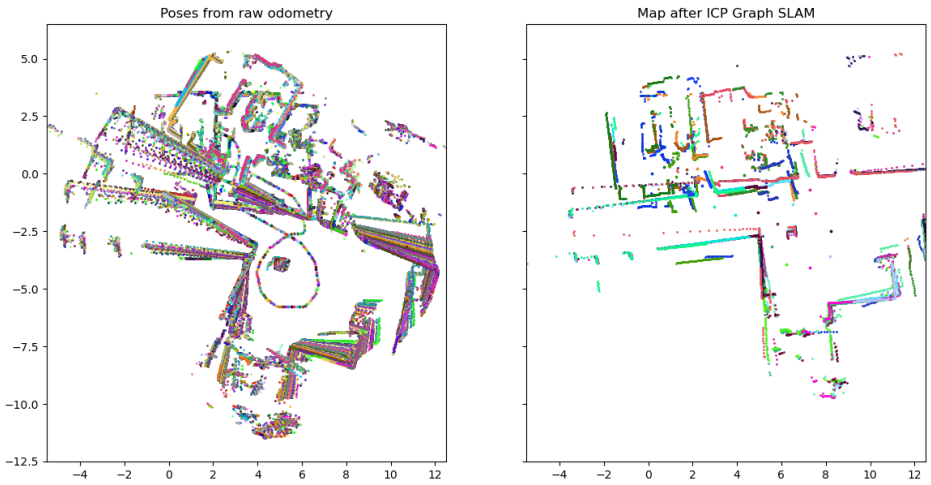


Figure 5: ICPGraphSLAM pour *dist_threshold_add* > *dist_threshold_match*

7 Q7

Pour les données issues de FR079, les algorithmes aboutissent aux cartes suivantes :

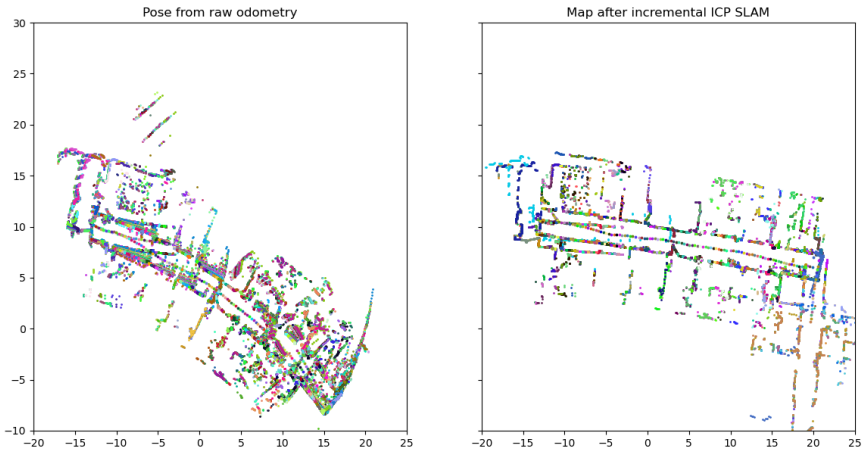


Figure 6: ICPincrementalSLAM pour FR079

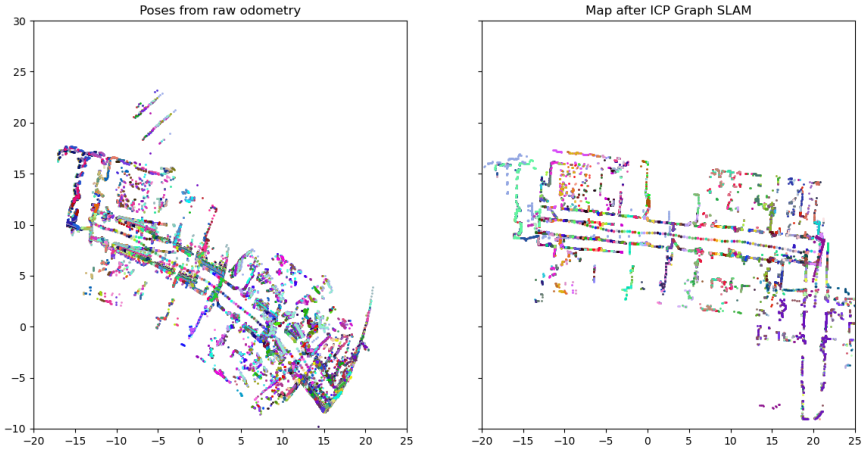


Figure 7: ICPGraphSLAM pour FR079

On remarque tout de suite que les deux algorithmes donnent de moins bons résultats pour ce jeu de données que pour le précédent, avec des points plus épars, moins denses, et des lignes globalement moins précises. ICP-GrapheSLAM semble tout de même créer une meilleure carte car le couloir principal est moins courbé que sur la carte de ICPincrementalSLAM. Dans tous les cas, l'angle droit formé par l'extrémité droite du couloir semble être une erreur d'appariement des scans car un tel angle n'est pas visible sur les scans mêmes.

La principale raison pour laquelle cet environnement est plus complexe que le précédent est la présence du long couloir, qui ne permet pas le bon repositionnement des scans avec ICP. De plus, le demi-tour à gauche du couloir crée une erreur d'orientation qui a pour effet de courber la carte alors que le demi-tour large du robot avec le jeu de données précédent laissait plus de temps à ICP pour trouver la bonne orientation.

Une bonne manière de résoudre ce problème serait d'utiliser Matchin Point Range ICP, que l'on sait converger plus rapidement en rotation.

Ici, on arrive à corriger le décalage, la fermeture de boucle et la courbure de la carte dûs au demi-tour avec ICPGraphSLAM et les paramètres suivants :

- `dist_threshold_add` = 0.1
- `dist_threshold_match` = 0.8
- `max_ICP_error` = 0.6

La figure 8 montre la carte obtenue pour les 200 premières itérations. On voit qu'avec ces paramètres, la carte ne présente plus le dédoublement du couloir observé sur les deux cartes précédentes :

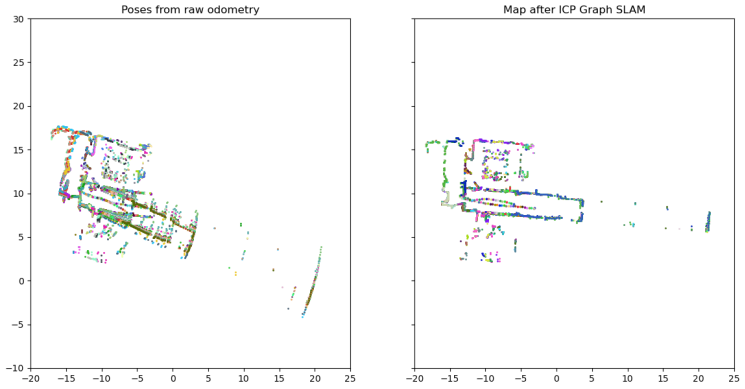


Figure 8: La carte corrigée pour FR079