Rapport Robotique Mobile Avancée - Sujet 2

Marius Ballot

Introduction

Il a fallu pour ce sujet 2 créer un algorithme de déplacement d'un robot mobile à travers 7 différents points avec l'aide d'algorithmes de calcul de trajectoire Bézier. Plus tard dans le sujet, un obstacle s'ajoute à la difficulté qu'il faut éviter tout en conservant au maximum la trajectoire initiale.

Partie 1 - Intégration d'un générateur de path Cubic Bézier

Il a fallu tout d'abord créer un générateur de courbe de Bézier.
Pour cela, j'ai choisi une méthode par l'utilisation des Cubic Bézier. C'est-à-dire 2 points avec 2 points de contrôles. Voici ma routine pour générer mes points de contrôles.

```
class BezierPath:
    # Assumption: WPList is a list of [ [x coord, y coord]  ]

    def __init__(self, CPs):
        self.CPs = CPs
        self.magn = 0.2
        self.handles = []

    def computeHandles(self):
        for i in range(len(self.CPs)):
            if i == 0:
                hx = self.CPs[i][0]
                hy = self.CPs[i][1]
                self.handles.append([hx, hy])
```

```python
def computeHandles(self):
    for i in range(len(self.CPs)):
        if i == 0:
            hx = self.CPs[i][0]
            hy = self.CPs[i][1]
            self.handles.append([hx, hy])

        elif i != len(self.CPs)-1:
            h1x = self.CPs[i][0] - \
                (self.CPs[i+1][0] - self.CPs[i][0])*self.magn
            h1y = self.CPs[i][1] - \
                (self.CPs[i+1][1] - self.CPs[i][1])*self.magn

            h2x = self.CPs[i][0] + (self.CPs[i][0] - h1x)

            h2y = self.CPs[i][1] + (self.CPs[i][1] - h1y)
            self.handles.append([h1x, h1y])
            self.handles.append([h2x, h2y])
    print("self.handles")
    print(self.handles)
```

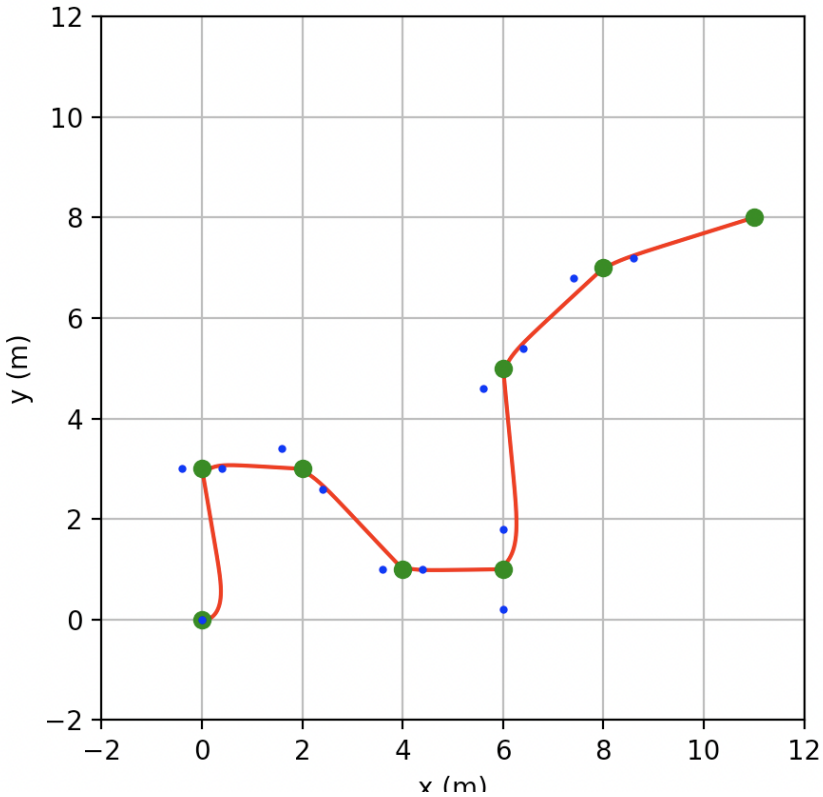
main.py × BezierPath.py Robot.py

main.py > ...

```python
# list of way points: list of [x coord, y coord]
CheckPoints = [[0, 0],
               [0, 3],
               [2, 3],
               [4, 1],
               [6, 1],
               [6, 5],
               [8, 7],
               [11, 8],
               ]
```

Avec l'array CPs étant les points par lesquels je souhaite passer, je souhaite générer un array contenant un nombre considérable de WPoints de sorte que le robot passe ensuite par ces points de passages. J'ai également modifié Robot.py pour afficher mes points de contrôles ici appelé Handles, représentés en bleu.

Grâce à la valeur magn, nous pouvons aussi modifier l'amplitude des Handles (leur distance par rapport au check point)

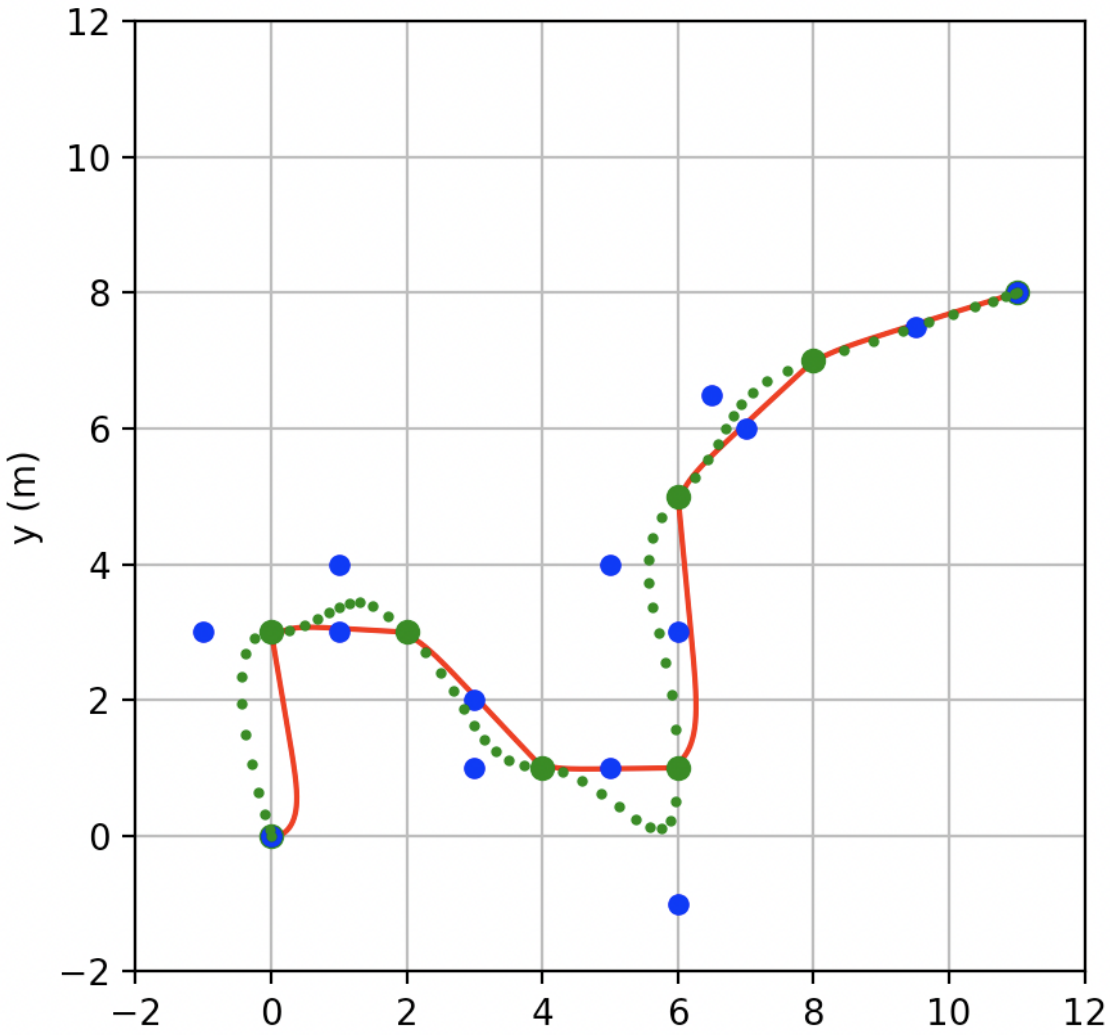


Après cela, il a fallu créer les Waypoints nécessaires pour que le robot sache où se rendre. En voici la routine algo inspirée de cet article :
https://www.f-legrand.fr/scidoc/srcdoc/graphie/geometrie/bezier/bezier-pdf.pdf

```
31      def computeWPs(self):
32          wps = []
33          for i in range(len(self.CPs)):
34              if(i < len(self.CPs)-1):
35                  bpoints = self.courbe_bezier_3(
36                      [self.CPs[i], self.handles[i*2], self.handles[i*2+1], self.CPs[i+1]], 10)
37                  print("New Point")
38                  for j in range(len(bpoints)):
39                      print(bpoints[j])
40                      wps.append(bpoints[j])
41
42          return wps
```

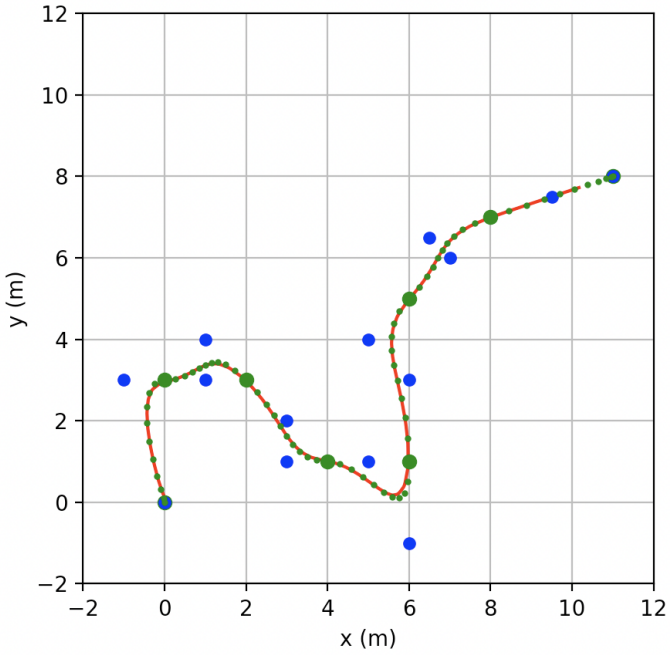
```python
    def combinaison_lineaire(self, A, B, u, v):
        return [A[0]*u+B[0]*v, A[1]*u+B[1]*v]

    def interpolation_lineaire(self, A, B, t):
        return self.combinaison_lineaire(A, B, t, 1-t)

    def point_bezier_3t(self, points_control, t):
        x = (1-t)**2
        y = t*t
        A = self.combinaison_lineaire(
            points_control[0], points_control[1], (1-t)*x, 3*t*x)
        B = self.combinaison_lineaire(
            points_control[2], points_control[3], 3*y*(1-t), y*t)
        return [A[0]+B[0], A[1]+B[1]]

    def courbe_bezier_3(self, points_control, N):
        dt = 1.0/N
        t = dt
        points_courbe = [points_control[0]]
        while t < 1.0:
            points_courbe.append(self.point_bezier_3t(points_control, t))
            t += dt
            # points_courbe.append(points_control[3])
        return points_courbe
```

À l'aide des checkpoints et handles, je créé un tableau de Waypoints qui décompose le path Bézier, ici en 10 points de passages, me donnant donc ce path bézier représenté par les points verts.



Partie 2 - Suivi de trajectoire

Malgré la présence graphique du path Bézier, nous pouvons voir que notre robot (représenté en rouge), ne suis pas du tout la trajectoire Bézier
Pour cela, j'ai dû ajuster notre algorithme de suivi de trajectoire et y passer en argument notre nouveau set de Waypoints:

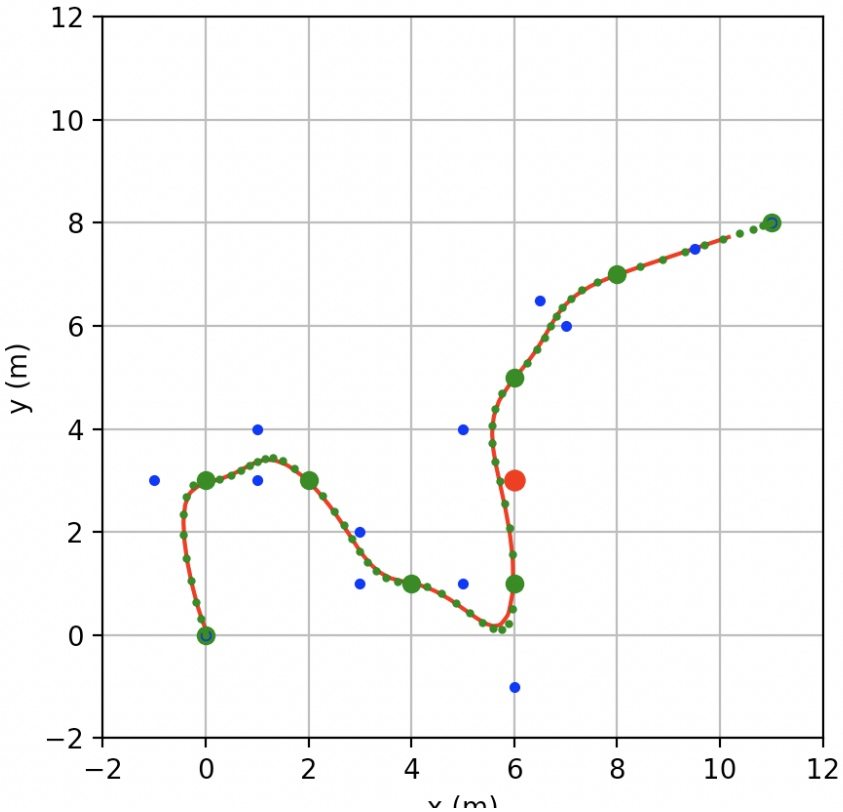


Voici toutes les modifications apportées a Robot.py pour l'affichage des data supplémentaires:

```
def plotXY(self, figNo=1, xmin=-2, xmax=12, ymin=-2, ymax=12):
    fig1 = plt.figure(figNo)
    graph = fig1.add_subplot(
        111, aspect='equal', autoscale_on=False, xlim=(xmin, xmax), ylim=(ymin, ymax))
    graph.plot(self.x, self.y, color='r')

    if(self.CPs):
        for p in self.CPs:
            graph.plot(p[0], p[1], 'ro', color='g', label='marker only')

    if(self.wpHandles):
        for p in self.wpHandles:
            graph.plot(p[0], p[1], 'ro', color='b',
                       markersize=5, label='marker only')

    if(self.WPlist):
        for p in self.WPlist:
            graph.plot(p[0], p[1], 'ro', color='g',
                       markersize=2, label='marker only')
    graph.grid(True)
    graph.set_ylabel('y (m)')
    graph.set_xlabel('x (m)')
```

Partie 3 - Correction de trajectoire à obstacle :

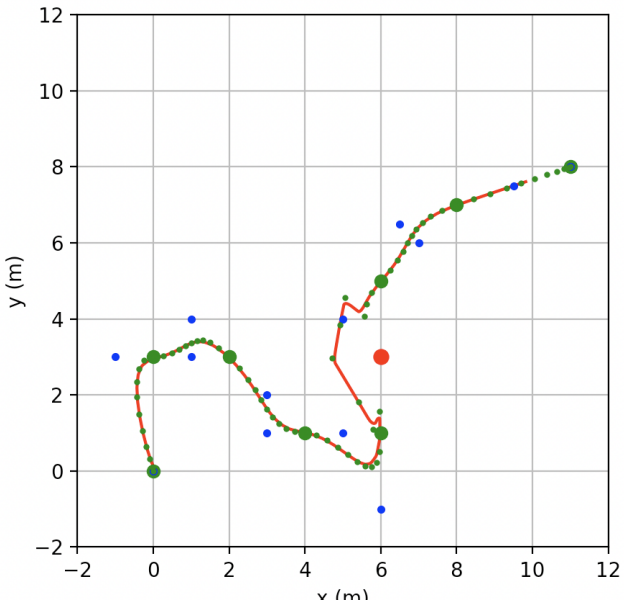
Voici déjà le placement de l'obstacle sur la carte, il nous faut un moyen de corriger l'output WPlist pour contourner cet obstacle



En regardant le tableau de points wpList, nous pouvons regarder la distance de chaque points par rapport à l'obstacle. D'ici, en prendre le vecteur directeur de ces points par rapport à l'obstacle, normaliser ce vecteur, et l'utiliser comme vecteur de displacement pour replacer ce WPpoint de sorte à ne pas être dans la zone d'obstacle. Le facteur de multiplication de displacement sera donc relié au radius de l'obstacle.

Voici la routine calculatoire:

```python
    def getDist(self, p0, p1):
        return math.sqrt(math.pow((p0[0]-p1[0]), 2)+math.pow((p0[1]-p1[1]), 2))

    def ObstacleCorrection(self, wps):
        for i in range(len(wps)):
            if(self.getDist(wps[i], self.obst) < 1):
                dirVecx = wps[i][0] - self.obst[0]
                dirVecy = wps[i][1] - self.obst[1]
                dirVecx /= self.getDist([0, 0], [dirVecx, dirVecy])
                dirVecy /= self.getDist([0, 0], [dirVecx, dirVecy])
                wps[i][0] += dirVecx
                wps[i][1] += dirVecy

        return wps

    def combinaison_lineaire(self, A, B, u, v):
```



Et voici le résultat. Nous pouvons remarquer que ce résultat pourrait être bien plus soigné, une des solutions serait d’interpoler entre correction WPlist et WPlist post Bézier.