Memòria NEATO

Alberto López Sánchez, Alejandro Adán Navarro

maig i juny del 2017

Robòtica (GEI) · FIB · UPC

Contents

1 Introducció

Amb aquest projecte es pretenia utilitzar un robot de neteja NEATO modificat amb una Raspberry Pi per tal que es desplacés dins els espais dels laboratoris del Departament d'Enginyeria de Sistemes, Automàtica i Informàtica Industrial de la UPC a l'edifici C5 del Campus Nord, reconeixent i sortejant els obstacles que es trobés en el seu cami.

2 Informació general

El projecte consta d'un programa en Python que s'executa a la Raspberry Pi, que controla el robot, fa l'odometria, rep la informació del làser i torna a calcular el millor camí per arribar al destí. D'altra banda, utilitzem el servidor HTTP bàsic de Python per servir una web amb un petit codi JavaScript que s'actualitza cada segon amb una imatge del plànol on es veu el resultat del camí a seguir, els punts per on passarà el robot, els obstacles detectats pel làser i la posició del robot.

3 Missió complerta

3.1 Entorn

L'entorn s'ha descrit en una imatge on cada píxel equival a 20 mm d'espai real, ha sigut feta a mesura repassant les mides dels plànols mesurant els espais amb una cinta mètrica. La part negra indica les parets, taules, portes i altres obstacles, la part gris es coixí que hem posat, per a que el robot no xoques contra les parets.

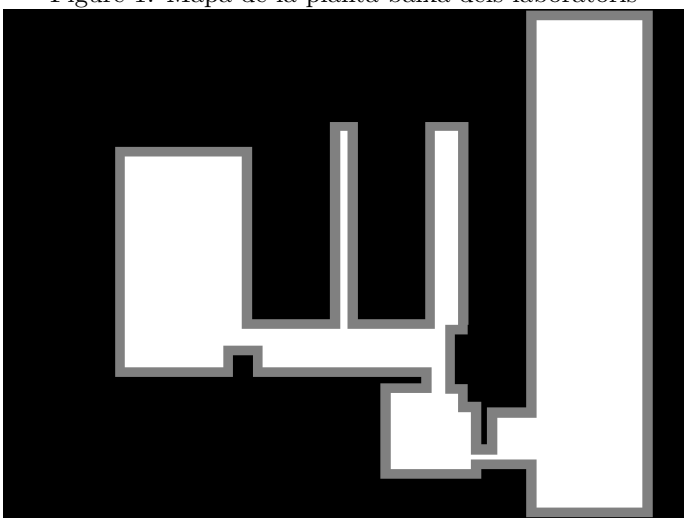
El punt (0mm,0mm) del plànol correspon a la cantonada d'un armari contra a un altre de la aula de Robòtica, que correspon al píxel (172,140) de la imatge.

Primer carreguem la imatge del mapa amb el següent codi:

```
pixels = mapper.load_image_in_pixels("
                                    plano_clase_m20jun2017_export5
                                    .png")
```

```
def load_image_in_pixels(filepath):
""" Carrega la imatge i la retorna com una matriu de pixels RGB
                                    . """

global WIDTH, HEIGHT

img = Image.open(filepath)
rgb_im = img.convert('RGB')

# Obtenim una llista de pixels
pixels = list(rgb_im.getdata())

# Obtenim la mida de la imatge
WIDTH, HEIGHT = rgb_im.size
```

Figure 1: Mapa de la planta baixa dels laboratoris



```
    # Convertirm aquesta llista de pixels en una matriu.
    pixels = [pixels[i * WIDTH:(i + 1) * WIDTH] for i in xrange(
                                        HEIGHT)]

    return pixels
```

Després, la matriu de píxels es passa a una matriu de números.

```
    original_map = mapper.pixelsToMap(pixels)
```

```
    def pixels_to_map(pixels):
    mapa = np.zeros((len(pixels),len(pixels[0])))

    for i in range(0, len(mapa) - 1 ):
        for j in range(0, len(mapa[i]) - 1):
            if pixels[i][j] == (255, 255, 255):
                mapa[i][j] = 0
            elif pixels[i][j] == (128,128,128):
                mapa[i][j] = 6
            else:
                mapa[i][j] = 1

    return mapa
```

Cada color te associat un número i cada número té un significat:

- 0: Espai lliure
- 6: Coixi
- 1: Paret / Espai no navegable.

3.2 Tècnica de navegació

Per a navegar, situem el robot en el punt inicial conegut i li diem que vagi a una posició en l'espai. Es converteixen les dues posicions a píxels i s'executa l'algorisme de *path-finding*.

Un cop fet això, l'algorisme retorna tots el punts per on ha de passar el robot i es fa un procés de simplificació eliminant tots els punts que no aporten un canvi de direcció al robot.

Després, el robot mira quina es la seva posició actual, quin es el punt on ha d'anar i calcula quant ha de girar i quant ha d'avançar per arribar-hi.

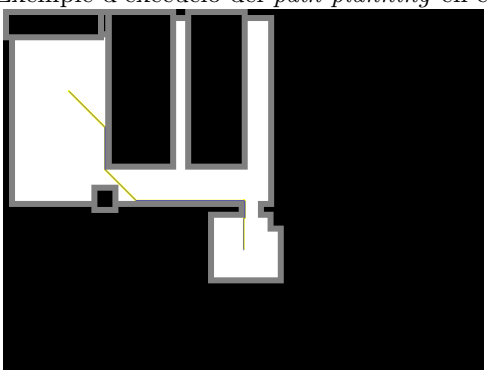
Finalment, executa el moviment. Si hi ha algun obstacle que ha estat marcat pel làser al mapa l'algorisme de *path-finding* ja ho té en compte.

3.2.1 Algorisme de *path-finding*

L'algorisme que hem fet servir és un BFS típic, però modificat per tal que, en comptes d'utilitzar una cua simple, utilitzés una cua de prioritat (*priority queue*) utilitzant com a prioritat la distància de Manhattan entre el punt en què ens trobàvem i la destinació.

Fent que l'algorisme només obri $3n$ nodes, sent n la longitud del camí, en el cas mitjà, aquest s'executa en qüestió de mil·lisegons, molt més ràpid que un BFS típic.

Figure 2: Exemple d'execució del *path-planning* en el cas mitja



```
    def FindPath(mapa, start, goal):
    """ Funcio per trobar un cami a un mapa, des de un punt inicial
                                        a un final. """

    start = convertToPixel(start)
    goal = convertToPixel(goal)

    print("Buscando:", start, "->", goal)
    queue = []
    visited = set()
```

```
    # Per veure a cada node des de quin altre s'ha arribat per
                                    poder tornar en enrere i
                                    obtenir el cami.
    predecesor = {}

    visited.add(start)
    pq.heappush(queue, (1,start))

    while queue:
        v = pq.heappop(queue)[1]

        # Per a tots els veins
        for p in [(-1,0),(1,0),(0,-1), (0,1)]:
            n = (v[0] + p[0], v[1] + p[1])

            c1 = n[0] < len(mapa) - 1
            c2 = n[1] < len(mapa[0]) - 1

            c3 = mapa[n[0]][n[1]] == 0 or mapa[n[0]][n[1]] == 4

            # Si compleixen totes les condicions (navegable), el
                                            posem a la cua.
            if n not in visited and n[0] >= 0 and n[1] >= 0 and c1
                                            and c2  and c3:
                mapa[n[0]][n[1]] = 5
                visited.add(n)
                pq.heappush(queue, (distManhattan(n, goal), n))
                predecesor[n] = v

            # Si es on voliem arribar, retornem el cami per on hem
                                            arribat.
            if n == goal:
                print "Goal"
                return True, get_path(mapa, predecesor, n)

    return False, []
```

```
    def get_path(mapa, predecesor, n):

        camino = []
        k = n
        if k not in predecesor:
            print("k no es en predecesor.")

        while k in predecesor:
            mapa[k[0]][k[1]] = 4
            camino.append(k)
            k = predecesor[k]


        return camino
```

```
    def distManhattan(a,b):
        return abs(a[0] - b[0]) + abs(a[1]- b[1])
```

3.2.2 Algorisme de simplificació

Segueix el camí punt a punt i fa que només es retornin finalment els punts que impliquen un canvi de direcció. Així, descarta tots els punts intermedis d'una mateixa recta.

```
    def simplifica_camino(camino):
    """ Funcio que donat un conjunt de punts a l'espai, elimina
                                    tots els que no aporten cap
                                    canvi de direccio. """
    if (len(camino) == 0):
        print "camino vacio"
        return []

    # Comencem amb el primer punt
    nuevo_camino = [camino[0]]
    prev_point = camino[0]

    # Guardem l'ultim punt del cami original, per si es descartat
                                    per l'algorisme, ja que
                                    podria estar en linia recta
                                    amb un altre.
    last_point = camino[len(camino) - 1]

    acum_giro = 0

    while camino:
        next_point = camino.pop(0)

        angu = mapper.cuantoGiro(prev_point, acum_giro, next_point)
        acum_giro += angu

        if abs(angu) > 0.001:
            nuevo_camino.append(next_point)

        prev_point = next_point

    if not last_point in nuevo_camino:
        nuevo_camino.append(last_point)

    return nuevo_camino
```

3.2.3 Movent el robot

Un cop tenim el camí calculat, el robot efectivament es mou seguint el camí, ordenant-li anar al primer punt del camí, esperant-se fins que hi arribi i així successivament.

```
    def ruta_demo(ser, mapa, orig_in_mm, dest_in_mm, simula):

    # Es marca al mapa els punts d'inici i final.
    mapper.markPoint(mapa, orig_in_mm, START)
    mapper.markPoint(mapa, dest_in_mm, END)
    #  184x140
```

```
    # Es preparen els punts, per a passar-los al path-planning
    offset_x = (172*mmPerPixel)
    offset_y = (140*mmPerPixel)

    orig_int = (int(orig_in_mm[0] + offset_x), int(orig_in_mm[1] +
                                    offset_y))
    dest_int = (int(dest_in_mm[0] + offset_x), int(dest_in_mm[1] +
                                    offset_y))

    # Es busca el cami.
    b_camino, camino = mapper.FindPath(mapa, orig_int, dest_int)
    if b_camino:
        print "Voy a simplificar..."
        camino = simplifica_camino(camino)
        print("camino simplificado:", camino)
    else:
        print("No hay camino", camino)


    # S'escriu el cami en un fitxer jSON, per a que el viewer el
                                    pugui mostrar.
    write_path(camino, "camino.json")
    #save_map_as_image(mapa, "salida.png")

    # Pasem el cami a la funcio que fara que es mogui el robot.
    go_for_it(ser, orig_in_mm, dest_in_mm, mapa, camino, simula)
```

```
    def go_for_it(ser, orig_in_mm, dest_in_mm, original_map, camino
                                    , simula):

    # Inicialitzem el punt previ com el punt origen on es troba el
                                    robot.
    prev_point = (orig_in_mm[0] + (172 * mmPerPixel), orig_in_mm[1]
                                     + (140 * mmPerPixel))
    # Suposem que el robot esta girat 0 graus.
    acum_giro = 0

    # Mentre quedin punts per anar al cami.
    while camino:

        #mapa = copy.deepcopy(original_map)
        next_point = mapper.convertToPoint(camino.pop())

        print("Go to:", mapper.convertToPixel(next_point))

        # Calculem quant ha de avanzar i girar el robot, per anar
                                        al seguent punt.
        dist = mapper.cuantoAvanzo(prev_point, acum_giro,
                                        next_point)
        angu = mapper.cuantoGiro(prev_point, acum_giro, next_point)

        print("Dist:", dist, " - Angu: ", angu)

        # Els afegim al nostre sistema de localitzacio independent
                                        de la odometria.
        prev_point = next_point
        acum_giro += angu
```

```
        # Si no estem en mode simulacio, efectivament mourem el
                                            robot.
        if simula == 0:
            # Fa gira el robot l'angle especificat a 75mm/s.
            giro_rad(ser, angu, 75)
            # Fa avancar el robot la distancia especificada a 150mm
                                                per segon.
            adelante(ser, dist, 150)

            # Fem la Pose Integration per actualitzar la odometria.
            readPoseIntegration(ser)
            print("Estoy en: (" + str(x + punto_A[0]) + "," + str(y
                                                + punto_A[1]) + ")",
                                                "Theta:", theta)

        # time.sleep(1)
```

A la versió amb làser el codi és una mica diferent.

La simplificació del camí no elimina tots els punts en línia recta, sinó que deixa com a mínim un punt cada 10 cm, per tal de que el robot es pugui anar aturant a passar el laser.

El robot recalcula el camí cada vegada que arriba al següent punt, per tal de tenir els obstacles actualitzats al mapa i poder esquivar-los si cal.

```
    def ruta_demo(ser, mapa_original, orig_in_mm, dest_in_mm,
                                        simula):

    # Preparem els punts per passar-los al path-planning
    offset_x = (172*mmPerPixel)
    offset_y = (140*mmPerPixel)

    orig_int = (int(orig_in_mm[0] + offset_x), int(orig_in_mm[1] +
                                        offset_y))
    dest_int = (int(dest_in_mm[0] + offset_x), int(dest_in_mm[1] +
                                        offset_y))

    # Iniciem el punt previ com el punt on es troba el robot
                                        inicialment.
    prev_point = orig_int
    acum_giro = 0

    # Fem una copia del mapa per no arrossegar els punts que han
                                        estat marcats per el laser
                                        previament.
    mapa = copy.deepcopy(mapa_original)

    # Mentres pugi:
    while True:

        # Llegeix els punts del laser.
        if simula == 0:
            print "Leiendo Laser"
            laser_points = map_laser(ser, mapa)
            # Escriu els punts llegits del laser en un fitxer jSON
                                                per poderlos
```

```
                                        visualitzar al viewer
                                        .
    write_laser(laser_points, 'laser.json')

# Busca el cami per anar al seguent punt, tenint en compte
                                    que l'origen es la
                                    posicio actual del robot
                                    donada per la odometria.
print "Buscando camino..."
b_camino, camino = mapper.FindPath(mapa, (x + offset_x,y+
                                    offset_y), dest_int)

# Si no troba cami ho indiquem.
if not b_camino:
    print("No hay camino", camino)

# Simplifiquem el cami per eliminar punts innecesaris.
camino = simplifica_camino(camino)
print "camino simplificado:", camino

# Escrivim el cami a un fitxer jSON per poder-lo
                                    visualitzar al viewer.
write_path(camino, "camino.json")

# Agafem el primer punt del cami
next_point = mapper.convertToPoint(camino.pop())

print("Go to:", mapper.convertToPixel(next_point))

# Calculem quant hem d'avancar.
dist = mapper.cuantoAvanzo(prev_point, acum_giro,
                                    next_point)
angu = mapper.cuantoGiro(prev_point, acum_giro, next_point)

print("Dist:", dist, " - Angu: ", angu)

# Els afegim al nostre sistema de localitzacio independent
                                    de la odometria.
prev_point = next_point
acum_giro += angu

# Si no estem en mode simulacio, efectivament mourem el
                                    robot.
if simula == 0:
    # Fa gira el robot l'angle especificat a 75mm/s.
    giro_rad(ser, angu, 75)
    # Fa avancar el robot la distancia especificada a 150mm
                                        /s.
    adelante(ser, dist, 150)

    # Fem la Pose Integration per actualitzar la odometria.
    readPoseIntegration(ser)
    print("Estoy en: (" + str(x) + "," + str(y) + ")", "
                                        Theta:", theta)


# time.sleep(1)
```

```
        # tornem a posar com a mapa el mapa original, netejem la
                                            lectura del laser.
        mapa = copy.deepcopy(mapa_original)
```

Les funcions utilitzades per veure quant ha d'avançar i girar el robot per arribar al següent punt són les que apareixen a continuació i es basen en la geometria.

Per anar d'un punt a un altre, el robot s'haurà d'orientar cap a la destinació i després desplaçar-s'hi en línia recta (recórrer la distància euclidiana entre ambdós punts).

```
    def cuantoAvanzo(odometria, theta, destino):
    return math.sqrt(math.pow(destino[0] - odometria[0], 2) + math.
                                        pow(destino[1] - odometria[1]
                                        , 2))
```

```
    def cuantoGiro(odometria, theta, destino):
    return math.atan2(destino[1] - odometria[1], destino[0] -
                                        odometria[0]) - theta
```

Les funcions que fan moure's al robot són aquestes:

```
    def giro_rad(ser, angulo, speed):
    """ Hace girar al robot el angulo especificado a esa velocidad.
                                         """
    comando = 'SetMotor LWheelDist ' + \
        str(-angulo * S) + ' RWheelDist ' + \
        str(angulo * S) + ' Speed ' + str(speed)
    envia(ser, comando, abs(angulo) * 500 / speed / math.pi, False)
```

```
    def adelante(ser, dist, speed):
    """ Hace avanzar al robot esa distancia a esa velocidad. """
    envia(ser, 'SetMotor LWheelDist ' + str(dist) + ' RWheelDist '
                                        +
          str(dist) + ' Speed ' + str(speed), abs(dist) / speed + 0
                                            .1, False)
```

Els temps d'espera d'aquestes funcions està calculat per tal que es correspongui (amb un cert excés) amb el temps que triga el robot en fer el desplaçament o gir demanat.

3.3 Localització del robot

Per a localitzar el robot utilitzem dues tècniques en paral·lel.

En primer lloc, fem ús de l'odometria llegint els paràmetres L i R dels motors, que ens dona un punt (x, y) a l'espai en mm, al qual l'hem de sumar el punt des d'on hem posicionat el robot a l'espai.

Hem observat que l'odometria té un error d'1 cm per metre en l'eix x i un error de 10 cm per metre en l'eix y amb els NEATOs amb què hem treballat.

```
    def init_odometry(ser):
    """Inicializa los valores de la odometria para la localizacion
                                        del robot en el espacio."""
```

```
    global Lprev, Rprev
    resu = envia(ser, 'GetMotors LeftWheel RightWheel', 0.1, True).
                                        split("\n")

    #print ("L:",L, "R:", R)
    # Inicialitzem els valors L i R de les rodes amb els primers
                                        que ens retorna el robot
                                        abans de comencar a mourens.

    Lprev = int(resu[4].split(',')[1])
    Rprev = int(resu[8].split(',')[1])
```

```
    def readPoseIntegration(ser):
    """ Llegeix els valors L i R que retorna el robot i els
                                        actualitza. """
    global Lprev, Rprev

    resu = envia(ser, 'GetMotors LeftWheel RightWheel', 0.1, True).
                                        split("\n")

    #print ("L:",L, "R:", R)
    L = int(resu[4].split(',')[1])
    R = int(resu[8].split(',')[1])

    L2 = L - Lprev
    R2 = R - Rprev
    # print "L:", L, "R:",R, "L2:",L2, "R2: ",R2

    # Calcula la nova posicio (x,y)
    poseIntegration(L2, R2)
    # print "X:", x, "Y:", y

    Lprev = L
    Rprev = R
    # print "Theta",theta
```

```
    def poseIntegration(R, L):
    """ Calcula la nova posicio de l'espai a partir del nous valors
                                        R i L. """
    global theta, x, y, S, Pose_t

    dT = (R - L) / (2 * S)
    theta = theta + dT
    dP = (R + L) / 2

    dx = dP * math.cos(theta)
    dy = dP * math.sin(theta)

    x = dx + x
    y = y - dy

    Pose_t = np.matrix([x, y, theta])
```

L'altra tècnica és acumular quant li hem ordenat avançar i quant li hem ordenat girar al robot. És a dir, si sabem que el robot es troba al punt $(10, 10)$ i la següent ordre és anar al punt $(20, 20)$, deduïm que el robot acabarà al punt

(20, 20) i que es trobarà a 45º.

Aquest codi ja ha sortit comentat abans:

```
    # Els afegin al nostre sistema de localitzacio independent de l
                                        'odometria.
    prev_point = next_point
    acum_giro += angu
```

3.4 Detecció i evasió d'obstacles

Primerament, degut que el nostre model tracta el robot com un objecte puntual, hem introduït una modificació bàsica consistent a posar uns coixins de 25 cm (una mica més que la meitat) a les parets del mapa del robot per a que no intenti fer un camí que el faci penetrar en les parets de la classe o a les taules.

Un cop que el robot sap on ha d'anar, primer fem un escaneig amb el làser per saber quins obstacles tenim al nostre voltant, que es marquen en el mapa, però només per a aquesta iteració.

Després, cridem l'algorisme de *path-finding* i ens retornarà els punts necessaris per a anar al nostre destí esquivant els obstacles detectats. Aquests punts estan separats per un màxim de 10 cm, per tal de no avançar molt sense consultar el làser, ja que ens hem d'aturar per llegir-lo.

Cada cop que s'arriba a un punt, el mapa temporal del punt anterior s'elimina i es torna a llegir el làser amb els obstacles nous i es crida l'algorisme de *path-finding*.

D'aquesta manera, si hi havia un obstacle en la lectura anterior que ja no hi és, potser podem fer un camí millor, perquè ja no cal esquivar aquest obstacle. Així, també evitem lectures errònies del làser, com ara gent que ha passat al voltant del robot.

```
    def map_laser(ser, mapa):
    global x,y
    """ Funcio encarregada de llegir el laser i posar les dades al
                                        mapa."""

    # Demanem les dades del laser al robot.
    msg_laser = envia(ser, "GetLDSScan", 0.1, True)
    angulo = 0

    laser_points = []

    # Per cada dada de laser llegida la tractem
    for line in msg_laser.split('\r\n')[2:362]:
        value = int(line.split(',')[1])
        # Si la distancia es superior a 4m descartem la dada.
        if value >= 4000:
            value = 0

        # Si la dada es una lectura valida.
        if value != 0:
```

```
            # Convertim la dada a cordenades x,y tenin en compte l'
                                                angle del robot
            x_l = value * math.cos(math.radians(angulo - math.
                                                degrees(theta)))
            y_l = value * math.sin(math.radians(angulo - math.
                                                degrees(theta)))

            # Afegim la posicio de la odometria a la posicio del
                                                laser
            laser_point = (x_l + x, y_l + y)
            print("Obstaculo:", laser_point)

            # Marquem l'obtacle al mapa
            mapper.markLaser(mapa, laser_point, 2)
            laser_points.append(laser_point)

        # Incrementem l'angle en un grau per tractar la seguent
                                            lectura.
        angulo += 1
    return laser_points
```

3.5 Exploració del passadís i detecció d'objectes

L'exploració del passadís és lliure i funciona només amb els obstacles detectats pel làser en l'espai. Donat un punt a l'espai, el robot intenta arribar-hi fent servir el *path-planning* i la informació del làser com a mapa temporal.

La detecció dels objectes no està implementada, però en cas que ho haguéssim de fer segurament algun sistema de visió per computador hauria servit per detectar els cercles en el mapa. Si no, sempre es poden matar mosques a canonades i fer servir una petita xarxa neuronal com a tècnica de *Machine Learning*.

3.6 Error en situar-se en la pose Punt Base

Sortint des del punt A, en arribar al Punt Base veiem que hem comès un error d'aproximadament 10 cm en l'eix x i de 100 cm en l'eix y.

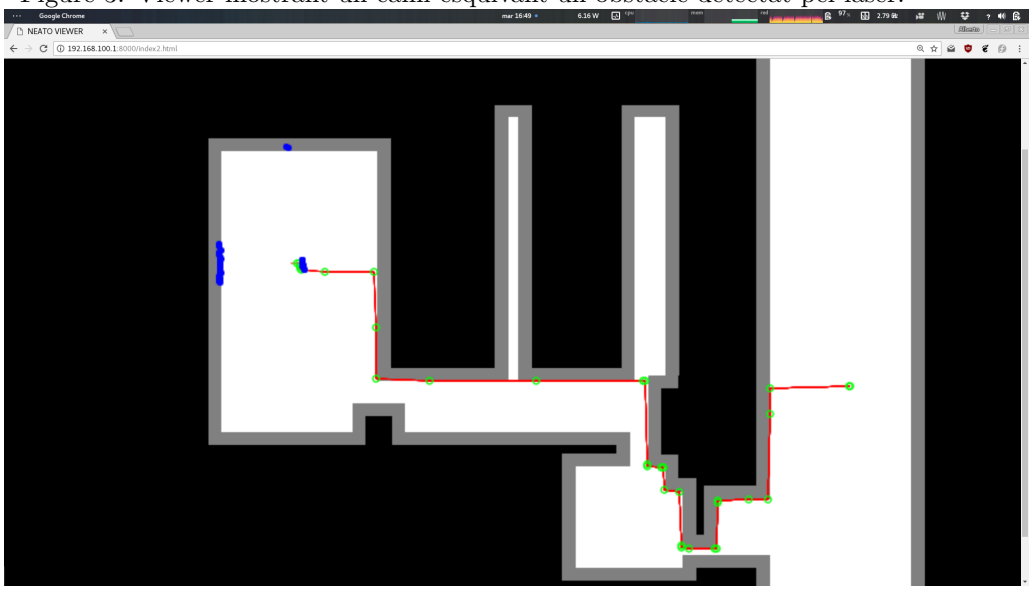
4 *Viewer*

Amb l'objectiu de poder visualitzar el camí que el robot ha planificat i els obstacles que ha detectat, cada cop que el robot planificava un camí escrivia els punts del camí a un fitxer JSON per tal de que el visor fet en HTML + JavaScript el pogués mostrar per pantalla.

Aquests fitxers són servits pel SimpleHTTPServer de Python, que s'inicialitza a la terminal amb una sola comanda. Els punts del làser també es poden visualitzar d'aquesta manera.

La visualització és en temps real: el web s'actualitza cada dos segons, però es pot fer més ràpid.

Figure 3: Viewer mostrant un camí esquivant un obstacle detectat pel laser.



5 Comentaris

5.1 Làser

Per tal de poder detectar els objectes que té a prop, el robot disposa d'un làser giratori. Un cop s'activa, el làser realitza una lectura i retorna 360 tuples de la forma (angle, distància, intensitat, codi d'error) amb els resultats.

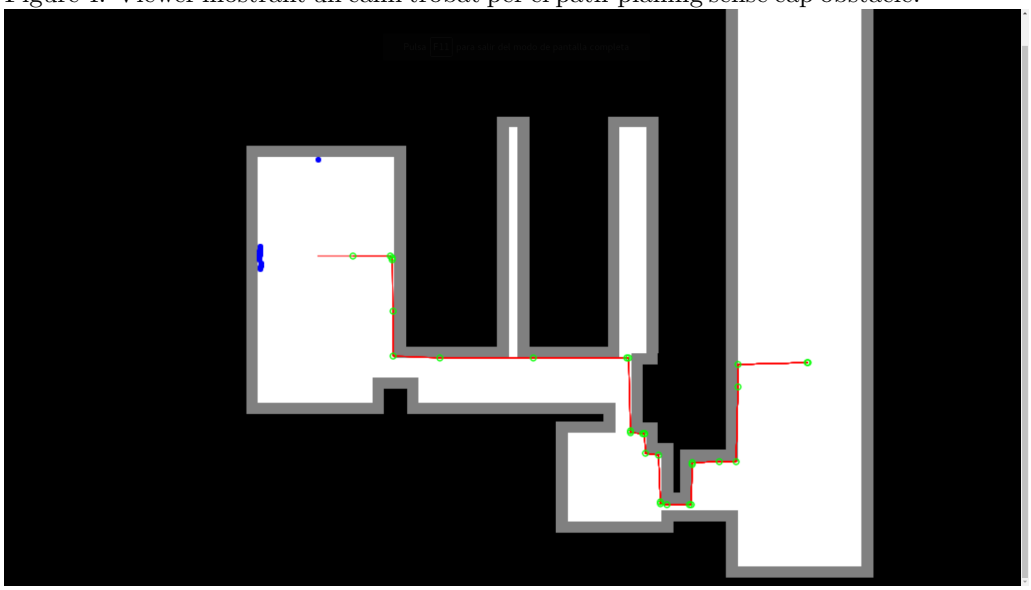
Malauradament, aquest làser és poc útil. No només té nombroses absències de dades (que poden suposar fàcilment més de dos terços dels resultats que s'haurien obtenir), sinó que a més molts cops les dades que sí que dona són absurdament altes o incorrectes ("objectes fantasma").

Tot i que és cert que les dades absurdament altes són fàcils de detectar i descartar (passen de setze metres) i que les dades que manquen solen indicar que no hi ha res a una distància de menys de mig metre del centre del robot, es fa difícil gestionar el robot amb aquesta informació exigua.

5.2 Odometria

Continuant en la mateixa tònica de l'apartat anterior, hem pogut veure com l'odometria tenia uns errors de precisió molt grans (d'uns 10 cm per metre), així que utilitzar l'odometria per guiar al robot en el nostre camí de punts era pràcticament impossible. En canvi, si sabíem que el robot havia de fer 10 metres, assumíem que quan el robot hagués completat la ordre estaria a la posició esperada.

Figure 4: Viewer mostrant un camí trobat per el path-planing sense cap obstacle.



5.2.1 Intents fallits

Teníem implementat un algoritme D* Lite per al robot en C++, que anava com un tret, i un visor en OpenGL, amb el plànol i la quadrícula que s'executava en un PC client i anava rebent tota la informació del robot, però després de dedicar-hi molt de temps vam abandonar la idea, perquè era molt complex fer quadrar els *threads* i els *sockets* de xarxa per enviar i rebre informació.

La arquitectura que utilitzàvem era un ordinador client que escoltava les connexions de la Pi amb el visor. La Pi tenia un "servidor" amb el D* que rebia l'odometria i el làser del robot feia els càlculs del *path-planning* i els tornava a la Pi, però a la vegada també enviava tota aquesta informació al visor.

Això ens va fer perdre molt de temps; ha estat una errada a la gestió del projecte.

6 Conclusions

No hem aconseguit tots els objectius que ens proposava el projecte, però hem pogut veure com implementar els components bàsics per al funcionament d'un robot d'aquest tipus, i tot i que el sistema amb el làser no acabava de funcionar bé era molt interessant. El projecte ens ha aportat coneixements sobre com gestionar un robot i la visió que, al contrari del que sembla, fer anar un robot no és trivial.