Can Distance Sim

July 3, 2021

1 Description of Simulation

Simulation: Given a 2.44 x 2.44 area, we have an ~18x18 grid of possible can placement positions if we consider all possible can orientations as a given grid square and using the standard size of a can. Each grid unit is 0.122682 x 0.122682 m in reality. (check yousif's notebook)

Run an algorithm that randomizes a can location in this 18x18 grid for 6 different cans. Placement examples: (0,1), (7,16), etc… up to (18,18).

The rocket is defined as a 4x3 silo arrangement with each silo - 3"x3". This equates to roughly a 3x2 grid unit area. We define **drop_off_pos** from the bottom left. So if drop off is at (0,0) we cannot place cans up to 3 grids in the right of and 2 grids above (0,0).

The robot is defined as as 10"x12" which equates to roughly 2x2 grid unit area. We define **start_pos** as the start location of the robot from bottom left. So if start pos of robot is (0,0), we cannot place cans up to 2 grids in the right of and 2 grids above (0,0).

For the simulation any cans generated that lie within the bounds of drop off or start pos will be randomly rearranged. Any cans that are the same as other can pos will also be randomly rearranged.

Calculate total distance traveled to collect cans with various techniques: * 1 at a time * All at once, from can to can (goes to first detected can first) * All at once, from can to can but smarter (goes to nearest can first) * add more later

Repeat simulation N times, averaging the total distance for N simulations (larger N is better, but takes more time…)

Why this simulation might not be a good indication of distances: * Doesn't account for area taken up by robot once it starts moving to can locations, ie. it assumes robot is 1 grid and moves it to another grid * Random can arrangement isnt representative of the competition can pattern, i.e. perhaps 2 cans cant be too close to each other, or some other rules for the pattern generation im missing here * The grid system is a rough approximation * Calculates straight line traversals only regardless of any other cans or rocket in path, i.e. in reality our robot would have to stop to pick them up or traverse around these

Regardless, this is better than my previous method…i think.

2 Code

2.1 Imports

[1]:
```
from random import randrange
import numpy as np
# example : print(randrange(10))
```

2.2 Constants

[2]:
```
########Constants#######
start_dx = 2
start_dy = 2

dropoff_dx = 3
dropoff_dy = 2

MAX_CANS = 6

GRID_NUM = 20

GRID_UNIT = 0.122682
#########################
```

2.3 Utilities

[3]:
```
#get random can location in 18x18 grid
def rand_can():
    x = randrange(GRID_NUM+1)
    y = randrange(GRID_NUM+1)

    return x,y

def dist(p1, p2):
    dx = p2[0] - p1[0]
    dy = p2[1] - p1[1]

    d = np.sqrt(dx**2 + dy**2)
    return d

def convert(num):
    return num * GRID_UNIT
```

2.4 Can Detection Sims

[4]:
```
#make sure can isnt in start zone
def validate_start(can, start):
    if(can[0] >= start[0]):
        return False
    if(can[0] <= start[0] + start_dx -1):
        return False
    if(can[1] >= start[1]):
        return False
    if(can[1] <= start[1] + start_dy -1):
        return False

    return True

#make sure can isnt in drop off zone
def validate_dropoff(can, dropoff):
    if(can[0] >= dropoff[0]):
        return False
    if(can[0] <= dropoff[0] + dropoff_dx -1):
        return False
    if(can[1] >= dropoff[1]):
        return False
    if(can[1] <= dropoff[1] + dropoff_dy -1):
        return False

    return True

#make sure can isnt in other can zone
def validate_can_locs(can, other_cans):
    for can_b in other_cans:
        if(can[0] == can_b[0] or can[1] == can_b[1]):
            return False

    return True
```

[5]:
```
#given a start pos and drop off pos for robot and rocket, calculate a random␣
↪can arrangement
def rand_can_arrangement(start_pos, drop_off_pos):
    cans = []
    while(len(cans) < MAX_CANS):
        passed = False
        can = rand_can()

        while(passed == False):
            can = rand_can()
```

```
            passed = validate_start(can, start_pos)
            passed = validate_dropoff(can, drop_off_pos)
            passed = validate_can_locs(can, cans)

        cans.append(can)

    return cans




####one at a time####

#total distance for one at a time
def distance_one(start_pos, drop_off_pos):
    cans = rand_can_arrangement(start_pos, drop_off_pos)

    distance = 0
    last_pos = start_pos
    for can in cans:
        distance += dist(last_pos, can) + dist(can, drop_off_pos)
        last_pos = drop_off_pos

    return distance

#N simulation for one at a time
def sim_one(N,start_pos,drop_off_pos):
    count = 0
    distance = 0
    while(count < N):
        d = convert(distance_one(start_pos, drop_off_pos))
        distance += d
        count+=1

    return distance/N




####all at once#####
def distance_all(start_pos, drop_off_pos):
    cans = rand_can_arrangement(start_pos, drop_off_pos)

    distance = 0
    last_pos = start_pos
```

```
    for can in cans:
        distance += dist(last_pos, can)
        last_pos = can

    distance += dist(last_pos, drop_off_pos)

    return distance

#N simulation for all at once
def sim_all(N,start_pos,drop_off_pos):
    count = 0
    distance = 0
    while(count < N):
        d = convert(distance_all(start_pos, drop_off_pos))
        distance += d
        count+=1

    return distance/N






####all at once BUT SMARTER#####
def get_closest(pos, cans):
    closest_can = cans[0]
    for can in cans:
        if(dist(pos, can) > dist(pos, closest_can)):
            closest_can = can

    return closest_can

def distance_all_smart(start_pos, drop_off_pos):
    cans = rand_can_arrangement(start_pos, drop_off_pos)

    distance = 0
    last_pos = start_pos

    for can in cans:
        closest_can = get_closest(last_pos, cans)
        distance += dist(last_pos, closest_can)
        last_pos = closest_can
        cans.remove(closest_can)

    distance += dist(last_pos, drop_off_pos)
```

```
    return distance

def sim_all_smart(N,start_pos,drop_off_pos):
    count = 0
    distance = 0
    while(count < N):
        d = convert(distance_all_smart(start_pos, drop_off_pos))
        distance += d
        count+=1

    return distance/N
```

2.4.1 Sim for all possible positions

[6]:
```
#simulation for one at a time for all positions
def sim_one_pos(N):
    best_distance = 1000000
    best_start = (0,0)
    best_drop = (0,0)
    count = 0

    for x_start in range(GRID_NUM):
        for y_start in range(GRID_NUM):
            for x_drop in range(GRID_NUM):
                for y_drop in range(GRID_NUM):
                    start_pos = (x_start, y_start)
                    drop_off_pos = (x_drop, y_drop)

                    d = sim_one(N,start_pos,drop_off_pos)

                    if(d < best_distance):
                        best_distance = d
                        best_start = start_pos
                        best_drop = drop_off_pos

                    count+=1
                    print('simulation %.3f             percent complete'␣
 ↪%(count/104976 *100),end='\r')

    return best_distance, best_start, best_drop
```

[7]:
```
#simulation for all at once for all positions
def sim_all_pos(N):
    best_distance = 1000000
    best_start = (0,0)
    best_drop = (0,0)
```

```
    count = 0

    for x_start in range(GRID_NUM):
        for y_start in range(GRID_NUM):
            for x_drop in range(GRID_NUM):
                for y_drop in range(GRID_NUM):
                    start_pos = (x_start, y_start)
                    drop_off_pos = (x_drop, y_drop)

                    d = sim_all(N,start_pos,drop_off_pos)

                    if(d < best_distance):
                        best_distance = d
                        best_start = start_pos
                        best_drop = drop_off_pos

                    count+=1
                    print('simulation %.3f           percent complete' %(count/
 ↪104976 *100),end='\r')

    return best_distance, best_start, best_drop
```

[8]:
```
#simulation for all at once but smart for all positions
def sim_all_smart_pos(N):
    best_distance = 1000000
    best_start = (0,0)
    best_drop = (0,0)
    count = 0

    for x_start in range(GRID_NUM):
        for y_start in range(GRID_NUM):
            for x_drop in range(GRID_NUM):
                for y_drop in range(GRID_NUM):
                    start_pos = (x_start, y_start)
                    drop_off_pos = (x_drop, y_drop)

                    d = sim_all_smart(N,start_pos,drop_off_pos)

                    if(d < best_distance):
                        best_distance = d
                        best_start = start_pos
                        best_drop = drop_off_pos

                    count+=1
                    print('simulation %.3f          percent complete' %(count/
 ↪104976 *100),end='\r')
```

```
    return best_distance, best_start, best_drop
```

2.5 Can Sweep Sim

[83]:
```
########Constants#######

SWEEP_WIDTH = 3
SWEEP_HEIGHT = 20

ROBOT_WIDTH = 2
ROBOT_HEIGHT = 2

AREA_W = 20
AREA_H = 20

#########################
```

[110]:
```
def get_bounds(p1, p2):
    x1=p1[0]
    x2=p2[0]
    if(x2 < x1):
        x1=p2[0]
        x2=p1[0]

    y1=p1[1]
    y2=p2[1]
    if(y2 < y1):
        y1=p2[1]
        y2=p1[1]

    return x1, x2, y1, y2

#given a list of can arrangments, and sweep path which is essentially points␣
 ↪along a path
def sweep_cans(sweep_path,sweep_width, cans):
    swept = []

    for i in range(len(sweep_path)):
        if(i >= len(sweep_path) -1):
            break

        p1 = sweep_path[i]
        p2 = sweep_path[i+1]
        x1,x2,y1,y2 = get_bounds(p1, p2)

        for can in cans:
            if(x1==x2):
```

```
                if((can[0] >= x1 and can[0] <= x1 + sweep_width)
                  and (can[1] >= y1 and can[1] <= y2)):
                    if(can not in swept ):
                        swept.append(can)
            if(y1==y2):
                if((can[0] >= x1 and can[0] <= x2)
                  and (can[1] >= y1 and can[1] <= y2 + sweep_width)):
                    if(can not in swept ):
                        swept.append(can)


    return swept

def path_distance(path):
    last = path[0]
    d= 0

    for point in path:
        d += dist(last, point)
        last = point

    return d

def gen_path(gap, sweep_width):
    x = 0
    y = 0
    dir_count = 1

    xlim = AREA_W - ROBOT_WIDTH + 1
    ylim = AREA_H - ROBOT_HEIGHT + 1

    path = [(0,0)]

    while(x <= xlim ):
        if( dir_count == 1):
            y+=ylim
        if(dir_count == 2 or dir_count == 4):
            if(x + gap + sweep_width <= xlim  ):
                x+=gap + sweep_width
            else:
                x=AREA_W
        if(dir_count == 3):
            y-=ylim

        dir_count+=1

        if(dir_count>4):
```

```
            dir_count=1

        path.append((x,y))

    return path


def sweep(sweep_gap, sweep_width):
    path = gen_path(sweep_gap,sweep_width)

    cans = rand_can_arrangement((0,0), (16,0))
    swept = sweep_cans(path,sweep_width, cans)

    return swept

def sweep_sim(sweep_gap,sweep_width, N):
    count = 0
    distance = path_distance(gen_path(sweep_gap, sweep_width))
    cans_collected = 0

    while(count < N):
        count+=1
        cans_collected += len(sweep(sweep_gap,sweep_width))

    return cans_collected/N , convert(distance)
```

2.5.1 Sweep Paths

[73]:
```
sweep_sim(2, 10000)
```

[73]: (4.0709, 14.10843)

[116]:
```
p = gen_path(2, 2)
c = rand_can_arrangement((0,0), (16,0))
s = sweep_cans(p,2,c)
print(p)
print(c)
print(s)
```

```
[(0, 0), (0, 19), (4, 19), (4, 0), (8, 0), (8, 19), (12, 19), (12, 0), (16, 0),
(16, 19), (20, 19)]
[(6, 1), (8, 7), (7, 15), (1, 12), (20, 17), (15, 13)]
[(1, 12), (6, 1), (8, 7)]
```

[120]:
```
for i in range(20):
    ans = sweep_sim(i,2, 10000)
    print("Sweep distance: %f, Cans collected: %f \n" %(ans[1], ans[0]))
```

```
Sweep distance: 25.763220, Cans collected: 5.914600

Sweep distance: 18.770346, Cans collected: 5.917800

Sweep distance: 14.108430, Cans collected: 4.428400

Sweep distance: 11.777472, Cans collected: 3.679900

Sweep distance: 11.777472, Cans collected: 3.651600

Sweep distance: 9.446514, Cans collected: 2.918100

Sweep distance: 9.446514, Cans collected: 2.948700

Sweep distance: 9.446514, Cans collected: 2.957800

Sweep distance: 7.115556, Cans collected: 2.211400

Sweep distance: 7.115556, Cans collected: 2.186000

Sweep distance: 7.115556, Cans collected: 2.174900

Sweep distance: 7.115556, Cans collected: 2.176700

Sweep distance: 7.115556, Cans collected: 2.151200

Sweep distance: 7.115556, Cans collected: 2.126200

Sweep distance: 7.115556, Cans collected: 2.106200

Sweep distance: 7.115556, Cans collected: 2.103300

Sweep distance: 7.115556, Cans collected: 2.095600

Sweep distance: 7.115556, Cans collected: 1.856400

Sweep distance: 4.784598, Cans collected: 1.348100

Sweep distance: 4.784598, Cans collected: 1.351500
```

3 Running the Simulation

[13]:
```
drop_off_pos = (0,0)
start_pos = (2,3)
N = 10000
print('one at a time: %f metre' %sim_one(N,start_pos,drop_off_pos))
print('all at once: %f metre' %sim_all(N,start_pos,drop_off_pos))
print('all at once but smart: %f metre'␣
 ↪%sim_all_smart(N,start_pos,drop_off_pos))
```

```
one at a time: 22.312594 metre
all at once: 10.344510 metre
all at once but smart: 8.906085 metre
```

[14]:
```
drop_off_pos = (0,0)
start_pos = (17,17)
N = 10000
print('one at a time: %f metre' %sim_one(N,start_pos,drop_off_pos))
print('all at once: %f metre' %sim_all(N,start_pos,drop_off_pos))
print('all at once but smart: %f metre'␣
 ↪%sim_all_smart(N,start_pos,drop_off_pos))
```

```
one at a time: 22.265833 metre
all at once: 10.289962 metre
all at once but smart: 8.388285 metre
```

[15]:
```
drop_off_pos = (8,0)
start_pos = (9,9)
N = 10000
print('one at a time: %f metre' %sim_one(N,start_pos,drop_off_pos))
print('all at once: %f metre' %sim_all(N,start_pos,drop_off_pos))
print('all at once but smart: %f metre'␣
 ↪%sim_all_smart(N,start_pos,drop_off_pos))
```

```
one at a time: 17.426372 metre
all at once: 9.416758 metre
all at once but smart: 7.417634 metre
```

[16]:
```
drop_off_pos = (8,7)
start_pos = (9,9)
N = 10000
print('one at a time: %f metre' %sim_one(N,start_pos,drop_off_pos))
print('all at once: %f metre' %sim_all(N,start_pos,drop_off_pos))
print('all at once but smart: %f metre'␣
 ↪%sim_all_smart(N,start_pos,drop_off_pos))
```

```
one at a time: 12.561398 metre
all at once: 9.001882 metre
all at once but smart: 6.986585 metre
```

```
[ ]:
```