Simultaneous Localization and Mapping (SLAM)

A Succinct Robotics Application of Sequential Monte Carlo Methods

Eduardo Joaquin Castillo
Electrical and Computer Engineering
Cornell Tech at Cornell University
New York, NY 10044
ec833@cornell.edu

1 Introduction

This report documents the development of a two dimensional robot localization and mapping algorithm based on a purpose-built Sequential Monte Carlo algorithm.

The model processes discrete wheel rotation data collected from self-contained wheel encoders, as well as light radar (LIDAR) detection data over a 270° horizontal visual field. In addition, a robot inertial measurement unit (IMU) provides linear and angular acceleration data.

Using these inputs, the model iteratively produces a correlation to observed map landmarks (i.e.: prior LIDAR returns from objects in the environment) to predict the most likely location of the robot. According to this best-estimate location, landmark location confidences are incrementally updated using Bayesian inference.

2 Problem Formulation

2.1 Input Data Review

The input data includes relevant data for three training maps (Maps 20, 21 and 23) as well as two testing maps (Maps 22 and 24). Timed wheel odometry data (wheel rotation), timed LIDAR object detection distances and inertial measurement unit (IMU) accelerations are included for each map.

The Laser Scanner was a Hokuyo UTM-30LX mounted in accordance with Appendix B. No calibration or tolerance information was used for sensors. All measurements were provided in SI units. The robot was assumed to start at absolute position (0,0) with a heading angle of 0 degree with respect to the horizontal axis.

3 Technical Approach

3.1 Stage One - Dead Reckoning

A dead reckoning algorithm was initially implemented. Namely, wheel odometry data was processed to compute displacement and heading angle updates over each time step.

These displacements and heading angles were transferred to a global coordinate frame, using applicable trigonometric relationships. Based on these measurements, LIDAR measurements were used to produce a rough occupancy map of the robot's environment. Results for this stage can be found in Appendix C.

3.2 Stage Two - SLAM Algorithm

The dead reckoning algorithm was enhanced to incorporate a Sequential Monte Carlo method, also known as a particle filter. Particle filters comprise a broad family of sequential Monte Carlo algorithms for inference in partially observable Markov chains [1].

In this particular application, the particle filter was leveraged to partially span the distribution of possible sensor noise levels in the linear (2D) and angular dimensions. Figure 1 includes a representative model of the particles paths .

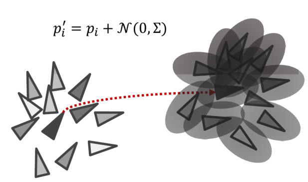


Figure 1: Particle Noise Model [2]

3.3 SLAM Model Implementation

The algorithm was trained to infer correlations between discrete odometry, LIDAR information and identified landmarks. The implemented algorithm can be described as follows:

1. The initial robot position and pose were specified (i.e.: coordinate (0,0) at heading angle 0°).
2. Particles were initialized around the initial position and heading angle and uniform weights were assigned.
3. Discrete displacements and rotations were processed based on the noise model specified in Figure 1.
   (a) A proportional amount of uniformly-distributed noise was added to each displacement and angle change.
   (b) A fixed amount of uniformly-distributed noise was added to each displacement and angle change.
4. A correlation was calculated based on the number and weights of known landmarks (pixels) hit by simulated rays propagated from each of the particles.
5. Particle scores were transformed to weights based on this correlation:

   $$weight_\alpha = \frac{e^{(s_\alpha - \beta)}}{\sum_\alpha e^{s_\alpha} e^{-\beta}}$$

   where s_α represents the score of the α particle and β represents the highest score for the time step.[1]
6. Global weights were updated based on the product of the weight at time = t and the weight at time = $t-1$.
7. Best-estimate robot center locations and heading angles were determined based on the particle with the highest weight.
8. Map landmarks (pixels) receiving LIDAR hits increasing in certainty. Conversely, landmarks along the line of sight of the impacted cells received a reduction in certainty.
9. If the effective number of particles ($n_{effective}$, defined below) becomes less than 50%, particles are resampled based on their weight distributions.

   $$n_effective = \frac{(\sum_\alpha w_\alpha)^2}{\sum_\alpha {w_\alpha}^2}$$

10. Iterate through all time steps.

4 Results and Discussion

4.1 Dead Reckoning Maps

Relevant visualizations for all training and testing maps are included in this section, with the robot originating at coordinate (0,0) and initially facing in the positive direction along the horizontal axis.

As expected, the maps are fairly rough and significant distortion of the map and associated robot trajectory are produced with this basic algorithm.

Appendix C contains enlarged annotated version of these dead reckoning maps.

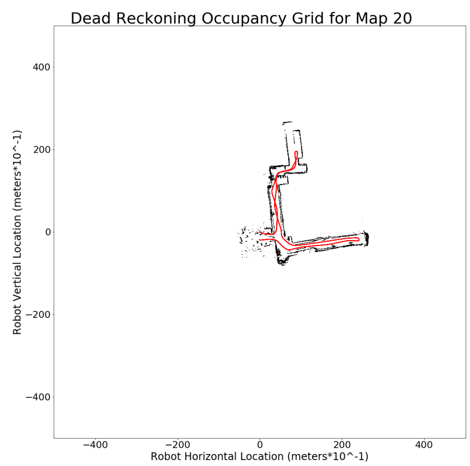


Figure 2: Dead Reckoning Trajectory - Map 20

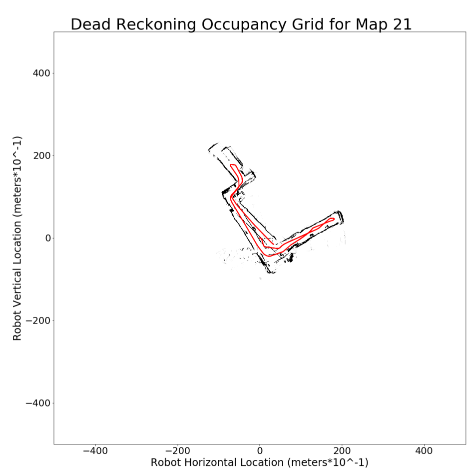


Figure 3: Dead Reckoning Trajectory - Map 21

[1] This is a β-shifted version of the generalized softmax function. Appendix A provides further background on the basis for the β-shift.

4.2 SLAM Results

The figures below display the maps obtained using the SLAM algorithm for maps 20 through 24. Each figure is hyperlinked to an animated image showing the robot's best-estimate position and pose over time. These animations also display the landmark detection process during mapping.

As previously, the robot is predicted to start at coordinate 0,0, at a pose angle of 0 degrees. Appendix D includes full-page versions of these figures as well as illustrations of the robot's best-estimate path .

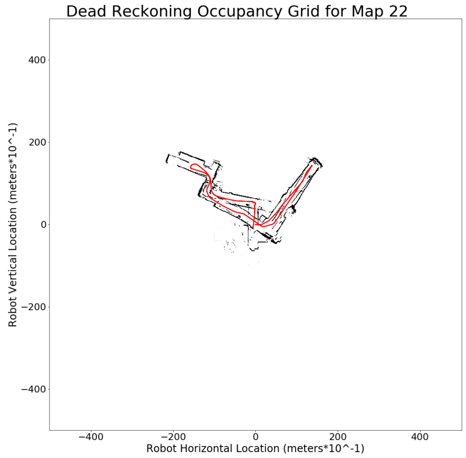


Figure 4: Dead Reckoning Trajectory - Map 22

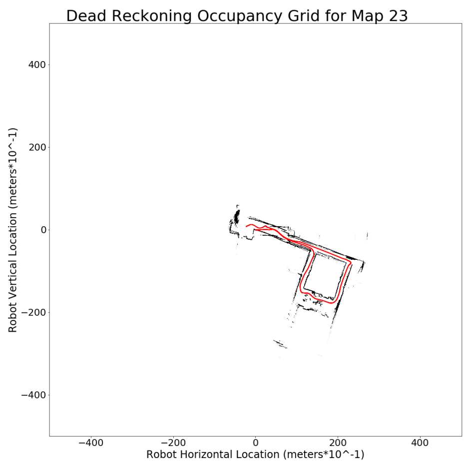


Figure 5: Dead Reckoning Trajectory - Map 23

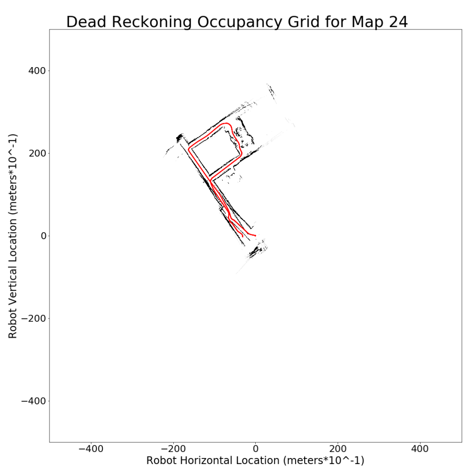


Figure 6: Dead Reckoning Trajectory - Map 24

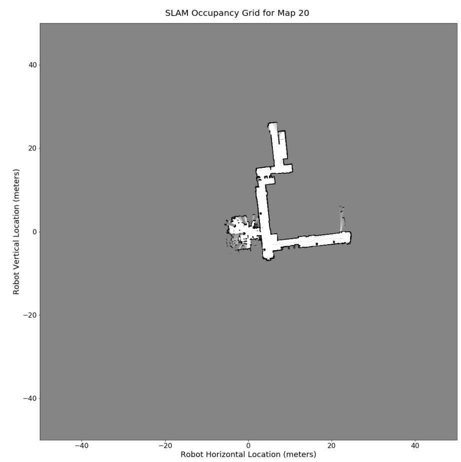


Figure 7: SLAM Prediction for Map 20

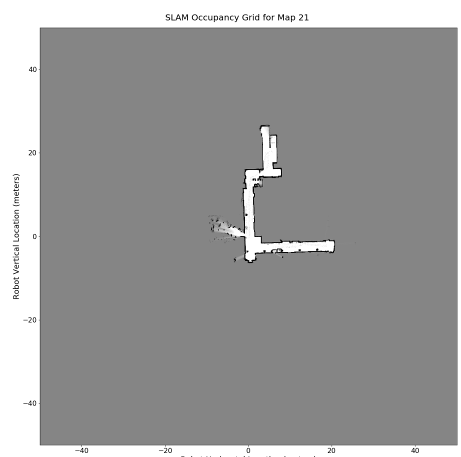


Figure 8: SLAM Prediction for Map 21

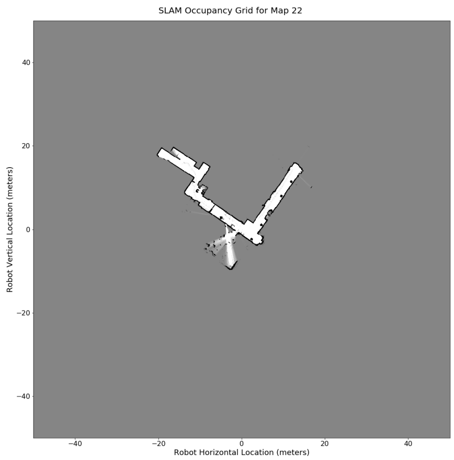


Figure 9: SLAM Prediction for Map 22

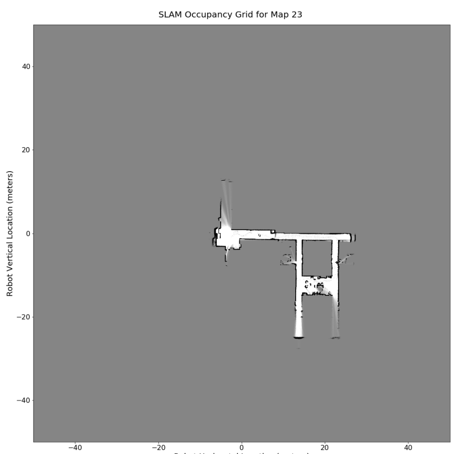


Figure 10: SLAM Prediction for Map 23

Figure 11: SLAM Prediction for Map 24

4.3 Odometry Results

Appendix C shows the results of the odometry algorithm for the training and test data.

5 Hyperparameters and Tuning

5.0.1 Particle Count

Given the sizable computational advantages of the LIDAR Downsampling implementation (described below), the algorithm was implemented using a particle count of 300 particles.

5.0.2 Map Resolution

A baseline map resolution of 10 pixels per meter was used to maintain computational cost within reasonable bounds.

5.0.3 Linear and Angular Noise Parameters

Uniformly-distributed fixed and variable noise were added to each particle at each time step. The fixed noise had a maximum range of 0.36 meters, 0.01625 meters and 0.072 radians (approximately 4 degrees) for the X, Y and θ (angle) parameters.

Likewise, the variable noise had a maximum range of 0.04, 0.01 and 0.15. These factors represent an amplification proportional to the amount of change for the X, Y and θ parameters.

5.0.4 The Downsampling Parameter

A downsampling parameter was introduced to the SLAM algorithm to reduce the angular resolution of the LIDAR measurements from 1081 to 540 (downsampling parameter of 5). This parameter determined a multiple of 108 that would be used to draw a random sample of the available 1081 readings per time step.

The implementation of this hyperparameter was beneficial from a computation and map fidelity standpoint. The processing time for map 24 was reduced for 108 minutes to approximately 36 minutes using a general purpose PC.

From a map fidelity standpoint, noise from LIDAR readings was reduced through this approach, given the sparse nature of LIDAR noise. Additionally, higher downsampling values were verified to also produce high fidelity results with significantly lower compute times.

5.1 Robot Width

An effective robot with of 0.733 meters was used for all odometry and SLAM algorithms.

5.2 Minimum Effective Particle Count

The minimum allowed effective number of particles was set to 50% of the specified particle count.

5.3 Log-odds Parameters

An increase of 10 units was assigned for every lidar hit at a given cell, while a reduction of 0.5 units was assigned for every lidar miss in line of sight with any impacted cell.

All cells were limited to a maximum score of 300 units and a minimum score of -300 units.

5.4 Minimum Lidar Range

A minimum lidar range of 0.33 meters was assigned to the robot's periphery for angles equal or greater than 90 degrees (with respect to the robot's longitudinal axis). As such, any detections in close vicinity to the robot's posterior area were ignored.

References

[1] Gordon N. Doucet A., de Freitas N. An introduction to sequential monte carlo methods. in: Doucet a., de freitas n., gordon n. (eds) sequential monte carlo methods in practice. statistics for engineering and information science. springer, new york, ny. 2001.

[2] Cornell Tech Electrical and Computer Engineering Department. ECE - 5242 Project 3 - Simultaneous Localization and Mapping, 03 2020.

Appendices

Appendix A General Shifted Softmax Derivation

Show that the softmax function is invariant to constant offsets to its input, i.e.,

$$\mathrm{softmax}\left(\mathbf{a} + c\mathbf{1}\right) = \mathrm{softmax}\left(\mathbf{a}\right),$$

where $c \in \mathbb{R}$ is some constant and $\mathbf{1}$ denotes a column vector of 1's.

Solution:

Note:

$$\sum_J e^{a_j} = [e^a]^T * \vec{1}$$

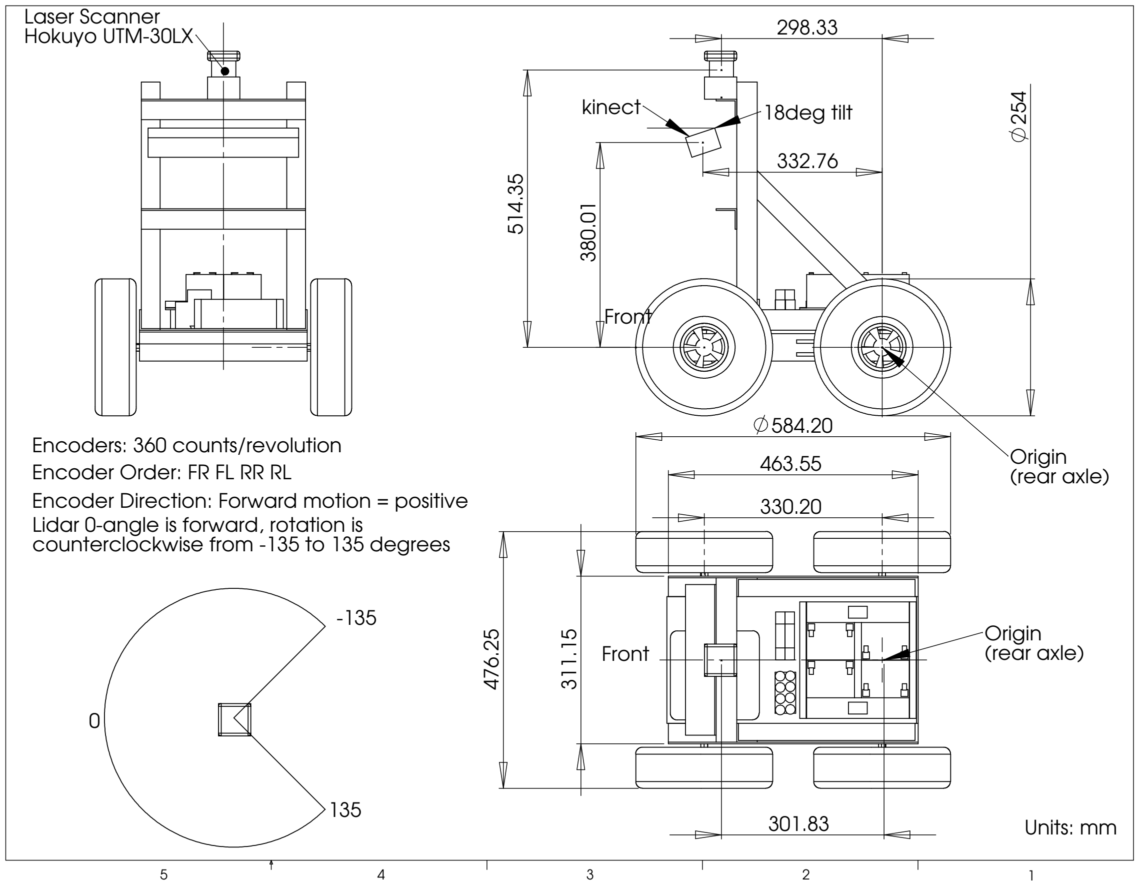
Hence,

$$softmax(a) = \frac{e^a}{\sum_J e^{a_j}} = \frac{e^a}{[e^a]^T * \vec{1}}$$

$$softmax(a + c\vec{1}) = \frac{e^{a + c\vec{1}}}{[e^{a + c\vec{1}}]^T * \vec{1}} = \frac{e^a e^{c\vec{1}}}{[e^a * e^{c\vec{1}}]^T * \vec{1}}$$

$$softmax(a + c\vec{1}) = \frac{e^a e^c}{\vec{1}^T e^a e^{c\vec{1}}} = \frac{e^a e^c}{\sum_J e^{a_j} e^c} = \frac{e^a}{\sum_J e^{a_j}} = softmax(a)$$

Appendix B Robot Configuration Drawing



Appendix C   Odometry Results for Training and Test Data

The maps shown below represent the dead reckoning occupancy maps for the training and test data without the particle filter processing.

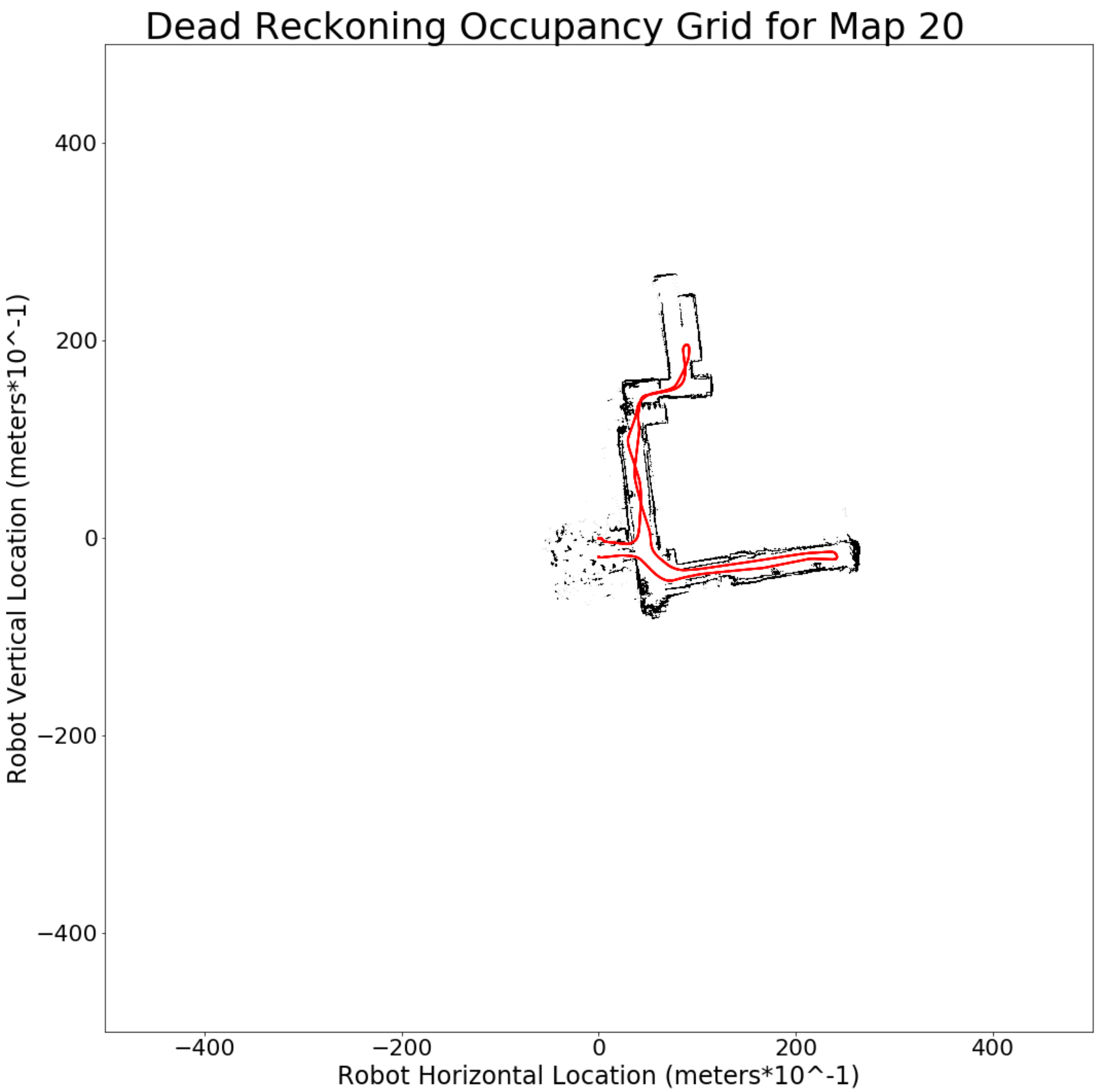


Figure 12: Dead Reckoning Trajectory - Map 20

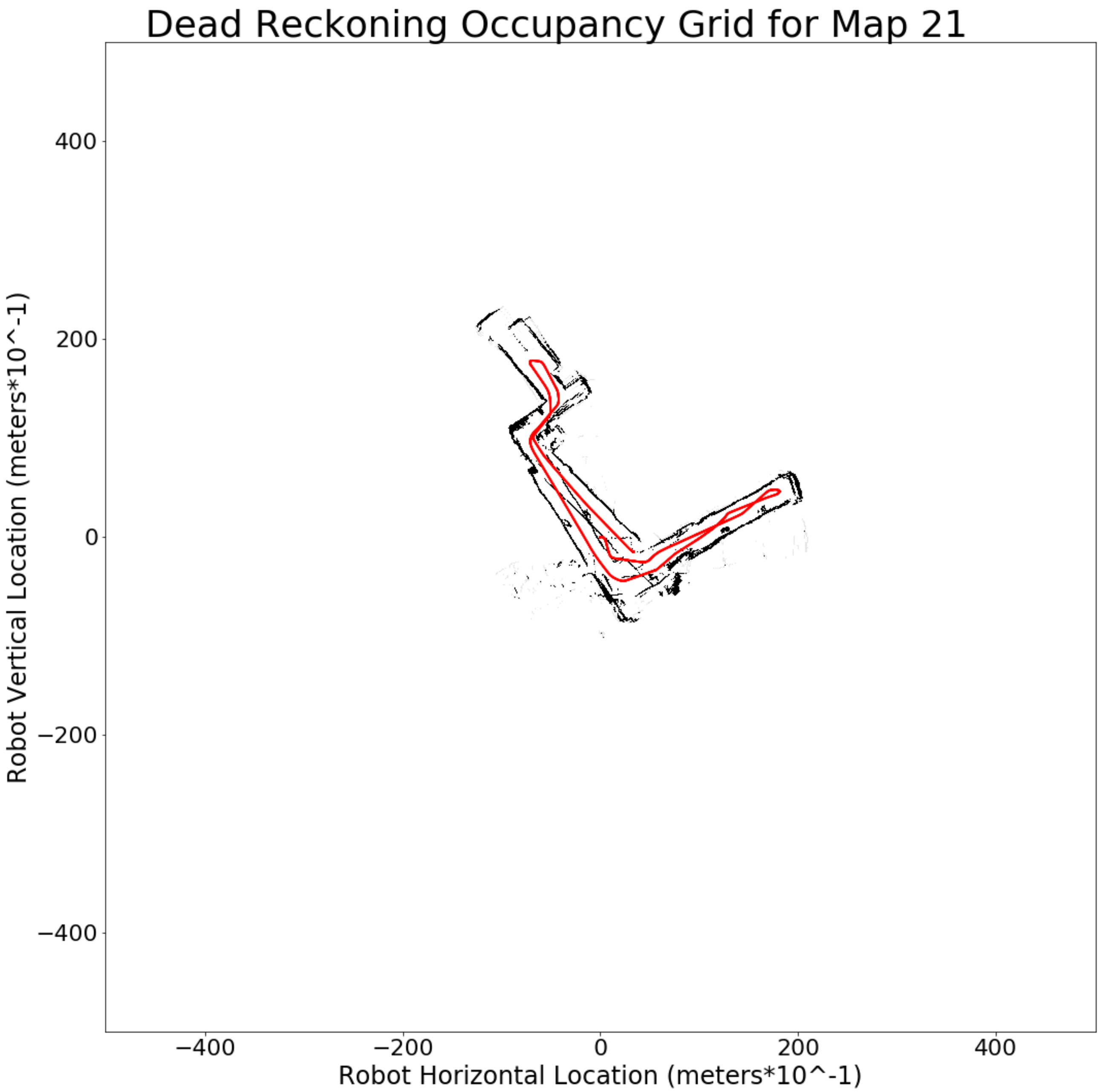


Figure 13: Dead Reckoning Trajectory - Map 21

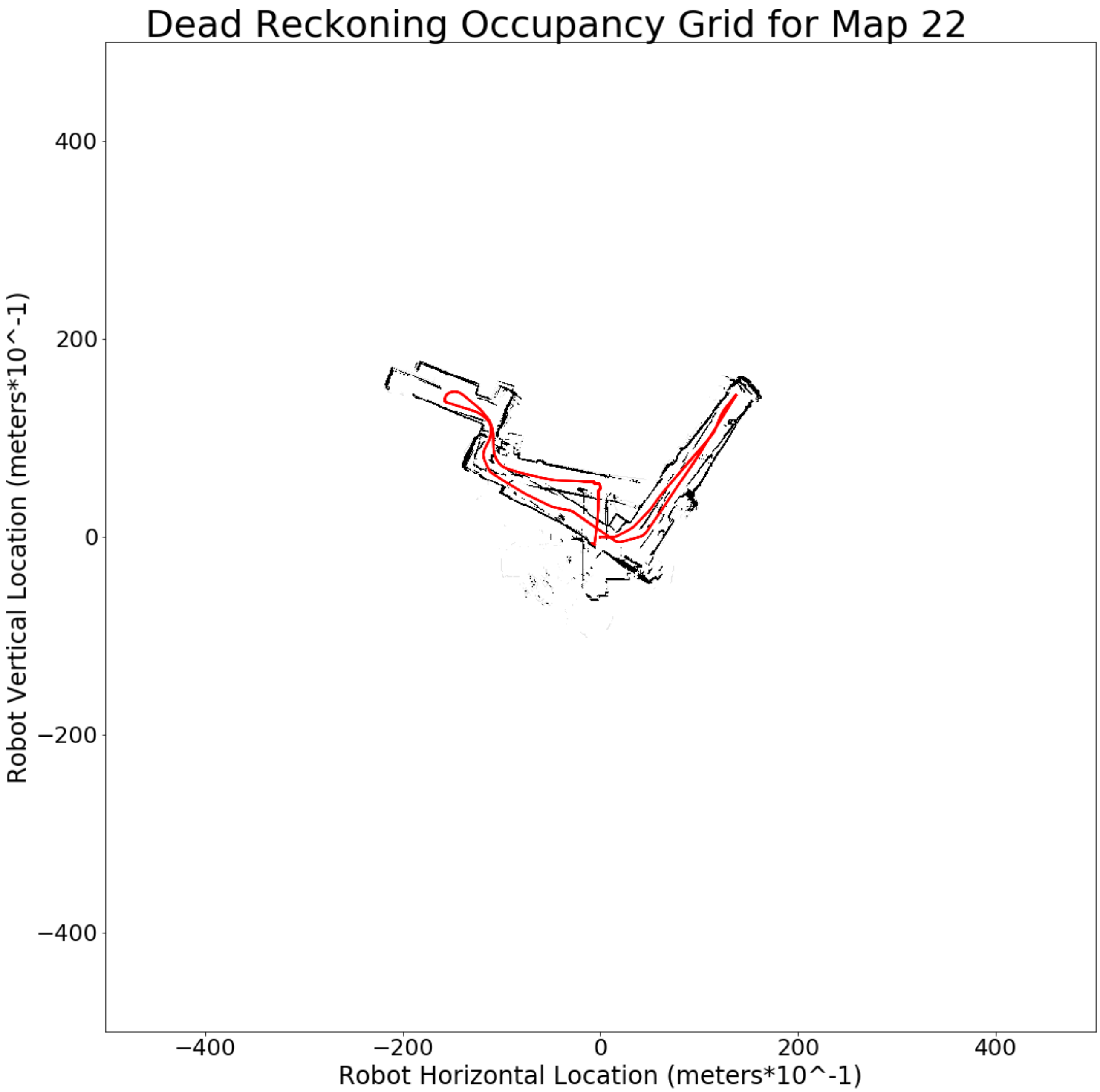


Figure 14: Dead Reckoning Trajectory - Map 22

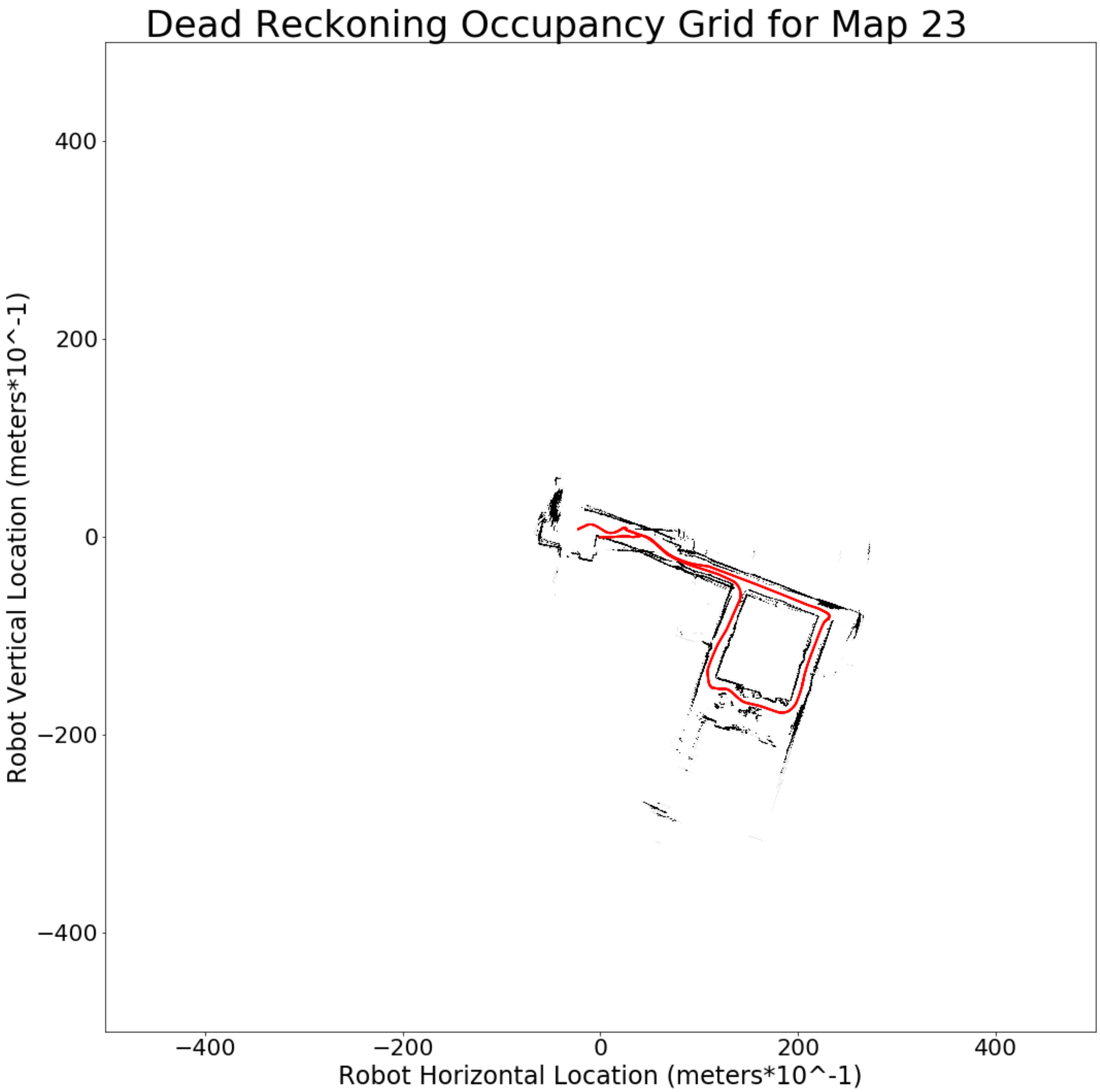


Figure 15: Dead Reckoning Trajectory - Map 23

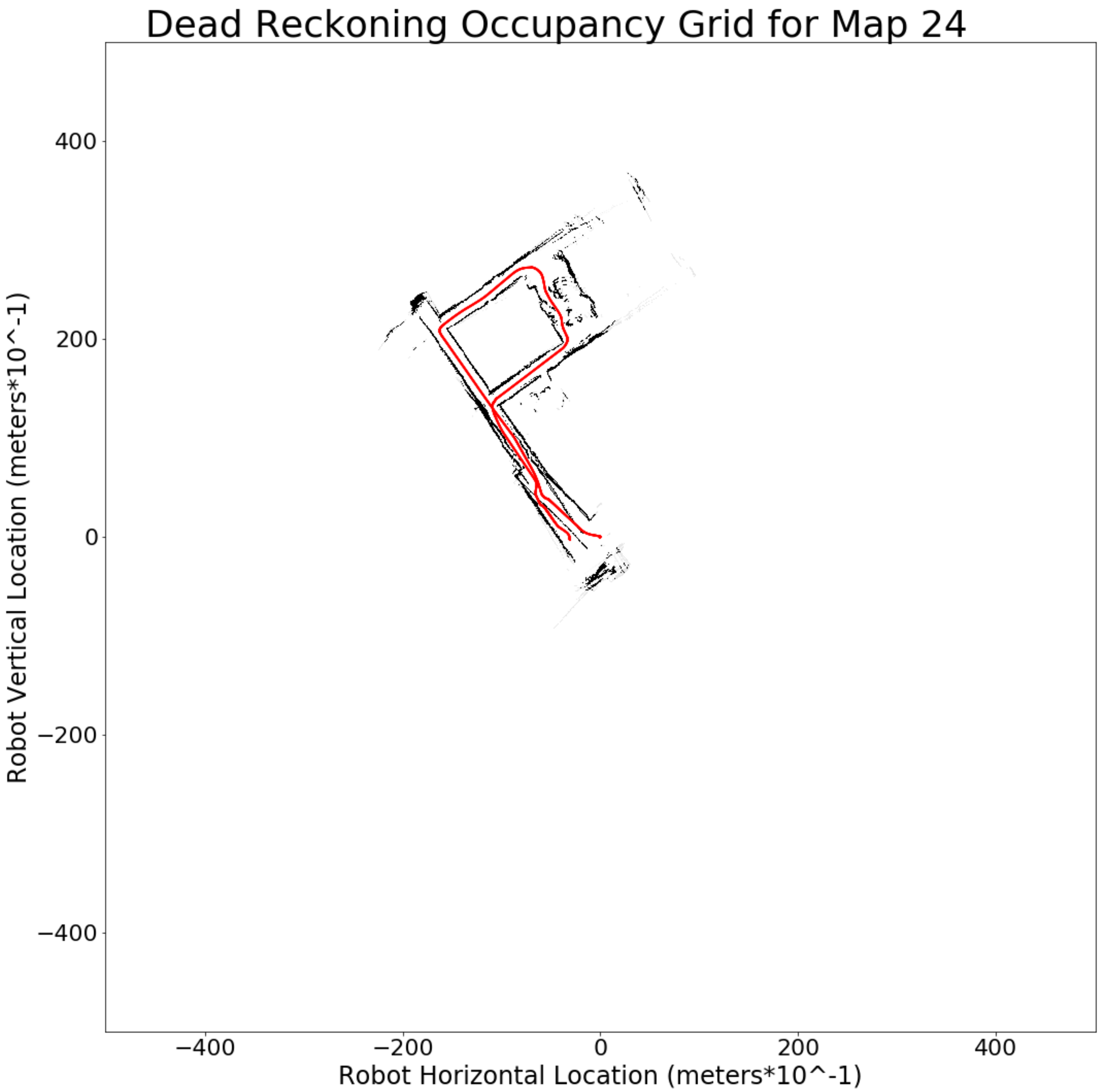


Figure 16: Dead Reckoning Trajectory - Map 24

Appendix D Slam Map Plots

Shown below are the occupancy grid plots for both the training and testing maps.

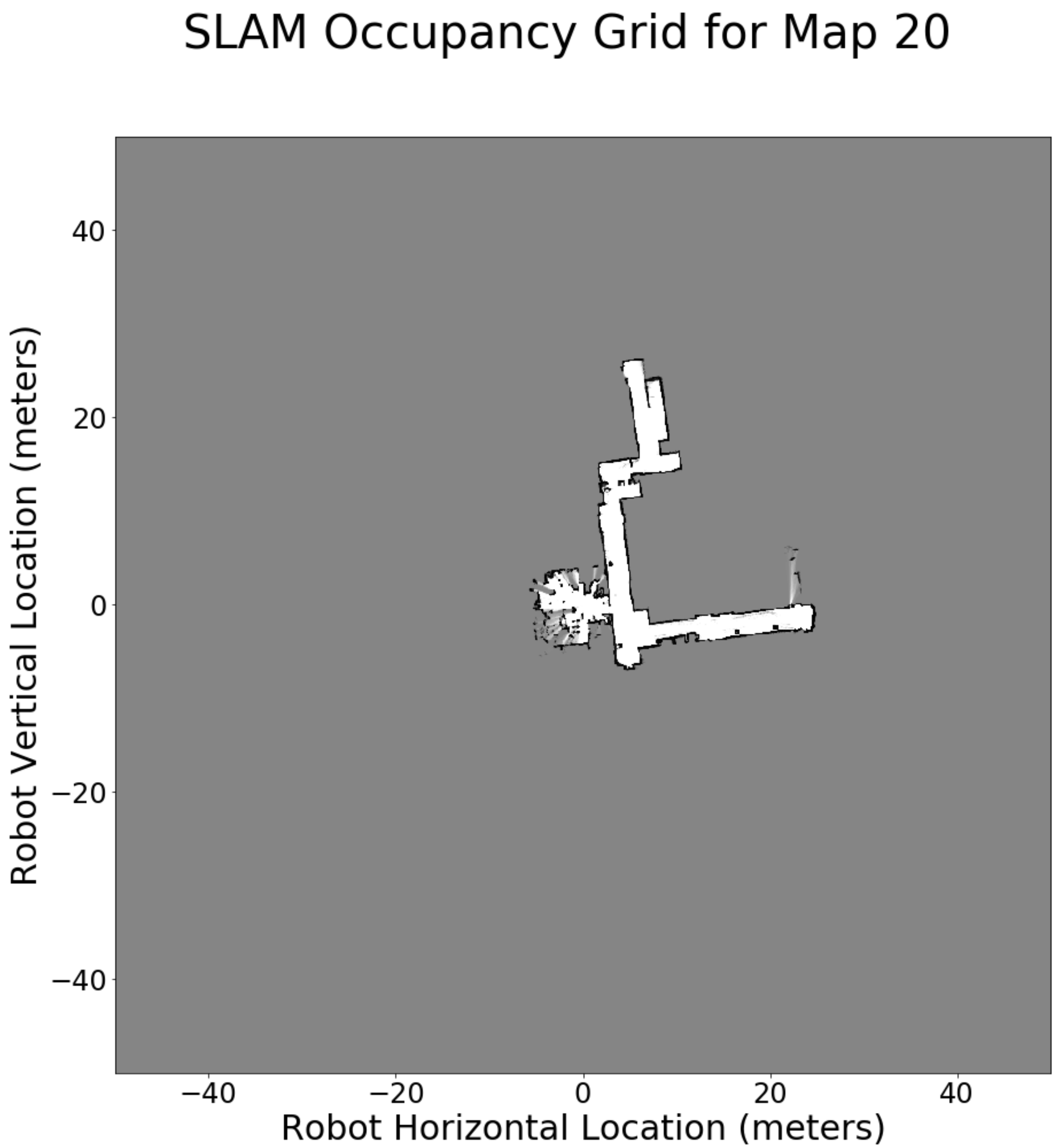


Figure 17: SLAM Map 20

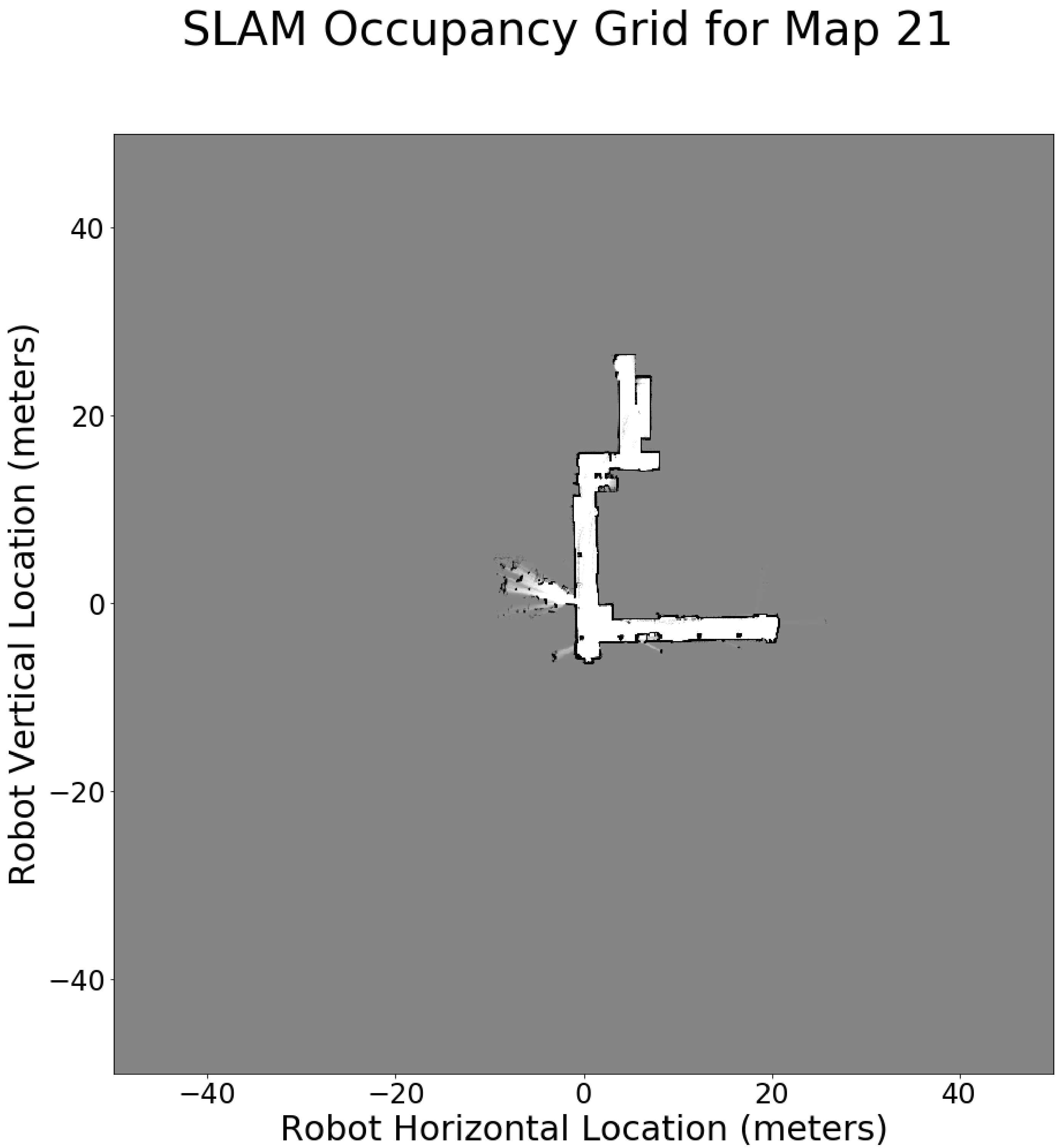


Figure 18: SLAM Map 21

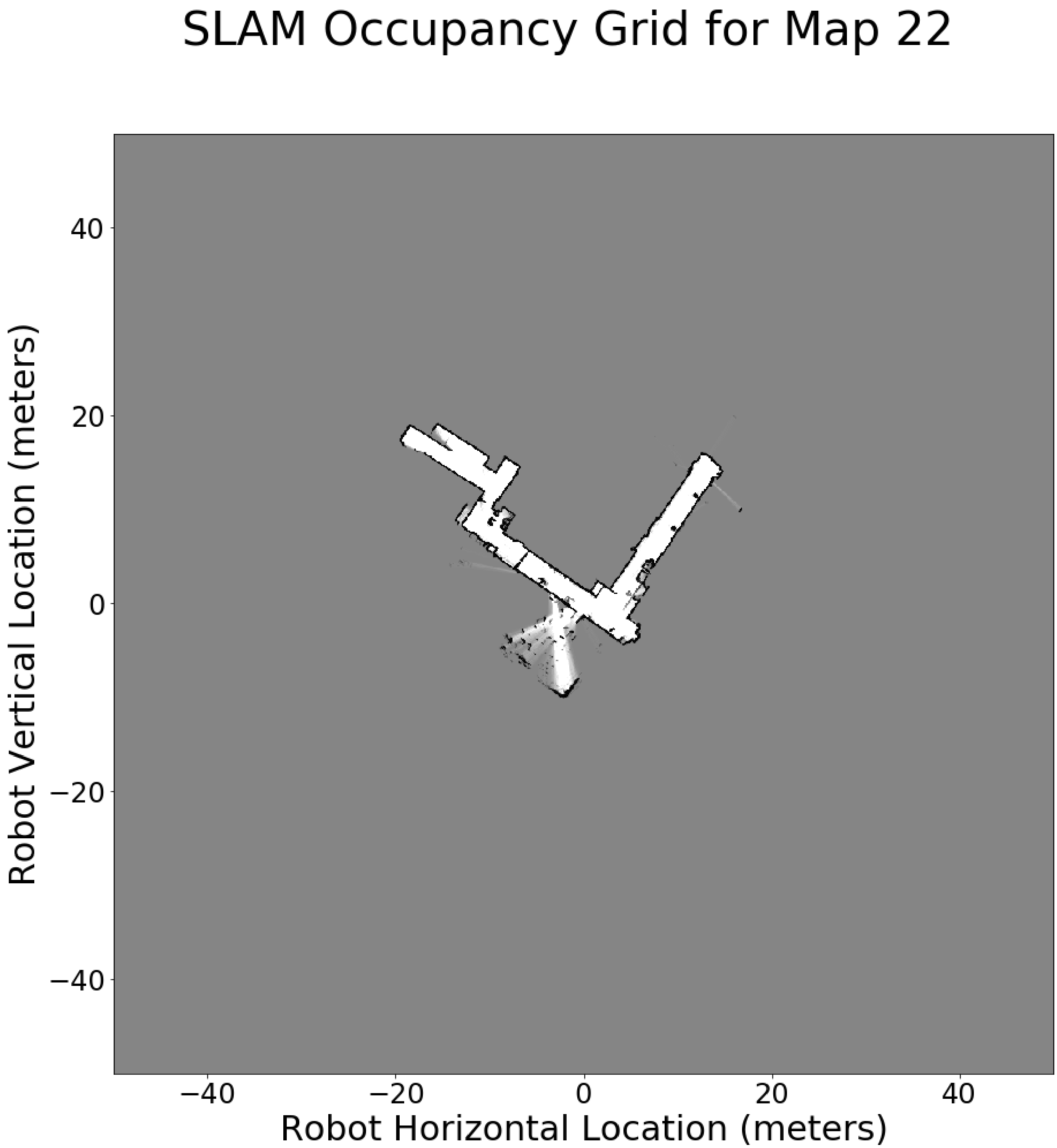


Figure 19: SLAM Map 22

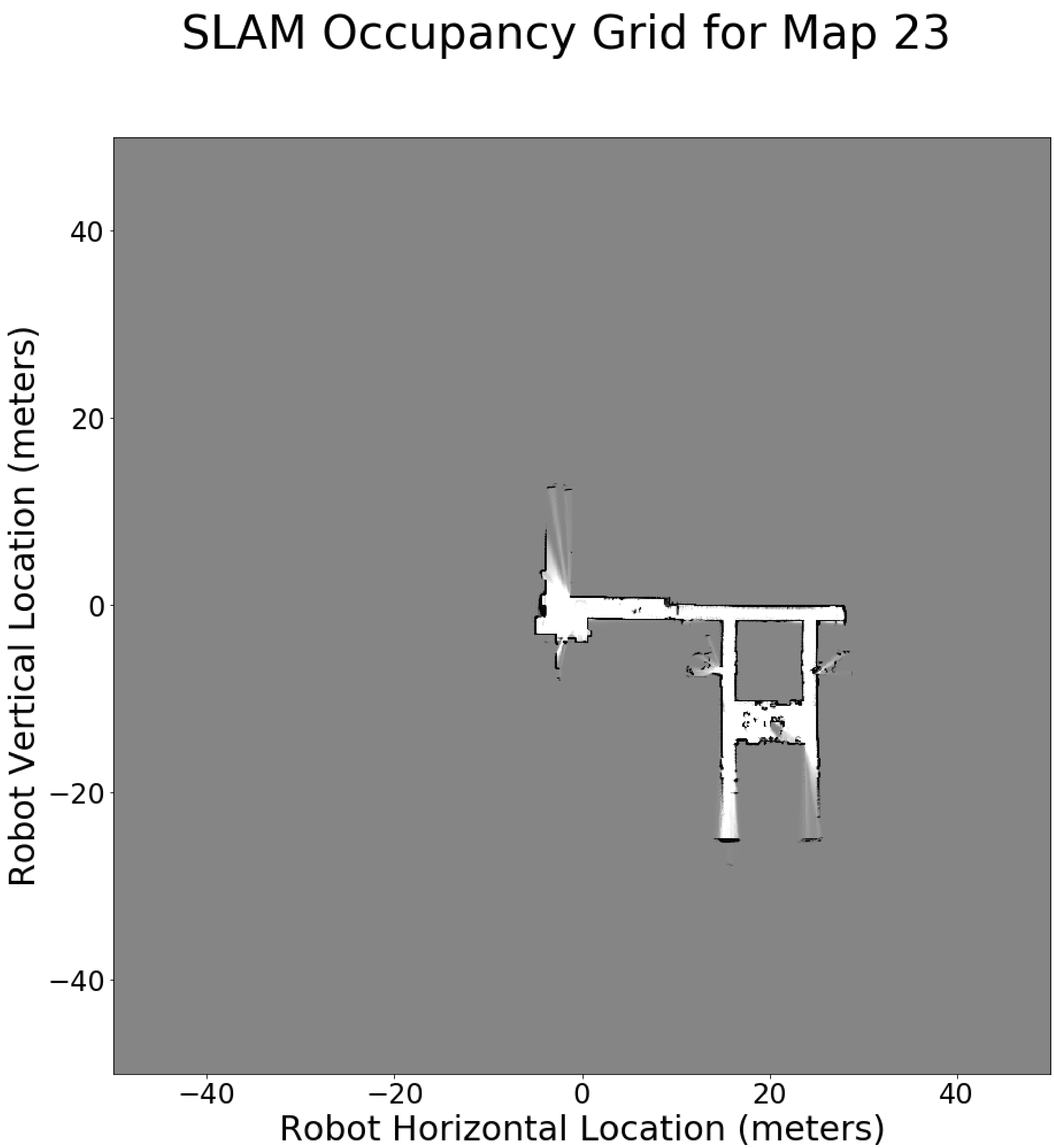


Figure 20: SLAM Map 23

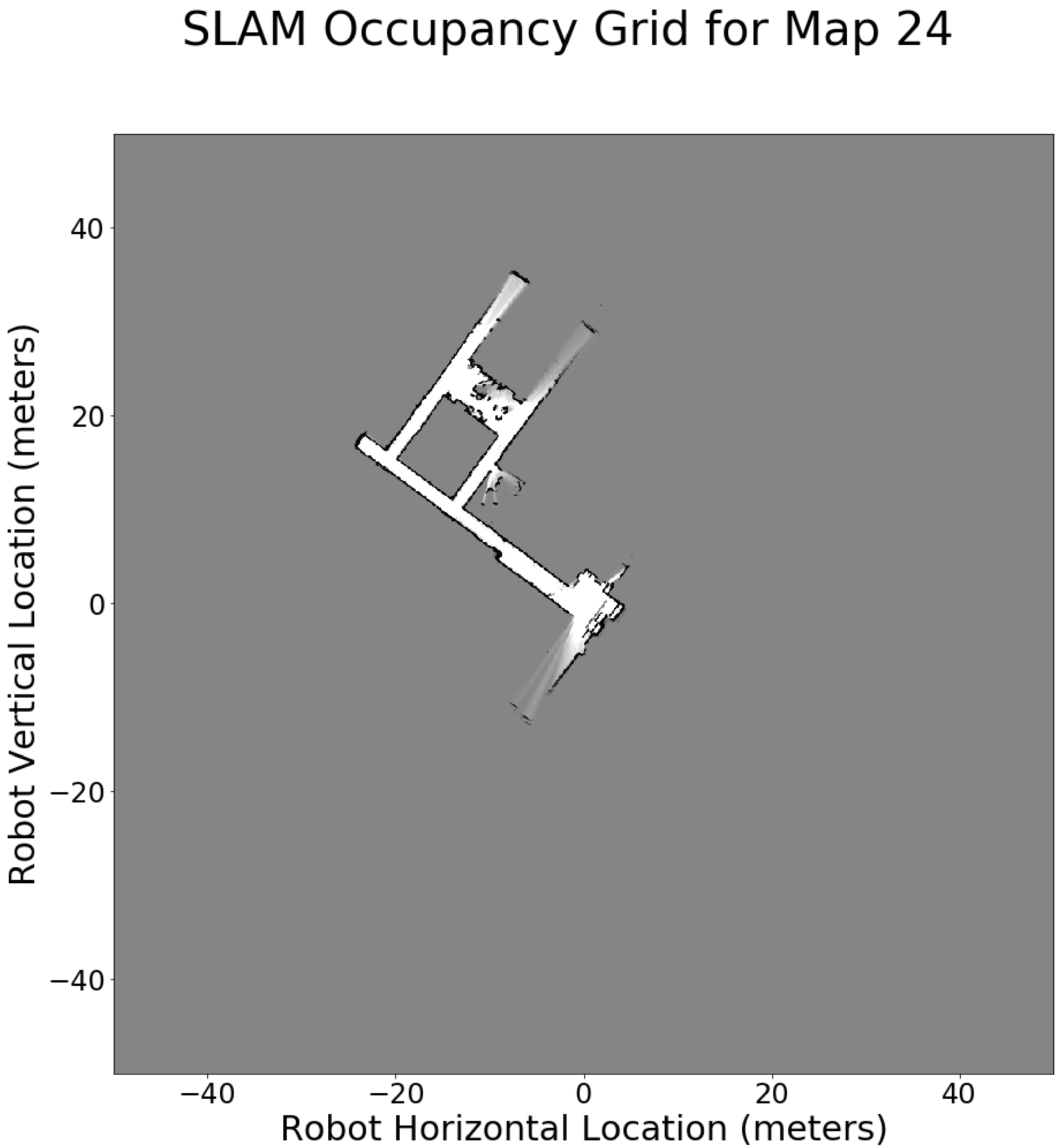


Figure 21: SLAM Map 24

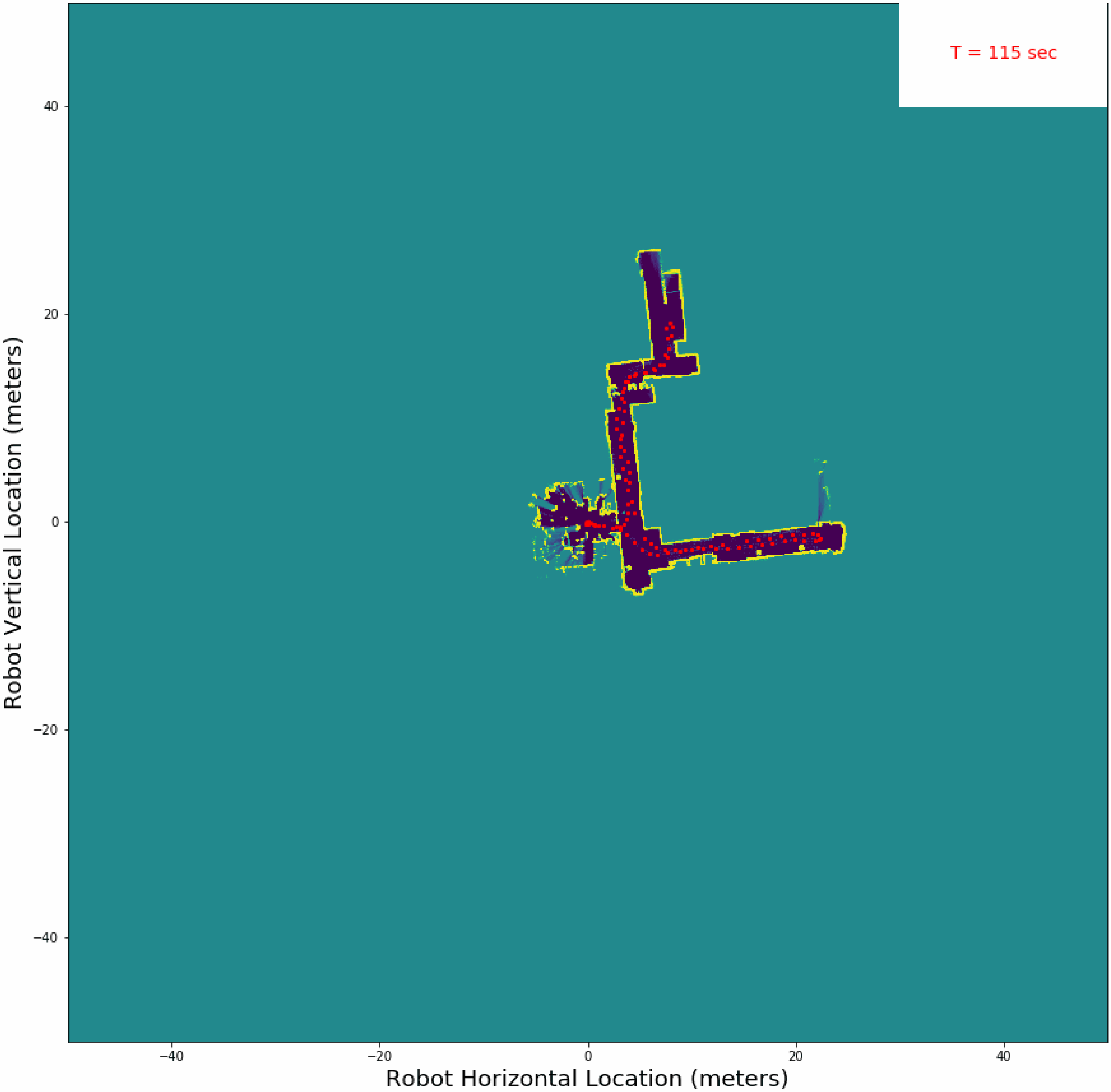


Figure 22: SLAM Robot Trajectory - Map 20

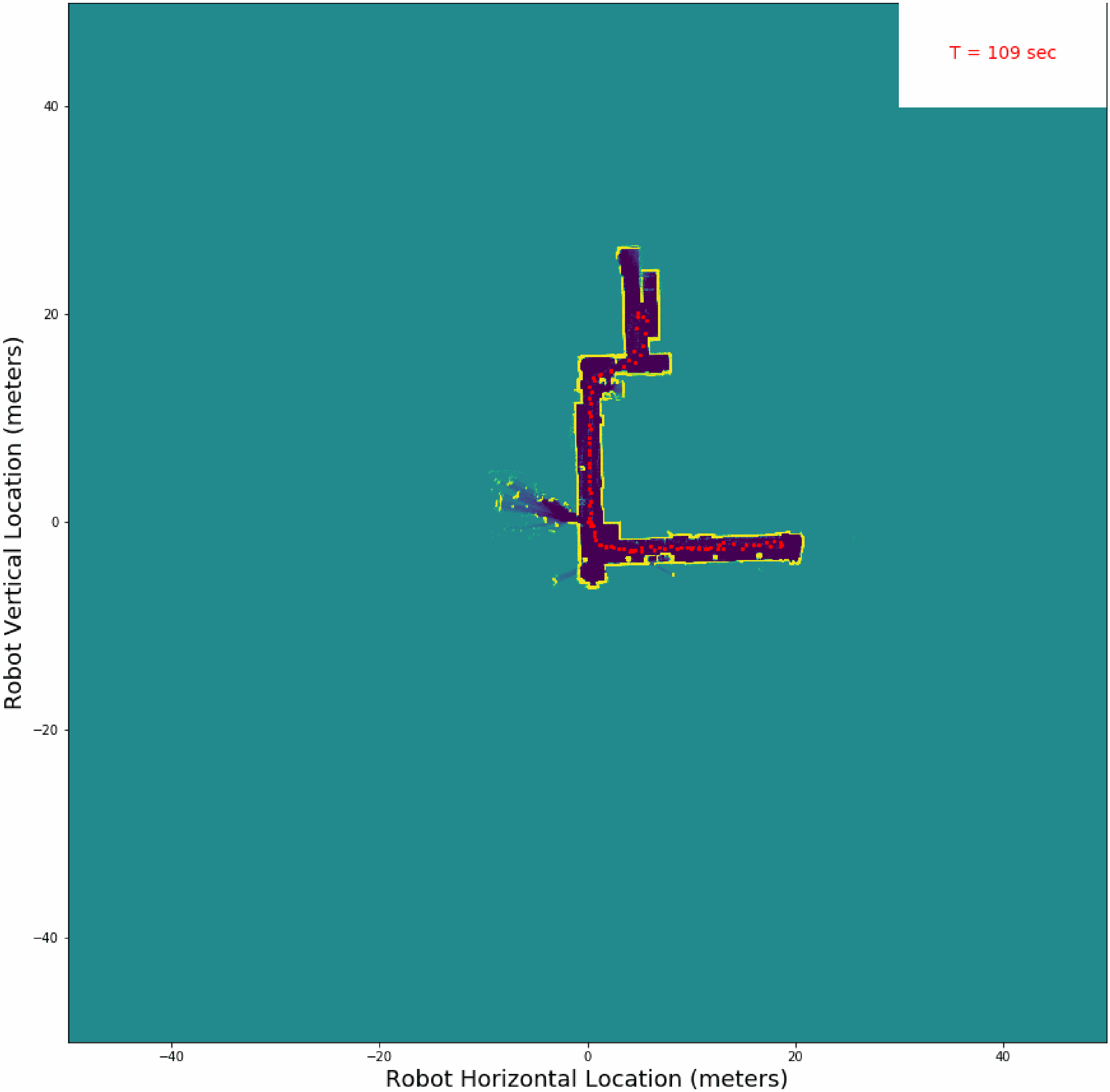


Figure 23: SLAM Robot Trajectory - Map 21

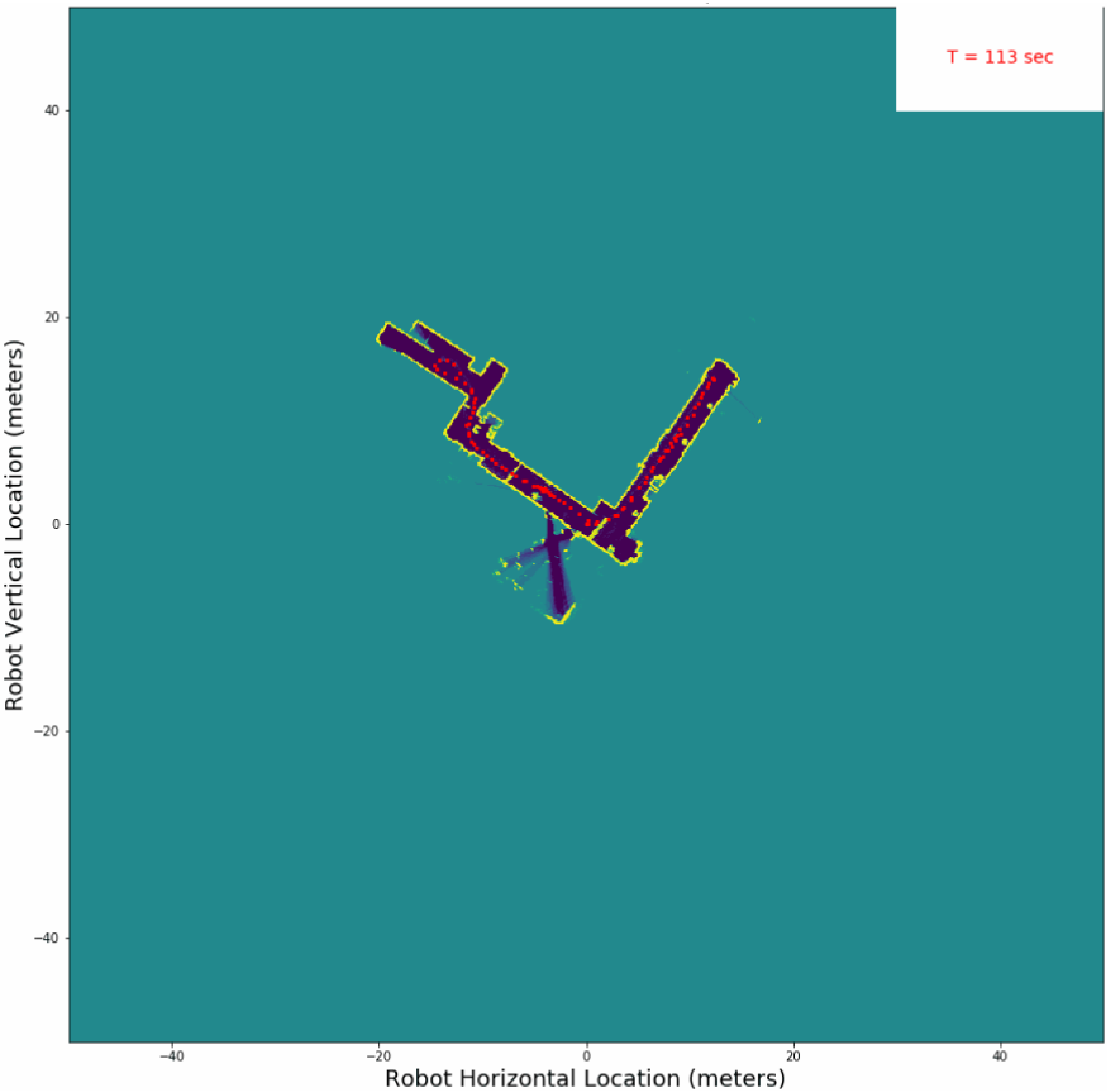


Figure 24: SLAM Robot Trajectory - Map 22

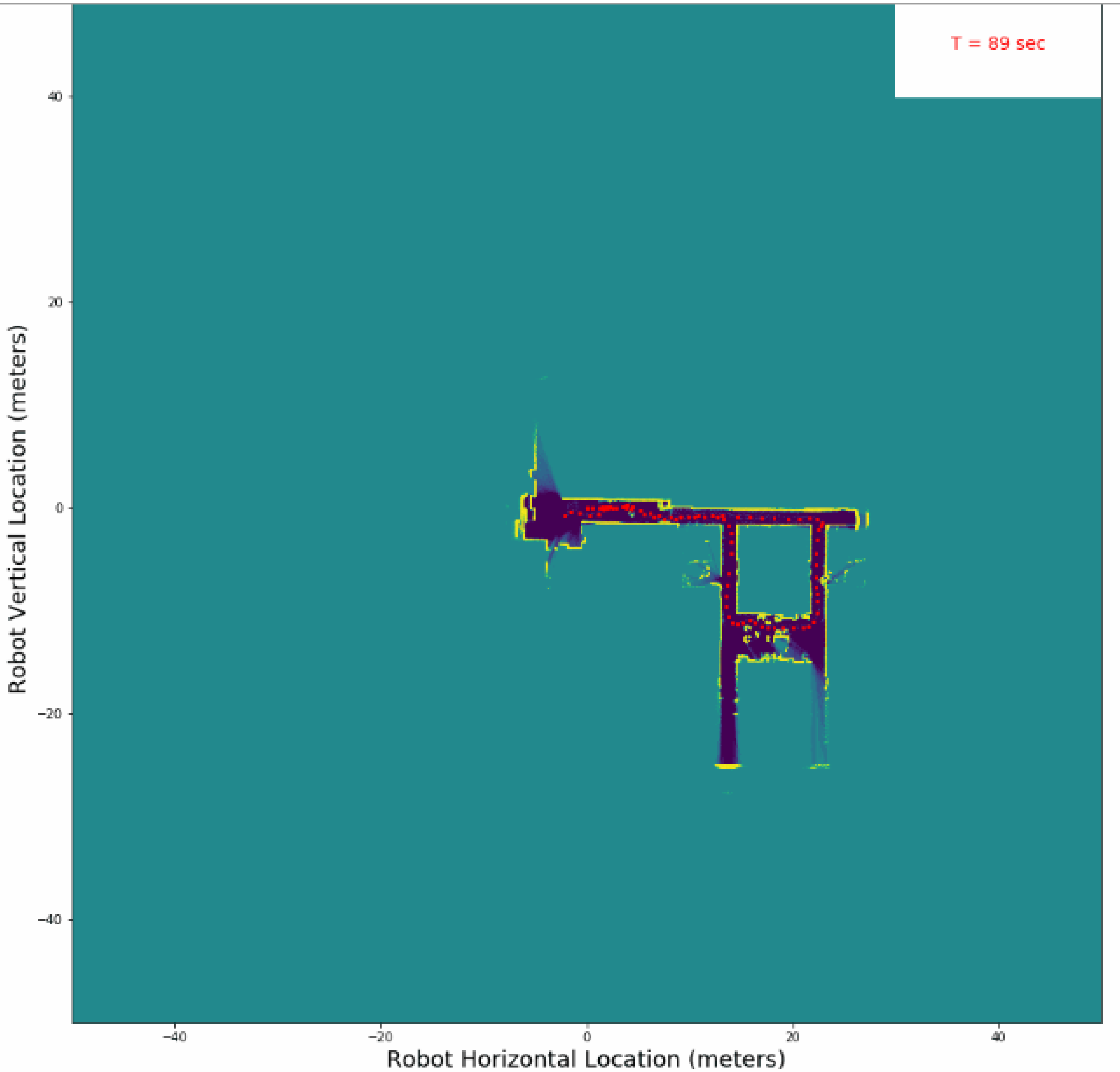


Figure 25: SLAM Robot Trajectory - Map 23

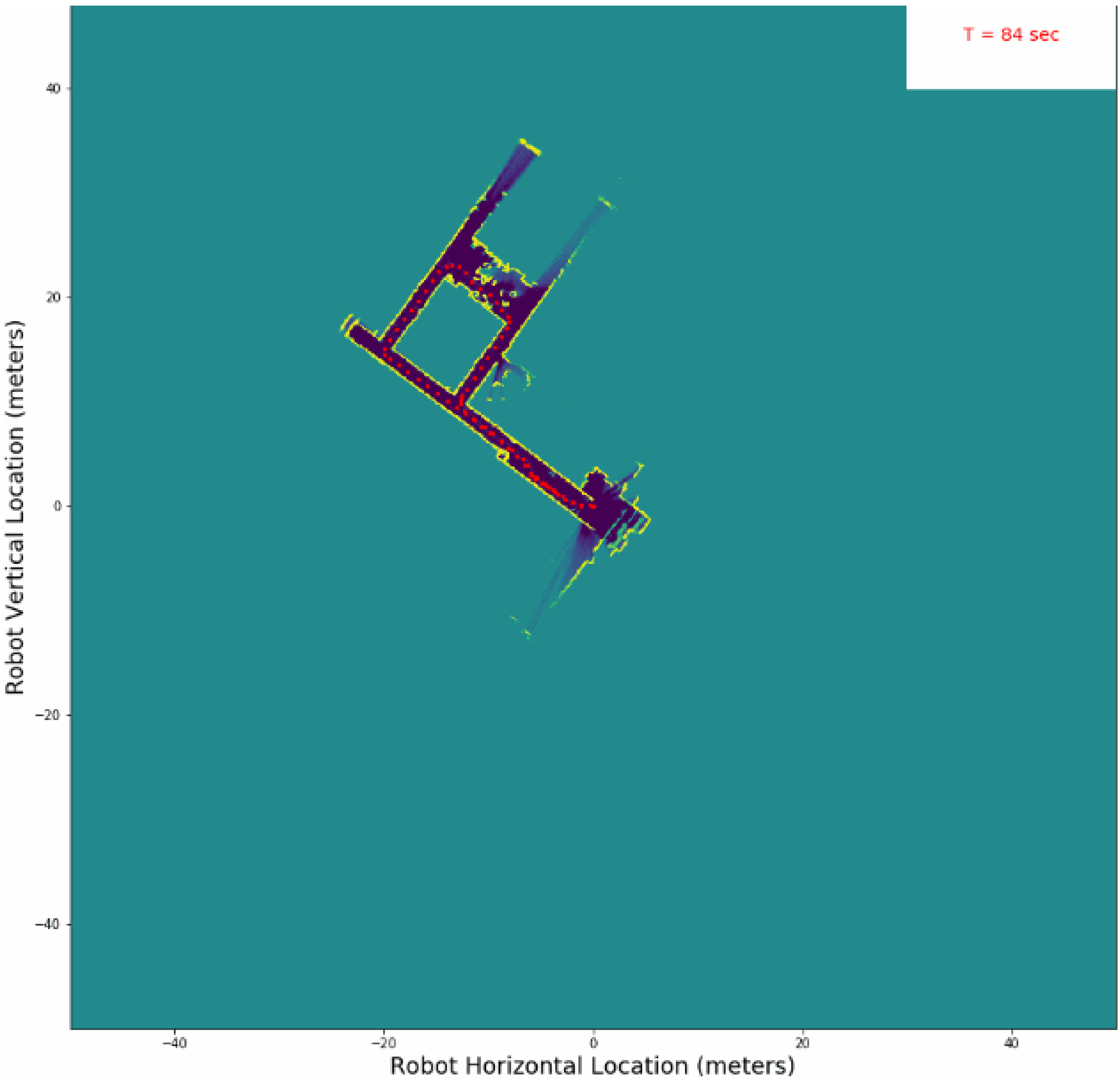


Figure 26: SLAM Robot Trajectory - Map 24

Appendix E SLAM Algorithm

```

# coding: utf-8

# In[1]:


# import built in modules
import sys
import numpy as np
import os
import matplotlib.pyplot as plt
import matplotlib
import zipfile as zf

get_ipython().system('pip install bresenham')
from bresenham import bresenham
import pandas as pd
import time
# !pip install tqdm -U --user
get_ipython().system('pip install tqdm')
# import tqdm
from tqdm import tqdm_notebook as tqdm
from scipy.stats import norm as norm_rv
from scipy.stats import rv_discrete
from matplotlib.animation import FuncAnimation


# In[2]:


''' NOTE: This code will be used when running code on AWS, to unzip
    file'''
# files = zf.ZipFile("ec833_project3_odom.zip", 'r')
# files.extractall('')
# files.close()


# In[3]:


''' NOTE: Must NOT separate custom module imports from raw data import
    '''

# import custom modules + data

import load_data, dataloader

custom_modules = ["load_data","dataloader"]

# DELETE: assures up-to-date local modules are always used
for module_par in custom_modules:
    del sys.modules[module_par]

# RELOAD: assures up-to-date local modules are always used
import load_data, dataloader


# load all data from target folder
data_folder = "data"

# produces list with objects of class "map_object" in it
map_list_train = list(dataloader.load_folder(data_folder))

for map_number in range(len(map_list_train)):

```

```python
    print("Map ID",map_list_train[map_number].id)
    print("Units are Meters. Robot is assumed to start facing right at coordinate (0,0)")
    fig1 = plt.figure(figsize=(15,15))
    plt.plot(map_list_train[map_number].cummulative_displacement[:,0],             map_list_train[map_number].cummulative_displacement[:,1])
    plt.xlabel('Robot Horizontal Location (meters)', fontsize=18)
    plt.ylabel('Robot Vertical Location (meters)', fontsize=18)
    plt.xticks(fontsize= 16)
    plt.yticks(fontsize= 16)

    fig1.suptitle('Odometry Navigation Map {0}'.format(map_list_train[map_number].id), fontsize=20)
    fig1.tight_layout()
    fig1.subplots_adjust(top=0.95)
    fig1.savefig('maps/odometry_map{0}.png'.format(map_list_train[map_number].id))

    plt.show()


# In[4]:


top = '\n##########################################################'
bot = "##########################################################\n"


# In[5]:


def get_effective_n(weights_current):
    return 1 / np.sum(weights_current**2)

     #how to test this:
    '''a = np.array([[4],[6]])
    print(a**2)'''



# In[6]:


def normalize_weights(weights_resampled_par):
    weights_resampled_par /= np.sum(weights_resampled_par)
    return weights_resampled_par

    #how to test this:
    ''' weights_test = np.ones(10)/10
        weights_test /= 2
        print((weights_test[0]))'''



# In[7]:


def resample_particles(weights_current,particles_par):

    # get the number of weights
    weight_cnt = weights_current.shape[0]

    weights_cum = np.asarray([np.sum(weights_current[0:i+1]) for i in range(weight_cnt)])
```

```
    # produce n uniform random samples
    random_density_samples = np.random.rand(weight_cnt)

    list_add = []
    for sample in random_density_samples:
        list_add.append(np.sum(weights_cum < sample))

    randomly_sampled_indexes = np.asarray(list_add)

    weights_resampled  = weights_current[randomly_sampled_indexes]
    weights_resampled = normalize_weights(weights_resampled)

    particle_resampled = particles_par[randomly_sampled_indexes]

    return particle_resampled,weights_resampled

    #how to test this:
    '''print(np.argsort(weights_current),"np.argsort(weights_current)
    ")
    print(np.argsort(np.bincount(randomly_sampled_indexes)),"np.
    argsort(np.bincount(randomly_sampled_indexes))")'''


# In[8]:


def plot_particles(x0, y0, theta0, x_noise, y_noise, theta_noise):

    # create a single 10 x 10 figure
    fig_particles, plt_particles = plt.subplots(figsize=(10,10))

    # print
    title_particles = str("pose_angle: "+str(pose_angle_old_SLAM*180/
    np.pi)+                               " | d_theta from ODOMETRY "+str(
    d_theta*180/np.pi)+"| dx from ODOMETRY "+str(dx)+
              " dy from ODOMETRY"+ str(dy))
    fig_particles.suptitle(title_particles, fontsize=16)


    plt_particles.set_xlabel('distance (PIXELS)')
    plt_particles.set_ylabel('distance (PIXELS)')

    plt_particles.quiver(x0,y0,np.cos(theta0),np.sin(theta0))
    plt_particles.quiver(x_noise,y_noise,np.cos(theta_noise),np.sin(
    theta_noise), color='m', alpha=1, headwidth=.3*conv_factor)

    plt_particles.ticklabel_format(useOffset=False)

    plt.show()

    #     fig_particles.show()
    print("TIME STEP FOR PLOT PARTICLE ABOVE:",time_step)



# In[9]:


def make_lidar_rays(lidar_global_hit_locs_par,particles_par):

    # extract the 0th column of all particles, which represents the
    angles
    angle_matrix = particles_par[:,0]

```

```
    # make an array where each column is the consine and sine of the angle associated with each particle
    transform_matrix = np.asarray([np.cos(angle_matrix),np.sin(angle_matrix)])

    '''lidar_global_hit_locs_par contains the rotation matrix corresponding to each ray (1080 of them) in its local
    coordinate frame (angles from -135 to 135). The dot product below computes the transformation to the local frame'''
    global_lidar_vecs_matrix = np.dot(lidar_global_hit_locs_par, transform_matrix)
    #print(global_lidar_vecs_matrix.shape) # (1810 x 2 by number of particles)
    return global_lidar_vecs_matrix


# In[10]:


def update_map(points_impacted_par,map_array_par):
    map_array_update = np.copy(map_array_par)
    if map_array_update[points_impacted_par[-1]]<100:
            map_array_update[points_impacted_par[-1]] += 10    #* (angle_factor)

    for i in points_impacted_par[0:-1]:
        if map_array_update[i] > -100:
            map_array_update[i] += -.5
    return map_array_update


# In[11]:


def make_noise(dx,dy,d_theta,particle_cnt):

    # make numbers between -1 and 1
    noise_x = 2 * (np.random.rand(particle_cnt)-0.5)
    # scale ONLY RANDOM noise by conversion factor
    random_noise_x = x_random_noise_factor * noise_x * conv_factor
    # recalculate percent noise based on GLOBAL factor
    noise_x = factor_x * noise_x

    # make numbers between -1 and 1
    noise_y = 2 * (np.random.rand(particle_cnt)-0.5)
    # scale ONLY RANDOM noise by conversion factor
    random_noise_y =   y_random_noise_factor* noise_y * conv_factor
    # recalculate percent noise based on GLOBAL factor
    noise_y = factor_y * noise_y

    # make numbers between -1 and 1
    noise_theta = 2 * (np.random.rand(particle_cnt)-0.5)
    # scale ONLY RANDOM noise by conversion factor
    random_noise_theta = theta_random_noise_factor * noise_theta
    # recalculate percent noise based on GLOBAL factor
    noise_theta = factor_theta * noise_theta

    noised_d_x     = dx * (1 + noise_x)          + random_noise_x
    noised_d_y     = dy * (1 + noise_y)          + random_noise_y
    noised_d_theta = d_theta * (1 + noise_theta) + random_noise_theta

    min_dx = np.amin(noised_d_x)
    max_dx = np.amax(noised_d_x)

    min_dy = np.amin(noised_d_y)
```

```
    max_dy = np.amax(noised_d_y)

    min_d_theta = np.amin(noised_d_theta)
    max_d_theta = np.amax(noised_d_theta)

    return noised_d_x, noised_d_y, noised_d_theta,            min_dx, min_dy, min_d_theta,            max_dx, max_dy, max_d_theta


# In[12]:


# number of particles for model
particle_cnt = 300

# threhold for robot-body lidar noise, in meters, compared with lidar readings
robot_self_dist = .33

# +/- angle (with respect to straight ahead direaction) that is not expected to result in robot-body lidar noise
angle_reducer = 90

# pixels per meter
conv_factor = 10

# wdith of map in meters
map_width = 100

# holds lists of arrays, which are used to produce plots
map_plot_list = []


# hold list of lidar parameters
map_lidar_list = []

# hold numpy arrays of all maps
all_map_list = []

# number of (particle / map) plots allowed
particle_plot_cnt_allowed = 300

# numeric parameter of width of current map in pixels, will be a function parameter at the end
map_cm = map_width * conv_factor
map_cm = map_cm + 1

# if the map width is not divisible by 2, issue warning
if ((map_cm - 1)%2) > 0:
    print ("ISSUE WITH THE MAP DIMENSION!")

# center in x and y
x_center = int((map_cm - 1)/2)
y_center = np.copy(x_center)

# start plotting information at this time step
start_time_master = 1700

# complete this number of steps before stopping plotting
time_steps_master = 5000

# number of time steps between plots
plot_interval = 3

# hyperparameter save file parameter
```

```
saved = 0



# number of angles sampled for every time step
downsampling_par = 2

# iterate through all the maps
for idx in range(len(map_list_train)):

    # capture start time
    start = time.time()

    plot_array_list = []
    # initialize particle weights as *** 1/number of particles
    weights = np.ones(particle_cnt)/particle_cnt

    # initialize weights old with the same uniform values
    weights_old = np.copy(weights)

    # print map id from filename
    print("Map ID:",map_list_train[idx].id)

#     if int(map_list_train[idx].id) != 24 and int(map_list_train[idx].id) != 23:
#         continue

# #      if map is not 23,skip (for testing only)
#     if int(map_list_train[idx].id) != 23:
#         continue

#     if int(map_list_train[idx].id) != 20:
#         continue

#     if map is not 21,skip (for testing only)
#     if int(map_list_train[idx].id) != 21 and int(map_list_train[idx].id) != 23:
#         continue

#     # if map is not 20,skip (for testing only)
#     if int(map_list_train[idx].id) == 20:
#         continue


    # list to hold IMU data lined up with encoder and lidar data
#     acceleration_list = []


    print("Execution Started")

    # map nummpy array (to be populated)
    map_array = np.zeros([map_cm,map_cm])

    # times, angles and ray return distances from a map id 'idx'
    lidar_data = map_list_train[idx].lidar

    # Number of time observations for this map
    time_ob_cnt    = len(lidar_data)

    # time index where we start aligning lidar and encoder using a subsampled version of the encoder time array
    range_start = 3

    time_match_list = []

```

```
    ''' meas_per_tstep: get number of lidar measurements for every time step.
          lidar_angle_vec: get lidar angle vector array (same angles spanned for
          all measurements, hence 17 hardcoded)

    17 is safely HARDCODED here because, no matter how many time steps there are,
    there will be many more than 17 (picked that number just 'because')
    all the measurements will have the same number of angular lidar measurements (1801)
    but didn't want to hardcode 1801, just in case.'''

    # lidar angle vector is from -135 to 135 ********in RADIANS*******
    meas_per_tstep  ,   lidar_angle_vec =   lidar_data[17]['angle'].shape[0]  ,    lidar_data[17]['angle']

    # make list of transformations for local to global coordinate frames
    ray_transform_vec =np.asarray([ [[np.cos(lidar_angle_vec[i]), -np.sin(lidar_angle_vec[i])],                                                                [np.sin(lidar_angle_vec[i]),  np.cos(lidar_angle_vec[i])]]
                                    for i in range(meas_per_tstep)])
    # transform list to array
    ray_transform_vec = np.squeeze(ray_transform_vec)

    '''print(ray_transform_vec[540+180])
       TESTED EQUALS [[ 0.70710678 -0.70710678]
                      [ 0.70710678  0.70710678]]'''

    #compute lidar off-center distance from wheelbase
    wheelbase_len   = 0.33020

    # locate distance of lidar lever from center of robot
    lidar_lever_len = 0.29833 - (wheelbase_len/2)

    # convert to small unit
    lidar_lever_len = lidar_lever_len * conv_factor

    # initialize old float (prior step) position for encoder
    x0_encoder_float_old = np.copy(x_center)
    y0_encoder_float_old = np.copy(y_center)

    # initialize old pose angle for encoder
    pose_angle_old = 0
    pose_angle_old_SLAM = 0

    # initialize old float (prior step) position for lidar basepoint
    x0_lidar_float_old = x_center + lidar_lever_len*np.cos(pose_angle_old)
    y0_lidar_float_old = y_center + lidar_lever_len*np.sin(pose_angle_old)

    # initialize old int (prior step) position for lidar basepoint. this is fine since the lidar
    # will hit a point different from 0,0 as soon as it starts running
    x1_lidar_int_old = 0
    y1_lidar_int_old = 0

    # counter of particle plots, to limit the number of plots generated at runtime
    particle_plot_cnt = 0

```

```
    # an array holding the ENCODER time values for the current map as indexed by idx
    time_array_encoder = map_list_train[idx].processed_encoder[:,0]

    # the IMU time for the current map as indexed by idx
    imu_time = np.transpose(map_list_train[idx].imu)[:,6]

    # the IMU data for the current map as indexed by idx
    imu_data = np.transpose(map_list_train[idx].imu)[:,:6]

    # random numbers for particle coloring, each color is 3 channels
    colors_rays = tuple(np.random.rand(particle_cnt,3))
    color_center = tuple(np.random.rand(3))

    list_test=[]

    displacements = map_list_train[idx].cummulative_displacement

    for time_step in tqdm(range(time_ob_cnt)):


        ## if we are testing, use this to stop after a certain number of time steps
#         if time_step > time_steps_master + start_time_master:
#             break

        # boolean variable: allow print if not too many plots have been generated
        allow_print = False #particle_plot_cnt < particle_plot_cnt_allowed and time_step >= start_time_master and time_step % plot_interval == 0

        # list of hits for a time step based on the x,y and angle prediction
        hit_list = []

        # if time is less than a small integer, find the smallest difference between the encoder time array
        # and current lidar time, return the index
        if time_step < range_start:
            lidar_time_closest_arg = (np.abs(time_array_encoder - map_list_train[idx].lidar[time_step]['t'])).argmin()

        # if time is larger, find the smallest difference between as smaller version (size is determined by range_start)
        # of the encoder time array and the current lidar time, return the index
        else:
            lidar_time_closest_small = time_array_encoder[lidar_time_closest_arg:lidar_time_closest_arg+range_start]
            lidar_time_closest_arg = (np.abs(lidar_time_closest_small - map_list_train[idx].lidar[time_step]['t'])).argmin() + lidar_time_closest_arg

        # extract the right index for the imu time based on the closed encoder time
#         imu_time_closest_arg   =  np.absolute(time_array_encoder[lidar_time_closest_arg]-imu_time).argmin()

        # TEST THE LINES ABOVE TO VERIFY TIMES ALIGN
#         print(imu_time[imu_time_closest_arg])
#         print(time_array_encoder[lidar_time_closest_arg])

#         if time_step > range_start:
#             vertical_acc      = imu_data[  imu_time_closest_arg,  2]
```

```
#              vertical_acc_vec  = imu_data[0:imu_time_closest_arg,  2]
#              vertical_std      = np.std (vertical_acc_vec)
#              vertical_mean     = np.mean(vertical_acc_vec)
#              vertical_outlier = (.995*vertical_acc < vertical_mean)*
    vertical_std >.005

        # robot pose angle in RADIANS
        pose_angle = map_list_train[idx].cummulative_angle[
    lidar_time_closest_arg]

#     TESTED: JUST COMMENT ALL BELOW THESE LIST/TWO PLOT LINES BELOW
#         list_test.append(pose_angle)
#     plt.plot(np.asarray(list_test)*180/np.pi)
#     plt.plot(np.zeros(len(list_test)))
#     plt.show()

     # get x coordinate of robot, initially based on encoder
        x0_encoder_float = map_list_train[idx].
    cummulative_displacement[lidar_time_closest_arg,0]

        # convert to small unit (i.e.: centimeters), converted
        x0_encoder_float = x0_encoder_float * conv_factor

        # account for scaling center shift, converted
        x0_encoder_float = x0_encoder_float + x_center

        # transfer to centroid of lidar, account for lidar being off
    center with encoder, BOTH IN SMALL UNITS
        x0_lidar_float = x0_encoder_float + lidar_lever_len*np.cos(
    pose_angle)

        # get y coordinate of robot, initially based on encoder
        y0_encoder_float = map_list_train[idx].
    cummulative_displacement[lidar_time_closest_arg,1]

        # convert to small unit (i.e.: centimeters)
        y0_encoder_float = y0_encoder_float * conv_factor

        # account for scaling center shift
        y0_encoder_float = y0_encoder_float + y_center

        # transfer to centroid of lidar, account for lidar being off
    center with encoder
        y0_lidar_float= y0_encoder_float + lidar_lever_len*np.sin(
    pose_angle)

        x0_lidar_int = int(np.around(x0_lidar_float))
        y0_lidar_int = int(np.around(y0_lidar_float))

        # measure distance travelled in time step
        distance_traveled = np.linalg.norm(np.array([[y0_lidar_float-
    y0_lidar_float_old],
           [x0_lidar_float-x0_lidar_float_old]]))

        '''if distance travelled is less than one micrometer,
        AND change is heading angle is less than .01 degree(s) --->
    cell updates based on previous reading'''
        if time_step != 0 and distance_traveled < conv_factor *
    .0000010 and abs(pose_angle-pose_angle_old)<np.pi/(18000):
            map_array = update_map(points_impacted,map_array)
            continue

        factor_x = .8/20
        factor_y = .1/10 #.35
        factor_theta = 3/20
```

```
        #aqui
        x_random_noise_factor = 0.03072
        x_random_noise_factor = x_random_noise_factor*30

        y_random_noise_factor = 0.025
        y_random_noise_factor = y_random_noise_factor /.5

        theta_random_noise_factor = 0.012
        theta_random_noise_factor = theta_random_noise_factor*5

        d_theta = (pose_angle - pose_angle_old)
        dx      = (x0_encoder_float - x0_encoder_float_old)
        dy      = (y0_encoder_float - y0_encoder_float_old)

        noised_d_x, noised_d_y, noised_d_theta,        min_dx,min_dy, min_d_theta,        max_dx,max_dy,max_d_theta = make_noise(dx,dy, d_theta,particle_cnt)

        if saved == 0:
            hyperparameters = (factor_x, factor_y, factor_theta, x_random_noise_factor, y_random_noise_factor, theta_random_noise_factor)
            hyper_filename = "hyper_parameters/hyperparameters_for_map_"+str(map_list_train[idx].id)+str(time.time())+".txt"
            hyperparameters = np.savetxt(hyper_filename,np.asarray(hyperparameters))
            saved = 1
        if get_effective_n(weights) < particle_cnt * .5:
#         if get_effective_n(weights) < 15:
#             print("resampling particles. Effective n:", get_effective_n(weights),"Time Step",time_step,"\n",  )
#             print(np.argsort(weights))
            particles,weights_old = resample_particles(weights, particles)

        if time_step == 0:
            pt_no_noise = pose_angle # store old ANGLE
            px_no_noise = x0_encoder_float # store old x encoder origin
            py_no_noise = y0_encoder_float # store old y encoder origin

            theta_particles = pose_angle       + noised_d_theta # update encoder angle with noise
            x_particles     = x0_encoder_float + noised_d_x     # udpate encoder x with noise
            y_particles     = y0_encoder_float + noised_d_y     # update encoder y with noise

        if time_step >0:
            # parameters in particles before any new displacements or angle changes are added
            pt_no_noise = particles[:,0] # store particles before noise, ANGLE
            px_no_noise = particles[:,1] # store old x encoder origin for particles
            py_no_noise = particles[:,2] # store old y encoder origin for particles

            # parameters in particles after new displacements or angle changes are added
            theta_particles = particles[:,0] + noised_d_theta # update encoder angle with noise
```

```
            x_particles     = particles[:,1] + noised_d_x     # update encoder x with noise
            y_particles     = particles[:,2] + noised_d_y     # update encoder y with noise

        # score the particle data in "particles variable"
        particles = np.hstack((theta_particles[:,np.newaxis], x_particles[:,np.newaxis],y_particles[:,np.newaxis]))

        if False and allow_print: # if printing is allowed, pring the particles.
            particle_plot_cnt +=1
            plot_particles(px_no_noise,          py_no_noise          , pt_no_noise,                           particles[:,1], particles[:,2], particles[:,0])

        '''update "old_encoder", "old lidar" and "old_pose angle". '''
        x0_encoder_float_old = np.copy(x0_encoder_float)
        y0_encoder_float_old = np.copy(y0_encoder_float)

        x0_lidar_float_old   = np.copy(x0_lidar_float)
        y0_lidar_float_old   = np.copy(y0_lidar_float)

        pose_angle_old       = np.copy(pose_angle)
        pose_angle_old_SLAM  = np.copy(pose_angle_old)

        # update the scan vector based on current time step, IN SMALL UNITS!
        lidar_scan_vec  = lidar_data[time_step]['scan'] * conv_factor

        '''At current time step, multiplies the hits (distances in meters) for every angle
        by the corresponding local (2x2) transform for that angle'''
        # returns scaled cosine and sine components of reading along the robot's local coordinate frame,
        # arranged in 2x2 matrices within a list, ready to transfer to global frame

        '''lidar_scan_vec holds the readings from the lidar from angles -135 to 135
        ray_transform_vec holds the transformation matrix at each of those angles'''
        lidar_global_hit_locs = [lidar_scan_vec[i]*ray_transform_vec[i] for i in range(meas_per_tstep)]

        # converts this list reading to numpy array, in meters
        lidar_global_hit_locs = np.asarray(lidar_global_hit_locs)

        # adds offset to x lidar distance reading (based on the lidar off-center mounting distance), in meters
        lidar_global_hit_locs = lidar_global_hit_locs +  np.array([[lidar_lever_len,0],[0,lidar_lever_len]])

        # converts to array and squeeze extra dimension
        lidar_global_hit_locs = np.asarray(lidar_global_hit_locs)
        lidar_global_hit_locs = np.squeeze(lidar_global_hit_locs)
#         print(lidar_global_hit_locs.shape,"        print(lidar_global_hit_locs.shape)")

        if time_step > 1:

            # transfer all lidar rays for all particles to local frame
            lidar_ray_matrix = make_lidar_rays(lidar_global_hit_locs, particles)

```

```
            # for each time step, make the particle score zero to begin with
            particle_score = np.zeros(particle_cnt)

            for reading ,ray in enumerate(lidar_ray_matrix):
                ''' for this given time step and angle, distance is less than 0.33 meters and abs(angle) > 90 degrees, SKIP THIS READING'''
                if reading % downsampling_par !=0                   or lidar_data[time_step]['scan'][reading] < robot_self_dist and (abs(180/np.pi*lidar_angle_vec[reading])> angle_reducer):
                    continue

                #for each of the 1081 readings, set the x and y endpoints to ray values plus the center of each particle
                x1_rays_float = ray[0,:]  + particles[:,1]
                y1_rays_float = ray[1,:]  + particles[:,2]

                # make these vectors integers
                x1_rays_int = np.around(x1_rays_float).astype(int)
                y1_rays_int = np.around(y1_rays_float).astype(int)

#                 particle_score += np.multiply(map_array[y1_rays_int,x1_rays_int]>0,map_array[y1_rays_int,x1_rays_int])
                particle_score += map_array[y1_rays_int,x1_rays_int]

            shift = np.amax(particle_score)
            weights = np.exp(particle_score-shift)
            weights = weights/np.sum(weights)
            weights = np.multiply(weights,weights_old)
            weights = normalize_weights(weights)

            weights_old = np.copy(weights)

            index_max = np.argmax(particle_score)
            pose_angle       =  particles[index_max,0]
            x0_encoder_float =  particles[index_max,1]
            y0_encoder_float =  particles[index_max,2]

            if allow_print:

                print("\n####################################################")
                print("Odometry dx", dx,"for time", time_step)
                print("Minimum dx", min_dx,"for time", time_step)
                print("Maximum dx", max_dx,"for time", time_step)
                print("Choosen dx", noised_d_x[index_max],"for time",time_step,"\n")

                print("Odometry dy", dy,"for time", time_step)
                print("Minimum dy", min_dy,"for time", time_step)
                print("Maximum dy", max_dy,"for time", time_step)
                print("Choosen dy", noised_d_y[index_max],"for time",time_step,"\n")

                print("Odometry d_theta", d_theta,"for time",time_step)
                print("Minimum d_theta", min_d_theta,"for time",time_step)
                print("Maximum d_theta", max_d_theta,"for time",time_step)
                print("Choosen d_theta", noised_d_theta[index_max],"for time", time_step, "\n")

```

```python
                print("##################################################\n")

            pose_angle_old_SLAM = np.copy(pose_angle)

            # transfer to centroid of lidar, account for lidar being off center with encoder, converted
            x0_lidar_float = x0_encoder_float + lidar_lever_len*np.cos(pose_angle)

            # transfer to centroid of lidar, account for lidar being off center with encoder, converted
            y0_lidar_float = y0_encoder_float + lidar_lever_len*np.sin(pose_angle)

            # convert all origin and end point to int variables
            x0_lidar_int = int(np.around(x0_lidar_float))
            y0_lidar_int = int(np.around(y0_lidar_float))

        global_transform_vec = np.array([[np.cos(pose_angle)],[np.sin(pose_angle)]])

#          transform lidar to global frame for every reading, in meters
        lidar_global_hit_locs = [np.dot(lidar_global_hit_locs[i], global_transform_vec) for i in range(meas_per_tstep)]

        # converts to array and squeeze extra dimension
        lidar_global_hit_locs = np.asarray(lidar_global_hit_locs)
        lidar_global_hit_locs = np.squeeze(lidar_global_hit_locs)

        for reading in range(len(lidar_global_hit_locs)):
            if reading % downsampling_par !=0                  or lidar_data[time_step]['scan'][reading] < robot_self_dist and (abs(180/np.pi*lidar_angle_vec[reading])> angle_reducer):
                continue

            # shift by encoder location AND convert to smaller unit
            x1_lidar_float = np.squeeze(lidar_global_hit_locs[reading,0]) + x0_encoder_float

#             print(x1_lidar_float,'x1_lidar_float')
            y1_lidar_float = np.squeeze(lidar_global_hit_locs[reading,1]) + y0_encoder_float

            # convert all end point to int variables
            x1_lidar_int = int(np.around(x1_lidar_float))
            y1_lidar_int = int(np.around(y1_lidar_float))

            ''' THIS IS DONE WITHIN SAME TIME STEP --> Origin has NOT CHANGED'''
            if x1_lidar_int == x1_lidar_int_old and y1_lidar_int == y1_lidar_int_old:
                continue

            x1_lidar_int_old = np.copy(x1_lidar_int)
            y1_lidar_int_old = np.copy(y1_lidar_int)
            hit_list.append([x1_lidar_int_old,y1_lidar_int_old])

            # generate list of impacted points
            '''Note that we use a (y0,x0,y1,x1 convention because of the way map_array is indexed (rows first))'''
            points_impacted = list(bresenham(y0_lidar_int,x0_lidar_int, y1_lidar_int,x1_lidar_int ))
            map_array = update_map(points_impacted,map_array)
```

```python

        if allow_print:
            particle_plot_cnt += 1

            plt.figure(figsize=(15,15))
            plt.imshow(map_array, origin="lower")
            plt.arrow(x0_encoder_float, y0_encoder_float, map_width*conv_factor/20*np.cos(pose_angle), map_width*conv_factor/20*np.sin(pose_angle), length_includes_head=True,    head_width=10, head_length=5)
            hit_vec_verify = np.asarray(hit_list)
            plt.scatter(hit_vec_verify[:,0],hit_vec_verify[:,1],c='r',alpha=1,s=1)
            plt.scatter(x0_lidar_int, y0_lidar_int,c='k',alpha=1)
            plt.show()
            print("TIME STEP FOR PLOT PARTICLE ABOVE:",time_step)


        if time_step % 40 == 0:
            plot_array_list.append([map_array,[x0_encoder_float,y0_encoder_float,pose_angle,time_step]])

    map_plot_list.append(plot_array_list)
    plt.close()

    all_map_list.append(np.copy(map_array))
    print("Total Execution Time: {0}".format(time.time()-start) )


    fig = plt.figure(figsize=(15,15))
    map_copy = np.copy(map_array)

    extent=(-map_width/2,map_width/2,-map_width/2,map_width/2)
    plt.imshow(map_copy,origin = 'lower', extent = extent,cmap='binary')
    plt.xlabel('Robot Horizontal Location (meters)', fontsize=30)
    plt.ylabel('Robot Vertical Location (meters)', fontsize=30)
    plt.xticks(fontsize= 24)
    plt.yticks(fontsize= 24)

    fig.suptitle('SLAM Occupancy Grid for Map {0}'.format(map_list_train[idx].id), fontsize=40)
#     fig.tight_layout()
#     fig.subplots_adjust(top=0.95)


    fig.savefig('maps/WORKS_map{0}_lidar.png'.format(map_list_train[idx].id,start))


# In[ ]:


top = 50
bottom = 40
left = 30
right = 50

ann_list = []

def update(i):

    for j, a in enumerate(ann_list):
        a.remove()
```

```
    ann_list[:] = []

    print(str(i/len(map_current)*100)[:5],"% Complete")

    im_normed = map_current[i][0]
    quiver_pars = map_current[i][1][:2]
    angle =  map_current[i][1][2]
    ax.imshow(im_normed, origin= "lower",extent=extent)
    t = map_current[i][1][3]


    if i == 0:
        rect = matplotlib.patches.Rectangle((right, bottom), left-right,top-bottom, angle=0.0,color=(1,1,1),)
        ax.add_patch(rect)

        texting = plt.annotate('T = '+str(int(t/40))+" sec",(.5*(left+right), .5*(top+bottom)),             ha='center',va='center', fontsize=14, color='red')

        ann_list.append(texting)


        ax.scatter((quiver_pars[0]-x_center)/conv_factor,(quiver_pars[1]-y_center)/conv_factor,c='r',s=5)
        return ax

    if i > 0:
        rect = matplotlib.patches.Rectangle((right, bottom), left-right,top-bottom, angle=0.0,color=(1,1,1),zorder = 3+i)
        ax.add_patch(rect)

        texting = plt.annotate('T = '+str(int(t/40))+" sec",(.5*(left+right), .5*(top+bottom)),             ha='center',va='center', fontsize=14, color='red',zorder = 4+i)

        ann_list.append(texting)


        ax.scatter((quiver_pars[0]-x_center)/conv_factor,(quiver_pars[1]-y_center)/conv_factor,c='r',s=5)
        return ax

for index, map_current in enumerate(map_plot_list):
    extent=(-map_width/2,map_width/2,-map_width/2,map_width/2)
    fig, ax = plt.subplots(figsize=(15, 15))
    ax.set_title('Robot Location Versus Time for Map '+str(map_list_train[index].id),fontsize=40)
    ax.set_xlabel('Robot Horizontal Location (meters)', fontsize=30)
    ax.set_ylabel('Robot Vertical Location (meters)', fontsize=30)
    plt.tick_params(axis='both', which='major', labelsize=24)

    print("Figure",map_list_train[index].id)
    anim = FuncAnimation(fig, update, frames=np.arange(0, len(map_current)), interval=300,repeat=True)
    anim.save('WORKS_gifs/map'+str(map_list_train[index].id)+".gif", dpi=80, writer='imagemagick')

    plt.close()
    print("DONE!")


# In[ ]:


```

```
# !tar chvfz notebook.tar.gz 'maps'
```