Comp 141 Probabilistic Robotics Homework 2: Image-Based Particle Filter for Drone Localization

Daniel Choate

To repreoduce the following results, run <`main.py`> with python 3. Requires <`HW2_utils.py`>, as well as numpy, openCV, and matplotlib libraries. To change the reference image shape, change variable 'm' (line 62 in <`main.py`>). To get an average of 10 trials, change variable 'trials' to 10 (line 58 in <`main.py`>) Beware of increased run time.

1 Simulation Environment

A simulation environment was created to enable a drone to randomly move through a known aerial map. The drone is assumed to always be oriented in the North direction, with no rotation considered.

Pixel to Unit Conversion: The origin of the map (0,0) is at the center of the image, with 1 unit of distance = 50 pixels. When a known map is input to the simulation, the range in each direction is automatically calculated. For example, when 'BayMap.png' is input into the simulation, the range is calculated to be:

$(x, y) = (12, 9)$ units in each direction

DFOV = 50 pixels: 1/2 of the reference image size is subtracted from each side to represent the drone field of view (DFOV).

Drone Starting Position: The drone's starting position x, y is randomly generated according to the uniform distribution. An example of a random position is shown in Figure 1, where $X = \begin{bmatrix} -9 \\ 0 \end{bmatrix}$ units, or $X = \begin{bmatrix} 198 \\ 450 \end{bmatrix}$ pixels.

Reference RGB Image: Based on the initial state location of the drone, a reference image is generated. The chosen size of the reference image is $m = 100x100$ pixels. In section 3, we experiment with different sized reference images to test the accuracy of our particle filter. This simulation can generate a reference image for any particular x, y position.

Random Movement: At each timestep, the simulator generates a random movement vector in the x and y direction, labeled as dx and dy such that $dx^2 + dy^2 = 1.0$. It is important to note that any movement vector (dx, dy)

Figure 1: Bay map image with drone location randomized, shown by a red circle

which moves the agent off the map is rejected, and a new movement is generated. The actual position of the drone $x_{t+1} = x_t + dx + \mathcal{N}(0, \sigma^2_{movement})$. Similarly, $y_{t+1} = y_t + dy + \mathcal{N}(0, \sigma^2_{movement})$. In this simulation, $\sigma^2_{movement} = 5$ (pixels). The movement is done in line 45 of the <`HW2_utils.py`> file. To advance time steps of the simulation, the user will press enter (and close any previously generated plots).

2 Particle Filter Implementation

This section will describe the implementation of the particle filter algorithm for image-based localization in our drone simulation.

Generate a set of particles: Initially, a set $\mathcal{P}$ of N particles is uniformly distributed across the entire map. The number of particles chosen for this simulation is $N = 10,000$ particles.

Assigning particle weights: The next step in the particle filter process is to assign a weight to each particle randomly distributed throughout the scene. This process consists of three steps.

1. Generate a mxm image for each particle p, where $m = 100$ pixels (for now)
2. Compare the reference image shown in Figure 2 with the newly generated picture for particle p
3. Assign a weight to each particle based on the image comparison

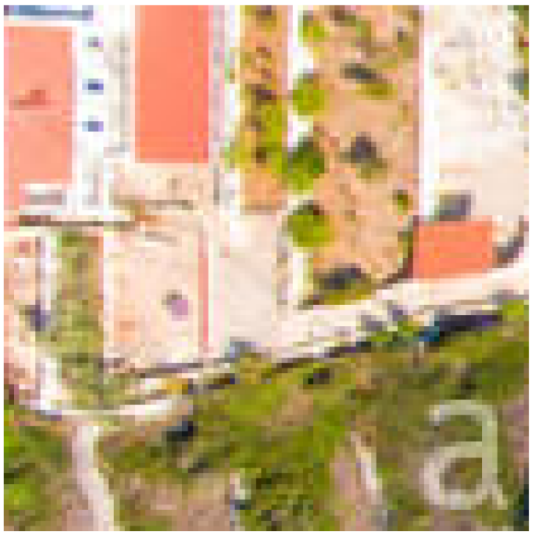

Figure 2: 100x100 pixel reference image at location $(x, y) = (-9, 0)$

For step two, an image comparison strategy is necessary for determining whether a specific particle is close to the actual drone location. For this investigation, a histogram of each image is calculated using openCV's 'calcHist' function [1], listed in line 136 of the <`HW2_utils.py`> file. The histograms analyze the distribution of pixel RGB values in each image, producing a count of pixel colors for a specific image.

Following the histogram calculation of the particle reference image, it is compared to the reference image using openCV's 'compareHist' function [1], listed in line 146 of the <`HW2_utils.py`> file. This function takes two histograms, along with a comparison method which determines how the similarity is calculated. The method used in this simulation is a correlation comparison (openCV's 'HISTCMP_CORREL' function).

Thus, a similarity score is returned for each particle. A higher similarity results in a higher score, resulting in a higher particle weight. Following the similarity calculation, each similarity w is squared, w^2, so as to emphasize the difference between particles with large differences in similarity scores. This is done in line 117 of the <`main.py`> file.

$$w_{\text{norm}} = \frac{w}{\sum w} \tag{1}$$

$$w_{\text{scaled}} = \frac{w_{\text{norm}}}{\max(w_{\text{norm}})} \tag{2}$$

After calculating a weight for each particle, the resulting weights are then normalized and scaled. This is done by first dividing each weight by the sum of all weights, then dividing the normalized weights by the maximum of all

Figure 3: Initial random particle distribution, with equal weights assigned.

normalized particle weights. This is done in lines 121-125 of the <`main.py`> file, also shown in equations 1 and 2.

Algorithm Low_variance_sampler($\mathcal{X}_t, \mathcal{W}_t$)**:**

$\bar{\mathcal{X}}_t = \emptyset$
$r = \text{rand}(0; M^{-1})$
$c = w_t^{[1]}$
$i = 1$
for $m = 1$ to M do
  $u = r + (m-1) \cdot M^{-1}$
  while $u > c$
    $i = i + 1$
    $c = c + w_t^{[i]}$
  endwhile
  add $x_t^{[i]}$ to $\bar{\mathcal{X}}_t$
endfor
return $\bar{\mathcal{X}}_t$

Figure 4: Pseudo-code for low variance sampler [2]

Resampling: Following the assignment of scaled weights to each particle, a resampling process occurs, to generate a resampled set $\mathcal{P}'$. To accomplish this, a low variance sampler is used, as described in *Probablistic Robotics*, by Thrun, Burgard, and Fox [2]. The pseudo-code is shown in Figure 4.

The low variance sampler works by first calculating the cumulative weights from the weighted particles. Next, it generates a uniform random sample from the interval defined by these cumulative weights. Then, particles are iteratively selected based on the random sample, forming a set of new particles. This ensures that the higher-weight particles are more likely to be selected while

maintaining the overall diversity. This is implemented starting in line 84 of the <`HW2_utils.py`> file.

The purpose of a low variance sampler is to select particles according to their weight, aiming to focus on more likely particles, eliminating less likely ones. The low variance sampler also helps mitigate sample depletion by ensuring that the resampling process retains diversity among the particles, while also being computationally efficient.

Particle Movement: Following the resampling process done at each timestep, each particle is moved according to the known movement vector dx, dy randomly generated, such that $dx^2+dy^2 = 1.0$. However, it is known that the drone moves with noise $\sigma^2_{movement} = 5$. Similarly, the particles move with noise $\sigma^2_{p,move} = 3$. This assumes that the particle movement has slightly less noise than the drone movement. At each stage after movement, the particles are drawn on the map, with radii proportional to the weight of the particle (multiplied by 5 for scaling). A larger radius denotes a larger weighted particle. This is implemented in line 135 of the <`main.py`> file.

Particle Visualizations: The visualizations in Figures 5-10 show a full trial with a series of 20 iterations of the previously described particle filter on the <`BayMap.png` > picture. As stated in the previous section, the particle radius is proportionate to the weight of the particle. The true position of the drone is shown in a green cirlce. The parameters for the scene include:

$m = 100$ particles for the mxm reference images
$trials = 1$
$iterations = 20$
$\mathcal{P} = 10,000$ particles

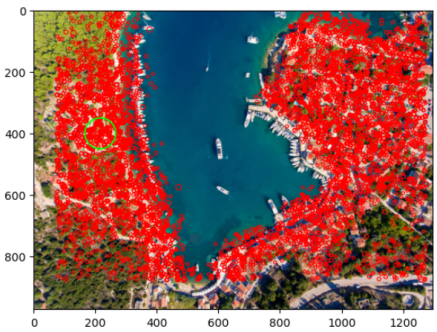

Figure 5: Iteration 1: particles, drone

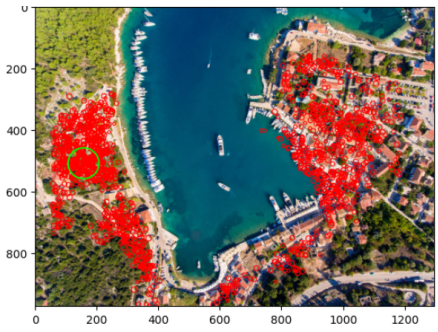

Figure 6: Iteration 5: particles, drone

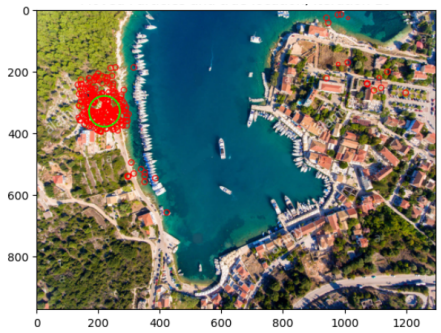

Figure 7: Iteration 10: particles, drone

Figure 8: Iteration 15: particles, drone

Figure 9: Iteration 20: particles, drone

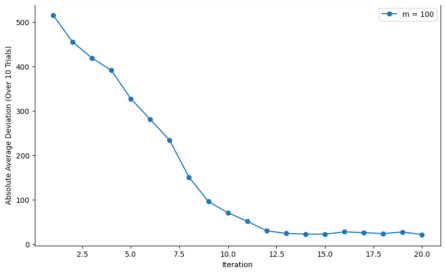

Figure 10: Variance vs. Iterations

3 Experiments and Evaluation

The following section explains an experiment in which an aspect of the particle filter was varied to test the effect on success.

Success Metric Following the convergence of our particles around the true location of the drone, the success of the filter is then determined. This is done by using the absolute average deviation metric, calculated as follows:

$$\text{absdev} = \sum_{i=1}^{N} \frac{1}{N} \sqrt{(x_i - x_{\text{drone}})^2 + (y_i - y_{\text{drone}})^2}$$

Where N is the total number of particles, $(x_{\text{drone}}, y_{\text{drone}})$ is the true position of the drone, and (x_i, y_i) is the location of each particle. This is implemented in line 151 of the <`HW2_utils.py`> file. As seen from Figure 10, our particle filter algorithm starts with an extremely high absolute average deviation, but begins to reach convergence at around iteration 12.

Experimental Conditions During experimentation, the size of the refer-

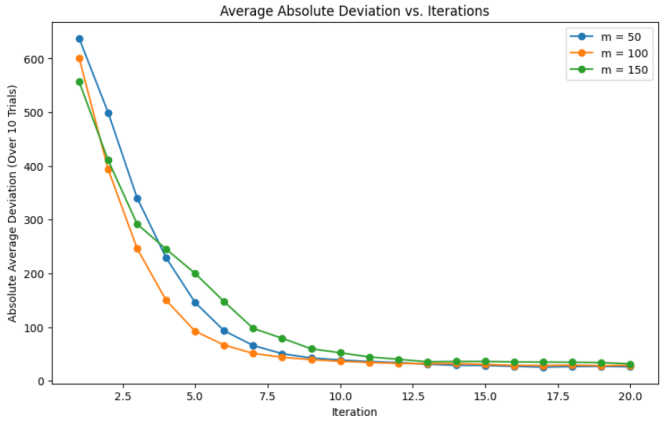

Figure 11: Absolute Average Deviation Comparison (Avg across 10 trials).

ence image was varied to test the effect on the absolute average deviation at each iteration. Our reference image was change from $m = 100x100$ pixels, to $50x50$, as well as $150x150$. For each test case, 10 trials were run for 20 iterations, and an average was taken across all 10 trials. To run these experiments, change the 'trials' variable to 10 in line 58 of the <`main.py`> file, as well as including 50 and 150 in line 62 of <`main.py`>. These values can be changed to meet any user preference. However, as trials and test cases increase, runtime significantly increases.

Figure 11 shows the results of the absolute average deviation across 10 trials for each mxm reference frame size [50,100,150]. As expected the initial iteration shows that on average, a smaller reference image results in a higher initial deviation, where as a larger image results in a smaller initial deviation. However, all three reference image sizes converge to roughly the same deviation. Using an image size of $m = 100$ pixels shows to converge quicker than both 50 pixels and 150 pixels. Using a larger image size like 150x150 may result in a smaller deviation initially, but converges slower than if a smaller reference image is used.

4 Experiments and Evaluation

[1] OpenCV. Open Source Computer Vision Library, 2024, opencv.org/.

[2] Thrun, Burgard, Fox, *Probabilistic Robotics*. The MIT Press, 2005.