Smart Phone Sensing - Final Report

Ingimundur Vilhjálmsson Jón Arnar Briem

Group id: 1 Group name: 5885522
June 20, 2019

Testing for Bayesian Localization was performed on CAT S41, running Android 8.0.0. Testing for Particle Filters was performed on a Samsung Galaxy A7 (2018), running Android 9. Code loosely based on examples provided by the TA, all other code is our own. Only makes use of standard Android and Java libraries

1 Bayesian Localization

1.1 Training Data Collection

Rather than collecting all training data on a single day we chose to collect data over multiple days in order to minimize the effect of changes in environment. As such we performed multiple small measurements, scanning each cell 4 times (facing in different directions each time), and then writing the BSSID, RSS and Cell to a CSV file. In total we made 7 scanning trips for a total of 28 scans per cell. The scans were performed on working days in the period of May 17th to June 6th roughly evenly spread over working hours. A noticeable improvement in accuracy after the addition of every new set of training data was observed, but this effect diminished as the amount of training data grew.

1.2 Data Processing

After collecting the training data, as described in the previous section, we processed them offline using Python. We assumed that WiFi signal strength follows a Gaussian distribution, allowing us to compute and use the mean and standard deviation of the RSS in each cell for every BSSID. In cases where we did not have enough samples, that is fewer than 5, to accurately estimate the mean and standard deviation we assumed a mean of -60 dBm and standard deviation of 100 dBm, this assures that the effect of missing data is limited. Since data was collected over multiple days this was enough to eliminate the effects of temporary changes such as hotspots and no further filtering was needed.

1.3 Results

When the user asks to locate himself a wifi scan is performed (see below) and, starting with an initial belief, a new posterior is calculated based on the signal strength of each `ScanResult` in a serial manner. Once every `ScanResult` has been processed in this manner the location is determined to be the most likely one.

1.3.1 WifiManager Scanning Issue

In our initial implementation of Bayes, we ran into an issue with getting old scanning data after invoking `startScan()`. It seems that the buffer containing the `ScanResult`s is infrequently and not regularly updated. This caused our implementation to randomly use outdated `ScanResult`s resulting in an inaccurate location estimate.

To avoid this issue we use an intent filter based on `SCAN_RESULTS_AVAILABLE_ACTION`, allowing for `getScanResults()` to be used after the scan has completed. This eliminated the unreliability of the scan but consequently added a wait time which can up to 2 seconds in some cases.

2 Particle Filter

2.1 Motion Model

For the particle filter to work properly, the users movements need to be monitored so that the particles will move relative to the users movement. At first we tested out the `STEP_DETECTOR` sensor, but it was very apparent that it was not able to record single steps at a time, for it has a long delay. This led us to develop our own step detector based on the accelerometer sensor.

The formula used to derive the acceleration is described as follows:

$$a = \sqrt{a_x^2 + a_y^2 + a_z^2} \quad (1)$$

where a is the net acceleration magnitude, and a_x, a_y, a_z are obtained from the corresponding axes of the accelerometer. We noticed that while walking a spike to approximately $a = 12.2$ would occur with every step and in between steps, a drops below 9.9, signifying that the step is complete. Thus our step detector would simply register a step when $a > 12.2$, then a flag is put to disallow the registering of another step until $a < 9.9$. This proved to be very accurate, responsive, and lightweight. However, the downside to it is that it is prone to disturbances such as sudden shaking of the phone.

Based on measurements of our own steps we fixed the length of a step at 0.7 meters but to account for possible differences in step length we also added an error margin of 5%.

Along with the step detection mechanism, we made use of the inbuilt `ROTATION_VECTOR` sensor to complete our motion model. The rotation vector is used to make a general alignment for the orientation/rotation of each particle with the phone, allowing them to move in the same direction as the user.

One issue that we discovered was that this sensor is susceptible to interference from magnetic fields. Two regions were found to have a noticeable effect on it; by the elevators and near the fire door (at cell 8). In other words, when facing the phone in the same direction in every cell, different azimuth values were sensed. We made several readings from each cell and found that the errors within in a cell were $\pm 20°$ on average while the differences between cells were up to $50°$. To compensate for the difference between cells we measured the offset for each cell and then apply that to any particle in that cell before rotating it, furthermore we randomly generate an error in the range $\pm 20°$ for the rotation of each particle.

To ensure that the app runs consistently smooth we use a loop which iterates every 500 ms. This loop takes the steps detected since last iteration and the current direction and updates the location of each particle accordingly. This loop also takes care of tasks such as movement constraint violation detection, respawning dead particles and convergence detection.

2.2 Map Generation

We first define a variable called `decimeter` by dividing the width of the screen by 400, the total distance of the hallway in decimeters (we assume that the width of the screen in landscape mode is the limiting factor, not the height), this enables us to consider every length in the map as a whole number avoiding any awkward `int` conversions. The `Room`, `Wall` and `Particle` classes are implemented as extensions of the `ShapeDrawable` class allowing for easy drawing.

2.3 Particle Deployment

On initialization the particles are created randomly within the map area. This is done using Java's `Math.Random()` class to generate pseudo-random numbers. The same instance of the `Math.Random()` class is used for all random numbers, including the ones used to add noise to the movement model, this is done to avoid multiple random numbers being generated using the same seed.

2.4 Movement Constraint Violation Detection

In order to detect movement constraint violations our `Rooms)` are implemented as a network, where every `Rooms)` is a node with edges to all directly reachable neighbors, including itself. Any move resulting in a room which is not a neighbor, or no room at all, is considered a violation of the movement constraints and that particle is "killed". After that it is respawned with the same location and direction as a randomly selecting surviving particle.

2.5 Convergence and Accuracy

When more than a certain percentage of particles are in the same cell we consider convergence to be reached. Currently this percentage is set at 50%. The particle filter is functional and works well. However, its accuracy is dependent on the user taking consistently long steps and holding the phone in a proper manner.

2.6 Novelties

To improve on the basic groundwork we implemented two novelties to enhance the performance of the program.

2.6.1 Convergence Assistance The inner workings of particle filters are not understood by the general public and the average user might therefore be unaware of how to reach convergence from the initial belief. In order to assist users

and keep them from walking in circles or some other non-unique patterns we developed a simple Convergence Assistant. When convergence has not been reached the Convergence Assistant keeps track of, for every particle, every direction which will kill the particle and then instructs the user to walk in the direction that will kill the most particles.

2.6.2 Parallelization Because thousands of independent particles are being dealt with, it would make sense to introduce parallelism in compute-intense regions. Moreover, since the particle filter operates in a real-time fashion, attaining higher performance is crucial.

We used Java's `parallelStream` to perform the rotation and movement of each particle in parallel, this process is responsible for roughly 60% of the runtime for each iteration. Implementing this for 10.000 particles running on 8 cores did however not result in a speedup but rather an increase in runtime of just over 75%. This is most likely due to the fact that each thread used the same instance of `Java.Util.Random` which is necessary to avoid the same seed being used more than once. Running the same implementation for 100.000 particles did show some improvement, but nonetheless having so many particles was too computationally heavy for the phone to be feasible with or without parallelization so we discarded the idea of parallelization.

3 Workload

Every task did receive some attention or input from both members but in this table we try to emphasize who did the majority of the work. Double X's denote that a lot of effort was put into that task.

	Jón	Ingimundur
Bayes Data Collection	X	
Bayes Data Processing		X
Bayes Calculating Result	X	X
PF Map Generation	X	
PF Motion Model		X
PF Movement Violations	X	
PF Convergence Assist	X	
PF Parallelization		X
PF fine-tuning	XX	XX

4 Attendance & Challenges

The biggest challenge was probably fine-tuning and debugging, both Bayesian and PFs are quite complicated and a lot of different variables influence the final outcome. Therefore it was difficult to determine the reason for unsatisfactory results and which parts to improve. For example, we had issues with particles falling behind as we walked along the hallway but when we increased the step size it became too long for shorter walks. Then we noticed that the particles were not moving in a straight line as we were but rather zig-zagged behind us taking a longer route. We tried fixing this by changing our error margin for the direction but that did not work either. In the end we determined that this was due to the different directional offset in each zone which lead this zig-zag route by the particles. So the eventual solution was to introduce a different directional offset for each zone. Both of us attended all lectures and all labs.

5 Possible Additions & Improvements

The Convergence Assistant provides plenty of opportunities for further development, including improving understanding of how users interpret "Turn Right", does that mean immediately rotate 90° and move forward or entering the next room on your right?

Another possible improvement is to use RSS data to influence the initial belief in PF. Even if different phones detect different strength of signals and temporary changes can have a big impact on the strength detected, surely the signal detected should give some indication of where the user is located.

6 Screenshots

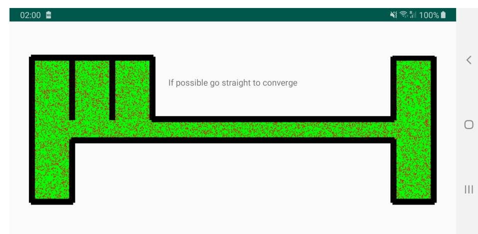

Figure 1: Initial State of the Particle Filter

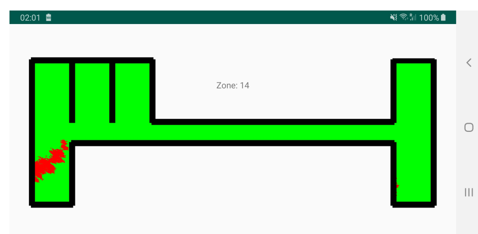

Figure 2: Converged State of the Particle Filter

Confusion Matrix for Bayesian

		App															
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Actual	1	1.0															
	2	0.3	0.6		0.1												
	3			0.9	0.1												
	4	0.1			0.8	0.1											
	5					0.7	0.3										
	6						1.0										
	7							0.7									
	8							0.2	0.5	0.3							
	9									0.8	0.2						
	10										1.0						
	11											1.0					
	12										0.2		0.8				
	13												0.2	0.5		0.1	0.2
	14														1.0		
	15												0.4			0.5	0.1
	16																1.0