Indoor Localization | CS6650 Report

Created	@May 18, 2021 7:16 PM
Status	
Tag	

Exploring Indoor Localization with Sensor Fusion of PDR and a Single RTT Wi-Fi with known Initial Position

Team Members

Sai Rohitth Chiluka:

Contribution:

- Kalman Filter Algorithm
- RTT Range Simulation
- Single RTT Localization
- PDR

- Bias and Variance Estimation for step length and orientation
- Walking Simulation on Circle and Line
- Post Analysis and plotting localized points
- All Python related scripts
- Report

Annie Marandi

Contributions:

- Accelerometer, Magnetometer and Gryoscope Data on Android
- Complimentary Filter for Orientation
- Step Detection
- Step Length Calculation
- Button for activating RTT
- UI Design Components (Plotting Groundtruth, AP, Localizated Points)
- PPT Slides

Introduction

This is a project on Indoor Localization without using trilaterization and using as less dependencies as possible (Range information from 1 AP and 1 IMU). Since a single range equation would give many solutions the location of the starting point is assumed as known. This can be a valid assumption in many cases (see application below). The position of the AP cannot be chosen at random and is discussed below for certain cases.

Application Examples

Supermarket Cart Location Analysis

The problem is to track the Carts in supermarkets so that we can gather information about what consumers are doing the data obtained can then be used in Data Analysis and Machine Learning Models to gather useful information.

In this case, the initial position can be taken as the entrance of the shop where all the carts are located. Some management of Carts has to be done to make it

work. 1 RTT can be used for medium sized shops depending on its range.

Localization for the Visually Impaired in Public Places

Here the problem is to track the location of Visually Impaired person, so that directions can be given to them using a speech assistant when they find thems in a new location. The base implementation would be the same as above where the entrance to a mall for example is taken as the initial point. Some additional infrastructure has to be added for starting the localization or exisiting infrastructure like metal detectors, cloth tag detectors can be modified for the purpose.

Getting IMU Data

We have used Androids TYPE_LINEAR_ACCELERATION to get acceleration data, a low pass filter was then applied to get better readings.

We first tried double integrating this data to get displacement information but that did not work well as a lot of errors got accumulated which led to a lot of drift maybe because the sampling frequency was not fast enough to minimize the effect of errors while integration, especially for a faster fluctuating signal like walking. If the dynamics was "*slower*", like constant acceleration for significant time then the errors would not be that bad. I think an independent accelerometer with faster sampling rate would have performed better for walking.

We then went forward with using step detection to find displacement. This is called PDR or **Pedestrian Dead Reckoning**.

Step Length

We used the accelerometer reading to detect a_{max} and a_{min} which is used to find step length using the formula shown below.

$$L_{step} = K_{vel} \times \sqrt[4]{A_{max} - A_{min}}$$

To estimate K we used a formula found from linear regression analysis shown below.

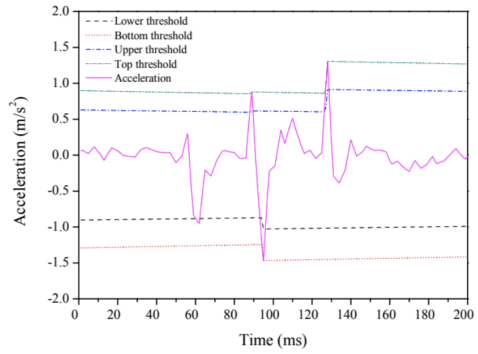
$$K_{vel} = 0.68 - 0.37 \times \bar{v}_{step} + 0.15 \times \bar{v}_{step}^2$$

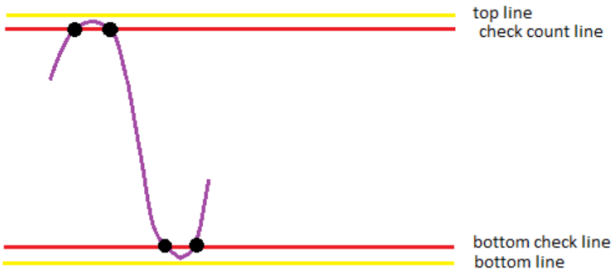
$$\bar{v}_{step} = \sqrt{\bar{v}_{stepX}^2 + \bar{v}_{stepY}^2 + \bar{v}_{stepZ}^2}$$

$$\bar{v}_{stepX} = \text{mean}\left(\int a_{r\_stepX}(t)\,dt\right)$$
$$\bar{v}_{stepY} = \text{mean}\left(\int a_{r\_stepY}(t)\,dt\right)$$
$$\bar{v}_{stepZ} = \text{mean}\left(\int a_{r\_stepZ}(t)\,dt\right)$$

Step Detection

To detect the steps itself, an implementation of a step detection liked below was used. This checks for various thresholds to see if it is actually a step.





Pedestrian Dead Reckoning

PDR was integrated into the Kalman Filter Algorithm in the Prediction Step. It was implemented as shown below.

$$X_t = X_{t-1} + L_t \begin{bmatrix} sin(\theta_t) \\ cos(\theta t) \end{bmatrix}$$

Complimentary Filter

For getting orientation for the above to work, we used 2 sources to get a good result and combined it using a **Complimentary Filter** as shown below.

$$angle = 0.98 * (angle + gyrData * dt) + 0.02 * (accData)$$

- First one by integrating gyroscope data to get orientation w.r.t. initial orientation.
- Second by using a combination of magnetometer and accelerometer to get azimuth which is then referenceded to the same initial orientation.
- 2% of magnetometer-accelerometer orientation is used to reduce drift and get correct orientation in the short term.

RTT Range Simulation

RTT stands for Round Trip Time which the system then uses to calculate range by multipling with time. This is designed such that only the client has knowledge of the range and not the RTT access point respecting privacy. RTT Wi-Fi gives an accurate results of within **1 meter** which is better that many other indoor positioning options like RSSI, etc. This is not mainstream yet so we don't have one to test it out, hence, we'll be simulating one.

gen_rtt_line(len, step) and **gen_rtt_circle(radius, step_deg)** in **rtt.py** generates the ranges on a line and circle respectively. The access point was taken to be at **(5,0)**.

For the line a total length of **15m** was taken for the simulation and every **3rd** step(**3m**) was taken as a measurement update point for the kalman filter. Step length was assumed as 1m for simplicity.

For the circle, a radius of **5m** was chosen, the circle was not added into the app as a large area would be needed to test it. Each step was assumed to rotate the heading direction by **20** degrees, hence **1.73m** would be the step length. Not practical but simplifies simulation. Every 3rd step which is **60 deg** would be the measure update point.

> After calculating range at each update point from the AP, a gaussian noise was added with **mean 0** and **standard deviation 0.5** which means that 95% of the time the error would lie between $+2 * sigma$ and $-2 * sigma$

Effect of Position of AP

Position of anchor point is important as it affects localization and increases errors. A similar effect of GDoP in trilateration happens depending on where the AP is located.

Example, for the circle, if the AP is close to the center then the points to the farther end have closer ranges hence added noise creates similar ranges meaning harder to pinpoint where on the circle the node is. By experimenting with different AP locations it was found that having the AP alteast radius distance away from the center, i.e., on circumference gives the best results. It doesn't affect much if it is moved further away.

For the line it is important to position the AP at any point on the perdendicular line on either ends of this line. Else there would be 2 potential points where the node could be which could have easily been avoided.

Effect of Radius of Circle

It is found that for a fixed AP location (on circumference), having a bigger circle gives more distance between the ranges (less GDoP) as each step is **20 deg** meaning the distance between coordinates of the current and the previous point keeps increasing as radius increases.

Hence range of Wi-Fi AP should be taken into consideration.

Effect of Update Frequency

Measurement Update frequency implies how often is the measurement update taken in the Kalman Filter. For a fixed AP location and radius of circle, it is found that **updating less often may be better!** This is because if updated frequently the value of the ranges would be numerically close (more GDoP) hence giving bad localization.

This holds for the line as well as updating every step may produce ranges which are numerically close.

It should also be noted that if update frequency is very slow then the PDR reading would drift and cause bad reading on the whole. Hence, a sweet spot has to be found.

Single RTT Localization

For the line the problem would seem easy because we positioned our AP in such a way. If AP was near the mid-point level of the line there there would have been 2 possible solutions. Hence only range is not enough.

For the circle the problem is evident, anywhere you position the AP there are always 2 potential solutions except when placed at the center of the circle, then there are **infinite solutions!**

Hence, we used the position given by PDR to estimate localization by **projecting the point onto the circle, i.e., finding the closest point on the circle to the PDR estimate**. This can be done by the below formula, it is basically scalling when looked at in polar form:

$$x = x_a + (x_p - x_a) * r/d$$

$$y = y_a + (y_{pdr} - y_a) * r/d$$

Downside: The downside to this is that is would give bad results if the PDR estimate itself is very bad. If the drift overpowers then we get bad answers. This would happen if the distance travelled is long. A possible solution to this to use landmarks to reset to actual location or use trilaterization (assumin a large area would use more than 1 RTT).

This is implemented in **read_rtt()**

Kalman Filter

A Kalman Filter is a Bayesian Filter which uses probability distributions to correct the Mean and Variances of the data being estimated. The Kalman Filter assumes that the errors are distributed Normally. The Kalman filter also assumes that all the models are **linear**. A standard implementation of the Kalman Filter looks something like one shown the below:

$$\textbf{Algorithm Kalman\_filter}(\mu_{t-1}, \Sigma_{t-1}, u_t, z_t)\textbf{:}$$

$$\bar{\mu}_t = A_t\ \mu_{t-1} + B_t\ u_t$$
$$\bar{\Sigma}_t = A_t\ \Sigma_{t-1}\ A_t^T + R_t$$

$$K_t = \bar{\Sigma}_t\ C_t^T (C_t\ \bar{\Sigma}_t\ C_t^T + Q_t)^{-1}$$
$$\mu_t = \bar{\mu}_t + K_t(z_t - C_t\ \bar{\mu}_t)$$
$$\Sigma_t = (I - K_t\ C_t)\ \bar{\Sigma}_t$$
$$\textit{return}\ \mu_t, \Sigma_t$$

It has 2 steps, the prediction step and the measurement update step.

The prediction step is taken as the PDR, here we have also used theta as one of the variables even though theta never gets updated and hence the covariance keeps increasing. This is done to **make sure the model stays linear** and the Kalman filter can be applied.

The correction step or the measurement update step is what calls **read_rtt()** and updates the means and variances using kalman gain.

For details on the matrices used, refer to the comments in the file **kalman.py**

One Downside: Here it is not know if the measurement model is linear because of the point projection onto circle equation. Hence an improvement to this would be to use an **EKF (Extended Kalman Filter)** which is not implemented.

Achieving Different Update Frequencies

To achieve different update frequencies, 2 functions **pdr()** and **pdr_with_rtt()** are created which do exactly what they say by passing it step length and heading and getting back the **position belief** and **covariance belief**. Calling **pdr_with_rtt()** every 3rd time and **pdr()** otherwise does what we want.

For initial location, an **initial belief covariance** of **0.04** in x and y, and **0.1** in theta was assumed. This was initialized inside **kalman_init_line()** and **kalman_init_circle()**

For the app though, since we don't know if every step is detected, a **button** is made that calles pdr_with_rtt().

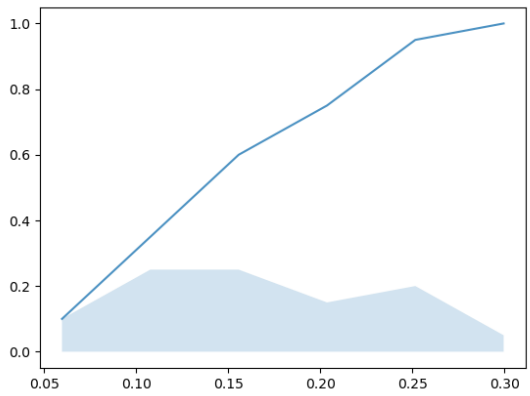
These can be found in **kalman.py**

Bias-Variance Estimation for Step Length and Orientation

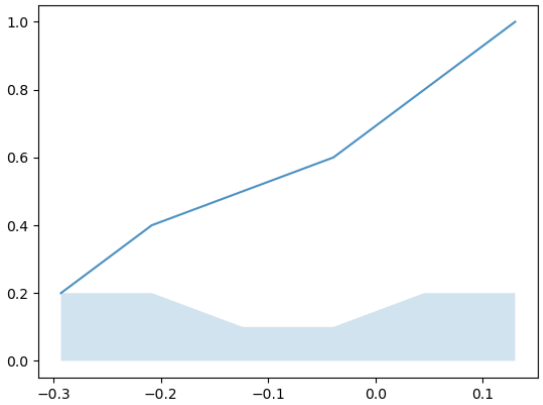
In each step covariance error matrices **R** and **Q** are used which are to be empirically determined. R is used for PDR(step length and orientation) errors and Q for RTT. Q is already known as we took it to be **0.5** (1 meter accuracy).

R was experimentally determined by:

- taking 20 readings of walking 5 steps and comparing it with actual measured length. Finally dividing by 5 to get error in each step. There are 2 aspects here error in step length and errors in step detection. To find errors for only step length, each of the 20 readings were divided by the number of steps detected and then calculating mean and variance of the errors.

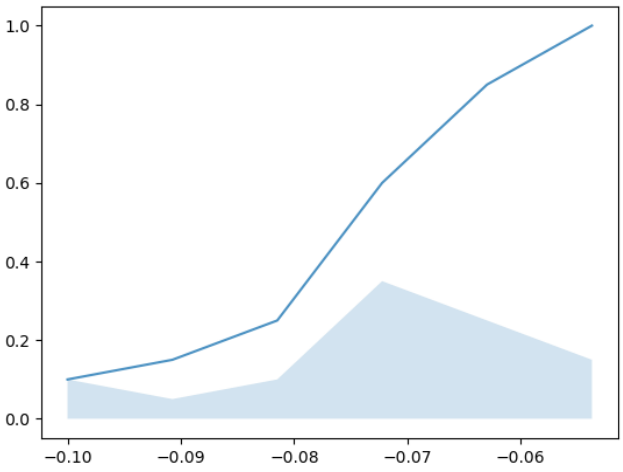


Errors in each step length — Almost Gaussian



Errors in step length and step detect combined

- taking 20 readings of turning the phone by 90 degrees and measuring orientation. Finding the variance and mean and then diving them by 90 which gives values per degree turned.



Almost Gaussian

The means found is taken as biases and subtracted off from the reading before giving it to the Kalman Filter, the variances are used in R.

This can be found in **localization_error.py** and **orientation_errors.py**.

Downsides: The PDF for step length and step detect is not exactly gaussian, hence not perfect for Kalman Filter. Even the step length errors look bi-modal. Maybe using a **Particle Filter** would help here as Gaussian is not assumed over there. But it could also mean the experiment was not conducted properly. Also

this was conducted for the Android step detection which was slow, but not against the one which we wrote which seems to be better.

Walking Simulation and Post Error Analysis

This is the part of the code that simulates walking (for loop one step at a time) and uses the functions discussed above just like any app or client would use. The line is simulated in **sim_line()** and the circle in **sim_circle()**.

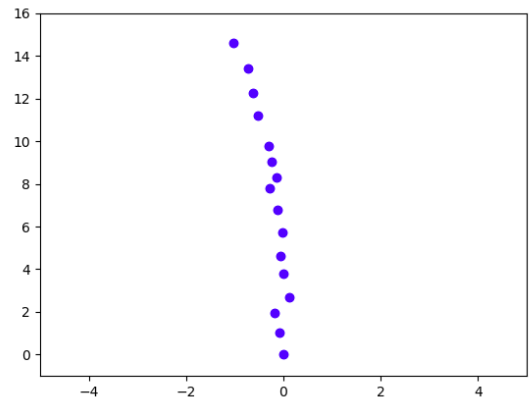
- For the line each step is taken to be **1m** and and every **3m** the RTT reading is also taken.
- For the circle each step is taken as **1.73m**, and a rotation of **20 degrees**. Every **60 degrees** the RTT is also taken into account.

For each of the above the biases and variances were appropriately added.

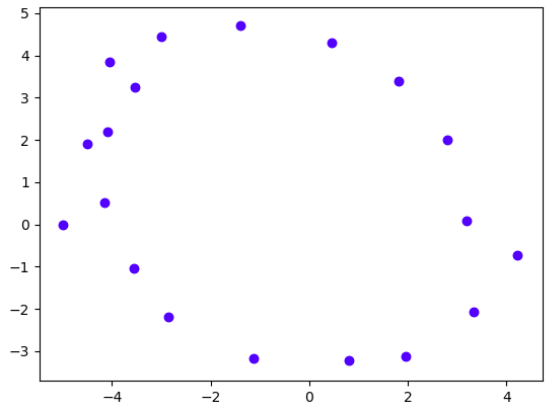
The localized points are then plotted as a scatter plot.

A sample output on the terminal screen would look something like this, this first tuple is **belief position** and second tuple is **belief covariance**:

In each case a **drift** is also simulated for the heading direction to check if the correction is working. Without proper drift the simulations seem to be working perfectly well without needing RTT which doesn't happen in real life.



Line Simulation



Circle Simulation

The output of every localization is stored in an array called ***results*** which is later used to find **mean error** and **max error** from the ground truth in the

functions **analyse_line()** and **analyse_circle()** in **localization_error.py**.

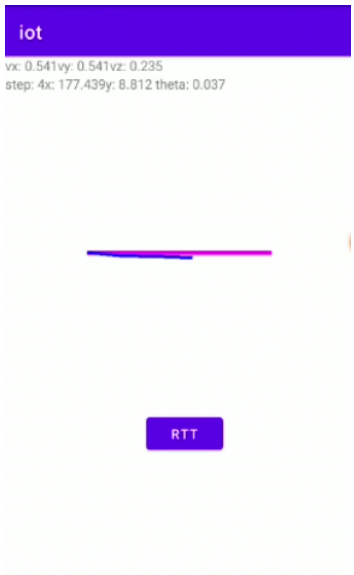
This can be seen by running either **simulate.py** or **localization_error.py**

Observation

- From the Line Simulation it can be seen that every 3 steps the drift is corrected untill the drift catches up at around **15m** this is as discussed in one of the downsides above. The projection of the drifted position is drifted. This can be reduced by adding landmarks.
- From the Circle Simulation, a sawtooth shape can be seen which is familiar in a sensor fusion like this. The drift caused due to rotation is shifted back.
- From the covariance data it can be seen that every time an RTT measuement is made, the covariance decreases which is expected off Kalman Filter.
- As noted before, error in theta is not updated or corrected, hence it keeps piling up which can be seen from the covariance data.
- The mean and max error data don't seem to give valuable insight at first glance when comparing with RTT and without RTT. Maybe analysing multiple of these simulations would give some insight.

App UI

- Button for activating RTT
- Button for stopping to Analyse data
- Graph plotting movement real-time (blue)
- Ground truth shown underneath (violet)
- Anchor Point Indication



App Testing

The app has only been tested for the case without RTT not much for the RTT case. It seems to be working nicely for few number of steps as can be seen in the video in the folder. Further testing and tweeking in necessary for getting in to work as expected.

Resources

- Step Detect: https://medium.com/@farissyariati/implement-a-simple-step-detection-algorithm-e6bfdcf8d669
- http://josejuansanchez.org/android-sensors-overview/gravity_and_linear_acceleration/README.html
- https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5038701/
- https://www.codeproject.com/Articles/729759/Android-Sensor-Fusion-Tutorial
- Probabilistic Robotics - Sebastian Thrun
- https://code-examples.net/en/q/49747