# Probabilistic Models For Mobile Phone Trajectory Estimation

by

Arvind Thiagarajan

S.M., Computer Science, Massachusetts Institute of Technology (2007)
B.Tech., Computer Science and Engineering, Indian Institute of Technology Madras (2005)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2011



Author . . . . . .
Department of Electrical Engineering and Computer Science
September 2nd, 2011

Certified by . . . . . .
Hari Balakrishnan
Professor of Computer Science and Engineering
Thesis Supervisor

Certified by . . . . . .
Samuel R. Madden
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by . . . . . .
Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

**Probabilistic Models For Mobile Phone Trajectory Estimation**

by

Arvind Thiagarajan

Submitted to the Department of Electrical Engineering and Computer Science
on September 2nd, 2011, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

## Abstract

This dissertation is concerned with the problem of determining the track or trajectory of a mobile device — for example, a sequence of road segments on an outdoor map, or a sequence of rooms visited inside a building — in an *energy-efficient* and *accurate* manner.

GPS, the dominant positioning technology today, has two major limitations. First, it consumes significant power on mobile phones, making it impractical for continuous monitoring. Second, it does not work indoors. This dissertation develops two ways to address these limitations: (a) sub-sampling GPS to save energy, and (b) using alternatives to GPS such as WiFi localization, cellular localization, and inertial sensing (with the accelerometer and gyroscope) that consume less energy and work indoors. The key challenge is to match a sequence of *infrequent* (from sub-sampling) and *inaccurate* (from WiFi, cellular or inertial sensing) position samples to an accurate output trajectory.

This dissertation presents three systems, all using probabilistic models, to accomplish this matching. The first, *VTrack*, uses Hidden Markov Models to match noisy or sparsely sampled geographic $(lat, lon)$ coordinates to a sequence of road segments on a map. We evaluate *VTrack* on 800 drive hours of GPS and WiFi localization data collected from 25 taxicabs in Boston. We find that *VTrack* tolerates significant noise and outages in location estimates, and saves energy, while providing accurate enough trajectories for applications like travel-time aware route planning.

*CTrack* improves on *VTrack* with a Markov Model that uses "soft" information in the form of raw WiFi or cellular signal strengths, rather than geographic coordinates. It also uses movement and turn "hints" from the accelerometer and compass to improve accuracy. We implement *CTrack* on Android phones, and evaluate it on cellular signal data from over 126 (1,074 miles) hours of driving data. *CTrack* can retrieve over 75% of a user's drive accurately on average, even from highly inaccurate (175 metres raw position error) GSM data.

*iTrack* uses a particle filter to combine inertial sensing data from the accelerometer and gyroscope with WiFi signals and accurately track a mobile phone indoors. *iTrack* has been implemented on the iPhone, and can track a user to within less than a metre when walking with the phone in the hand or pants pocket, over 5$\times$ more accurately than existing WiFi localization approaches. *iTrack* also requires very little manual effort for training, unlike existing localization systems that require a user to visit hundreds or thousands of locations in a building and mark them on a map.

Thesis Supervisor: Hari Balakrishnan
Title: Professor of Computer Science and Engineering

Thesis Supervisor: Samuel R. Madden
Title: Associate Professor of Computer Science and Engineering

*To my parents, to whom I owe everything.*

## Acknowledgments

First, I must thank both my advisors, Hari Balakrishnan and Sam Madden, without whose guidance, support and mentoring (each in their own unique way) this thesis would not have been possible.

Hari has been an inspiration from when I interned with him in my undergraduate days, through all of graduate school, and I have learned more about how to approach research from him than anyone else. His high standards, optimistic approach and exhortations to jump in and tackle the hardest and most meaningful problems have helped me become progressively more adventurous, risk-taking and confident in my research as well, and most importantly helped me have a lot of fun doing it. His excellent writing and presentation skills have also helped me learn a little about how to give talks and write papers. Last but not least, he has been a very supportive mentor, never hesitating to give help immediately when asked.

Sam has been an extremely helpful and encouraging mentor, and I can never thank him enough for his positive feedback, encouragement and constant support. He has always had the time and willingness to sit down and discuss any detail of a research problem, be it over person, phone, or e-mail, and provide much needed feedback, structure and guidance on concrete directions to pursue and baby steps to take towards solving a big problem. His guidance on FunctionDB and my Masters thesis, in particular, helped me a lot with getting started with research. What never ceases to amazes me is that I would often go to him for advice on a problem I'd spent a long time thinking about without success, and he'd instantly point me in the right direction after thinking for a minute or two. It was his vision and energy that helped transform VTrack from an algorithm in the lab to a real system that continuously processes data from the Cartel testbed and computes traffic delays. I also owe a lot to him for teaching me how to hack iPhone apps in the early stages of working on iTrack, which has been rewarding and fun in its own right.

I thank Seth Teller for serving on my thesis committee at incredibly short notice, reading and offering suggestions and comments. A lot of my work on indoor localization has been inspired by previous work he has done in this space. I thank Yoni Battat and Seth for their help with securing digital floorplans of the Stata Center for our iTrack project, and Sachi Hemachandra for allowing me to test drive their awesome robotic wheelchair.

Lenin Ravindranath has been the most incredible collaborator and lab-mate I could have asked for. His energy levels, propensity to generate new ideas every minute, and level of excitement about research are absolutely infectious, and hopefully this has rubbed off a little on me too. I would say it was his coming to MIT that reminded me that research can and should be fun, and helped shape my thesis topic at a time when I was still shopping around for ideas to work on. I have really enjoyed working with him and learning from him — both hacking and brainstorming, on a number of projects, including VTrack, CTrack and Code in the Air. Even the tiniest details of research and smallest of problems (like how to tweak an algorithm or design an experiment) turned into long and fun brainstorming sessions with him, and led to unexpected new directions to pursue.

Jakob Eriksson, my former office-mate, has been an awesome collaborator from the day he came to MIT. I admire Jakob's application-driven approach to research and his ability to quickly build useful, real systems at scale. It was his painstaking work that helped get PlanetTran taxicabs on board and build the Cartel testbed. The testbed has supplied me and countless other students with a rich vein of real data, without which this thesis would simply not exist. Later, I really enjoyed traveling to Chicago and working with him and his students James and Tomas on the Transitgenie bus tracking project, in a collegial, friendly and fun atmosphere. He has always been willing to help with advice on wireless networks, system building and the like, and we have had many fruitful and interesting discussions.

I thank Sivan Toledo and Katrina LaCurts for helping out with the experiments and writing for

the VTrack paper, and Bret Hull for access to Cartel data for the paper, as well as for patiently answering all my questions on Cartel. I also thank Sejoon Lim for his help with sharing data and thoughts, and getting me started with thinking about the map-matching problem.

Lewis Girod has been an extremely helpful resource on all things hardware related, and it has always been instructive to talk to him. I learn something new each time I go to his office. He was instrumental in getting me started off with research by mentoring me on the WaveScope project, and I am grateful to him for that.

I thank PlanetTran and Seth Riney for providing us with data and agreeing to run our software and devices in their cabs. Seth was always available and prompt to help us re-power and reboot the devices on-site when they failed or experienced errors.

I thank all the anonymous reviewers and shepherds of the papers I have co-authored over the years. Their comments and constructive suggestions have helped make this thesis stronger.

I thank Sheila Marian for her prompt help with all administrative issues — travel, reimbursements, visa paperwork, and many more. Cannot neglect to mention her awesome cakes.

Dorothy Curtis has been a life saver on innumerable occasions when my laptop or the server has gone down, offering prompt help and fixes. She has also been helpful whenever we needed to sort out issues with the Cartel database, free up disk space or buy new hardware.

I thank my office mates over the years at MIT: Emil Sit, Mike Walfish, Mythili Vutukuru, Lenin Ravindranath, Katrina LaCurts, Raluca Popa, Ramki Gummadi, Shuo Deng and Jakob Eriksson for their noise tolerance (particularly to my feverish keystrokes!), and many, many fun discussions. Mike, Jakob and Ramki in particular have given me plenty of high-level guidance on writing, presentation and picking problems to work on.

I thank all my other floor-mates and friends from CSAIL over the years — Adam Marcus, Arnab Bhattacharya, Dan Abadi, Dan Myers, Micah Brodsky, Alvin Cheung, Eugene Wu, Evan Jones, Eddie Nikolova, Vladimir Bychkovsky, Allen Miu, Stan Rost, Rahul Hariharan, George Huo, Jayashree Subramaniam, Yuan Mei, Bret Hull, Calvin Newport, Ramesh Chandra and others I've no doubt left out, for the many many engaging and fun discussions, help with practice talks, support and advice.

Paresh Malur and Tim Kaler have both been amazingly committed and fun undergrads to work with, and I have really enjoyed working, hacking and brainstorming with them on topics ranging from traffic prediction to smartphones.

I thank Michel Gorazcko, who was instrumental in helping me get set up, guiding me, and helping me learn the ropes of research when I first came to MIT as a wide-eyed undergrad intern seven years ago.

I thank my former internship mentors at Yahoo! Research — Utkarsh Srivastava, Brian Cooper, Adam Silberstein and Raghu Ramakrishnan for helping me get excited about research at a time when I was a bit unsure of what to work on. I enjoyed my experience at Yahoo! working on the PNUTS system a lot, thanks in no small part to the people there. I also thank Roopesh Ranjan, whom I first met during that internship, for being an awesome friend and sounding board over the years.

A special thanks to the baristas at the Forbes cafe in the Stata Center where I (and I imagine, many others like me) have faithfully consumed morning coffee over my years at MIT. Without you, no research would probably happen in this building.

I thank Krishna, Prabha, Jayku, Vivek, Mythili, Lavanya and Aditi for the awesome company, great home-cooked food, and many great memories during my stay in Ashdown. The cooking group, outings, and nightly discussions on random topics were, and will remain, some of the best times of my PhD experience at MIT. Vivek Jaiswal's incredibly tasty home-made *dal* and *fried rice* in particular sustained me for the better part of two years, I think. I also thank Aditi, Basky and

Shashi for the fun times talking music, and practising, performing and planning musical pieces for Diwali Nite, the Ashdown Concert and the like. I thank Aruna, Murali, Arvind Shankar, Karthik, and Anna for the fun times, outings, potlucks and ski trips during my last two years at MIT. I also thank my apartment mates (and friends) Vijay, Pranesh, Sharat and Ivan for the company and good times over the years.

I thank Krishna for introducing me to the incredible ocean that is Carnatic Music, which has become a big part of my life over the years. I can never be indebted enough to him for that. I thank my guru, Smt. Geetha Murali, for helping me get a real foothold in learning music and teaching me patiently in spite of my deficiencies. I thank Hari Arthanari for the many great musical memories — all the padams, Balu's concerts, and the *impromptu* Brindamma and Vishwa listening sessions we used to have in his car. I also thank Chintan Vaishnav for the Bade Ghulam Ali Khan and Amir Khan sessions. It would not be out of place here to thank the great masters of Carnatic Music who have (posthumously) given me innumerable hours of listening pleasure.

I thank Raju Mahodaya and Sharada Mahodaya for teaching me whatever little of Samskritam (Sanskrit) I know today, and Raju in particular for exhorting me to take up teaching Samskritam to learn it better. The Wednesday sessions of teaching and learning about this incredible language were a welcome break from the routine of graduate student life.

I thank my close friends, Vikram and Kamesh, for their support and amazing level of belief in me at all times, and for the many fun times over the years.

I am indebted to all my near and far family members. In particular my *Patti* (grandmother) Chajja, Sudhaman *Thatha*, my *Kollu Patti* (great-grandmother) Dhamma, and Jayamani and Jalaja *Patti* for their support, love, advice, never-shaking belief, and their constant prayers for me. Seethalakshmi Patti has always been razor sharp about making sure I was focused on finishing my PhD, as has Krishnamoorthy *Thatha*, and their contribution to my finishing is as big as anything else.

My wife Poorna's selfless love, affection, patience, home cooked meals, filter coffee, and constant support and belief in me while juggling her own exam preparations and volunteer work during my last year of PhD research at MIT is what has made this thesis possible. My in-laws' love, support and encouragement has also meant a lot to me during this time.

I owe everything in life that is good to my parents. Some debts are simply not possible to repay. *Appa* and *Amma* have been rock solid in their emotional support, optimism, constant encouragement and even detailed technical advice from thousands of miles away — through the worst of times, and the lowest of the lows. *Appa* has often been more excited and enthusiastic about my research than I have been. Both of them have had unshakeable belief in me. I dedicate this thesis to them.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

This dissertation is concerned with the problem of estimating the trajectory, or *sequence* of locations visited by a mobile phone, in an accurate and energy-efficient manner.

Two key trends — the proliferation of mobile smartphones, and the availability of a wide variety of location sensors on these phones, in particular global positioning technology (GPS) — have led to the emergence of many mobile applications that use trajectory estimation as a fundamental primitive.

For example, crowd-sourced traffic applications like iCartel [39] (Figure 1-1(b)) collect raw (latitude, longitude) location data from end user smartphones and match them to a sequence of road segments to obtain travel time estimates for individual roads. They then share these estimates with other users, who use them for applications like *traffic-aware route planning*, i.e., finding the best path to a destination given real-time traffic conditions and travel times.

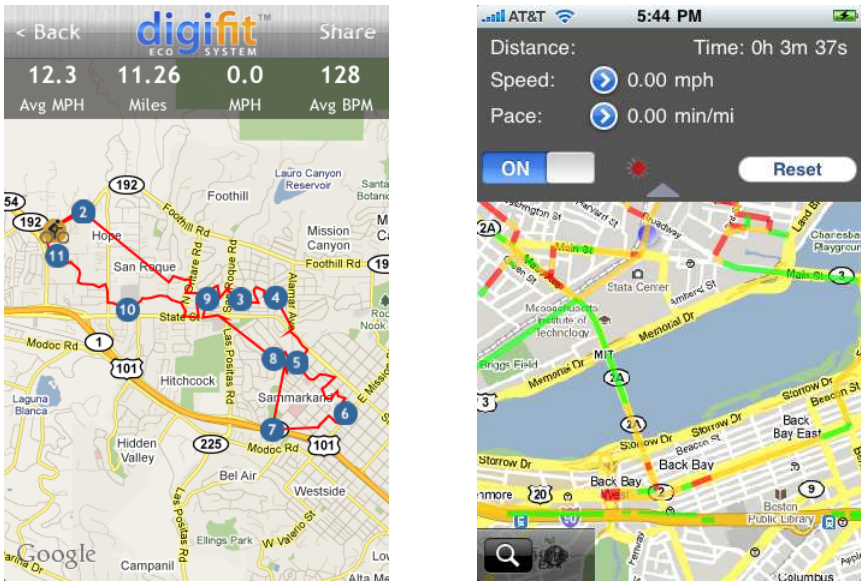
Personal fitness applications like Digifit [27] (Figure 1-1(a)) and MapMyRun [59] allow smartphone users to record and view their biking or running tracks, calorie consumption and other fitness-related metrics on a map. In a similar vein, personalized commute applications like Carweb (Figure 1-2) allow smartphone users to visualize and optimize their commute paths.

Trajectory estimation is also useful in many indoor sensing and analytics applications. Today's indoor location systems primarily use custom hardware or dedicated sensor devices [62, 74, 30], but are poised to become a reality on commodity mobile smartphones in the near future. For example, researchers have trialed mobile sensing systems that find movement patterns of people indoors, and use them to improve workplace efficiency [18]. There exist commercial products that track the location of patients, doctors or medical equipment indoors to identify and rectify system inefficiencies and improve patient care — for example, the Massachusetts General Hospital has trialled such an analytics system from Radianse [74].

## 1.1 Why Is Trajectory Estimation Hard?

Trajectory estimation on mobile phones is a non-trivial problem because *today's phones have no known position sensor that is accurate everywhere and energy-efficient.* The position sensors available on phones today include:

- GPS, which uses the time difference of arrival between transmissions from multiple satellites orbiting the earth to calculate the (latitude, longitude) position of a mobile phone accurate to within a few metres.




(a) Digifit: record and view your running route.

(b) iCartel: download traffic delay information and contribute them to other users.

**Figure 1-1: Screenshots of two track-based applications for mobile phones.**

- WiFi and cellular localization, which use nearby wireless base stations and cellular towers for positioning. A phone looks up the set of nearby base stations or cell towers it can hear, and their signal strengths, in a pre-existing training database (built using GPS) to find its position.
- Inertial MEMS (micro-electro-mechanical) sensors such as accelerometers, gyroscopes and magnetic compasses, which measure the *change* in a mobile phone's position or orientation and can be used to indirectly compute its location.

However, all of the above sensors fail to meet one or the other of three key requirements of the applications mentioned earlier — *accuracy*, *energy-efficiency* and *wide availability*. As we show:

- GPS is accurate, but is not energy-efficient and is not available everywhere.
- WiFi and cellular localization are more energy-efficient than GPS and widely available, but are highly inaccurate.
- Inertial MEMS sensors are highly energy-efficient, but also inaccurate.

We elaborate on each of the above below.

**Limitations of GPS.** While GPS is accurate to within a few metres outdoors, it is energy-intensive and drains a mobile phone's battery when kept switched on. A common feature of all the applications mentioned earlier is that they need to *continuously* monitor the location of a mobile device to find its trajectory. This is impractical with GPS because it results in poor battery life. For example, from our own experience building and deploying the iCartel system [39], we found that end users' phones barely last for a few hours with GPS switched on all the time. This is unacceptably low and

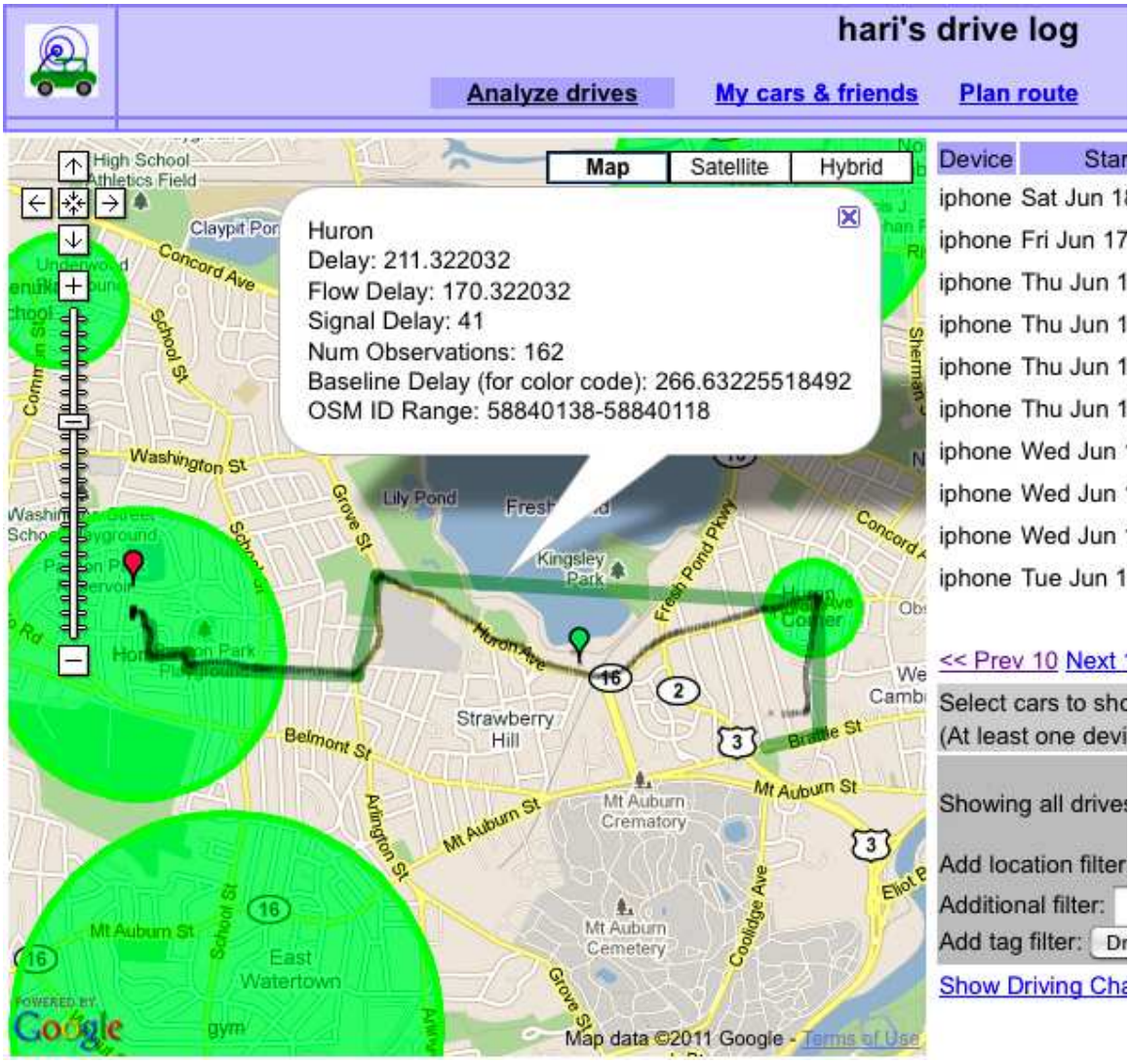



**Figure 1-2: Screenshot of Carweb, a web application that allows users to visualize and compute statistics for their commute paths.**

makes users reluctant to install the application on their phones. A similar concern exists for all of the outdoor tracking and fitness applications mentioned earlier.

We note that GPS energy consumption is a fundamental limitation that does not show a trend of reducing with improvements in underlying GPS hardware. This is because GPS satellites orbit 11,000 miles above the earth's surface with a transmission power of approximately 50 W, resulting in only $2 \times 10^{-11}$ $\mathrm{mW/m^2}$ of effective radiated power (ERP) at a receiver on the earth's surface. GPS receivers need sophisticated signal processing to successfully extract information from a signal with such low power (in contrast, the typical cell phone signal has an ERP of $10$ $\mathrm{mW/m^2}$ [35]). This processing is computationally-intensive and consumes significant energy on a mobile device.

The second fundamental limitation of GPS is that it does not work indoors, or in areas with significant obstructions that lower the GPS signal-to-noise ratio, such as downtown areas of cities with many tall skyscrapers. This makes GPS unusable for indoor trajectory estimation, and inaccurate (at times) for outdoor applications as well.

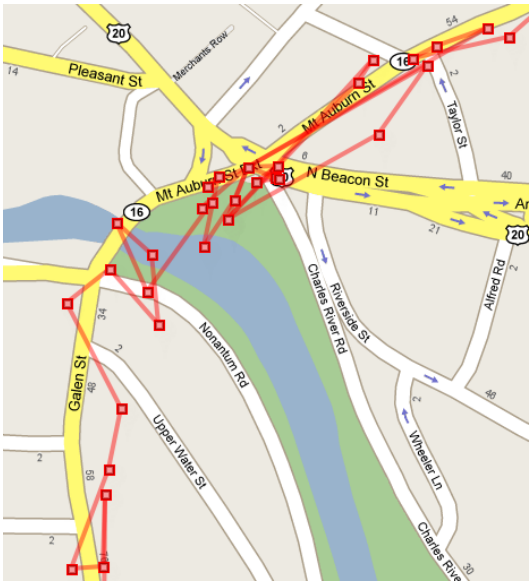
**Limitations of WiFi and Cellular.** Both WiFi and cellular localization consume less energy than GPS because of the higher ERP of WiFi and cellular signals, which are transmitted from much closer to the earth's surface than GPS satellites. Cellular localization is particularly attractive because it consumes little or no *marginal energy*: a phone's cellular receive circuits are typically always on anyway to receive calls.

The flip side is that these sensors are *much* less accurate than GPS, owing to the propagation of WiFi and cellular signals. We find that raw WiFi position estimates can have errors of up to 50-100 metres outdoors, and raw cellular position estimates have errors ranging from a few hundred metres to as much as a kilometre. Figure 1.1 shows examples of WiFi and cellular (GSM) localization estimates. In the figures, the red points are raw locations obtained by matching observed WiFi or cellular signals to their closest match in a pre-built training database. The actual roads traversed are shown in black. It is non-trivial to recover the true sequence of road segments from the raw position estimates, let alone recover accurate travel times for each individual segment.

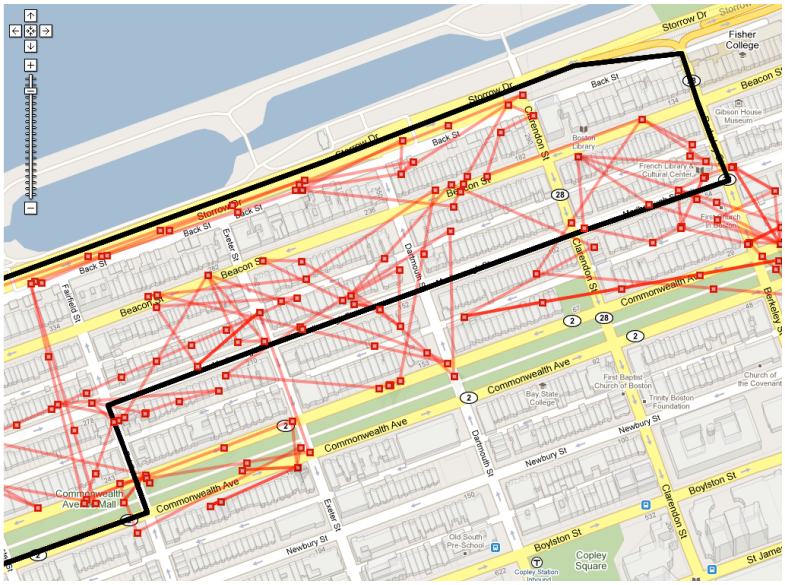
Both WiFi and cellular localization work indoors, where GPS does not work. However, they are also inaccurate indoors. Our experiments show a mean positioning error of 5-6 metres indoors when using WiFi localization, also corroborated by other studies on both WiFi and cellular localization [28, 8]. The real problem is not the 5-6 metre mean location error, but that a significant fraction of position estimates can have errors ranging from 10-20 metres. This is too high to be useful for applications that want to locate where a person is in a store, hospital or workplace, or find room-level location in a building. Figure 1.1 compares the true trajectory walked by a person indoors to the output of WiFi localization. WiFi can approximately find the outline of the trajectory walked, but is subject to significant errors at multiple points in the walk.

A second major limitation is that WiFi localization indoors requires extensive manual training effort to associate known locations to WiFi signal measurements. Outdoors, training is relatively easy because ground truth can be obtained from GPS. Building an indoor map of WiFi, on the other hand, is *much* more tedious and difficult to scale, because training requires dedicated personnel or volunteers to survey hundreds or thousands of locations in a building, and manually mark them on a map to associate them to WiFi signals at that location.

**Limitations of MEMS sensors.** MEMS sensors such as the accelerometer and the gyroscope are relatively energy-efficient, and they work indoors and outdoors. In theory, it should be possible to




(a) WiFi localization estimates.




(b) GSM localization estimates.

**Figure 1-3: WiFi and GSM localization are highly error-prone.**


(a) True trajectory walked by a person indoors.


(b) Trajectory estimated from WiFi localization.

**Figure 1-4: WiFi localization is error-prone indoors.**

perform *dead reckoning* with inertial sensors by integrating angular velocity estimates from a gyroscope to obtain the orientation of the phone at any time instant, and using this orientation to subtract the effect of the earth's gravity from measured acceleration. The resulting "user acceleration" when integrated twice should give the displacement, and hence location of the phone. However, the problem is that even a small measurement error by an MEMS sensor results in a large, rapidly growing *drift error* when the output of the sensor is used as input to higher-order integration, because the errors add up. The net effect is that the estimates from integration are unusable for localization. Figure 1.1 illustrates upward sloping drift in user acceleration estimated from accelerometer and gyroscope measurements on an iPhone.

## 1.2 Contributions Of This Dissertation

This dissertation describes systems that address all the above challenges, using measurements of:

- GPS
- WiFi signals
- Cellular signals

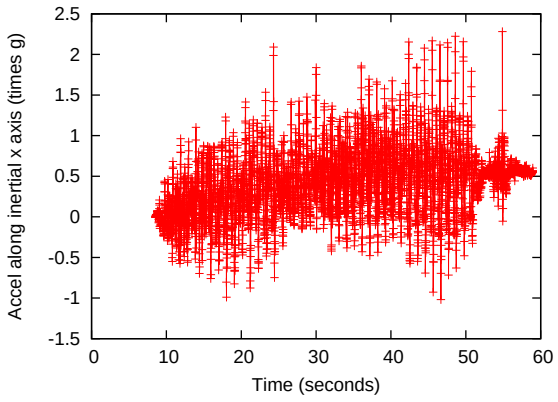



**Figure 1-5: Upward sloping drift in user acceleration measured on an iPhone.**

- Inertial sensors (accelerometer and gyroscope)

to estimate the trajectory of a mobile device accurately and energy-efficiently, indoors or outdoors. These systems develop and use two key techniques:

- Techniques that can extract accurate trajectories from both *sparsely sampled* and *noisy* location estimates. The ability to handle sparse input is important because it permits us to use *intermittently* sampled (sub-sampled) GPS, and thereby duty cycle the GPS when not in use to save a significant amount of energy. Handling noise permits us to use inaccurate WiFi and cellular localization estimates as input, and still estimate trajectories accurate at the level of individual road segments.
- Sensor fusion techniques that *combine* data from one or more of the above sensors — GPS, WiFi, cellular, and MEMS. While each individual sensor does not have all three desired properties of accuracy, energy efficiency and wide availability, sensor fusion enables us to achieve all the properties.

The systems we present all rely on a class of probabilistic model called a Hidden Markov Model (HMM). An HMM is a Markov process that models a sequence of noisy observations as coming from a sequence of (unknown) hidden states with a known probability distribution. It also models *transitions* between hidden states at each time step using a different distribution. Given an HMM and a sequence of noisy observations, a dynamic programming algorithm called Viterbi decoding can find the most likely sequence of hidden states corresponding to the observations.

While a HMM is a well-known tool to model and correct noise in input data, it is only as good as the underlying choice of hidden states, observations and probabilities used in it. We show in this dissertation that a careful choice of HMM is essential to handle sparsely sampled and noisy location data, and to perform effective sensor fusion. In the following sections, we describe the three systems we have developed: *VTrack*, *CTrack* and *iTrack*, and the key insights from each of the systems.

## 1.3 VTrack: Map-Matching Noisy and Sparse Coordinates

*VTrack* matches noisy position samples in the form of geographic $(lat, lon)$ coordinates to a sequence of road segments on a map, and extracts travel time estimates for each output road segment.

*VTrack* uses an HMM where the observations are noisy coordinates and the hidden states are road segments. *VTrack* uses two insights that are critical to making the HMM work well:

- It uses a *speed constraint* that enforces the fact that a minimum amount of time must be spent on each road segment. This prevents the output of the HMM from "jumping around" to follow the noise in the input data. We show that the speed constraint is particularly important for matching the noisiest trajectories, reducing the error of map-matching (a term we define precisely in Chapter 3) by a factor of $3\times$ for the noisiest $10\%$ of input trajectories.
- It *interpolates* its input location samples before feeding them to the HMM. This simplifies the design of the HMM because it can enforce an *adjacency constraint* on the output road segments. This helps ensure output a continuous sequence of road segments even if the input has outages or is intermittently sampled. Interestingly, we find that a simple linear interpolation strategy performs as well as more sophisticated strategies such as shortest path interpolation.

*VTrack* is optimized offline for each mobile phone hardware configuration it runs on: it uses offline measurements of GPS and WiFi energy costs on a mobile phone in conjunction with a given energy budget to determine whether to use GPS sub-sampling, or WiFi, or a combination of duty-cycled GPS and WiFi location samples on that phone hardware.

*VTrack* has been deployed as part of the Cartel [17] mobile sensing system at MIT. Our evaluation is over GPS and WiFi location data from a testbed of 25 taxicabs, and from mobile phone users who run the iCartel iPhone application and contribute crowd-sourced traffic data for the Boston area. It has been running live for nearly two years and has been used to map-match thousands of trajectories of raw position samples. The travel time estimates produced by *VTrack* are served to both users of the iCartel application and the Carweb web portal.

We evaluate the accuracy and energy-efficiency of *VTrack* on 800 hours of driving data from the Cartel testbed. We use *VTrack* to extract trajectories from GPS and WiFi location samples, and evaluate the accuracy of the trajectories produced by *VTrack* by comparing to ground truth GPS. In addition, we evaluate the usefulness of the output trajectories in the context of two applications: using travel times extracted from the trajectories for better route planning, and for detecting individual traffic "hotspots". Our key findings are:

- Using a Hidden Markov Model for map-matching is robust to significant amounts of noise, producing trajectories with a median error of less than $10\%$ on WiFi localization samples that have a median error of over 50 metres. Using an HMM is much more robust to noise in the input data than a simple strategy like matching each point to the closest segment on a road map, which breaks down when map-matching WiFi positioning or noisy GPS data.
- We find that *VTrack*'s interpolation technique is suitable for map-matching location data consisting of sub-sampled GPS. When GPS is available, sampling GPS periodically to save energy is a viable strategy. On our data, for up to 30-seconds duty cycling (i.e., turning on the GPS every 30 seconds to get a location sample , then turning it off for 30 seconds to save energy) sub-sampled GPS can produce high quality trajectories if processed with a HMM.
- Travel times from WiFi localization alone are accurate enough for route planning, even though estimates for individual road segments are poor. This is because groups of segments are typically traversed together, and our estimation scheme ensures that errors on adjacent or nearby segments cancel out.

- If a phone is also WiFi-equipped, the tradeoff between sampling GPS every $k$ seconds, sampling WiFi or a hybrid strategy (*both* sub-sampled GPS and WiFi) depends on the energy costs of each sensor. Chapter 3 of this thesis explores these tradeoffs in detail.

## 1.4 CTrack: Accurate Tracks From Soft Information

As mentioned earlier, extracting tracks from cellular signals alone is attractive because the *marginal* energy consumption of cellular localization is close to zero on mobile phones, since the cellular radio is always on anyway. However, raw cellular position samples (such as in Figure 1-3(b)) have errors ranging from a few hundred metres to as much as a kilometre, making them challenging to match to a map. As we show, the one-pass HMM used in *VTrack* breaks down when run over data with so much noise.

The second component of this dissertation is *CTrack*, a system that makes it possible to extract accurate tracks from cellular signals. *CTrack* uses two novel insights to achieve this:

- Using *soft* information in the form of signal strengths from cellular (or WiFi) radios. *CTrack* uses a two-pass HMM that is very different from the HMM used in *VTrack*. It divides space into grid cells, and determines the most likely sequence of traversed grid cells corresponding to a sequence of cellular base station observations.

  The *CTrack* HMM uses training data to build a probabilistic model of which base stations are seen from which grids, and with what signal strengths, in contrast to the simple model of a propagation error in a latitude/longitude coordinate that the *VTrack* HMM uses. This model is better than *VTrack* for cellular signal observations, because a given cellular "radio fingerprint" (consisting of cell towers and their signal strengths) is seen from a wide range of locations, sometimes spaced as much as hundreds of metres apart. Averaging this soft information to produce a single "hard" location estimate loses a lot of information.

- A probabilistic model for the HMM that uses *sensor hints* from MEMS sensors on the phone — the accelerometer to detect movements, and the compass or gyroscope to detect turns.

We have implemented *CTrack* on the Android smartphone platform, and evaluated it on nearly 126 hours of real drives (1,074 total miles) from 20 Android phones in the Boston area. We find that *CTrack* is good at identifying the sequence of road segments driven by a user, achieving 75% precision and 80% recall accuracy (these terms are defined and explained in Chapter 4 of this dissertation). This is significantly better than state-of-the-art cellular fingerprinting approaches like Placelab [57] applied to the same data, reducing the error of trajectory matching by a factor of 2.5×. We also measure the energy consumption of *CTrack* on Android and find that *CTrack* has a significantly better energy-accuracy trade-off than other candidate approaches, including sub-sampling GPS data, or using VTrack, reducing energy cost by a factor of 2.5× for the same level of mapping accuracy. We also show that sensor hints can correct some common systematic errors that arise with cellular localization.

*VTrack* is still useful when the only data available is "hard" information in the form of coordinates, which is the case on some platforms. For example, on Apple's iOS, WiFi and cellular location can only be accessed indirectly via a location API that provides geographic position samples.

## 1.5 iTrack: Accurate Indoor Tracks From Inertial Sensing and WiFi

Raw WiFi location samples indoors are subject to an average error of the order of 5-6 metres, and can occasionally experience errors of up to 10 or 20 metres. It is possible to use a Hidden Markov Model to extract tracks from this data (similar to *VTrack* or *CTrack*) but the extracted tracks still have limited accuracy. Secondly, as mentioned earlier, WiFi localization requires extensive and cumbersome manual training, either dedicated or crowd-sourced, before it can be deployed in a building.

*iTrack* is a system that addresses both these limitations by using a novel sensor fusion approach that combines data from inertial MEMS sensors with data from WiFi signals. While either of these sensors taken on its own is inaccurate, we show that *combining* data from these sensors using a carefully designed Markov Model can achieve high trajectory estimation accuracy. *iTrack* can accurately find the trajectory of a person walking steadily indoors with a smartphone in his/her hand or pants pocket to within less than a metre of error.

*iTrack* also significantly reduces the manual training effort required for WiFi localization by allowing tracks extracted from walks to be used to contribute training data for WiFi localization automatically, without requiring users to manually mark their location on a map.

*iTrack* uses two key novel ideas. The first key novelty of *iTrack* is how we avoid accumulation of drift error when dealing with data from MEMS sensors. *iTrack*'s approach works in four steps: *walking detection*, *step counting*, *shape extraction* and *map-matching*:

- Walking detection uses acceleration measurements to detect walking and distinguish it from other movements of a mobile phone such as talking, picking up and other normal use. A periodic up-and-down or sideways swaying motion is characteristic of walking, whether the phone is held in a users' hand, pants pocket, or bag, and causes accelerometer data to exhibit a periodic pattern with a peak on its Discrete Fourier Transform (DFT).
- Step counting uses the acceleration signal to accurately determine the number of steps walked using an iterative peak-finding algorithm.
- Shape extraction integrates angular velocity from the gyroscope to find *changes* in orientation (turns), and hence the *approximate shape* of a user's walk. The key insight is that using estimates of *changes* in orientation is more robust to drift error than using absolute orientation.
- Map-matching matches the extracted shape and size from the MEMS sensors to the contours of a floorplan (assumed to be known) using a *particle filter*. A particle filter is essentially a generalization of an HMM to a continuous state space, here the $(x, y)$ coordinate of the phone on the floorplan. Unlike *VTrack* and *CTrack* (where the state space is discrete, consisting of road segments or grids on a map), the Viterbi algorithm cannot be used for a continuous state space. Instead, a particle filter simulates a large number of paths with similar shape and size to that determined from the MEMS sensors, and eliminates paths that cross a wall or obstacle in the floorplan to try to find a good match.

The second key novelty of *iTrack* is the actual Markov model used in the particle filter. While particle filters with a Gaussian error model have been previously used for indoor pedestrian tracking with foot mounted sensors [67, 29], we show that a *mixture model* for stride length and turn angle are preferable to a Gaussian error model in the mobile phone context. This is because a person's

walking pace typically varies significantly even within a given walk, and it is difficult to extract accurate length information for each stride from a mobile phone not rigidly mounted on a users' person.

The particle filter used in *iTrack* can optionally fuse WiFi signal information to improve the quality of the tracks it extracts. *iTrack* uses a small amount of "seed WiFi data" on each floor (typically 4-5 walks known to be on that floor) to identify the floor and building a person is on. The initial WiFi helps to initially extract tracks of adequate quality from inertial sensing.

As more WiFi data is collected, this sensor fusion has a feedback effect that enables *iTrack* to achieve the goal of indoor tracking with very little manual input. *iTrack* continuously adds tracks extracted from inertial sensing to a training database containing WiFi access points, their signal strengths, and locations in the floorplan they were seen from. As more tracks become available, *iTrack* iteratively uses the extracted WiFi information to *re-extract* tracks from previously seen walks, and rebuild the WiFi training database. This has the effect that the quality of extracted tracks improves, and further improves the quality of the WiFi training database, resulting in a positive feedback cycle.

Therefore, once the seed information has been populated for each floor of interest, the rest of the process is entirely automated and does not require manual input — dedicated operators simply walk around the floor to add more training data, and volunteers can contribute "crowd-sourcing" data easily by simply downloading and running a background phone application. Since crowd-sourcing is entirely automated and requires no intervention on behalf of the user beyond normal use of the application, this is a significant improvement over techniques requiring extensive manual input from a human to indicate where he/she is.

We evaluate *iTrack* on 50 walks collected on the 9th floor of the MIT CSAIL building in the Stata Center. We use tape markings made on the ground to accurately estimate ground truth for each trajectory. We find that with seed WiFi data from just 4 walks (that took only 5-10 minutes to collect), *iTrack* can extract accurate trajectory data from 80% (40 out of 50) of the walks (20% of the walks fail to produce any match). The walks extracted have a median error of $\approx 3.1$ feet, corresponding to approximately one stride. This contrasts to a median error of over 18 feet if using just WiFi.

## 1.6 Road Map

The rest of this dissertation is organized as follows. Chapter 2 describes energy measurements of the different localization technologies and sensors used in this dissertation (GPS, WiFi, cellular and inertial sensors) on different phone platforms. These measurements motivate and drive the design of the algorithms subsequently described in the dissertation. Chapter 3 describes the *VTrack* system for map-matching a sequence of noisy geographic location samples to road segments, and extracting travel time information from these segments. Chapter 4 describes and evaluates *CTrack*, a system that can accurately map-match soft information from cellular (GSM) signals. Chapter 5 first provides background on how MEMS inertial sensing technology works, and then describes and evaluates *iTrack*. Chapter 6 concludes the dissertation and outlines directions for future work.

# Chapter 2

# Energy Studies

We return to the first problem mentioned in the introduction:

- *Given an energy budget and a requirement on accuracy, what is the best combination of sensor(s) and sampling rate(s) to use for trajectory mapping?*

We will show that the answer to this question depends on the specific mobile device hardware being used, and in particular on the energy costs of sampling different sensors. For this reason, for any given mobile device whose trajectory we want to map, it is necessary to first measure the energy costs of each sensor and use the measurements to drive an energy-efficient algorithm design.

In this chapter, we first present a discussion of the fundamental factors that drive the on-device energy cost of each of the sensors under consideration — GPS localization, WiFi localization, cellular localization and inertial/orientation sensors (accelerometer, magnetic compass and gyroscope). As part of the discussion, we survey a number of experimental studies by other researchers that measure the energy costs of these sensors on different mobile devices.

We then present three of our own experiments that measure the energy costs of these sensors when sampled at different rates on actual phone hardware. The first experiment measures the energy costs of GPS and WiFi localization on the iPhone 3G. The second experiment measures the energy cost of accelerometer sampling relative to the cost of GPS sampling on the iPhone 3G-S and iPhone 4. The third experiment measures the energy cost of GPS, WiFi localization, cellular localization, accelerometer and magnetic compass on an Android G1 phone.

The results of the energy experiments in this chapter drive the design of the energy-efficient trajectory mapping algorithms described in Chapters 3 and 4. We do this by building cost models for the energy consumption of each sensor, and using the cost models to choose the right parameters for the algorithm i.e., which sensor(s) to use and at what sampling rate(s).

## 2.1 Theoretical Factors Driving Energy Cost

### 2.1.1 GPS Energy Cost

The energy cost of GPS is rooted in the need for processing gain to acquire positioning signals from GPS satellites. On modern GPS chipsets, there are two distinct parts of this process which

have different energy consumption profiles. To understand the energy consumption requirements of GPS, it is first necessary to understand how a GPS receiver works.

To estimate its location, a GPS receiver on earth measures the transmission delay from multiple GPS satellites orbiting the earth's surface. The receiver uses the measured transmission delay to calculate its distance (also called *pseudo-range*) to each visible satellite. For each satellite, the receiver has prior knowledge of the precise orbit of that satellite, and hence knows that its location must lie on a sphere with that satellite's location as centre and the measured pseudo-range as radius. If the receiver can measure its pseudo-range for at least 3 (preferably 4) satellites, it can use *triangulation* to solve for the intersection of the spheres and determine its precise latitude, longitude, and altitude above the earth's surface.

The GPS triangulation process requires the following pre-requisites before it can work:

- At least 3 (preferably 4) visible GPS satellites i.e., satellites that the GPS receiver can hear and decode transmissions from.
- Knowledge of the precise orbits of the visible satellites so that their location can be calculated exactly. This is called *ephemeris* information.

When a GPS receiver first powers up, its first task is to determine a set of 3 or more visible satellites to triangulate from. The naive method of doing this (and the fallback in case quicker techniques fail) is brute-force search. Each satellite is assigned a unique binary sequence called a *Gold code*. The GPS signal is decoded after demodulation using modulo 2 addition of the Gold codes corresponding to each satellite. When performing brute-force search, a GPS receiver cycles through *all* the possible Gold codes until at least one of the satellite transmissions can be decoded. However, as might be expected, this process is extremely time-consuming. On older GPS receivers with only one reception channel, obtaining a brute-force satellite lock can take between 7.5 to 15 minutes. Modern GPS receivers can receive on up to 12 channels in parallel, significantly reducing the brute-force fix time. However, even on modern chipsets, obtaining a brute-force satellite lock remains a fairly time consuming process, requiring 3-5 minutes [34].

To reduce the time to get a fix, GPS receivers maintain a cache of visible satellites and their approximate coarse-grained locations, called an *almanac*, which helps narrow down the brute-force search. The GPS receiver uses a brute force search to obtain a lock the first time it is used. On first use, it downloads an almanac from the GPS satellite it is connected to. Once downloaded, the almanac information remains valid for a few months.

Once the almanac information is available and the GPS has narrowed down a set of 3 or more visible satellites, it must next download the orbital (ephemeris) information for these satellites before it can triangulate its position. Each GPS satellite transmits its ephemeris information every 30 seconds. The satellite link over which the transmission takes place has limited bandwidth, and the maximum possible transmission rate over the link is only 50 bytes per second. This means it can take up to 40 seconds to download the ephemeris information from each satellite. Older GPS receivers cannot download this information in parallel (since they can only receive on one channel) but newer chipsets can receive on up to 12 channels and hence can download the ephemeris information from 3 or 4 satellites in parallel. Thus, a modern GPS receiver can obtain a so-called "GPS lock" (i.e., knowledge of ephemeris information for at least 3 visible satellites), and compute a first GPS location estimate in about 40-45 seconds [34]. This process of downloading the ephemeris information

is called a *cold start* in GPS terminology, and must be repeated each time a GPS is powered on in a new location.

A further optimization modern GPS chipsets use is *warm start*. If a GPS chipset is turned off and then turned on again within a certain (short) time interval, and is at a location near where it was turned off, it is likely to be able to hear the same set of satellites it downloaded ephemeris information for previously. Hence, it can use ephemeris information saved from the previous cold start to reduce the time required to obtain a fix. The time to fix is still not instantaneous, because the receiver needs to verify that the ephemeris data is valid and that at least 3 satellites it has data for are still visible in the sky. A "warm start" on a modern GPS receiver takes between 6-15 seconds.

The simplest way we have developed to measure the time to a GPS warm start on a mobile device is to drive through a tunnel in an urban area (or other known region with poor GPS reception). The device will lose its GPS lock and re-acquire it when the vehicle emerges from the tunnel. It is easy to measure the time required for the device to re-acquire its GPS position estimate from the instant the vehicle emerges from a tunnel. Using this method, we estimated an average "warm start" time of 6 seconds on the iPhone 3G, and about 12 seconds on the Android G1.

Once a GPS chipset has downloaded the necessary ephemeris information, it switches to "continuous tracking mode". In this mode, the main function of the GPS is to use periodic satellite transmissions to perform triangulation and continuously determine its position.

**GPS Energy Requirements.** The process of obtaining a GPS lock from scratch ("cold start") is the most energy-intensive part of GPS operation, since it requires significant processing gain over a period of 40-50 seconds to decode and download ephemeris information from the relatively weak ($-160$ dB) GPS signal. This energy requirement is somewhat lower for warm start, but warm start still consumes energy to download and verify satellite orbital information during the 6-15 second initialization period. As might be expected, this energy consumption increases in conditions where the signal is weaker and requires more processing gain e.g., on a cloudy day without clear satellite visibility, or in areas with interference from other obstacles, such as downtown urban areas with tall skyscrapers. The power consumption of continuous tracking mode once a fix is obtained is typically lower, but still significant on a mobile device. Most GPS receivers report a power consumption requirement of about 50-75 mW in continuous tracking mode. In practice, when including the energy required to obtain a fix, we have found in our experiments that the average power consumption of GPS chipsets on mobile smartphones is much higher, closer to 350-400 mW. These results are also corroborated by studies by other researchers, presented in Section 2.1.5.

### 2.1.2 WiFi Localization

WiFi localization on mobile phones works by periodically scanning for nearby WiFi access points. The phone periodically looks up the set of access points it sees and their signal strengths (aka "WiFi signature") in a local or remote database to determine its approximate location.

WiFi localization consumes energy because the WiFi radio must be on to periodically scan for nearby WiFi access points. Localizing with WiFi does not actually require associating to an access point or transmitting data over WiFi. Duty-cycling the WiFi radio incurs an initialization and shutdown energy cost on most phones [50]. Studies on a Nokia N95 mobile phone [45] and [50] have found the maximum power consumption of WiFi localization on a mobile phone to be of the order of 1 Watt, during the time period when the phone is actually scanning for WiFi access points.

Fortunately, a device using WiFi localization need not scan all the time. The *average* power consumption of WiFi localization over *all time* i.e., not just the time period when a scan is being

performed, is lower than the peak power consumption of 1 Watt, and is mainly driven by the time duration of each WiFi scan and the interval between successive scans. The authors of [50] found an average scan time of 0.7 seconds on an AT&T Tilt phone. We have found the number to be smaller on more modern iPhone and Android devices, closer to 0.5 seconds or even smaller. A reasonable assumption is that that the maximum possible rate of WiFi sampling corresponds to obtaining a new WiFi localization estimate every 2 seconds. Typically, more frequent scanning than 0.5 Hz is not useful since more frequent scans tend to return cached results or result identical to the previous scan. Assuming a scan interval of 2 seconds and a mean scan duration of 0.5 seconds, we can derive a theoretical average power draw of approximately $\frac{1}{4} \times 1 = 250$ mW, smaller than 1 Hz GPS sampling (about 400 mW).

Our actual experiment with WiFi energy consumption measured the energy cost on an Android G1 phone with a request interval of approximately 2 seconds between WiFi scans. We found that battery life of the phone was close to 10 hours with WiFi scanning, compared to only 6 hours with 1 Hz GPS sampling. This corroborates the back-of-the-envelope theoretical calculation above, as well as the power consumption findings in [50, 45].

A second driver of energy cost with WiFi localization is the network transmission cost of looking up WiFi access points if using a remote server-side database to do the lookup. For example, each time an Android phone makes a "network location" lookup, it contacts a Google server with a list of the access points it can see nearby to determine its location. The request must be transmitted and a reply received from the server over the internet. This uses either WiFi or 2G/3G transmission and consumes energy.

Fortunately, in practice it is possible to store a locally relevant cache of the WiFi access point database on the phone, thus significantly reducing or eliminating the energy cost associated with the lookup request. Current implementations of WiFi localization all appear to use caching of WiFi tiles to reduce latency and energy consumption [41]. For this reason, we ignore the energy cost of network lookups in all of our subsequent analysis.

To summarize, the power consumption of WiFi localization is often 50-60% lower than GPS (assuming both sampled at the highest possible rate), with the exact savings depending on the mobile phone hardware. Previous work [77] corroborates these end-to-end energy consumption numbers on a Nokia N95, finding that WiFi localization can provide 2-3 times the battery life of GPS localization. The iPhone 3G, which we used for one of our energy experiments (§2.2.2), appears to be an exception to this approximate trend, with GPS being an order of magnitude more expensive than WiFi — perhaps owing to poor GPS power management.

### 2.1.3 Cellular Localization

Because phones continuously track cell towers as a part of their normal operation, it costs very little extra energy to keep the cellular radio switched on, and log the cell towers seen and their signal strengths. Hence, the marginal energy cost of cellular localization is small, and driven primarily by CPU load. On the Android OS, for example, a list of neighbouring towers and their signal strengths can be obtained by querying the `TelephonyManager` API. An application or service that provides/uses cellular localization periodically reads the list of neighbouring cell towers and their signal strengths using the API. Keeping such an application running continuously consumes a small amount of energy since the phone cannot put the CPU in low-power mode or frequency scaling mode while it is running. Processing a cell tower signature might require at most 100,000 instructions, which costs 5 nJ on a current generation 1 GHz Qualcomm Snapdragon processor.

In embedded (non-phone) applications that do not need the cellular radio on as part of their normal operation, it is possible to track only the signal quality and cell ID portions of the GSM protocol. This requires observing only the BCH slots of the GSM beacon channel, which are 4.6 ms long and are transmitted once per each 1.8 second cycle. A 10% GSM receiver duty cycle should be adequate to track the strongest towers. Assuming a GSM receiver uses 17 mA at 100% duty cycle, this represents an additional power consumption of 5 mW (1.7 mA @ 2.7 V) [11, 79].

Our experiments measuring battery lifetime with an Android G1 phone found that the phone lasted nearly 3 days when running a background application that periodically requested and wrote a list of neighbouring cell towers and their signal strengths to flash storage. This is close to the phone's "stand by" lifetime when not running any applications.

### 2.1.4 Inertial Sensors: Accelerometer, Compass and Gyroscope

Accelerometers, magnetic compasses and gyroscopes on today's mobile phones use low-energy MEMS (micro-electro-mechanical sensing) technology. These devices have a relatively low energy overhead. A significant portion of the overhead comes from CPU cost for processing inertial sensing data on the phone, rather than the sensor sampling cost. For example, ADXL 330 accelerometers use about 0.6 mW when sampling at the highest possible rate, and at 10 Hz can be idle about 90% of the time, suggesting a power overhead of around .06 mW for sampling the accelerometer [13]. The MicroMag3 compass uses about 1.5 mW when sampling at the maximum possible rate, suggesting a power consumption of .15 mW or less at 10 Hz [73].

Our experiments with battery life on the Android G1 have found that sampling the accelerometer and compass at 20 Hz add a negligible amount of energy consumption. In fact, running cellular localization, accelerometer and the compass all together yields a lifetime of over 60 hours, or about $10\times$ the lifetime when sampling GPS at 1 Hz. A different experiment we performed on the iPhone 3G-S and iPhone 4 (also described later) showed a similarly insignificant energy overhead from sampling the accelerometer at 20 Hz on the iPhone.

### 2.1.5 Other Energy Studies and Discussion

In [45], the authors showed that Nokia N95 phones use about 370 mW of power when GPS is left on, versus 60 mW when idling. They also found that 1 Hz GPS sampling results in 9 hours of total battery life. Several other papers [82, 60, 32, 37, 77] suggest similar numbers for N95 phones (battery life in the 7–11 hour range) with regular GPS sampling. On a more recent AT&T Tilt phone [50], the authors found that 1 Hz GPS sampling used 400 mW, a single GPS fix costs 1.4-5.7 J of energy (depending on whether previously downloaded satellite ephemeris information is cached or not) and a WiFi scan consumed about 0.55 J of energy. These numbers are consistent with our discussion above.

### 2.1.6 Summary

We summarize the discussion above:

- The theoretical power consumption of cellular scanning plus sensors on a phone should be less than 5 mW.
- The power consumption of inertial sensors alone should be less than 1 mW, low enough that it does not reduce the phone's overall lifetime even when in standby mode.

- In contrast, the best-case theoretical power consumption (as reported by datasheets) for GPS is 75 mW in "continuous tracking" mode when a fix is already acquired, but in practice is closer to 400 mW on most mobile phone chipsets, *when including the energy to periodically re-acquire GPS fixes*.

- WiFi scanning every second or two requires keeping the radio on, consuming 50-60% of the energy cost of GPS on some platforms. On other platforms like the iPhone 3G, WiFi localization is significantly cheaper than GPS.

## 2.2 Energy Experiments

### 2.2.1 Does GPS Sub-Sampling Save Energy?

Continuously sampling the GPS is energy-intensive and constitutes a dominant fraction of the energy cost for trajectory mapping. A natural question that arises is whether the strategy of *sub-sampling* i.e., querying the GPS for position estimates intermittently rather than at the maximum possible rate, can save energy.

Saving energy with sub-sampling depends on the ability of a phone to turn off, or *duty-cycle* its GPS unit when not in use. Unfortunately, the location sensing APIs of most current mobile smartphone platforms do not provide direct low-level control over the GPS sensor. For example, Apple's iOS does not even provide direct control over whether GPS is used or not for determining location. Instead, mobile phone APIs such as iOS and Android allow only indirect control of the GPS via the operating system's location API. A phone application can request location updates with a specified accuracy or at a specified time interval, and the operating system takes care of the low-level details of whether GPS is duty-cycled or not. Hence, the energy savings in practice from sub-sampling depend on both the chipset and the policy used by the operating system.

Our initial goal was to obtain an understanding of whether sub-sampling GPS can save energy in a *stand-alone* setting, independent of the quirks of the underlying operating system. To do this, we performed a simple experiment with a stand-alone GPS unit to measure how well current GPS sensors perform in a setting where we have direct control over them.

We used a Bluetooth GPS unit with an MTK GPS two-chip chipset (MTK MT3301 and MT3179). This device has a power switch. We measured the time for the device to power on and acquire a "warm fix" after it had previously obtained a "cold fix" and saved it. As mentioned earlier, this is the typical mode of operation when a GPS is duty-cycled frequently. We found that with a power-off time of anywhere from 5 seconds to 30 minutes, it took the GPS receiver about 6 seconds to power on, acquire a "warm fix" (i.e., verify all its cached satellite ephemeris data), and report the first reading over Bluetooth to a laptop computer.

Based on this experiment, it is possible to estimate the theoretical cost of GPS sub-sampling on a hypothetical mobile device that performs *perfect* GPS power management — by turning off the GPS immediately after a location sample is reported to the application, and turning it on 6 seconds before the application needs the next location sample. If such a hypothetical device were to acquire a GPS "warm fix" every $k > 6$ seconds, it would use approximately $\frac{6}{k}$ of the power of 1 Hz GPS sampling.

Our subsequent phone experiments, described next, measure the savings due to GPS sub-sampling on the iPhone 3G and the Android G1 phones. As we shall see, the two phones' implementations

| Location Mechanism | Sampling Period | Lifetime |
|---|---|---|
| None | - | 7 h |
| GPS | 1 sec | 2 h 24 m |
| GPS | 30 sec | 2 h 27 m |
| GPS | 2 min | 2 h 44 m |
| WiFi | 1 sec | 6 h 30 m |

**Table 2.1: Energy experiments comparing GPS and WiFi localization on an iPhone 3G.**

of GPS power management are very different. The iPhone 3G's firmware/OS — at least as of the time we performed our experiment — appears to do poor GPS power management, resulting in negligible or no savings due to GPS sub-sampling. In contrast, the Android G1 seems to perform good GPS power management, and sub-sampling the GPS actually saves energy in accordance with a formula similar to the one mentioned above.

### 2.2.2 Experiment on iPhone 3G

Our first experiment on actual phone hardware measures the energy consumption of GPS and WiFi localization on the iPhone 3G. We did not test cellular localization on the iPhone since we do not know of a way to programmatically retrieve neighbouring cell towers on the iPhone. It appears to be possible to retrieve the connected cell tower using an undocumented API, and a list of nearby cell towers when the phone is put in "field test" mode, but no other information appears to be available during normal operation.

We wrote a simple iPhone application that repeatedly requests location estimates at either GPS or WiFi accuracy, with a user-specifiable periodicity, until the battery drains to a fixed level from a full charge (in our experiments we drain the battery to 20%). The iPhone 3G produces a localization accuracy estimate for each location it reports to the application. The "vertical accuracy" field of this estimate has a non-zero (valid) value when the phone uses GPS localization, and has an invalid value when the phone uses WiFi or cellular localization to obtain a fix.

Because the iPhone 3G does not permit background applications (unlike the newer iPhone 4 which does), we ran this application in the foreground with the phone's screen turned on at minimum brightness. Therefore, we ran a control experiment and measured the total lifetime with the screen on at minimum brightness, but without requesting location estimates.

Our results are summarized in Table 2.2.2. On the iPhone 3G, GPS is extremely power hungry and reducing the sampling period does not reduce energy consumption appreciably. The reason appears to be that the iPhone OS leaves the GPS on for about a minute even when an application de-registers for position estimates, rather than duty-cycling it, perhaps to avoid the additional time delay incurred for a warm or cold start. Hence, sampling GPS every 30 seconds is as expensive as sampling once a second, and even obtaining one GPS sample every two minutes does not save a significant amount of power. In contrast, the iPhone seems to do a much better job of aggressively managing WiFi power consumption when no data is being sent and the radio is being used only for localization.

**Estimating WiFi Cost.** While GPS sub-sampling does not save a significant amount of energy on the iPhone 3G, WiFi localization does turn out to save a significant amount of energy over sampling GPS. We use the numbers in the table above to solve for the WiFi energy cost on the

| Sensors Used | iPhone 3G-S | iPhone 4 |
|---|---|---|
| No sensors | 18.6 hr (1.7) | 16.6 hr (0.8) |
| Accelerometer 1Hz | 19.5 hr (1.4) | 17.3 hr (1.8) |
| Accel. 20Hz | 18.3 hr (3.1) | 16.9 hr (0.8) |
| GPS | 6.1 hr (0.6) | 10.1 hr (0.3) |

**Table 2.2: Battery lifetime: GPS vs accelerometer on the iPhone.**

iPhone as a fraction of the GPS@1 Hz sampling cost. Suppose the battery has capacity $c$ Joules. The baseline lifetime, without any location estimates but with the screen on, is 7 hours (25,200 seconds). Therefore, the baseline (with screen on) power consumption $b = c/25200$ Watts. The lifetime with both GPS and the screen on is 8,640 seconds, so $g + b = c/8640$. Solving, we get $g = c/13147$. On doing a similar analysis for WiFi, we get $w = c/327360$ and $g/w = 24.9$—that is, the cost per sample of GPS is $24.9\times$ the cost per sample of WiFi. This also suggests that WiFi sampling is about 8 percent of the total power consumption when the phone is running continuously with the screen on (since $w/g = .08$), which means that a foreground application that samples WiFi is unlikely to dramatically alter the battery life of the iPhone 3G compared to a foreground application that does not. This is partly because WiFi is more energy-efficient, and partly because the additional energy consumption due to WiFi is drowned in the cost of keeping the screen switched on, even at minimum brightness.

### 2.2.3 Experiment on Multiple iPhones

The experiment in the previous section only measures GPS and WiFi energy cost on the iPhone 3G. We now describe the results of a second energy experiment on the iPhone platform that measures the energy cost of the accelerometer in comparison to GPS. This experiment was performed by the author of this dissertation in collaboration with other researchers for a project not part of this dissertation [5].

This experiment measures energy consumption of 1 Hz GPS sampling and accelerometer sampling at 1 Hz and 20 Hz on the iPhone 3G-S and the iPhone 4. As in the previous experiment, it was not possible to turn off the iPhone's screen and at the same time keep the sensors running. Hence, as before, all the experiments were run with screen brightness set to a minimum

Table 2.2.3 reports the measured battery life for four sensing configurations:

- No sensors (i.e., "stand-by" lifetime).
- Accelerometer sampled at 1 Hz.
- Accelerometer sampled at 20 Hz.
- Continuous GPS sampling.

Each number reported in the table is the mean of 5 measurements. The standard deviation of each number is indicated in parentheses next to it.

The table shows that accelerometry has a negligible effect on energy consumption: the battery duration was similar for running the accelerometer at 1Hz or 20 Hz, and for discharging the battery with no sensors turned on. There was considerable variance between multiple runs of the experiment.

The phone with accelerometry enabled having *higher* lifetime than without (as seen from the table) is an artifact of this variance.

We note here that the lifetimes in this experiment are not directly comparable to the results for the iPhone 3G in the earlier experiment. The lifetimes in this experiment are higher because the phone models we use — iPhone 3G-S and iPhone 4 — are newer models with improved battery life. Moreover, the earlier experiment measures time discharge to 20% of maximum battery life, while this experiment measures time to complete battery discharge.

### 2.2.4 Experiment on Android G1

We also performed an experiment to quantify the energy consumption of each of the sensors of interest — GPS, WiFi localization, cellular localization, magnetic compass and the accelerometer on an Android G1 phone. For each sensor, we wrote an Android application to sample the sensor at some given frequency, and query the battery level indicator periodically. We charged the phone to 100%, configured the screen to turn off automatically when idle (this is the default behaviour), and started the application. We used the Android `TelephonyManager` API to retrieve neighbouring cell towers and their associated signal strength values.

Figure 2-1 shows the remaining battery life reported by the Android OS as a function of time for four sensor sampling configurations:

- GPS sampled every second.
- GPS sub-sampled every two minutes.
- WiFi scanning at the max possible rate, every 2 seconds.
- A configuration using three sensors: logging GSM cell towers every second, compass at 20 Hz, and accelerometer at 20 Hz.

The last configuration above measures the cost of cellular scanning bundled with the compass and accelerometer because the experiment was originally performed in the context of evaluating *CTrack*, the trajectory mapping algorithm discussed in Chapter 4 that uses cellular localization in conjunction with "hints" from the accelerometer and the compass for trajectory mapping.

We highlight three key points from Figure 2-1:

- **Cellular localization is extremely energy-efficient.** Even when also simultaneously sampling the compass and accelerometer, cellular (GSM) localization results in a saving of approximately 10× in battery life compared to sampling GPS every second, and over 6× compared to WiFi. The lifetime of over 60 hours is close to the stand by lifetime of the Android G1 (not shown in the figure) which is a little over 3 days.
- **GPS sub-sampling saves energy.** Unlike the iPhone 3G, the Android G1 seems to have a sensible GPS power management policy, appearing to duty-cycle the GPS when it is not in use. For example, we see that sub-sampling the GPS every 2 minutes saves a significant amount of battery life compared to sampling it once a second.

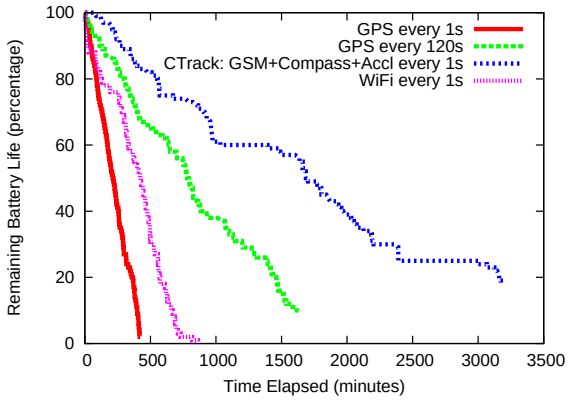



**Figure 2-1: Energy consumption: GPS vs WiFi vs cellular on an Android phone.**

- **WiFi saves some energy compared to GPS.** On the Android G1, WiFi localization is cheaper than GPS, with a battery lifetime of over 10 hours, compared to only 6 hours for GPS sampled at 1 Hz. This is significant, but not as much of a saving as in the iPhone 3G.

The battery drain curves in Figure 2-1 look irregular because the G1 phone does not estimate remaining battery life accurately. We performed a similar experiment on a Nexus One, a more recent Android phone model. The experiment on the Nexus One showed a very similar trend in terms of relative costs of the different sensors, but the battery drain curves looked like straight lines.

## 2.3 Conclusion

The energy consumption profiles of different sensors are different on the different phones we tested. For example, because the Android G1 permits background applications to run without the screen having to be kept switched on, WiFi sampling in the background on the G1 *does* significantly alter the G1's lifetime, unlike the iPhone 3G experiment where WiFi sampling cost was drowned out by the power consumption of the screen. Similarly, because the GPS power management policy is different on the iPhone 3G and the Android G1, the energy savings due to GPS sub-sampling are very different on these two platforms.

The main lesson is that an algorithm suitable for energy-efficient trajectory mapping on one platform may not be energy-efficient on a different platform, and vice versa. For example, an algorithm based on GPS sub-sampling may be suitable for the Android G1 phone but not for the iPhone 3G. Therefore, different platforms require different algorithms for energy-efficient trajectory mapping. The trajectory mapping strategy described in this dissertation adapts to different platforms using an *offline optimization* strategy, where a battery drain experiment similar to the ones presented in this chapter is performed for each distinct phone model or platform we want to optimize for. The experiment estimates the *relative* energy costs for each sensor (GPS, WiFi, or Cellular) on that platform. Chapter 3 discusses how the measured relative energy costs from such an offline experiment can be used as input to an *sensor selection framework* that decides what sensor(s) and sampling rate(s) to use on that platform to achieve a desired level of trajectory mapping accuracy.

# Chapter 3

# Map-Matching With Markov Models

The energy studies presented in Chapter 2 highlight two fundamentally different ways to reduce the energy requirement for trajectory mapping:

- Sub-sample GPS, i.e., obtain a GPS location sample every $k$ seconds for some value of $k$.
- Use GPS alternatives: WiFi localization, cellular localization, and/or inertial sensing.

One can also imagine strategies that use a combination of the above approaches.

With sub-sampling, the trajectory mapping system needs to process a stream of *infrequent* position samples and map them to an accurate trajectory. If using GPS alternatives, the key challenge is to process a stream of *inaccurate* position samples and map them to an accurate trajectory, since WiFi and cellular localization are typically much less accurate than GPS.

This chapter presents *VTrack* [7], a trajectory mapping system for the outdoor setting. *VTrack* uses a Hidden Markov Model (HMM) to accurately match a stream of *infrequent and/or inaccurate* position samples to a sequence of road segments on a map.

*VTrack* is most suitable when raw localization data is already in the form of geographic $(lat, lon)$ coordinates — either GPS coordinates obtained from sub-sampling, or coordinates obtained by looking up WiFi or cellular radio signals in a training database. If more detailed "soft" information such as WiFi or cellular base station sightings and their signal strengths is available, *CTrack*, the algorithm presented in the next chapter, is a better strategy than converting the soft information to coordinates and processing it with *VTrack*. However, *VTrack* is very useful in practice because soft information is not available in many situations. For example, Apple's documented iPhone API does not provide low-level access to WiFi or cellular signal strengths — WiFi and cellular localization can only be accessed as $(lat, lon)$ coordinates indirectly by querying a location API.

*VTrack* produces good trajectory estimates with coordinates from either GPS, sub-sampled GPS, or WiFi localization. It has been deployed as part of a *real-time traffic monitoring* system developed as part of the Cartel project [17], which uses smartphones as *traffic probes*, as do other systems [1, 55, 17, 71]. Unlike professional fleets and tracks which contribute to much of the currently available commercial traffic data, end users travel on the roads and at the times that are most useful to monitor for the purpose of reducing end-user commute duration. The idea is to obtain

timestamped position estimates from smartphones carrying GPS, WiFi or cellular radios, and deliver the estimates to a central server. The *VTrack* server processes position estimates to estimate *driving times on individual road segments of the road network*.

In the context of a traffic monitoring system, it is possible to give a precise characterization of the accuracy requirement for trajectory mapping. The end goal of *VTrack* is to produce an accurate sequence of road segments traversed by a vehicle, and accurate travel times for individual segments in the output path. In other words, it is important to be able to identify the road segment that each position sample came from accurately, and to estimate travel times accurately.

*VTrack* uses a probabilistic model, called a Hidden Markov Model, to model a vehicle trajectory over a detailed road map of the area, and performs *map-matching*, which associates each position sample with the most likely point on the road map, producing travel time estimates for each traversed road segment. The travel times are shared with other users of the system who would like to view or use the real-time traffic data.

We evaluate *VTrack* on a data set of GPS and WiFi location samples from nearly 800 hours of drives gathered from 25 cars. The data was gathered from two sources: an iPhone application, and from embedded in-car computers equipped with GPS and WiFi radios. We use a cleaned version of the GPS data without outliers and outages, sampled every second, to estimate the ground truth for these drives with high confidence, and use the ground truth in our evaluation.

The main question we investigate in the evaluation is how good the *end-to-end* accuracy of *VTrack* is. In particular, *how does the accuracy of travel time estimates depend on the sensor(s) being sampled and the sampling frequency?*

We answer this question by evaluating *VTrack* in the context of two applications that use the travel time estimates produced by the system. The first application reports *hotspots*, or roads with travel times far in excess of that expected from the speed limit. The second is a traffic-aware route planner that finds paths with the shortest expected travel times using the road segment travel times estimated by *VTrack*. Our key findings are:

- *VTrack*'s HMM-based map-matcher is robust to noise, producing trajectories with median error less than 10% when run over noisy WiFi localization data, as well as for simulated Gaussian noise with standard deviation up to 40 metres.
- *Travel times from WiFi localization alone are accurate enough for route planning, even though individual segment estimates may be poor.* When location samples are noisy, it is difficult to attribute a car's travel time to small stretches of road—for example, time estimates from WiFi for individual segments have a median error of 25%. However, somewhat counter-intuitively, using these times to find shortest paths works well—over 90% of shortest paths found using WiFi estimates have travel times within 15% of the true shortest path. This is because groups of segments are typically traversed together, and *VTrack*'s estimation scheme ensures that errors on adjacent or nearby segments "cancel out." *Moreover, estimating real drive times actually matters—for congested scenarios, using just speed limits to plan paths yields paths that are up to 35% worse than optimal.*
- *Travel times estimated from WiFi localization alone cannot detect hotspots accurately, due to outages present in WiFi data.* We find that a hotspot detection algorithm based on *VTrack* misses many hotspots simply because of a lack of data on those segments. This problem is not as apparent in route planning, since in that scenario we are focused on choosing a path that

has a travel time closest to the shortest path, rather than worrying about particular segments. However, on the subset of segments for which we *do* have WiFi data, we are able to accurately detect more than 80% of hotspots, and flag fewer than 5% incorrectly.

- *When GPS is available and free of outliers, sampling GPS periodically to save energy is a viable strategy for both applications.* On our data, for up to $k = 30$ seconds (corresponding to roughly 2 or 3 road segments), sampling GPS every $k$ seconds produces high-quality shortest paths, assuming GPS is available whenever it is sampled. If the device also has WiFi, the tradeoff between sampling GPS every $k$ seconds, sampling WiFi or a hybrid strategy depends on the energy costs of each sensor, as mentioned in Chapter 2. Later in the chapter, we discuss this tradeoff and present a sensor selection framework for *VTrack* that makes this decision based on energy costs measured offline.

Using Hidden Markov Models for map matching is not a new idea [36, 44]. However, previous research has focused mainly on map matching frequently-sampled GPS data with low noise, and on qualitative studies of accuracy. A key contribution of our work on *VTrack* is a quantitative evaluation of the end-to-end quality of time estimates from noisy and sparsely sampled locations.

The rest of this chapter is organized as follows. Section 3.1 provides background on traffic monitoring, explaining the motivation and original context for *VTrack*. Section 3.2 describes two key applications that use the travel time estimates produced by *VTrack* and uses them to concretize the "how accurate does map-matching need to be" question. Section 3.3 describes the architecture of the *VTrack* trajectory mapping system. Section 3.4 formally states the map-matching problem and describes the *VTrack* algorithm. This section describes the Hidden Markov Model used to model a car's trajectory and solve for the best match to a map. Section 3.5 explains how *VTrack* estimates travel times for individual segments from the output of trajectory matching.

Section 3.6 evaluates *VTrack* using two kinds of accuracy measurements:

- End-to-end accuracy measurements on the two real applications mentioned earlier, route planning and hotspot detection.
- Micro-benchmark evaluations of how robust *VTrack*'s Hidden Markov Model is to varying levels of simulated noise in its input data.

Section 3.7 uses the results of Section 3.6, mainly from the end-to-end applications, to revisit and provide some answers to a central question of this dissertation:

- *What is the best strategy for energy-efficient trajectory mapping given an end-to-end accuracy requirement?*

We answer the question of what sensor(s) and sampling rate(s) should be chosen as input to *VTrack* to achieve a given accuracy, given a set of energy cost measurements of sensors on the mobile device being used.

Section 3.8 describes related work to *VTrack*, and Section 3.9 concludes this chapter.

## 3.1 Background: Traffic Monitoring

Traffic congestion is a serious problem facing the road transportation infrastructure in many parts of the world. With close to a billion vehicles on the road today, and a doubling projected over the next decade [25], the excessive delays caused by congestion show no signs of abating. Already, according to much-cited data from the US Bureau of Transportation Statistics, 4.2 billion hours in 2007 were spent by drivers stuck in traffic on the nation's highways alone [20], and this number has increased by between $3\times$ and $5\times$ in various cities over the past two decades. In addition, various surveys from news organizations show that in the developed world, dependence on roads and cars remains very high. For example 90% of US workers drive to work, the typical metro commuter spends on average 100 minutes driving per day, 33% of commuters in US cities are stuck in heavy traffic at least once a week, and worst-case delays are more than double the average delay [10, 9].

*Real-time traffic information*, either in the form of travel times or vehicle flow densities, can be used to alleviate congestion in a variety of ways: for example, by informing drivers of roads or intersections with large travel times ("hotspots"); by using travel time estimates in traffic-aware routing algorithms to find better paths with smaller expected time or smaller variance; by combining historic and real-time information to predict travel times in specific areas at particular times of day; by observing times on segments to improve operations (e.g., traffic light cycle control), plan infrastructure improvements, assess congestion pricing and tolling schemes, and so on.

An important step for all these tasks is the ability to *estimate travel times on segments or stretches of the road network*. Over the past few years, the idea of using *vehicles as probes* to collect travel time data for roads has become popular [1, 40, 71]. Here, vehicles equipped with GPS-equipped mobile devices or embedded computers, or individual commuters carrying mobile smartphones, log the current time and position periodically as they travel, sending this data to a server over some wireless network. This approach is better than flow-monitoring sensors, such as inductive loops, deployed on the roadside because vehicles can cover large areas more efficiently.

Estimating travel times on individual segments of the road network requires energy-efficient and accurate map-matching, which is the subject of the algorithms described in this chapter. In the crowd-sourced, mobile phone-based traffic monitoring context, keeping the GPS on all the time is not a viable strategy owing to energy concerns. If all users keep their phones powered while driving, then energy isn't as much of a concern, but imposing that constraint is unreasonable barrier to large-scale deployment of a "crowd-sourced" traffic monitoring system. As we have discussed, energy concerns force the use of either GPS sub-sampling, or lower energy sensors such as WiFi or cellular radios, but these are accurate only to tens or hundreds of meters. Travel time estimation is a challenging problem in this context because *the closest road in the map to a position sample is often not the road that a vehicle actually drove on*.

We note that using smartphones as traffic probes raises some privacy concerns, which are out of scope for this dissertation; see Section 3.8 for a discussion of other work that addresses this concern.

To give a flavour of how *VTrack's* estimates are actually used, the next section discusses two key applications that use travel time estimates: *hotspot detection* and *traffic-aware route planning*. This also helps gain a more precise understanding of how accurate the trajectories produced by *VTrack*, and the travel time estimates for these trajectories, need to be.

## 3.2 Applications

### 3.2.1 Detecting And Visualizing Hotspots

We define a "hotspot" to be a road segment on which the observed travel time exceeds the time that would be predicted by the speed limit by some threshold. Hotspots are not outliers in traffic data. They occur every day during rush hour, for example, when drivers are stuck in traffic. The goal of hotspot detection is to detect and display all the hotspots within a given geographic area which the user is viewing on a browser. This application can be used directly by users, who can see hotspots on their route and alter it to avoid them, or it can be used by route avoidance algorithms, to avoid segments that are frequently congested at a particular time.

For hotspot detection, the travel time estimates produced by a trajectory mapping algorithm need to be accurate enough to keep two metrics numerically low:

- The *miss rate*, defined to be the fraction of hotspot segments that the algorithm fails to report.
- The *false positive rate*, the fraction of segments reported as hotspots that actually aren't.

If the trajectory mapping algorithm maps one or more input location estimate(s) to the wrong road segments, this may increase *both* the miss rate and the false positive rate for hotspot detection.

We give a simplified, but realistic, example (one that occurs quite often in the real data we have collected) to illustrate this point. Suppose the trajectory mapping algorithm needs to map a sequence of noisy WiFi location samples to one of two stretches of a freeway: A or B, with the segments lying on either side of a major exit on the freeway. Suppose in addition that what really happened was that segment A was congested — the vehicle being mapped spent a significant amount of time on segment A, *before* the exit, and sped up on segment B, immediately after traffic cleared at the exit. However, if the trajectory mapping system is confused by the error in the WiFi location samples and maps most or all of them to segment B, *after* the exit, then segment B will be flagged incorrectly as a hotspot instead of segment A. This causes an increase in the miss rate because the algorithm missed flagging segment A, and an increase in the false positive rate because the algorithm flagged segment B as a hotspot incorrectly. The consequence is that a route avoidance algorithm using the hotspot data would try to avoid segment B and perhaps route cars via segment A, which is ineffective and probably counter-productive (since the cars would get stuck on segment A in any case).

### 3.2.2 Traffic-Aware Route Planning

With the exception of hotspots, users are generally more concerned about their total travel time, rather than the time they spend on a particular road. Traffic-aware route planning that minimizes the expected time is one way for users to find a good route to a particular destination. Also, real-time route planning is useful because it allows users to be re-routed mid-drive.

Route planning using time estimates produced by *VTrack* has been deployed as part of the *iCartel* iPhone app as well as the *Carweb* personal driving portal, and has been running live for nearly two years now, providing route guidance to users of these applications. iCartel shows the user his/her position, nearby traffic, and offers a navigation service, while continuously sending position estimates back to a set of central server(s). The Carweb personalized driving portal is a website which shows users their own drives to/from their workplace and home, and also allows users to

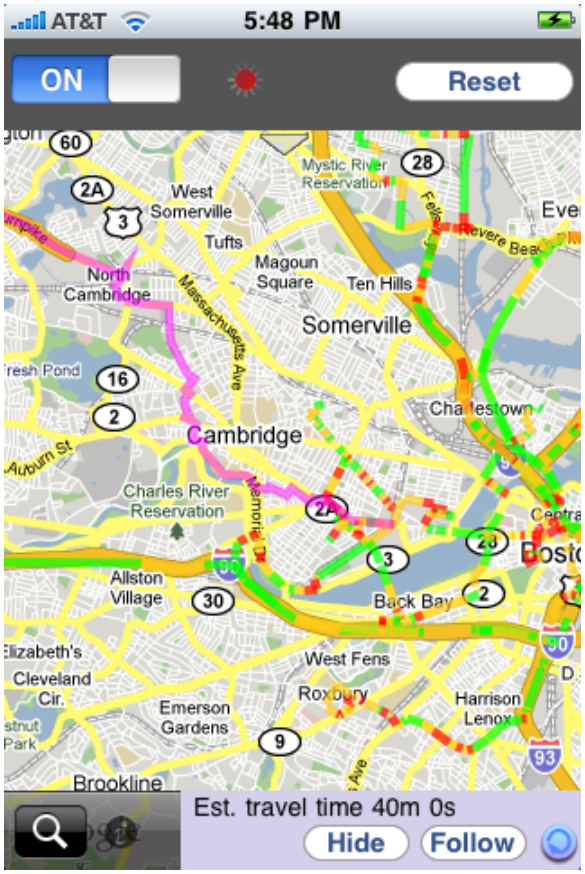



**Figure 3-1: iCartel application showing real-time delays and congestion information.**

view current traffic delays and receive route planning updates. Figure 3-1 shows a screenshot of the iCartel iPhone app and Figure 3-2 shows a screenshot from Carweb.

The goal of many route planning systems is to provide users with routes that minimize expected travel times. To meet this goal, *how accurate do the trajectories mapped need to be?* To analyze how accurate a set of travel time estimates are in the context of route planning, we use the travel time estimates to compute shortest paths in the road network — in terms of time — and study *how much worse* the shortest paths found are when compared to the true shortest paths, assuming the ground truth travel time is known. This is a realistic measure of how the quality of trajectory mapping estimates impacts the quality of end-to-end route planning. We believe that errors in the 10-15% range are acceptable — corresponding to at most a 3 to 5 minute estimation error on a 30 minute drive.

A second requirement is that the map-matching algorithm be efficient enough to run in real time as data arrives. Some existing map-matching algorithms run A*-style shortest path algorithms multiple times per point, which we found to be prohibitively expensive.

## 3.3 *VTrack* Architecture

Figure 3-3 shows the architecture of the *VTrack* system. Mobile smartphones report position data to the *VTrack* server periodically while driving. *VTrack* works with GPS and WiFi position estimates

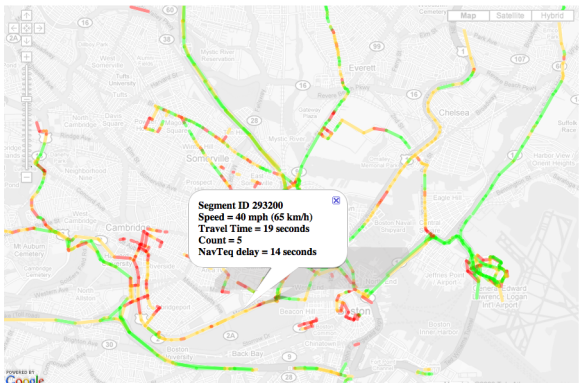



**Figure 3-2: Web application showing traffic delays.**

in the form of $(lat, lon)$ coordinates at some (constant or variable) sampling interval. The server runs a Hidden Markov Model and travel-time estimation algorithm that uses these noisy position samples to identify the road segments on which a user is driving and estimate the travel time on these segments. The estimates are then used to identify hotspots and provide real-time route planning updates back to participating phones, as well as to a web service that provides traffic updates. The real-time estimates are also periodically logged to a database that stores historical delays. The historical travel time database is used in concert with real-time information to perform *traffic prediction:* i.e. predict what travel times might look like at a given time of day, or five minutes from now. Traffic prediction algorithms have been developed and are running live on the *VTrack* platform, but they are out of the scope of this dissertation.

Figure 3-4 shows the *VTrack* server in more detail. *VTrack* uses WiFi for position estimation as follows. Access point observations from WiFi in smartphones are converted into position estimates using a "centroid localization" algorithm that we shall describe shortly. This algorithm uses a training (also known as "war-driving") database of GPS coordinates indicating where WiFi access points been observed from in previous drives. Positions from these sensors are fed in real-time to our estimation algorithm, which consists of two components: a *map-matcher*, which determines which roads are being driven on, and a *travel-time estimator*, which determines travel times for road segments from the map-matched trajectory.

## 3.4 The *VTrack* Algorithm

This section formally states the map-matching problem and then describes the Hidden Markov Model-based algorithm used in *VTrack*.

### 3.4.1 Problem Statement

The trajectory mapping problem in *VTrack* is the problem of matching input location samples to output road segments. Given the following as input:

- A known map of the road network that contains the geography of all road segments in an area of interest — examples include OpenStreetMap [68], and NAVTEQ [66]. While we have implemented *VTrack* on both map platforms, all the results in this dissertation use the NAVTEQ road map.

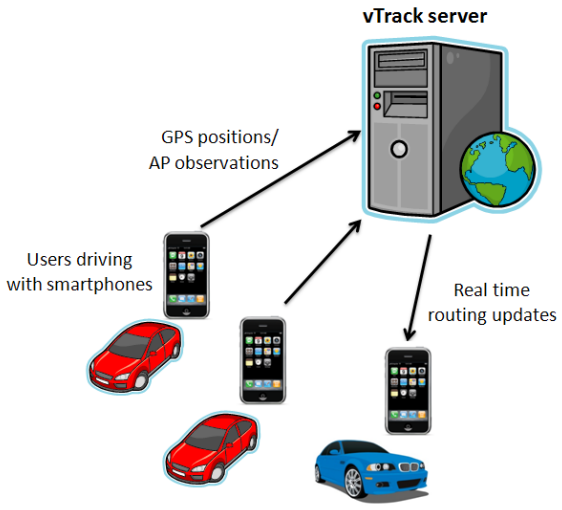



**Figure 3-3: VTrack system architecture.**

- A sequence of infrequent and/or inaccurate location coordinates with a possibly varying time interval between samples.

The goal is to produce the following as output:

- The most likely sequence of road segments traversed in the network. A given input point is matched to exactly one road segment. Multiple consecutive input points can be matched to the same road segment.

When our goal is to produce accurate travel time estimates from *sparse* input data, we need to modify the above definition slightly. If the location samples input to map-matching are very sparse (e.g., separated by minutes), then producing one road segment for each input location sample may not be sufficient to produce an unbroken sequence of road segments as output, especially if some of the road segments in the network are small segments.

In these cases, it is desirable to *interpolate* the output in some way to produce an unbroken sequence of road segments as output. Here, "unbroken" means each segment in the output sequence must be identical to, or adjacent to the next segment in the sequence. With this in mind, we use the following, slightly modified definition of "output" in *VTrack*:

- The most likely continuous sequence of road segments (guaranteed to be adjacent) traversed in the road network. A given input point is matched to exactly one road segment, but there can exist segments in the output not associated with any input point.

With the above modified definition, the system can (and often does) output *interpolated* road segments not corresponding to any particular input sample, especially if samples are spaced far apart.

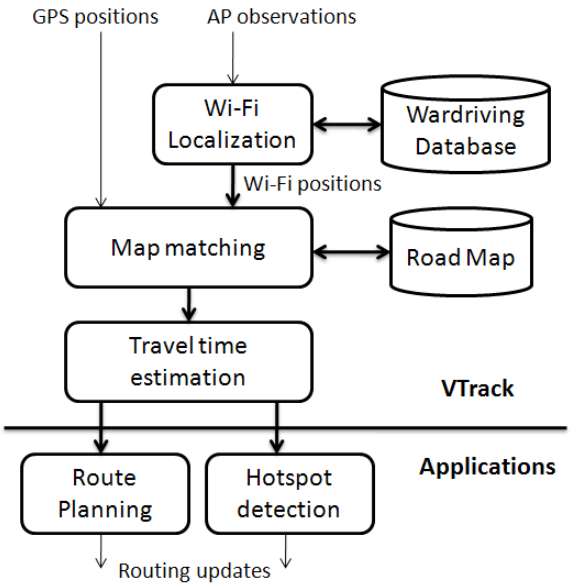



**Figure 3-4: VTrack server.**

### 3.4.2 Modeling Error In Location Samples

The input to *VTrack* is in the form of geographic $(lat, lon)$ coordinates, irrespective of whether they are obtained from GPS, WiFi or cellular localization. *VTrack* also uses a simple *Gaussian* model of error in all of its input location samples irrespective of the localization technology used to compute them. It uses a different variance for the Gaussian varies based on which technology is used (smaller for GPS and larger for WiFi and cellular).

While the Gaussian model of error may be reasonably accurate for GPS [33], the error distribution of WiFi and cellular "location samples" depends on the procedure used to compute them. In the case of radio localization, the lowest-level raw data is usually in the form of a *radio signature* — a set of WiFi or cellular base station observations and their signal strengths. How this data was converted to geographic coordinates will determine what the errors in the coordinates are.

Our deployment of *VTrack* uses WiFi localization data from phones and from Linux-based access points re-purposed as in-car embedded devices. On the iPhone, WiFi position samples were obtained using Apple's location API. On the embedded devices, WiFi scanning data was converted to coordinates using an averaging procedure that we call *centroid localization*. The procedure relies on a training database of access points and their signal strengths, and the ground truth GPS locations they were seen from. First, an approximate location is computed for each access point by finding the centroid of *all* GPS points in the training database from which that access point was ever seen. A given WiFi signature is converted to a location estimate by computing the "centroid of centroids", i.e., the centroids of the individual access point centroid locations in the signature. The training database used for centroid localization was built over a period of over one year using 25 taxicabs part of the Cartel [17] testbed. Each taxicab in the testbed was equipped with an embedded Linux access point and a GPS device.

If there are multiple locations in the training database from which a WiFi signature can be seen, all far apart, then the averaging process in centroid localization is likely to introduce significant error. In particular, a single stray sighting of an access point from a far off location can introduce a large

error in the centroid estimate. Even if the Gaussian assumption no longer holds true, in practice, we find that in practice, *VTrack*'s Hidden Markov Model is able to correct errors in WiFi location estimates, and yields fairly accurate trajectories when matching location samples computed by the centroid algorithm.

### 3.4.3 Hidden Markov Models

In this section, we provide a brief introduction to Hidden Markov Models and explain how they apply to the map matching problem.

The simplest way to understand HMMs and how they apply to map-matching is the following. Suppose we first ignore the problem of infrequent location samples, and assume that location samples are frequent enough that it is sufficient to map each input location sample to an output road segment to produce an unbroken sequence of output segments. In this scenario, the simplest approach to map matching would be to map each position sample to the nearest road segment. However, as we show in Section 3.6.5, this scheme fails even in the presence of small amounts of noise in the input location estimates. What we want is not an approach that matches individual location samples, but an approach that matches a *sequence* of samples and exploits constraints on the transitions a moving vehicle or phone can make between the location measurements. This is exactly what a Hidden Markov Model enables.

A Hidden Markov Model (HMM) is a discrete-time *Markov process* with a discrete set of *hidden states*. The HMM takes as input a sequence of *observables*. Each observable is modeled as coming from an *emission probability distribution*, which is the probability distribution of the observable conditioned on the (unknown) hidden state. A HMM also permits *transitions* between its hidden states at each time step, governed by an adjacency graph that specifies which hidden states can transition to which other hidden state(s). These transitions are also governed by a different set of probability distributions, called *transition probabilities*.

In the context of *VTrack*:

- The hidden states are the (unknown) road segments a vehicle drove on that we want to identify.
- The observables are raw latitude and longitude position samples from GPS or WiFi.
- The emission probability for a given (segment, position) pair $\langle S, P \rangle$ pair represents the probability distribution of seeing position sample $P$ given that the vehicle is on road segment $S$.
- The transition probability for a given pair of segments $\langle S1, S2 \rangle$ represents the probability of transitioning (driving) from segment $S1$ to segment $S2$.

A Markov Model satisfies the following important independence property: the probability distribution of the observable at a given time $t$ is conditionally independent of the distribution of past and/or future hidden states conditioned on the hidden state at time $t$. This makes it tractable to find the *most likely sequence* of hidden states corresponding to a given sequence of input observables. Due to conditional independence, the likelihood of any sequence of a hidden states can be analytically shown to be equal to the product of emission and transition probabilities for that sequence. Hence, the most likely sequence of hidden states is simply the sequence that maximizes the product of emission and transition probabilities.

This most likely sequence of hidden states can be found exactly using an efficient *dynamic programming* technique called Viterbi decoding without having to exhaustively search all the possible paths through the HMM [4].

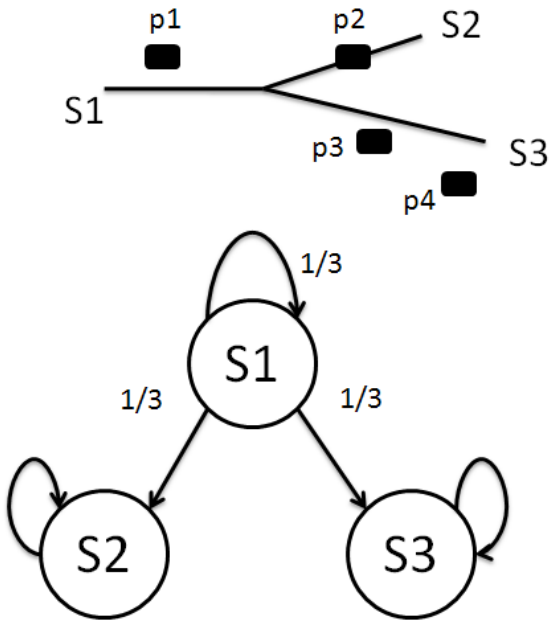



**Figure 3-5: Example illustrating an HMM.**

Figure 3-5 shows an example of an HMM where S1, S2, and S3 are road segments and p1, p2, p3, and p4 are position samples. There is an equal probability of transitioning from S1 to S1, S2, or S3. Because the emission probability density function is a decreasing function of distance from a road segment, assuming the transition constraints as shown in the state diagram, the maximum likelihood sequence of segments for the given sequence of position samples is S1, S3, S3, and S3. Hence, *although p2 is closer to S2*, the most probable hidden state of the point is S3 given the transition constraints.

As this simple example illustrates, a HMM makes sense for map-matching noisy location data because:

- It is robust to noisy position samples that lie closer to a *different* road segment than the one from which they were observed.
- It is able to guarantee a continuous (unbroken) output trajectory using transition constraints, unlike techniques that match individual samples to a sequence of output segments that may or may not be adjacent.

The HMM used in *VTrack* differs from previous HMM-based approaches to map-matching [36, 44] in two key ways:

- How it handles *gaps* and outages in the input data.
- How it models transition probabilities between road segments.

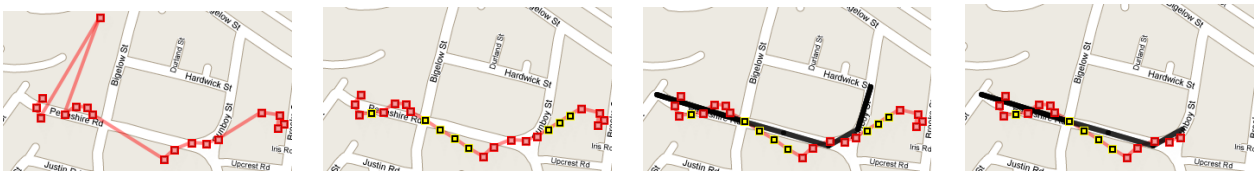



**Figure 3-6: The map-matching process used by *VTrack*. A raw location trace is gradually refined to produce a final, high quality street route. In this example, due to noise and outages, only the first three segments produced quality estimates.**

An additional difference is that *VTrack* uses *both* pre-processing and post-processing in addition to a HMM: it pre-processes input position samples to remove outliers and post-processes the HMM output to remove low-quality matches. This prevents it from producing inaccurate travel time estimates, as we shall explain below.

The following sections discuss the various stages of the *VTrack* algorithm: pre-processing, Viterbi decoding, and post-processing. Figure 3-6 provides a visual illustration of the stages of *VTrack*. We use this illustration as a running example through the sections that follow.

### 3.4.4 Pre-Processing and Outlier Removal

Prior to using a Hidden Markov Model, *VTrack* pre-processes its input location data to eliminate outliers. We say a location sample $p$ is an *outlier* if it violates a speed constraint; that is, for some threshold speed $S_{outlier}$, the car would have had to travel faster than $S_{outlier}$ kmph to travel from the previous sample to $p$ in the observed time between $p$ and the previous sample. We choose $S_{outlier} = 400$ km per hour, well over twice the maximum expected vehicle speed on any road. This threshold is intentionally conservative and accommodates for errors in subsequent noisy GPS or WiFi samples.

After outlier removal, the next challenge is dealing with gaps in the input data. Gaps in input location data can occur due to two reasons:

- *Outages*, which occur when the data is missing. For example, GPS location data can be unavailable when the vehicle being tracked is in a tunnel or "urban canyon" with tall skyscrapers, and WiFi localization can experience outages when a mobile device is unable to find any access points in its vicinity that have been previously seen in the training database.
- *Sub-sampling*. If using a strategy such as GPS sub-sampling to save energy, the mobile device intentionally duty-cycles its GPS unit and only provides a location sample to *VTrack* sporadically, e.g., one GPS sample every 30 seconds or every minute.

*VTrack* uses a simple, computationally efficient, pre-processing scheme to deal with *all* gaps in its input location data — whether they be from outages or from sub-sampling. It uses *linear interpolation* to insert interpolated points in regions where location data is missing. The algorithm generates interpolated samples at a pre-determined interval along the line segment connecting the last observed point before the gap and the first following the gap, assuming a constant speed of travel along this line. The interpolated points are then fed to the Hidden Markov Model along with

the sampled points, and the HMM has the (relatively) easier task of matching each input location sample to a unique hidden state (road segment).

The pre-determined interpolation time interval used by *VTrack* is governed by the need to ensure the frequency of input to the HMM exceeds a threshold. Specifically, there is a danger that if the time between interpolated points is too high, the HMM cannot accurately match candidate paths that pass through small road segments. The input frequency must be high enough to ensure that even if driving through the *smallest* road segment in the road network database, at least one input point is associated with that segment. The smallest segment in the OpenStreetMap [68] and NAVTEQ road maps we use is roughly 30 metres. Assuming a maximum speed of 65 mph = 105 kmph = 29 m/s, the minimum frequency required is about once-a-second. Higher speeds than 105 kmph generally occur on freeways where segments are almost always longer than 30 metres. For this reason, *VTrack*'s linear interpolation scheme uses an interpolation interval of 1 second.

One limitation of linear interpolation is that it can fail in some situations where the interval of time between interpolated points is high. For example, suppose there are two points A and B in the road network with two candidate paths connecting them. One path (C) is curved while the other path (S) is straight. If *VTrack* is given a location sample at A, and then its next sample only at B, linear interpolation will always assume the vehicle drove on the straight path S even if the true path happened to be C. In this kind of scenario, a technique that computes the most likely path from A to B based on other external knowledge (e.g., an A* shortest path, or a path that drivers actually take most often in the road network) may work better than linear interpolation.

In practice, we have found that if using sub-sampled GPS, the *VTrack* HMM matches a straight-line interpolation to the *approximate shortest path* between the points being interpolated quite well for short gaps. This is because vehicles typically tend to follow shortest paths, *especially* at short time scales (a few seconds to a minute).

The outlier removal and interpolation steps used in *VTrack* are illustrated in the second panel of Figure 3-6 on an example trajectory. Interpolated points are shown in green.

We note that previous work that uses HMMs for map-matching either does not consider gaps [36], or deals with them by performing multiple computationally expensive shortest path calculations, or by invoking a route planner [44] to find the best transitions in the HMM. This is a computationally intensive process compared to linear interpolation.

### 3.4.5 Viterbi Decoding

Once outliers have been removed and interpolation has been applied, the Viterbi algorithm is used to determine the most likely sequence of road segments corresponding to the observed and interpolated points (third panel of Figure 3-6).

The output of *VTrack* depends quite critically on the choice of emission and transition probabilities for the HMM. We now explain our choice of emission and transition probabilities.

**Emission Probabilities.** The emission probabilities used in *VTrack* reflect the notion that it is more likely that a particular point was observed from a nearby road segment than a segment farther away, while not necessarily forcing the map-matcher to always output the closest segment to that point. Concretely, the emission probability density of segment $i$ at position sample $\ell$ is $N(\text{dist}(i, \ell))$ where $N$ is a Gaussian distribution with zero mean, and $\text{dist}(i, \ell)$ is the Euclidean distance between

$i$ and $\ell$. The standard deviation of $N$ depends on the sensor that produced the sample. We use different standard deviations for GPS and for WiFi because they have different error distributions. For GPS samples, we use a standard deviation of 10 metres (as measured in previous studies of GPS accuracy), and for WiFi localization estimates, we use 50 metres (this number was obtained by computing the average error of WiFi localization over a subset of empirical data). The absolute value of the variances used does not matter (except for numerical precision) because the Viterbi decoder only uses the emission probabilities to *compare* the overall probabilities of paths, rather than using the absolute values of the probabilities.

**Transition Probabilities.** The transition probabilities in *VTrack* reflect the following three notions:

- For a given road segment, there is a probability that at the next location sample, the car will still be on that road segment.
- A car can only travel from the end of one road segment to the start of the next if it uses the same intersection (taking into account one-way streets). This constraint ensures that the output road segments form a continuous path through the road network.
- A car cannot travel unreasonably fast on any segment.

We define the transition probability $p$ from a segment $i$ at sample $t-1$ to a segment $j$ at sample $t$ as follows:

- If $i = j$, $p = \epsilon$ (defined below).
- If $j$ does not start where $i$ ends, $p = 0$.
- If $j$ starts where $i$ ends, $p = \epsilon$ or 0 (this reflects the third notion, and is explained below).

An important point to note here is that *VTrack* sets *all* of the non-zero transition probabilities to a constant. The reason we do this is to avoid preference for routes with low-degree segments — routes with intersections without too many outgoing roads. An alternative approach, used by some other map matching algorithms such as [36], is to partition one unit of probability between all the segments that start at the end of $i$. This approach seems intuitive but results in overall higher transition probabilities at low-degree intersections than at high-degree intersections, which is *not* a faithful model of the underlying process. Section 3.6.7 of our evaluation shows that the partitioning approach has poorer map-matching accuracy than setting all the non-zero transition probabilities to a constant value.

Therefore, we want to ensure that the non-zero transition probabilities are constant across *all segments*, but at the same time sum to 1 for a given segment. To model this, we use a dummy "dead-end" state $\emptyset$ and a dummy observable $\perp$. We set $\epsilon$ to be $\leq 1/(d_{\max}+1)$, where $d_{\max}$ is the maximum number of segments that start at the end of the same segment (i.e., the maximum out-degree of a graph with intersections as vertices and road segments as edges). We set the transition probability from $i$ at $t-1$ to $\emptyset$ at $t$ so that the sum of the transition probabilities for segment $i$ at $t$ normalizes to 1. The transition probability from $\emptyset$ at $t-1$ to $\emptyset$ at $t$ is 1 — thus effectively assigning zero probability to all paths that transition to $\emptyset$.

The third item in the transition probability definition reflects the fact that it is desirable to prohibit cars from traveling unreasonably fast on any segment. If we are considering a transition from segment $i$ to segment $j$, *VTrack* calculates the time it would have taken the car to travel from $i$ to $j$, based on the times at which the positions were observed. If this time implies that the car would have had to travel at higher than a certain threshold speed, $S_{outlier}$, we conclude that this transition is impossible and assign it a probability of 0; otherwise, the transition is possible and we assign it a probability of $\epsilon$. As mentioned before, we use a relatively relaxed value of 400 kmph for $S_{outlier}$ to avoid over-constraining the Hidden Markov Model in the presence of noisy position estimates. Section 3.6.6 of our evaluation shows that the speed constraint is essential to achieving good map-matching accuracy with noisy data.

The third panel in Figure 3-6 illustrates the most likely route found by the Viterbi decoder on our running example, in black. Note that the output is actually a sequence of ⟨point, road segment⟩ pairs, where consecutive points can lie on the same road segment.

**Performance Optimization.** The running time of the Viterbi decoder is $O(mn)$ where $m$ is the number of input location samples after interpolation, and $n$ is the number of search states i.e. the number of candidate road segments that may lie on the path we are matching to. Including every segment in the road network in the search space is not feasible computationally, so *VTrack* uses a simple *geographic pruning* strategy to reduce the state space of the Hidden Markov Model. It does so by dividing the entire road network into relatively coarse-grained grids, which are a few hundred metres in size. We consider all the grids that include *any* point in the input trajectory being map-matched, and include all the segments in these grids in the search space. The grid size we use is approximately 500 metres, which is a sweet spot: using smaller grids sometimes fails to include relevant segments (in case of extreme localization errors in WiFi localization), and using a larger grid size increases computational complexity significantly if the road network is dense. The computational complexity increases with road density.

We have implemented *VTrack* in both C++ and Java. In practice, we have found both the implementations to be comfortably faster than real-time, map-matching hour-length drives within 2 minutes on a MacBook Pro with 2.33 GHz CPU and 3 GB RAM. Our live deployment of *VTrack* as part of Cartel [17] is able to map-match real-time data collected from an iPhone app and provide near real-time routing delays to end users, while running on data from the entire US road network.

### 3.4.6 Bad Zone Removal

The final step in *VTrack* is a *post-processing* step called "bad zone removal" whose goal is to remove zones where the output of map-matching is *known* to be bad or unreliable. Removing such zones is useful in the context of applications like travel time estimation for traffic monitoring. The idea is that if it is possible to use only data from segments of the output that are likely to be reliable, this improves the accuracy of travel time estimates and reduces the likelihood of using a wrong travel time estimate for routing.

Bad zone removal in *VTrack* uses a simple *confidence metric*: $d(P)$, the distance of each *observed* position sample $P$ from the segment it is matched to in the output. The intuition is that if an observed position sample is too far from the segment it is matched to, it is extremely unlikely to be correctly matched. We use a conservative threshold of $d(P) \geq 100$ metres for bad zone removal (the 100 metres represents twice the approximate expected noise of 50 metres from WiFi localization estimates). Bad zone removal in *VTrack* works by eliminating entire zones around peaks in this "distance function" $d$. When the algorithm finds a candidate "bad zone" point $P$, it *backtracks* and

computes values of $d(x)$ for points $x$ in the output preceding $P$. As long as $d(x)$ keeps decreasing, it continues tagging points, stopping only when $d(x)$ reaches a local minimum. The same strategy is also applied to points following $P$ in the output.

The intuition behind this removal strategy is that when a position sample has high raw error, it is often likely that some samples preceding the sample have had increasing error, and vice-versa for following samples. This is the case with both GPS localization and WiFi localization. It is therefore prudent to eliminate the entire range of location samples from the output.

As we shall see in Chapter 4, bad zone removal works well mainly on GPS and WiFi localization data: no simple bad zone removal technique or confidence metric works effectively while map-matching cellular data, because the raw error of position samples is very high.

Bad zone removal is illustrated on our running example in the fourth panel of Figure 3-6.

## 3.5 Travel Time Estimation

The next step of *VTrack* is a *travel time estimator* that extracts travel times for individual segments from the output of map-matching.

The output of the map-matching step is a continuous sequence of segments in the road network. Each segment of the output has a corresponding (possibly empty) set of input point(s) matched to it. The traversal time $T(S)$ for any segment $S$ in the output consists of three parts:

$$T(S) = T_{left}(S) + T_{matched}(S) + T_{right}(S)$$

$T_{left}(S)$ is the time between the (unobserved) entry point for $S$ and the first observed point (in chronological order) matched to $S$. $T_{matched}(S)$ is the time between the first and last points matched to $S$. $T_{right}(S)$ is the time between the last point matched to $S$ and the (unobserved) exit point from $S$.

As stated in Section 3.4.3, map matching adds interpolated points to ensure that each segment in the output has at least one corresponding input point. Hence, if map matching outputs a continuous sequence of segments, both $T_{left}(S)$ and $T_{right}(S)$ are upper-bounded by 1 second, and for segments that are not too small, $T_{matched}(S)$ is the main determinant of delay. To perform time estimation, we first assign $T_{matched}(S)$ to the segment $S$. We then compute the time interval between the first point matched to $S$ and the last point matched to $S_{prev}$, the segment preceding $S$ in the map match, and divide it equally[1] between $T_{right}(S_{prev})$ and $T_{left}(S)$, and similarly for the segment $S_{next}$ following $S$.

Map matching does not always produce a continuous sequence of segments because bad zone detection removes low confidence matches from the output. We omit time estimates for segments in, immediately before, and immediately after a bad zone to avoid producing estimates known to be of low quality.

### 3.5.1 Travel Time Conservation

The strategy presented above, of partitioning the total travel time among individual segments, is only one possible way to estimate traffic on a road segment. An alternative, equally intuitive, approach

[1] How we divide does not affect estimates significantly because the interval is bounded by 1 second.

to estimating traffic or congestion levels on individual segments might be to compute approximate vehicle *speeds* and associate the observed speeds to the map-matched segments. However, speeds computed from raw GPS data or even map-matched points produced by the HMM tend to be inaccurate, particularly on small segments where a small number of points (sometimes only one or two) are matched to the segment. More importantly, if not computed carefully, average speeds may not satisfy the following important *conservation principle*:

- The sum of estimated travel times for individual segments on *any* stretch of road segments $S$ always equals the total observed travel time on $S$.

Note that the simple strategy of computing average speed on each traversed segment does not always satisfy the principle (it depends on how "speed" is defined).

The conservation principle is actually non-trivial: it turns out to be important in cases where the algorithm is confident of the travel time on a longer stretch, but unsure of the *distribution* of time on shorter sub-segments in that stretch. For example, given two road segments separated by a traffic light, we may not know precisely whether a car waited for a red light at the end of one segment, or crossed the intersection quickly but slowed down at the beginning of the next segment. However, as long as the *VTrack* travel time estimator conserves travel time, it will estimate the correct end-to-end travel time on any path that involves *both* segments — even if individual estimation errors on the two segments are high, they cancel out in aggregate. This cancellation of errors would not occur if a simple average speed estimator were used to compute travel time.

As our evaluation of *VTrack* in Section 3.6 shows, this "cancellation behaviour" actually occurs with travel times extracted from WiFi localization data: while time estimates on individual segments often have high errors, these errors tend to cancel out when the estimates are aggregated together for an application like end-to-end route planning.

The next section explains the types of estimation errors made by *VTrack*.

### 3.5.2 Estimation Errors

The main source of error in *VTrack*'s time estimates is inaccuracy in the map-matched output, which can occur for two reasons:

**Outages during transition times**. Transition times are times when a car is moving from one road segment to another. Without observed samples at these times, it is impossible to determine the travel time on each segment exactly. While map-matching can use interpolation to determine the correct sequence of segments, accurate time estimation for individual segments is harder than just finding the correct trajectory. For example, we cannot know whether a car waited for a red light at the end of one segment, or crossed the light quickly but slowed down at the beginning of the next segment.

**Noisy position samples.** Suppose that the location of a car is sampled just after it enters a short length segment, but a noisy sensor estimates the location of the car to be near the end of the segment. If this is the only sample matched to that segment, the location error is likely to translate into an extremely inaccurate time estimate. In particular, determining travel times for *small segments*, whose lengths are of the order of the standard deviation of the noise in the location data, is impossible with our approach. For example, consider the case of using GPS sampled every 60 seconds. In the absence of a sophisticated model for car speed, it is impossible to determine the travel time on a segment more accurately than to within 60 seconds, even if *all the location samples*

*are perfectly accurate*. Similarly, if a 100 metre error occurs in raw location data, it is impossible to estimate accurate travel time on a segment whose size is not much larger than this "resolution" of 100 meters.

Although *VTrack* is unable to estimate accurate times or speeds for individual small segments, Section 3.6 shows that the conservation principle is helpful here. Travel time estimation errors on adjacent or nearby segments tend to have *opposite* signs as long as the overall trajectory from trajectory mapping is correct. Because groups of segments (representing commonly traversed sub-paths) tend to be reused repeatedly in end-to-end routes, the *end-to-end travel time estimates* for routes produced by *VTrack* are highly accurate even with noisy or sub-sampled input data, and prove to be adequate for applications like route planning (Section 3.6).

## 3.6 Evaluation of *VTrack*

We have evaluated *VTrack* on a large data set of 800 drive hours of GPS and WiFi location estimates from real vehicular drives, obtained from the testbed of the MIT Cartel project [17].

This section is organized as follows. We first describe the Cartel testbed and the evaluation data set obtained from the testbed (Section 3.6.1). Since an important challenge in the evaluation was obtaining reasonable ground truth for the vehicular drives, this section also explains the procedure used to clean and obtain reasonable ground truth. We then overview the sensor sampling strategies that we compare in the evaluation (Section 3.6.2).

The first part of our evaluation is based on the two end-to-end applications mentioned earlier in this chapter:

- **Traffic aware routing**, whose goal is allow end users to compute shortest paths between a given source and destination using real-time traffic data collected from *VTrack*'s trajectory mapping system.
- **Hotspot detection**, whose goal is to detect road segments that are unusually congested or slow compared to their own historical mean, enabling users to manually avoid these hotspots.

These applications were built on *VTrack*. In each case, we use relevant *end-to-end metrics* from the application's point of view to evaluate the quality of travel time estimates produced by *VTrack* for different sensor configurations with different GPS and WiFi sampling rates.

At a high level, we show that *VTrack* can achieve good accuracy for route planning using either WiFi localization or sparsely sampled GPS, saving energy in both cases (Section 3.6.3). We drill down into the results and show that this accuracy is achieved *in spite of large travel time estimation errors on individual segments.* We also show that sub-sampled GPS is effective up to certain sub-sampling frequencies for hotspot detection, but WiFi localization is less useful for hotspot detection owing to outages in WiFi data (Section 3.6.4).

The second part of our evaluation presents four micro-benchmarks that evaluate the impact of *VTrack*'s algorithm design choices on map-matching accuracy:

- We first (Section 3.6.5) address the question of how robust *VTrack* is to noise in input data. We show using simulations that *VTrack*'s HMM-based map matching algorithm is robust to significant amounts of simulated Gaussian noise, but breaks down beyond a certain levels of input noise, such as that found in cellular localization estimates.

- We next show that the speed constraint used in *VTrack* is essential to achieving good map-matching accuracy (Section 3.6.6).
- We then show that *VTrack*'s transition score improves map-matching accuracy over a simple partitioning approach (Section 3.6.7).
- Last, we evaluate the impact of the linear interpolation strategy used in *VTrack* and show that it compares favourably to more sophisticated strategies such as shortest-path interpolation in terms of accuracy on intermittently sampled input data (Section 3.6.8).

### 3.6.1 Data and Ground Truth

**The Cartel Testbed.** We use location data from the Cartel testbed [17], drawn from two sources:

- 25 taxicabs equipped with embedded computer running Linux with on-board GPS and WiFi, manufactured by Meraki [61]. The computer fits in a box, and draws power supply from the OBD connector on-board the vehicle.
- iPhone users running iCartel who contribute GPS data.

The Cartel testbed has been running for over 4 years as of the date of this thesis (since 2007). It has been used to collect and analyze a wide variety of sensor data — wireless network throughput, real-time traffic data, accelerometer data for detecting potholes [42], etc. Most of the taxicabs are operated by a Boston-area company called PlanetTran, who have been using Cartel's visualization interface to visualize the current location and historical drives taken by their taxicabs. 10 livery vehicles from JB Livery in South Boston have contributed data since 2010.

For the evaluation, we begin with a collection of 3998 drives collected from 25 of the PlanetTran taxicabs. GPS location data is sampled from an attached GPS unit that plugs into the Meraki computer in each car. The Meraki computer runs a WiFi driver which periodically scans for WiFi access points. The GPS and WiFi data are joined in time and uploaded back to the Cartel server either via any available WiFi uplink.

We first pre-processed the WiFi access point sightings to produce location estimates to feed to *VTrack*. The location estimates were computed using a "centroid of centroids" approach (as discussed in Section 3.4.2). Figure 3-7 shows the coverage map of our evaluation data, i.e., the set of distinct road segments on which our vehicles drove. The data set *after* cleaning (using a procedure described below) amounted to nearly 800 distinct hours of driving with both GPS and WiFi location estimates.

**Ground Truth.** Obtaining ground truth is a fundamental challenge in the evaluation of any map-matching and travel time estimation system. Approaches such as recording additional data in test drives are accurate but very time consuming, not scaling beyond relatively small amounts of test data. For example [48] makes video recordings to record road segment transitions. Another approach is to simply use GPS sampled at the highest possible rate as ground truth [57]. This works well a significant fraction of the time, but fails in regions where GPS is subject to errors and outages (e.g., near large buildings and in tunnels), making it inappropriate to use such data for ground truth.

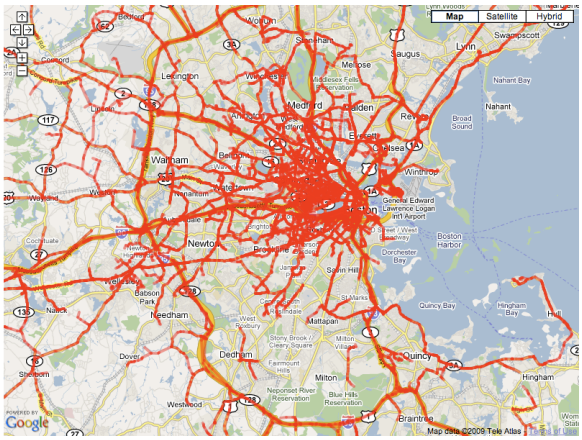



**Figure 3-7: Coverage map of our evaluation drives.**

As others have noted [44], it is difficult to get perfect ground truth for a large volume of data. We use GPS for ground truth, but rather than using it directly, we aggressively clean the data to produce ground truth with reasonable confidence for a subset of our drives. The goal of the cleaning is to *identify a subset of GPS data that can be treated as highly accurate with high confidence.*

The steps we use to clean raw GPS data are as follows:

- For each GPS point $g$ in a drive, we consider the set of road segments $S_g$ within a 15 meter radius of $g$ (we picked 15 metres because it is the width of a typical road segment).
- We search the space of these segments to match the sequence of points $g$ to a continuous sequence of segments $X_g$, such that each $X_g \in S_g$. Therefore, each GPS point is matched to one of its neighbours found in the previous step. We throw out all points $g$ that cannot be matched this way (for example, if there are no segments within a 15 meter radius).
- We now look for outages of more than a certain duration (we used 10 seconds, approximately the median traversal time for a segment) and split the drive into multiple drives on either side of the outage, ignoring the duration of the outage.
- Finally, we project each $g$ to the closest point on $X_g$ to obtain a corresponding ground truth point $g'$.

The output traces from data cleaning, which we term *clean drives*, satisfy three key properties:

- No gap in the data exceeds 10 seconds.
- Each GPS point is matched to a segment at most 15 meters from it.
- The resulting segments form an unbroken drive.

These three properties taken together define a subset of the GPS data that can be treated as ground truth with high confidence. It is possible that systematic errors in GPS could cause this approach to fail, but we assume these are rare.

**Limitations.** The main limitation of the cleaning approach to ground truth is that using only clean GPS data (known to be within 15 metres of a road segment) may bias the comparison of GPS and WiFi localization. When using only clean GPS data, *GPS sub-sampling* will also yield accurate longitude/latitude coordinates, and is likely to do better on the subset of data we have selected compared to the entire data set. To avoid biasing our evaluation in favour of GPS and to realistically understand how it compares to WiFi, we perturb GPS samples with additive Gaussian noise before feeding them to the *VTrack* algorithm, as described below.

The advantage of the perturbation approach is that it is cheap because it only uses easy to collect GPS data, but realistic, because it restricts the evaluation to drives with near-perfect travel time data. Importantly, the evaluation does *not* test the system on these nearly-perfect samples, which would introduce bias by making GPS look artificially good, but instead tests on noisy versions of the samples to simulate real GPS behaviour.

The limitation of the perturbation approach is that it cannot model the impact of outliers in GPS, because there is no ground truth for regions where GPS fails. Hence all the results in this dissertation involving GPS are for the *outlier-free* case. For cases where GPS has significant outliers — *e.g.*, "urban canyons" or tunnels — using WiFi localization may be preferable. This is because it is possible to collect WiFi localization training data in urban canyons using expensive, high-accuracy dedicated GPS receivers that can decode GPS signals in urban canyons better than commodity mobile phone GPS chipsets.

**Ground Truth Validation.** We attempted to validate the above ground truth cleaning procedure on a small data sample (about 1 hour of driving) using a simple field drive in the Boston area. We used an Android phone equipped with GPS to record the location of the phone, and a phone application to mark the locations of turns. A human operator pressed a button whenever the vehicle crossed an intersection, signal or stop sign. We compared the travel times between successive turns *i.e.*, successive button presses, obtained from the application — which is as close to ground truth as human error would permit — to that obtained from the cleaned version of the GPS data obtained using the procedure described above. The average error in travel times from the cleaned version was 4.7% for one such 30 minute drive in Boston with approximately 30 segments, and 8.1% for another 30 minute drive in the Cambridge/Somerville area with approximately 50 road segments. Manual inspection of some of the errors revealed that a significant portion of the error was from human reaction time when marking turns, rather than from residual errors not filtered by the cleaning procedure. While this is a small sample, it gives us some confidence that the basic cleaning procedure is sound, and useful to establish ground truth.

**Cleaning.** We cleaned the raw data from the Cartel testbed using the procedure described above. In addition, we also omitted traces shorter than 2 kilometres, with fewer than 200 location samples (about 3 minutes) or fewer than 10 road segments. For the purpose of the travel time estimation applications, we also eliminated portions of traces where a car traveled slower than 2 km/h for 60 or more consecutive seconds, which we interpreted as parked. Typical traffic signals in the urban United States do not last more than this, but this number could be modified to accommodate longer signal wait times.

### 3.6.2 Strategies We Compare

We use the clean drives obtained using the ground truth procedure described above to obtain accurate ground truth travel estimates for each segment on each drive. We then evaluate the accuracy of travel

time estimates produced by four different sensor sampling strategies when using *VTrack*. The last of these is a *control strategy*. The four strategies are:

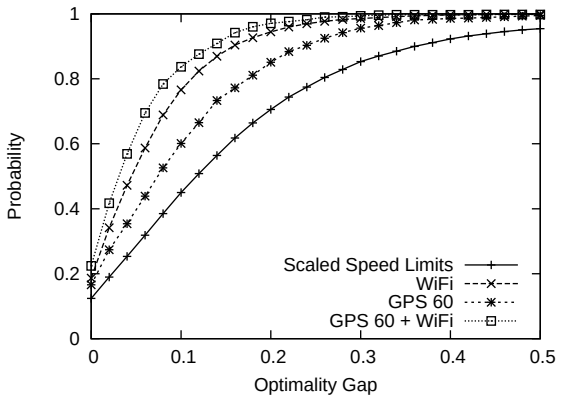
- **Continuous WiFi.** We use *VTrack* to map-match WiFi location data, compute travel times from the match, and compare to travel times obtained from the clean drives.
- **GPS every $k$ seconds.** We sub-sample ground truth GPS data at intervals of $k$ seconds, discarding $k-1$ samples and use every $k^{th}$ sample as input to the *VTrack* trajectory mapper. We add Gaussian noise with a standard deviation of approximately 7 meters to every $k^{th}$ GPS sample (7 metres of Gaussian noise is acknowledged to be a reasonable model for GPS [33]).
- **GPS every $k$ seconds + WiFi in between.** This is a combination of the sampling strategies above, where the *VTrack* trajectory mapper is given input samples *both* from sub-sampled GPS, and from WiFi location data in between GPS samples.
- **Scaled speed limits.** This approach is a "control" that does not use sensors. It instead uses speed limits from the NAVTEQ road database [66] to produce static travel time estimates for road segments. Since drivers do not drive exactly at the speed limit, directly using speed limits to compute travel times incurs a systematic bias. To alleviate this problem, we scaled all the speed limits by a constant factor $k$, and chose the best value of $k$ — the value that pegs the mean difference between time estimates from speed limits and time estimates from ground truth to zero. We find the value of $k$ to be close to 0.67, reflecting that drivers in our data drove at 67% of the speed limit on average. This strategy is intended to simulate the *theoretically best possible accuracy* that a strategy for route planning using only speed limits can achieve.

### 3.6.3 Route Planning

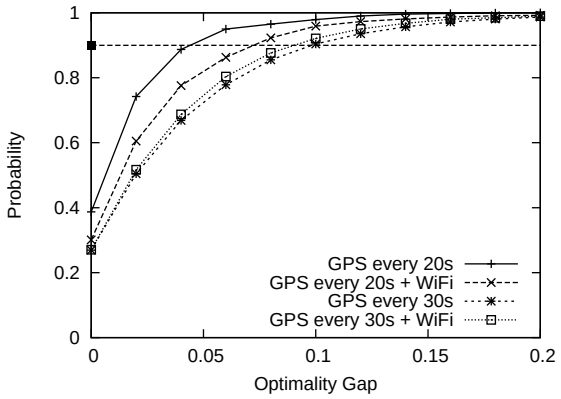
We evaluate the travel time estimates produced by *VTrack* for different combinations of GPS sampling rates and WiFi localization, in the context of route planning. In route planning, we are interested in finding the best end-to-end routes between source-destination pairs in the road network that optimize some metric. In this evaluation, we choose a popular metric: minimizing the expected drive time.

For each sensor setting, our evaluation takes as input a set of clean drives $D_{gt}$ and a corresponding set of noisy or sub-sampled drives $D_{noisy}$ — for example, using WiFi, or sub-sampled GPS, or a combination. We run *VTrack* on $D_{noisy}$ to produce travel time estimates for each segment on those drives. Our goal is to understand how errors in these estimates from VTrack affect the quality of the shortest path estimate, i.e., *how much worse the paths the route planner finds using these inaccurate travel times are*, compared to the best paths it can find from ground truth travel times.

We consider the set of road segments for which we have a ground truth travel time estimate from at least one clean drive in $D_{gt}$. Call this set $S_{gt}$. We construct the *induced graph* $G_{gt}$ of the entire road network on the set $S_{gt}$, i.e., the subset of the road network defined by the segments with ground truth and their end points. Note that $G_{gt}$ may be disconnected: if so, we divide it into connected components. We pick a connected component at random and simulate a drive between a random source-destination pair within that component. In our simulation, we divided the graph into zones and picked the pair from different zones to ensure drives that have a minimum length of 2 kilometres. For each segment $S$ in the graph, we pick two shortest path "weights":




(a) GPS vs WiFi vs Scaled Speed Limits.




(b) GPS vs Hybrid (GPS + WiFi).

**Figure 3-8: CDF of optimality gap when route planning using *VTrack*. Larger optimality gaps are worse.**

- A ground truth weight for $S$ picked from a clean drive $D_{gt}$ in which $S$ appears. $D_{gt}$ is chosen at random from all clean drives containing $S$.

- An estimated weight, picked to be the VTrack travel time estimate for $S$ from the noisy or sparse version of $D_{gt}$, $D_{noisy}$. This quantity is available whenever *VTrack* includes $S$ in its estimated trajectory for $D_{noisy}$. If the *VTrack* trajectory for $D_{noisy}$ omitted $S$, we fall back on estimating the travel time using the scaled speed limit.

We now run Dijkstra's algorithm on $G_{gt}$ with the two different weight sets to find two different shortest paths, $P_{gt}$ and $P_{noisy}$. To evaluate the quality of route planning, we compare the *ground truth times* for these two paths and compute the following "optimality gap" metric for each source-destination pair:

$$Optimality\ Gap = \frac{Time(P_{noisy}) - Time(P_{gt})}{Time(P_{gt})}$$

Note that the ground truth time for a path is always available because we use the graph induced by ground truth observations as the basis for route planning.

Figure 3-8(a) shows CDFs of the optimality gap across 10,000 randomly selected source-destination pairs for different combinations of sensors and sampling rates, as well as for a strawman which performs route planning using just scaled speed limits. As mentioned earlier in this chapter, we believe that an optimality gap of up to 10-15% is reasonable. This means that for an energy-saving strategy to be useful, it must produce a shortest path no worse than 35 minutes in travel time when the true shortest path takes 30 minutes in travel time.

Figure 3-8(b) shows the same CDF, but compares *GPS k*, the strategy of sampling GPS every $k$ seconds and interpolating in between, to *GPS k + WiFi*, the strategy of using *both* sub-sampled GPS and WiFi location estimates in between.

Below, we state the most important conclusions from the results above:

- Travel times from WiFi localization alone are good enough for route planning. The 90th percentile of the optimality gap is 10-15% for WiFi, implying that 90% of simulated commutes found paths that were no worse than 10-15% compared to the optimal path, when using travel times estimated from WiFi localization alone.
- Travel times from GPS sampled every 30 seconds (or slightly more than 30 seconds) are good enough for high quality route planning. The 30-35 second value appears to reflect that interpolating raw GPS locations works very accurately for up to 3 or 4 missing road segments, but suffers more significant errors if interpolating over a long time scale.
- As we would expect (and hope), both GPS sub-sampling and WiFi localization significantly outperform the control strategy of using scaled speed limits for route planning. Using speed limits works reasonably well in the median, but incurs a significant tail of poor path predictions. All of these poor predictions in the tail correspond to scenarios of traffic congestion, which are precisely the important scenarios to evaluate a traffic delay estimation system.
- A hybrid strategy using *GPS 30 + WiFi*, i.e., both sub-sampled GPS every 30 seconds, and WiFi in between, improves performance over just sub-sampling GPS every 30 seconds (*GPS 30*) or over using only WiFi. However, as we shall see shortly, the gains are tempered by the significantly higher energy costs of using both sensors simultaneously.
- Using *GPS 20* i.e., sub-sampling GPS every 20 seconds with interpolation in between, surprisingly outperforms *GPS 20 + WiFi*, i.e., sampling and using WiFi location estimates in between GPS samples. This is because outliers in WiFi hurt map-matching accuracy. In contrast, *GPS 30 + WiFi* outperforms *GPS 30*. This suggests a "break-even" point where using WiFi is better than using interpolation of 30 seconds. The intuition is that *VTrack*'s map-matching interpolation technique works better than using WiFi localization over a time scale of 20 seconds or smaller. This time interval corresponds to one (or at most two) road segment(s). For the *GPS 20* case, there is no inherent reason why adding WiFi information should hurt. We believe it should be possible to modify *VTrack* to filter out WiFi outliers and use only interpolated GPS information.
- For GPS sub-sampling intervals beyond 60 seconds, route planning starts to perform worse, with at least 10% of commutes finding paths at least 20-25% worse than optimal. This suggests that while sampling GPS less often than a minute might save significantly more energy, it is likely to yield medium-quality (reasonable but not very good) route predictions, at least for urban areas. Sampling GPS every minute corresponds to approximately half a kilometer in terms of distance at typical city driving speeds.

We now drill down into our results and show that, somewhat counter-intuitively, WiFi localization works adequately for route planning *in spite of large estimation errors on individual road segments* because *VTrack* correctly matches WiFi localization data to the correct trajectory most of the time.

Figure 3-9 shows a CDF of per-segment delay estimation errors, reported relative to ground truth travel time, on individual road segments when using only WiFi localization. WiFi localization has close to 25% median error and 50% mean error. However, the errors in the CDF are actually two-sided because *VTrack* when run on WiFi localization estimates often finds the *correct trajectories*. The main errors it makes are to mis-assign points on either side of segment boundaries. Hence, errors on groups of segments traversed on a given drive tend to cancel out, making end-to-end estimates more accurate than individual segment estimates. For example, the *VTrack* algorithm might distribute a travel time of 30 seconds as 10 seconds to segment $A$ and 20 seconds to an adjacent segment $B$ when the truth is actually 20 on $A$ and 10 on $B$. However, if $A$ and $B$ are always traversed together, *this error does not affect the end-to-end travel time estimate*.

Figure 3-10 shows the accuracy of only trajectory mapping in isolation. This graph shows the CDFs for two metrics:

- *Point Error Rate (PER)* measures the frequency with which any given position sample in the input is matched to the wrong segment in the output of *VTrack*.
- *Segment Error Rate (SER)*, defined as the ratio $\frac{ED}{G}$, where $EG$ is the edit distance between the map-matched output of *VTrack* and the ground truth trajectory, and $G$ is the number of segments in the ground truth trajectory. *SER*, unlike *PER*, is not concerned with errors in matching individual points as long as the overall trajectory is correct.

We see from the figure that *VTrack* performs *significantly* better on $SER$ than on $PER$.

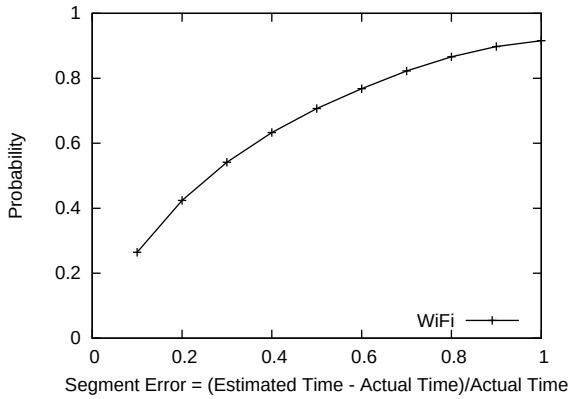



**Figure 3-9: CDF of errors in time estimation for individual segments on WiFi localization estimates.**

Figure 3-11 shows, in contrast, the errors in the actual estimated times for *end-to-end* routes found using WiFi localization. Here, we compare the estimated time for a path predicted using travel time estimates from *VTrack* with the ground truth time for the same path. The figure shows a plot of the CDF of the relative error in prediction. The graph shows that the end-to-end travel times predicted using WiFi localization are highly accurate, even at the 90th percentile, in contrast to times predicted using scaled speed limits.

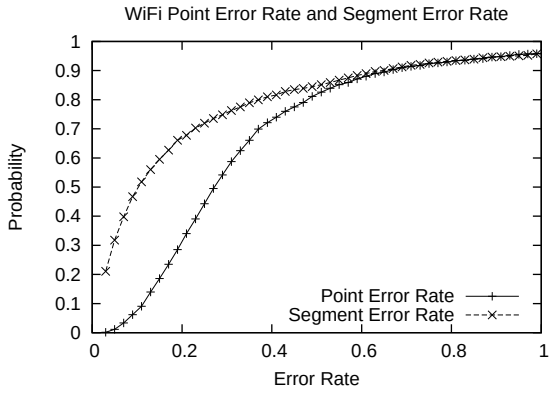



**Figure 3-10: Map matching errors for WiFi localization.**

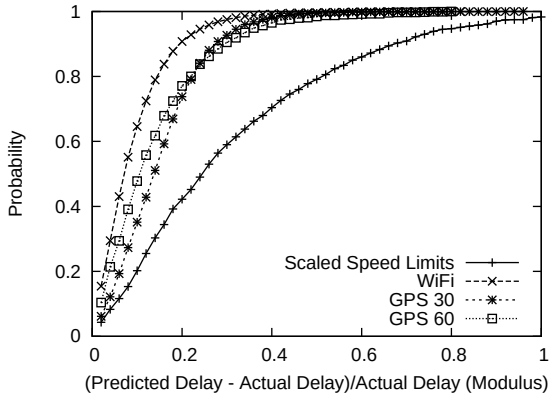



**Figure 3-11: End-to-end time estimation accuracy using WiFi localization.**

The results above validate the hypothesis that the trajectories produced by *VTrack* for WiFi localization data are often correct. It is the matching of points to segments within a trajectory that is more often wrong, which is why the strategy of following a *conservation principle* when calculating travel times (as discussed in Section 3.5.1) is a good idea.

**Spurious Segments.** The optimality gap metric presented above captures the impact of *both* incorrect predictions and missed segments in prediction. This is because *VTrack* falls back on a less accurate travel time, calculated from a scaled speed limit, when it misses a segment in its output. However, a limitation of the evaluation here is that it cannot directly capture the impact of *spurious segments*, segments produced by the map-matching algorithm that are absent in the original ground truth. We have found in separate experiments that the "error rate" for spurious segments i.e., number of spurious segments as a fraction of total number of segments in an output trajectory — is less than 15% for most of the sensor sampling strategies discussed in this evaluation — including WiFi alone, GPS sub-sampling more often than a minute, and combinations of GPS and WiFi (see Figure 3-12).

### 3.6.4 Hotspot Detection

This section *VTrack*'s time estimates in the context of hotspot detection, i.e., detecting which road segments are highly congested so that drivers can avoid them. We say a road segment is a "hotspot"

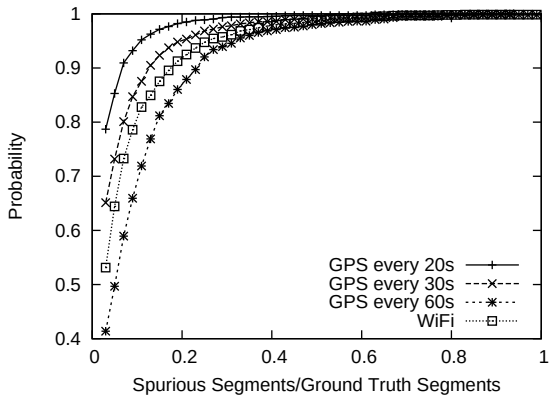



**Figure 3-12: Spurious segment rates from map-matching for different sensors and sampling rates.**

if the true travel time on a segment exceeds the travel time estimated from scaled speed limits by at least $threshold$ seconds. The hotspots for a particular threshold value are the set of road segments which have high travel time based on the ground truth data. We evaluate *VTrack* by examining how many of those hotspots we can detect using the different sensor sampling strategies: *GPS k*, WiFi sampling, and *GPS k + WiFi* for different values of $k$.

An alternative definition of a hotspot would be *multiplicative*: a road segment in which the observed travel time is more than $threshold$ *times* the travel time estimated with scaled speed limits. We found that this multiplicative definition is excessively biased by the length of the segment. Small segments have comparable variance in travel times to large segments. However, because they have a small travel time, they are more likely to be flagged as hotspots than large segments, which have a higher travel time. Therefore, we chose to use the first (additive) definition of "hotspot", as it more accurately reflects the road segments that drivers truly view as hotspots.

To detect hotspots using a trace of WiFi, GPS+WiFi, or sub-sampled GPS data, we first map-match the data using *VTrack*. We then find the segments that meet the additive definition, i.e., have a travel time that exceeds the time estimated from scaled speed limits by some threshold. In addition to classifying each of these segments as a hotspot, we also classify the segments *adjacent* to each such segment as hotspots if they appear in the map-matched output. We use this strategy of flagging "groups" of segments as hotspots because, as we have seen earlier, the travel time estimation algorithm is not always certain as to precisely which road segment a high travel time should be attributed to, and almost always tends to be off by one segment, such as in a scenario with two segments on either side of a congested intersection.

To measure the accuracy of *VTrack*, we use two metrics:

- *Success rate*, defined as the fraction of hotspots found using ground truth travel time data that are detected from the output of *VTrack*.
- *False positive rate*, defined as the fraction of segments output by *VTrack* as hotspots, that are *not* hotspots in the ground truth.

A false positive amounts to suggesting that a driver avoid a segment that is not actually congested.

False positives and groups interact as follows. We record a false positive if we flag a group of segments as a hotspot, but that *group* is not a hotspot. We define a group as a hotspot if the total travel time on the group is more than $threshold \times number\ segments\ in\ group$ seconds above the travel time estimated by scaled speed limits.

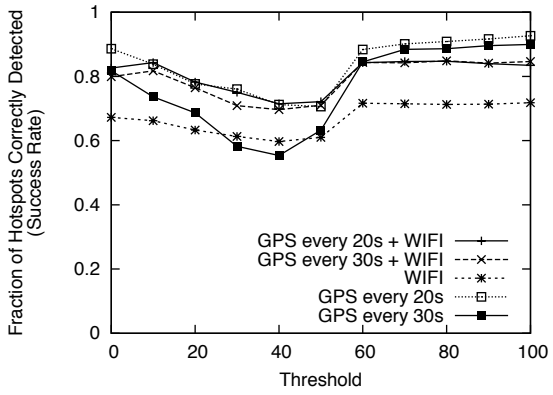



**Figure 3-13: Success rate of hotspot detection with *VTrack*.**

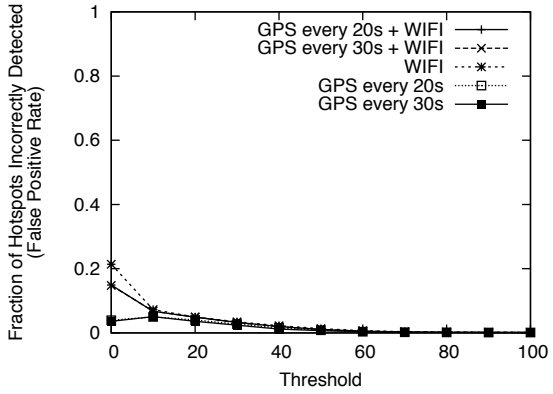



**Figure 3-14: False positive rate of hotspot detection with *VTrack*.**

Figure 3-13 shows the success rates when running *VTrack* on each sensor sampling strategy, and Figure 3-14 shows the false positive rates for each strategy.

There are a few interesting conclusions from these graphs:

- First, for the strategies involving GPS, the success rate is consistently above 0.8, and often around 0.9, implying that strategies using GPS sub-sampling can consistently detect between 80% and 90% of hotspots.
- The success rate for WiFi localization is much worse, frequently around 0.65, because WiFi localization experiences significant outages when there are no WiFi access point results returned in a scan. Hotspot detection cannot find a hotspot in regions for which it has no data. In contrast, GPS data has more complete coverage *even when sub-sampled*.
- If we restrict statistics to road segments where WiFi data does exist, WiFi localization has

a success rate comparable to all the GPS schemes. Hence rather than errors contributing to incorrect map matches, it is outages that affect the accuracy of hotspot detection.

- In all schemes, the false positive rate remains low. This is a desirable property as we do not want users to avoid road segments that are not actually congested.
- It is interesting to note that, with the exception of GPS every 30 seconds, *VTrack*'s success rate in every strategy remains relatively consistent across different values of the hotspot "threshold". This indicates that our algorithm is fairly robust to a variety of applications that may have different requirements for what constitutes a hotspot.
- *VTrack*'s false positive rate also remains low for most threshold values, and for the most part only increases for small thresholds. This is due to the fact that, with a small threshold, *VTrack* is likely to flag many groups of segments as hotspots, and these groups may include some segments that do not have high travel time, but were included because they were adjacent to a segment with high travel time.
- We note that there is a dip in all of the strategies at around 40 seconds. At small thresholds, *VTrack* flags many segments as hotspots, and thus has both a high success rate and a relatively high false positive rate. As the threshold begins to increase, the algorithm starts to miss some hotspots, but the false positive rate decreases dramatically. This explains the portion of the graph before a threshold of 40.
- The second portion of the graph and the dip can be explained by examining the total number of hotspots. As the threshold increases, the number of hotspots naturally decreases. At a threshold of about 40 seconds, the rate of decrease slows, and from 60 seconds on, the number of hotspots remains fairly constant. This means that many of the road segments that are hotspots with a threshold of 60 are also hotspots with a threshold of 100; their observed time differs from their estimated time by over 100 seconds. As a result, *VTrack* does very well flagging hotspots at larger thresholds, since they are the more "obvious" hotspots, in some sense, and relatively resistant to small errors in travel time estimation.

**Discussion of WiFi Outages.** In our data, we found that WiFi localization has an outage rate of 42%, i.e., 42% of the time which we are trying to use WiFi localization, we do not get a WiFi scan result. This raises the question: how can a sensor that is so unreliable still perform well in some applications? In particular, we saw that although WiFi sensors did not work particularly well for hotspot detection, they did work well for route planning.

The reason for this is that in route planning, using the scaled speed limit estimates on segments where there is no WiFi data is generally sufficient to do reasonably well. Outages in WiFi tend to cause missed data points on segments that are relatively small in size. These are exactly the segments where scaled speed limit estimates are reasonable. Using scaled speed limit estimates on an *entire* path does not perform well because they cannot account for any variance or traffic congestion. However, using scaled speed limits as a back-up for segments missing WiFi localization *can* work well in certain cases.

In hotspot detection, on the other hand, we can never use the scaled speed limit estimates in place of actual location or map-matching data. After all, we define a hotspot as a road segment where the observed time estimate *differs* from the scaled speed limit estimates. This explains why WiFi localization alone is unsuitable for hotspot detection.

### 3.6.5 Robustness To Noise

In this section, we *simulate* the performance of *VTrack* using micro-benchmarks that test the algorithm for varying levels of noise in input data. To provide a reference strawman and to understand how much a Hidden Markov Model actually helps, we compare *VTrack* to the simple approach of simply matching each point to the nearest road segment in the road map. We demonstrate that *VTrack* is relatively robust to noise, much more so than nearest-segment matching.

In these micro-benchmark experiments, the input is once again "clean drives" cleaned using the procedure described in Section 3.6.1. Each location sample in each of these drives is labeled with the ground truth segment it came from.

To perform the micro-benchmarks, we generate a *perturbed* version of each cleaned drive by adding random zero-mean Gaussian noise with different standard deviations of 15, 40, and 70 meters to both the X and Y coordinates of each point in the drive. As a reference point, 40 meters noise roughly corresponds to the average standard deviation of WiFi localization data, though of course WiFi localization errors are generally non-Gaussian.

We compare the accuracy of *VTrack* and the simple nearest-segment matching strategy on each of the perturbed drives, in terms of the *point error rate* (PER) of each approach. As defined previously, PER is the fraction of points on a drive that are assigned to the wrong segment.

Figure 3-15 shows a CDF of the PER over all the perturbed drives, for each of the strategies under varying amounts of added noise.

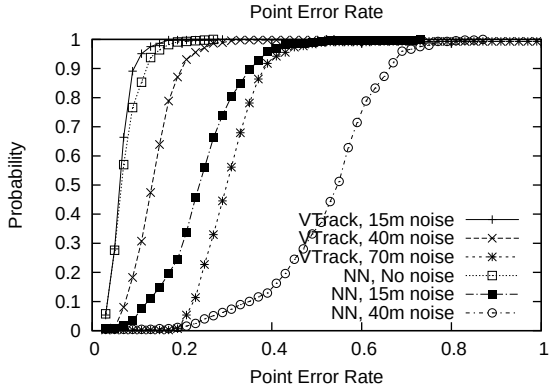



**Figure 3-15: Accuracy of *VTrack* vs nearest segment matching (denoted by NN).**

The results show that *VTrack* performs extremely well with 15 meter noise—about as well as nearest-neighbor matching with no noise. The median PER is less than 5%, and the 90th percentile PER is less than 8%. For 40 meter noise, these values are approximately 8% and 10%, not significantly worse.

Nearest segment matching performs much worse — even with just 15 meters noise, the median point error rate is almost 20%. Even with 70 meters of input noise, *VTrack* does better than nearest neighbor with 40 meters noise, achieving a median PER of about 20%.

In general, a Markov Model will be more accurate than nearest neighbor matching for the reasons described in Section 3.4.3; in particular, in cases like that in Figure 3-5, the HMM is able to correctly assign noisy point $P2$ (which is nearest to segment $S2$) to segment $S3$.

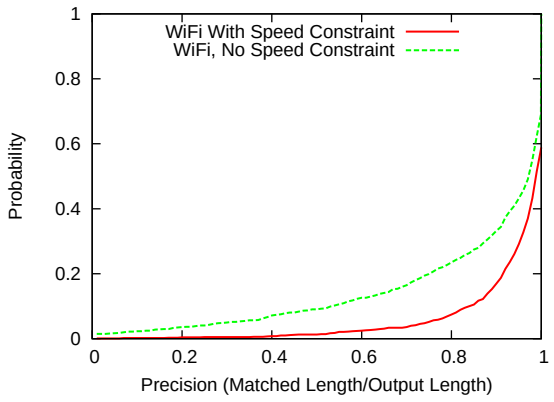



**Figure 3-16: Speed constraint improves map-matching precision.**

Beyond 70 meters noise (not shown in the figure), *VTrack* performs poorly, with a point error rate significantly higher than 20%. This suggests that the *VTrack* algorithm is not suitable to map match location estimates from cellular triangulation, which has errors of the order of hundreds of metres in urban areas.

In Chapter 4, we evaluate *VTrack* on cellular location data, and show that the *VTrack* approach of converting radio measurements (be they cellular or WiFi) into (latitude,longitude) coordinates is flawed when estimates are very noisy.

### 3.6.6 Impact Of Speed Constraint

In this section, we show that the speed constraint used in *VTrack* is critical to achieving good map-matching accuracy for the noisiest input drives.

We ran *VTrack* on WiFi localization data with and without the speed constraint enabled and compared the accuracy of map-matched outputs. For this evaluation, we used *precision*, a closely related metric to SER (segment error rate). Given a ground truth trajectory $G$ and an output trajectory $X$, we match the road segments of $G$ and $X$ using a dynamic program that finds the closest alignment between the trajectories. We define the precision to be the fraction of the output trajectory $X$ that is matched to some part of the ground truth $G$. A higher precision means better accuracy.

Figure 3-16 shows a CDF of the precision of map-matching with and without the speed constraint. The graph shows that while the speed constraint does not matter much in the median, it improves precision significantly — from 55% to 85% — for the worst 10% of input trajectories. The speed constraint is especially important for input data with significant amounts of noise because the Hidden Markov Model tends to "jump" from segment to segment following the noise without it.

### 3.6.7 Impact Of Transition Score

As mentioned earlier, *VTrack* assigns a transition probability equal to a small constant $\epsilon$ for all transitions out of a given intersection. The alternative would be to assign a transition probability of $\frac{1}{n}$ to each of $n$ road segments leaving an intersection.

Figure 3-17 compares the two alternatives, showing a CDF of the map-matching precision. The figure shows that similar to the speed constraint, using an $\epsilon$ transition score is important to achieving good map-matching accuracy in the tail end of the CDF, i.e. for the noisiest of the input drives.

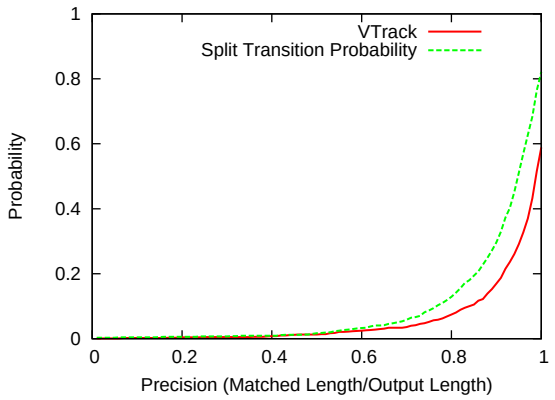



**Figure 3-17: *VTrack*'s transition probability is better than a partitioned transition probability.**

For the worst 10% of input trajectories, *VTrack*'s transition score improves precision over a simple partitioning strategy from 70% to 85%. This is because a simple partitioning strategy tends to overly bias in favour of paths containing intersections with low degree, i.e., with fewer major junctions.

### 3.6.8 Impact Of Interpolation Strategy

*VTrack* uses a simple linear interpolation strategy to deal with gaps in input data. One can also imagine alternative, more sophisticated interpolation strategies, such as computing the *shortest path* in the road network between intermittent location samples and using this path to "fill in" the sequence of road segments output by map-matching.

We performed a simple experiment to compare simple linear interpolation to shortest paths interpolation on sub-sampled GPS data. The shortest paths interpolation strategy interpolates between two location samples $L_1$ and $L_2$ as follows: it finds the closest segment $S_1$ in the road network to $L_1$ and the closest segment $S_2$ to $L_2$. It then computes the shortest path $P$ between $S_1$ and $S_2$ using Dijkstra's algorithm, run with times from scaled speed limits used as weights, and fills in interpolated location samples along $P$ at a frequency of at least one per second. The interpolated samples are fed together with the original samples to *VTrack*.

We use two metrics for evaluation: *precision*, as defined above, and *recall*. Given an output trajectory $X$ and a matched region $M$ between $X$ and the ground truth trajectory $G$, the recall is defined as $\frac{length(M)}{length(G)}$, i.e., the fraction of the ground truth (in terms of length) recovered by the map-matching algorithm.

Table 3.6.8 shows the precision and recall of the two interpolation strategies for different frequencies of sub-sampling GPS. The table shows that, somewhat counter-intuitively, simple linear interpolation performs as well or better than shortest paths interpolation at all frequencies. This is likely because the shortest paths implementation uses nearest-segment matching to find the end points for shortest path computation, which is prone to error.

It may be possible to design a better shortest paths interpolation strategy. We chose to use linear interpolation in *VTrack* since it appears to perform at least as well as (or slightly better than) a simple shortest paths implementation, and has the added advantage of being simpler to implement and faster than shortest paths interpolation. Shortest path interpolation requires issuing multiple shortest path queries in each run of the map-matching algorithm.

| GPS Sampled At | Precision | | Recall | |
|---|---|---|---|---|
| | Linear | SP | Linear | SP |
| 60 Hz | 93.9% | 91.3% | 85.4% | 84.6% |
| 120 Hz | 83.9% | 74.8% | 55% | 57.9% |
| 240 Hz | 63.4% | 60.5% | 30.4% | 34.0% |

**Table 3.1: Linear interpolation is as good, or better than SP (shortest paths).**

## 3.7 Revisiting the Energy Question

Now that we are armed with:

- Energy measurements from a sampling of two different mobile platforms, and
- A characterization of how accurate trajectory mapping needs to be in real applications,

we are equipped to provide an answer to a central question of this dissertation: determining the most energy-efficient strategy for trajectory mapping on a given mobile device, and for a given application.

In this section, we combine the results presented above with the empirically determined model of energy costs for WiFi and GPS presented in Chapter 2. We compare three different strategies: GPS sub-sampled periodically, WiFi, and a hybrid strategy that combines the two. We show how the answer of which strategy to use depends on the following factors:

- The ratio of WiFi to GPS energy cost on the mobile device.
- The end-to-end application we care about (e.g., route planning vs hotspot detection).
- An "energy budget" that specifies the maximum energy that can be used (e.g., in the form of "the algorithm should not drain more than X% of the battery").

*Discussion Of Results*

Based on Figure 3-8(a) presented earlier, we can infer that on devices where GPS is significantly more expensive than WiFi localization in terms of energy, WiFi sampling is the best strategy. The figure shows that WiFi localization is more accurate for route planning than GPS sampled every minute (*GPS 60*) and worse than GPS sampled every 30 seconds (*GPS 30*). It is approximately equivalent in accuracy to GPS sampled every 40 seconds (*GPS 40*). Hence, if `GPS 40` uses more energy than WiFi, WiFi should always be used in preference to GPS.

For example, on the iPhone 3G, it makes sense to use WiFi localization for route planning because GPS sub-sampling does not save energy on the iPhone (Chapter 2). However, even on a hypothetical iPhone with the ability to duty cycle GPS, it turns out WiFi localization would be preferable. Assuming GPS takes 6 seconds to acquire a "warm fix", sampling GPS at frequencies up to two minutes per sample would use more energy than WiFi. Sampling GPS less often than every two minutes is *much* less accurate than WiFi — resulting in a higher optimality gap than WiFi), and sampling GPS more often than that on the iPhone 3G drains the battery quicker than using WiFi. Our prototype iPhone implementation using WiFi estimation would use about $\frac{1}{6}^{th}$ of the total charge of the

| GPS Cost | Power Budget | Optimal Strategy |
|---|---|---|
| 0.5 | 0.1 | GPS 30 |
| 6 | 1 | GPS 36 |
| 7 | 1 | WiFi |
| 24.9 (iPhone) | 5 | GPS 30 |
| 24.9 (iPhone) | 3.5 | GPS 60 + WiFi |

**Table 3.2: Strategies for different power budgets and GPS to WiFi energy cost ratios.**

phone if it were run by a user for an hour a day with the screen on, which seems to be a reasonable level of battery consumption.

Similarly, Figure 3-13 shows that for hotspot detection, if WiFi sampling is very cheap such as on the iPhone it may be a good idea to *supplement* GPS sampling with WiFi localization. The figure shows that the *GPS 30 + WiFi* strategy yields a $5\times$ reduction in energy compared to GPS at 1 Hz, better than *GPS 20* (only a $3.3\times$ reduction) while having *equivalent accuracy*. Using WiFi alone for hotspot detection does not work owing to outages in the WiFi data, as we have discussed previously.

The results are quite different on the Android G1 phone where the relative cost of GPS is much lower than WiFi localization. On the G1, WiFi sampling is less than $2\times$ cheaper in terms of energy consumption than GPS, with the phone lasting a little over 10 hours with WiFi switched on, compared to a lifetime of 6 hours for GPS sampled at 1 Hz (Figure 2-1 in Chapter 2).

The G1 phone is also good at duty cycling GPS, acquiring a GPS "warm fix" in an average of 12 seconds each time the GPS is powered back on. Therefore, *GPS 24* is cheaper than or equivalent to WiFi localization in terms of energy consumption, while also being more accurate. On the Android G1 phone, GPS sub-sampling is consequently always a better strategy than WiFi localization.

### *Offline Optimization*

Given a characterization of the energy costs of WiFi and GPS localization on *any* mobile device and a power budget, the evaluation results presented in this chapter make it possible to perform this kind of analysis to derive an optimal sampling strategy for that device. *VTrack* uses such an offline optimization strategy, first measuring the energy costs of GPS and WiFi sampling using a battery drain experiment (such as that presented in Chapter 2) and then using the cost values to decide the sensor(s) and sampling rate(s) to use.

For any device, we measure the power consumption of sampling GPS at the max possible rate ($g$) and WiFi ($w$) and determine the ratio $g/w$ of the power consumed by GPS to that consumed by WiFi. Now, suppose we are given a target power budget $p$ and the ratio $g/w$, it is possible to perform an analysis to determine the best sensor(s) and sampling strategy to use on that device to maximize accuracy for the target application (e.g., routing or hotspot detection) for that particular combination of $p$ and $g/w$. The space of options we consider in this discussion are GPS every $k$ seconds for some $k$, WiFi or a combination of GPS every $k$ seconds with WiFi. It is possible to extend a similar analysis easily to other combinations of sensors, such as WiFi sub-sampling.

Table 3.7 shows the results of solving this optimization problem with route planning as the target application for some sample values of $p$ and $g$, assuming the WiFi cost $w$ is 1 unit.

For example, a power budget of $p = 2$ units in Table 3.7 means that *VTrack* is allowed to use at most twice the power consumed by WiFi.

Clearly, some settings of these parameters don't make sense — for example, $p < 1$ and $g > 1$ — so we only consider meaningful parameter settings. We see that there is a threshold of GPS cost $g$ beyond which using WiFi is the best option, as one would expect. Also, given a small power budget on a phone like the iPhone where GPS is power-hungry, *VTrack* starts to use WiFi localization.

Figure 3-18 illustrates the solution to the optimization problem visually to make the solution space to the energy optimization clearer. The solution is presented visually as a function of the power budget $\frac{p}{w}$ and the GPS sampling cost $\frac{g}{w}$, both expressed as ratios to WiFi sampling cost. To read the graph easily, first look at the case when $p = w$, i.e., the part of the graph along the x axis. Here, it is impossible to use both GPS and WiFi together, and there is a clear choice between using $GPS\ k$ and WiFi. For values of $g$ below a certain threshold, $GPS\ k$ is preferable and for values above, WiFi is preferable. Next, when $p \geq w$, $GPS + WiFi$ is always preferable to just WiFi, because the additional GPS points that can be sampled with the extra power budget never hurt accuracy.

The next choice we consider is between $GPS\ k$ for some sampling interval $k$, and $GPS\ k' + WiFi$ at a higher sampling interval $k'$, where $k'$ is chosen so that the energy cost of $GPS\ k' + WiFi$ and $GPS\ k$ are equal. First consider the case when $\frac{g}{w}$ is very large. In this case, the cost of WiFi is negligible, and for any $k$, $GPS\ k$ and $GPS\ k + WiFi$ have approximately the same energy consumption, so it purely comes down to which one is better in terms of accuracy. From our empirical data, for approximately $k = 20$ (labeled $k_{critical}$ in the graph), using *VTrack*'s interpolation with GPS is preferable to map-matching both GPS and WiFi together. Hence, whenever the power budget exceeds approximately that of GPS 20 (the dotted line) it is preferable to use only sub-sampled GPS. The graph also shows that for lower values of $g$, or beyond a certain power budget, it is better to use the power to increase the GPS sub-sampling rate (i.e., to reduce $k$) than to use WiFi.

In the figure, the iPhone 3G corresponds to the region of the graph where $\frac{g}{w}$ is large, and hence using `GPS + WiFi` or `WiFi` is preferable to just sub-sampling GPS. The Android G1, on the other hand, falls in the region of the graph where $\frac{g}{w}$ is relatively low and it is preferable to always use GPS sub-sampling and never use WiFi localization.

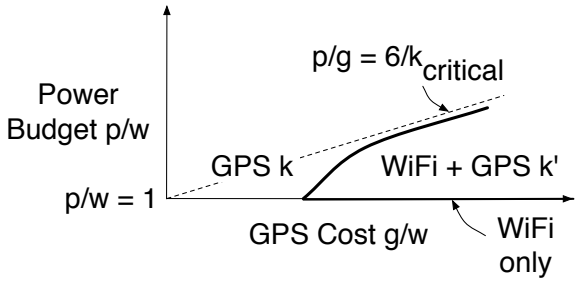



**Figure 3-18: Diagram showing optimal strategy as a function of power budget and GPS to WiFi energy cost ratio.**

**Limitations of Analysis.** We note that the accuracy results for routing and hotspot detection, and hence the optimal sensor sampling strategy to use, also depend on other factors not studied here. In a non-urban setting with lower road density, lower GPS sampling rates may be sufficient to estimate travel times accurately. There may be fewer or more WiFi hotspots in some areas, which would make WiFi a less or more viable strategy, respectively. At the opposite extreme, in very dense urban environments, WiFi localization might even be preferable to GPS. GPS in these settings may

perform worse than indicated by our results, because "urban canyon" effects from tall buildings impair its accuracy.

## 3.8 Related Work

As in the Cartel project, the Mobile Millennium project at UC Berkeley [1] built software to report traffic delays using mobile phones as probes. They focused on real-time traffic reporting. Claudel et al. [22, 21] develop a model for predicting flows and delays in the near future from traces of probe data. However, they assume GPS data and do not look into the effects of noisy data.

The NeriCell project [71] focused on monitoring road conditions and traffic using smartphones. Their goal was to combine data from GSM localization, GPS, and accelerometers to get a picture of road surface quality as well as traffic conditions, such as locations where users are braking aggressively. Their primary contributions are in processing accelerometer data, as well as power management. They do not provide algorithms for travel time estimation or map-matching.

Using Hidden Markov Models and Viterbi decoding for map-matching has been proposed by Hummel et al. [36] and Krumm et al. [44]. However, their work has mainly focused on map-matching GPS data with low noise. To the best of our knowledge, there have been no quantitative studies of accuracy. There has been some work in the database community on the map-matching problem [24, 76], which has largely focused on efficient algorithms for matching points to road segments when data is regularly sampled and not particularly noisy (comes from GPS or sensors without outages).

Gaonkar et al. [77] present the idea of a "micro-blog" where users can annotate locations and pick up other users' annotations as they travel. They use an energy-efficient location sampling technique, but focus on providing approximate estimates of location, rather than performing map matching or estimating travel time.

Yoon et al. [48] use GPS data to classify road conditions as "good" or "bad," both spatially and temporally (which reflect the steadiness and speed of traffic, respectively). They take frequently-sampled GPS points and calculate how a car's delay is distributed over each segment. This "cumulative time-location" data is converted to spatio-temporal data and then classified. This work differs from ours in that it assumes the ability to get relatively accurate travel time estimates on individual segments. Their method would likely fail on noisy data such as WiFi localization estimates, because the cumulative time-location measurements would be incorrect.

Privacy is an important concern in location-based smartphone applications, but is out of the scope of this dissertation. For completeness, we refer the reader to some related work on privacy-preserving smartphone traffic monitoring here. Using approaches inspired by the notion of *k-anonymity* [52], Gruteser and Grunwald [54] show how to protect locational privacy using spatial and temporal cloaking. A number of recent works show how to protect locational privacy while collecting vehicular traffic data [14, 43, 53] and in GPS traces [15]. In addition, some recent papers [16, 15] have developed tools to quantify the degree of mixing of cars on a road needed to assure anonymity (notably the "time to confusion" metric). The virtual triplines scheme [16] proposes a way to determine when it is "safe" from the standpoint of privacy for a vehicle to report its position using such a quantification. Many of these techniques could be used in *VTrack*.

## 3.9 Conclusion

This chapter presented *VTrack*, a system for trajectory mapping on mobile phones that can accurately estimate road travel times from a sequence of inaccurate and/or infrequent position samples

using a map-matching algorithm based on Hidden Markov Models. *VTrack* uses a measurement-driven optimization strategy to choose the best location sensor(s) (GPS or WiFi localization) and sampling rate(s) to use, on any given mobile device.

We evaluated *VTrack* on two end-to-end traffic monitoring applications: route planning and hotspot detection. We presented a series of results that showed an approach based on Hidden Markov Models can tolerate significant noise and outages in location estimates, while still providing sufficient accuracy for end-to-end traffic monitoring applications.

We presented an analysis using the evaluation results in this chapter that provides some answers to the question of which sensor(s) and sampling rate(s) to use for a given energy budget and mobile phone platform. A key result was that GPS sampled once every 40 seconds was approximately equivalent in accuracy to WiFi localization, and both WiFi and GPS sampled at a frequency up to once a minute worked reasonably well for map-matching.

One limitation of *VTrack* and the Hidden Markov Model we identified is that it begins to break down for increasing amounts of noise in the input data (Section 3.6.5), and in particular does *not* work for map-matching cellular location estimates. Since cellular localization is virtually free in terms of energy consumption on most phones (Chapter 2) this leaves open the following question:

- *Can we accurately map-match highly inaccurate cellular location data? How?*

This question is the focus of the next chapter.

# Chapter 4

# Map-Matching With Soft Information

The *VTrack* system presented in the previous chapter matches noisy geographic coordinates from GPS or WiFi localization to a sequence of road segments. With WiFi or cellular localization, the raw data collected is usually in the form of base stations and their signal strengths. If this *soft* information is available, it is possible to use it to significantly improve accuracy over *VTrack*-like strategies that use only "hard" information, i.e., $(lat, lon)$ coordinates. This chapter presents *CTrack* [6], a system that can use soft information from radios to perform more accurate map-matching than *VTrack*.

We implement and evaluate *CTrack* in the context of map-matching cellular signal data. Soft information is crucial to achieving reasonable accuracy for map-matching cellular signals. Cellular fingerprints can be seen from locations as far apart as hundreds of metres to a kilometre, even in densely populated urban areas with many cell towers. *Averaging* these locations using a process like centroid localization (described earlier) or fingerprinting [57] results in large errors in individual "location samples". As we have seen earlier, *VTrack* produces very inaccurate output trajectories when its input location coordinates have more than 70-80 metres of error.

## 4.1 Why Cellular?

The major advantage of a trajectory mapping system that can use purely cellular signals is that on a mobile phone, the marginal energy consumed for trajectory mapping is nearly zero (Chapter 2).

A second reason why cellular is interesting is that a large number of phones today do not have GPS or WiFi on them—85% of phones shipped in 2009, and projected to remain over 50% for the next five years [19]. The users of these devices, a disproportionate number of whom are in developing regions, are largely being left out of the many new location-based applications.

## 4.2 How CTrack Works

*CTrack* incorporates soft information by using a *two-pass* Hidden Markov Model. The first pass matches raw radio tower and signal strength data to a sequence of *grid cells* in the area of interest. The second pass uses *VTrack* to match the sequence of grid cells output by the first pass to a sequence of road segments on the road map.

The second contribution of *CTrack* is that it augments radio fingerprints with "sensor hints" from *inertial sensors* on a smartphone: the magnetic compass and the accelerometer. Specifically, *CTrack* extracts two kinds of binary hints:

- A *movement hint* that indicates if the phone is moving or not at a given time instant.
- A *turn hint* that indicates if the phone is turning or not at a given time instant.

*CTrack* fuses these "sensor hints" into its Markov Model to improve the accuracy of trajectory mapping. We show in this chapter that using movement and turn hints helps correct some of the systematic errors that arise with cellular localization, while consuming almost no energy.

### 4.2.1 Summary Of Results

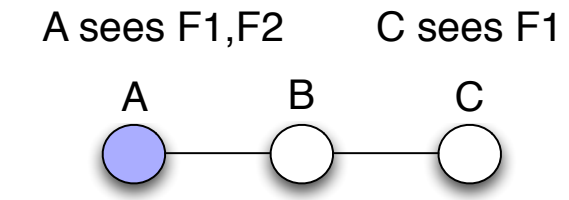
We have implemented *CTrack* on the Android smartphone platform, and evaluated it on nearly 125 drive hours of real cellular (GSM) signal data (1,074 total miles) from 20 Android phones in the Boston area. We find that:

- *CTrack* is good at identifying the sequence of road segments driven by a user, achieving 75% "precision" and 80% "recall" accuracy (these terms are defined later in this chapter). This is significantly better than state-of-the-art cellular fingerprinting approaches such as Placelab's algorithm [57] applied to the same data, reducing the error of trajectory matches by a factor of 2.5×.
- Although *CTrack* identifies the exact segment of travel incorrectly 25% of the time, trajectories produced by *CTrack* are on average only 45 metres away from the true trajectory. This implies that *CTrack* is more than adequate for applications like route visualization, or personalized drive monitoring applications that need to know which of $k$ possible routes a user took — for example, to find the best of $k$ different routes from a user's home to workplace in terms of travel time [64]. In this respect, *CTrack* is 3.5× better than map-matching "hard decision" coordinates from cellular fingerprints, which has a median error of 156 metres.
- *CTrack* on cellular signal data has a significantly better energy-accuracy trade-off than *GPS k*, the GPS sub-sampling strategy described in Chapter 3. For example, it reduces energy cost by a factor of 2.5× compared to *GPS k* for the same level of accuracy on the Android platform.
- In absolute terms, *CTrack* with cellular signal data uses very little energy on a modern smartphone. Our experiments on both the Android G1 and Nexus One phones show that these phones sampling data for *CTrack* have lifetimes close to their standby lifetime when not sampling any sensors.

The rest of this chapter is organized as follows. Section 4.3 explains how and when soft information helps map-matching. Section 4.4 describes the architecture of the *CTrack* system. Section 4.5 describes the actual *CTrack* algorithm. Section 4.6 describes sensor hints. Section 4.7 evaluates *CTrack* on real driving data collected on the Android platform. Section 4.8 discusses related work to *CTrack*, and Section 4.9 concludes this chapter.

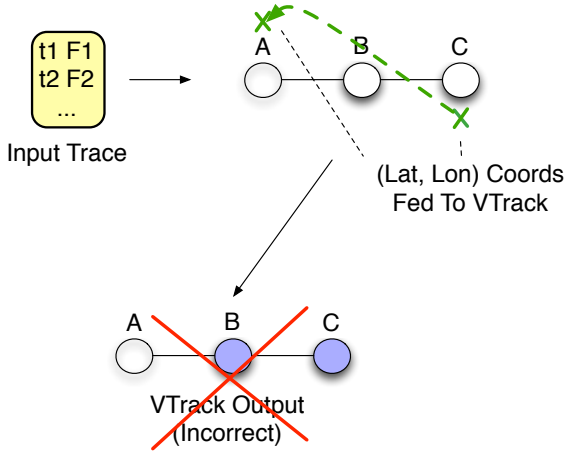
## 4.3 Why Soft Information Helps

We use the term *fingerprint* to refer to a set of observed IDs of WiFi or cellular base stations and their associated received signal strength (RSSI) values at any time instant. *CTrack* uses training data to compute a detailed probabilistic model of which fingerprints are seen from which geographic




Ground Truth = A

(a) Ground truth: device at location A.




(b) What *VTrack* would do.

**Figure 4-1: Example demonstrating the benefits of soft information for map-matching.**

locations. It uses this probabilistic model to match a sequence of fingerprints to a sequence of grid cells on a map.

An approach that does not use soft data would instead convert each fingerprint to the closest or most likely geographic location, and run *VTrack* over the resulting locations. There exist different approaches to converting a radio fingerprint to a geographic coordinate. For example, Placelab and RADAR [57] identify the closest matching fingerprint in their training databases for a given WiFi or GSM fingerprint, and output its location. One can also use a centroid or averaging approach as described earlier. All of these "hard decision" approaches must reduce a fingerprint to a single geographic location, which *loses valuable information* and results in lower trajectory mapping accuracy.

To understand why, consider a simplified world where there are only three discrete possible locations: A, B, and C, all of which lie on a straight line, with B midway in between A and C (as shown in Figure 4-1(a)). A mobile device can only transition one location in one time step (it can go from A to B, or B to C, but not directly from A to C). Suppose further that the device whose trajectory we are trying to map lies at location A throughout the experiment (though the algorithm we want to design does not know this).

Suppose we are given two radio fingerprints $F_1$ and $F_2$ from the device at two instants of time $t_1$ and $tF_2$. Our goal is to determine the trajectory of the mobile device. The true answer is that the device stayed at location A at both time instants. Both A and C see fingerprint $F_1$, but only A sees fingerprint $F_2$ (perhaps because they are equidistant from the same cell tower). The closest location

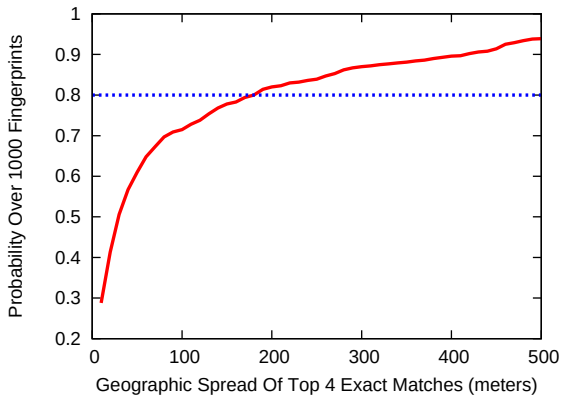



**Figure 4-2: Geographic spread of exact matches. The dashed line shows the 80th percentile.**

corresponding to fingerprint $F_1$ is C, while the closest location corresponding to fingerprint $F_2$ is A. In this situation, a centroid or closest-match algorithm that first converts the fingerprints to the closest matching location obtains $CA$ as the sequence of geographic locations to map-match (as shown in Figure 4-1(b)). When such a sequence is input to an algorithm like *VTrack*, it is likely to output the wrong answer (perhaps CB as shown in the figure or BA, but unlikely to be AA).

What we really need to do is to *not* throw out the "soft" information that $F_1$ is also likely to be seen from A, which is what a hard decision approach effectively ends up doing. Since fingerprint $F_1$ is likely to be seen from *both* A and C, the trajectory AA is most likely *even though the closest location to $F_1$ is C*.

To quantify this intuition on real cellular signal data, we selected approximately 1000 cellular fingerprints at random from a large training data set collected in the Boston area from Android G1 phones (as described later in this chapter). For each fingerprint $F$, we found all the *exact matches* for $F$, i.e., locations $F'$ with the exact same set of towers in the training data as $F$. We ordered the matches by similarity in signal strength, most similar first, and computed the geographic *diameter* of the top $k$ matches for each fingerprint (using $k = 4$).

The figure shows that more than 20% of matching sets have a diameter exceeding 150 meters, and more than 10% have a diameter exceeding 400 meters. The centroid or closest match approaches result in large "conversion error" for fingerprints with such a large geographic spread of locations.

Hence, the right approach is to use the soft information by keeping track of *all* possible likely locations a fingerprint is seen from, and using a continuity constraint to *sequence* these locations. This is the key intuition behind the *CTrack* algorithm we present later in this chapter.

A good analogy to understand *CTrack* vs *VTrack* is that of *soft-decision decoding*, as opposed to *hard-decision decoding* in wireless communication [47]. In hard-decision decoding, a sequence of symbols received on the wireless channel is converted to the most likely bit — either 0 or 1 without retaining any soft information about the likelihood that the symbol was in error. Soft-decision decoding, in contrast, preserves this likelihood information and passes it to the error correcting decoder.

The rest of this chapter describes and evaluates our implementation of *CTrack* on cellular (GSM) signal data.

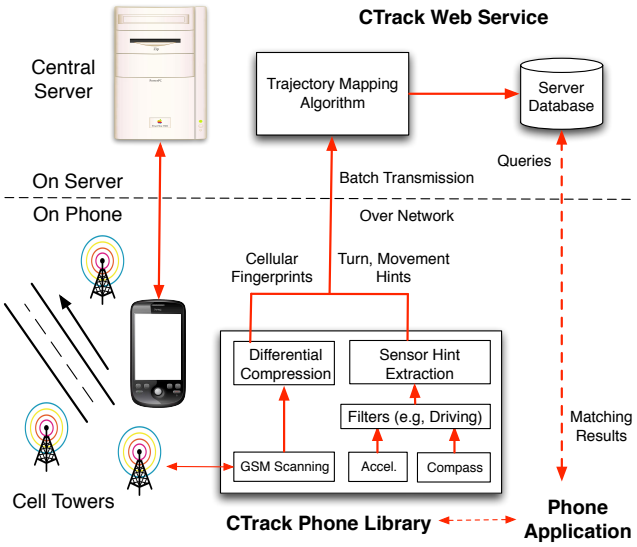



**Figure 4-3: *CTrack* system architecture.**

## 4.4 *CTrack* Architecture

Figure 4-3 shows the system architecture of *CTrack*. *CTrack* consists of two software components, the *CTrack Phone Library*, and the *CTrack Web Service*. The library collects, filters, and scans for GSM and sensor data on the phones, and transmits it periodically via any available wireless network (3G, WiFi, etc.) to the web service, which runs the trajectory mapping algorithm on batches of sensor data to produce map-matched trajectories. The mapping algorithm runs on the server to avoid storing complete copies of map data on the mobile device, and to provide a centralized database to which phone or web applications can connect to view and analyze matched tracks e.g., for visualizing road traffic or the path taken by a package or vehicle.

### 4.4.1 Phone Library

The phone library collects a list of neighbouring GSM towers sampled every 2-3 seconds approximately — because a cellular scan yields a new signature only once every 2-3 seconds. Optionally, if accelerometer, compass, or gyroscope sensors are available on the phone, the library also collects current values of *sensor hints* extracted from these inertial sensors. These sensor hints are binary values indicating if the phone is moving and/or turning. Section 4.6 describes how we extract sensor hints. The phone library uses the accelerometer to filter out portions where a user is stationary or walking, as described in [75, 5] and in Chapter 5 of this dissertation. This is to ensure we supply only relevant data to applications that want data only from moving vehicles. The library may also be configured to periodically collect GPS data for use in the training phase of *CTrack* from users who wish to contribute training data.

Since the goal of running *CTrack* on cellular data is to minimize energy consumption, it is important to minimize the energy consumption of transmitting cellular data over the network to the map-matching server. Our current implementation collects and transmits about 120 bytes/second of raw ASCII data on average. This quantity varies because the number of cell towers visible varies with location. In practice, it is possible to use two additional ideas to minimize the energy consumption of delivering sensor data over the network:

- Simple compression. Using gzip compression on our data set of 125 hours of test drives resulted in an average of just 11 bytes/second of data to be delivered.
- Batching the sensor data and uploading a batch every $t$ seconds. At 11 bytes/sec, with even small batches, using a 3G uplink with an upload speed of 30 kBytes/s (typical of most current 3G networks in the US) results in very low 3G radio duty cycles—for example, setting $t$ to 60 seconds results in the radio being awake only 0.03% of the time in theory. The duty cycle in practice is higher because the 3G radio does not immediately switch to a low power state after a transmission [63], but a large enough batching interval consumes a tiny amount of additional power. Once in 5 minute ($t = 300$) reporting is sufficient for most trajectory mapping applications (many of which use data from historical tracks in any case) — including recovering user tracks, traffic reporting, package tracking, and vehicular theft detection.

We chose not to run trajectory matching on the phone because it results in negligible bandwidth savings, while consuming extra CPU overhead and energy. For low data rates, the primary determinant of 3G or WiFi transmission energy is the transmitter duty cycle [63], making batch reports a good idea. However, we do extract sensor hints on the phone because the algorithms for hint extraction are simple and add negligible CPU overhead, while significantly reducing data rate. The raw data rate from sampling the accelerometer/compass without compression or hint extraction is about 1.3 MBytes/hour, which means that an application collecting this data from a user's phone for two hours a day could easily rack up a substantial energy and bandwidth bill without on-phone computation.

### 4.4.2 Web Service

The *CTrack* web service receives GSM fingerprints and converts them into map-matched trajectories. These matched trajectories are written into a database. Optionally, for real-time applications, the user's current segment can be sent directly back to the phone. A detailed description of the trajectory mapping algorithm is given in the next section.

## 4.5 *CTrack* Algorithm

The *CTrack* algorithm for map-matching a sequence of radio fingerprints differs from previous approaches in two key ways, as mentioned earlier. First, *CTrack* uses a two-pass algorithm which first matches cellular fingerprints to a sequence of spatial grids on a map, and then matches $(lat, lon)$ coordinates from the grids to road segments. The goal of the two pass algorithm is to avoid the loss of information from reducing a fingerprint to a single geographic location.

Second, *CTrack* optionally fuses sensor hints from the accelerometer and the compass to improve trajectory matching accuracy. Specifically, we show that turn hints can help remove spurious turns and kinks from GSM-mapped trajectories, and movement hints can help remove loops, a common problem with GSM localization when a vehicle is stationary.

### 4.5.1 Algorithm Outline

*CTrack* takes as input:

- A series of radio fingerprints from a mobile device, one per second in our implementation. In our implementation for GSM localization, the Android OS gives us the cell ID and the RSSI of up to 6 neighboring towers in addition to the associated cell tower. Each RSSI value is an integer on a scale from 0 to 31, where higher means a higher signal-to-noise ratio.

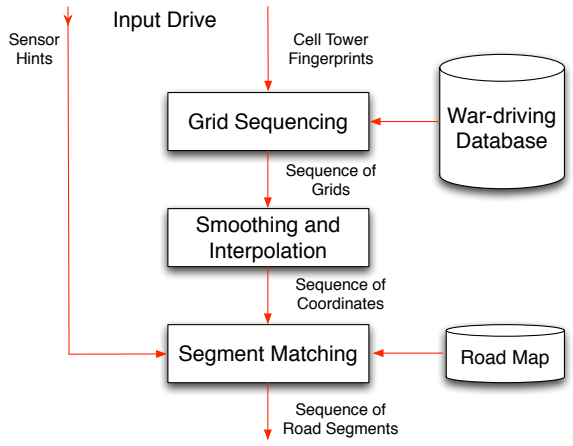



**Figure 4-4: Steps in *CTrack* algorithm.**

- If available, time series signals from accelerometer, and compass or gyroscope sampled at 20 Hz or higher. These are converted to "sensor hints" using on-phone processing as explained later in this chapter.
- A known map database that contains the geography of all road segments in the area of interest, similar to that used by *VTrack*. We use the NAVTEQ road map database for the experiments in this chapter.

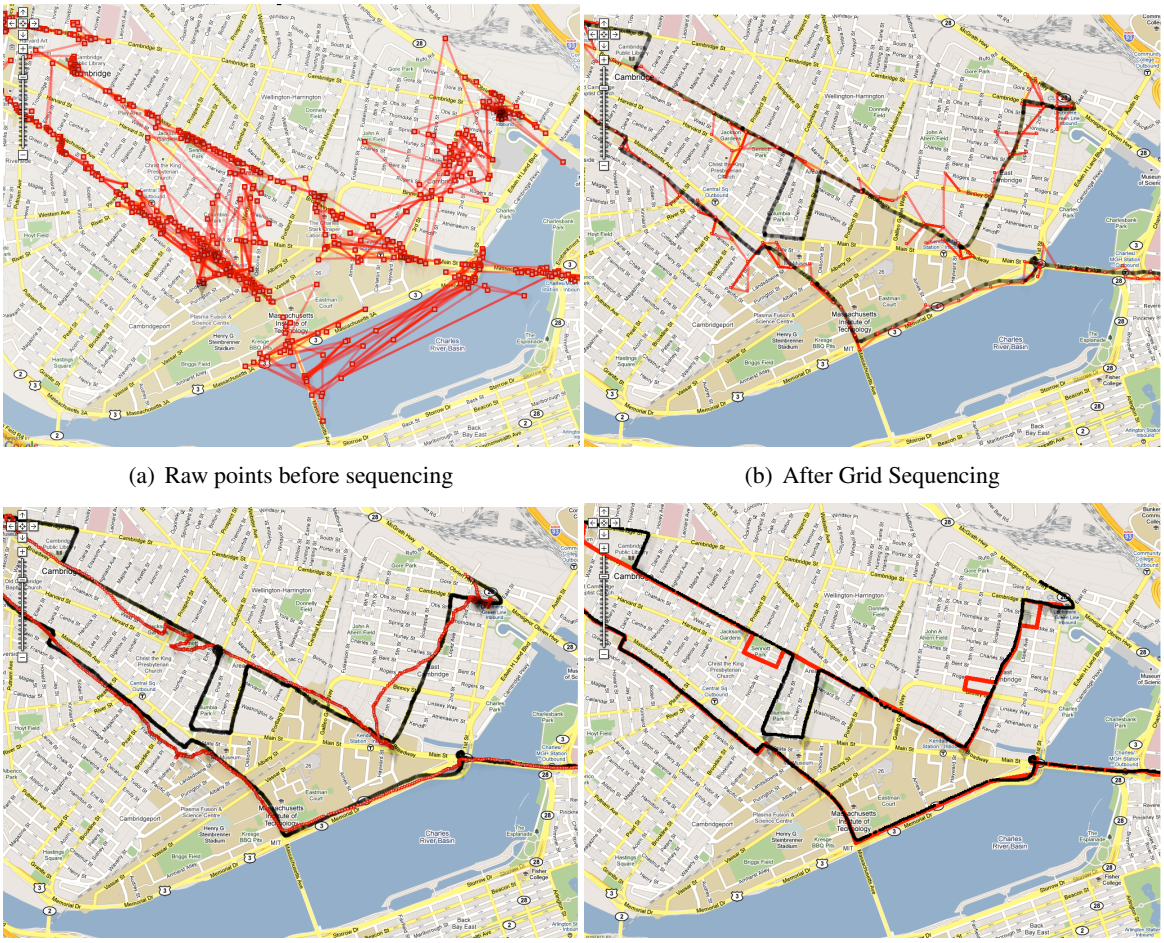
The output of *CTrack* is the likely sequence of road segments traversed, one for each time instant in the input.

Figure 4-4 shows the different components and passes of the *CTrack* algorithm. *Grid Sequencing* uses a Hidden Markov Model (HMM) to determine a sequence of spatial grid cells corresponding to an input sequence of radio fingerprints. The output of grid sequencing is *smoothed*, *interpolated*, and fed to *Segment Matching*, which matches grid cells to a road map using a *different* HMM. This stage also uses the output of a *Hint Extraction* phase (not shown), which processes raw accelerometer and compass data to extract *movement hints* and *turn hints* at different time instants in the drive. Offline, the *Training* phase (not shown) builds a training database, which maps ground truth locations from GPS to observed base stations/cell towers and their RSSI values.

Before we go into the details of *CTrack*, we point the reader to Figure 4-5. This figure illustrates the *CTrack* algorithm visually with an example. We shall use this trajectory as a running example throughout this section to explain how *CTrack* works.

We begin by noting that the input "raw points" in Figure 4-5(a) are shown only to illustrate the extent of noise in the input data. They are not actually used by *CTrack*, which uses radio fingerprints as inputs to its first step. The "raw points" in Figure 4-5(a) were computed by using the Place-lab/RADAR fingerprinting algorithm [57], where a radio (here, cell tower) fingerprint is assigned a location equal to the centroid of the closest $k$ fingerprints in the training database (we used $k = 4$). This approach is the state-of-the-art technique for computing a single closest match to a cellular fingerprint, and is shown here mainly to illustrate the extent of error in the raw location data without using any probabilistic model to sequence the data.

The next sections describe each stage of *CTrack*.



(a) Raw points before sequencing

(b) After Grid Sequencing

(c) After smoothing

(d) Final map-matched output

**Figure 4-5: *CTrack* map-matching pipeline. Black lines are ground truth and red points/lines are obtained from cellular fingerprints.**

### 4.5.2 Training

We divide the geographic area of interest into uniform square grid cells of fixed size $g_s$. We associate with each cell an ordered pair of positive integers $(x, y)$, where $(0, 0)$ represents the south-west corner of the area of interest. We use $g_s = 125$ meters. As we shall see, a smaller grid size results in higher accuracy if there is sufficient training data within each grid. However, too small a grid size can also result in lack of sufficient training data within each grid, as well as higher computational cost. Our grid size of 125 metres was chosen to balance running time and density of training data against the absolute algorithm precision (though we did find a reasonable range of grid sizes smaller and bigger than 125 metres to work reasonably well).

We train *CTrack* for the area of interest using software on mobile phones that logs a timestamped sequence of ground truth GPS locations and associated fingerprints. For each grid $G$ in the road map, our training database stores $F_G$, the set of distinct fingerprints seen from some training point in $G$. We did not use special rules to decide which training paths to drive, but we tried to cover all the road segments within the area of interest at least once.

Training can be done out-of-band using an approach similar to the Skyhook [78] fleet, which maps

WiFi access points to ground truth GPS coordinates. Once the training database is built, it can be used to map-match or track any drive, and needs to be updated relatively infrequently. We can also collect new training data in-band from consenting participating phones that use the *CTrack* web service whenever the user has enabled GPS.

### 4.5.3 Grid Sequencing

*CTrack*'s grid sequencing step uses a Hidden Markov Model to determine the sequence of grid cells corresponding to a timestamped sequence of fingerprints. For an overview of how Hidden Markov Models work, we refer the reader to Section 3.4.3 of Chapter 3.

In the HMM used for grid sequencing, the hidden states are grid cells and the observables are radio fingerprints. The emission score, $E(G, F)$ captures the likelihood of observing fingerprint $F$ in cell $G$. The transition score, $T(G_1, G_2)$, captures the likelihood of transitioning from cell $G_1$ to $G_2$ in a single time step.

*CTrack* first pre-processes input fingerprints using the *windowing* technique described below. *CTrack* then uses Viterbi decoding on the HMM to find the maximum likelihood sequence of grid cells corresponding to the windowed version of the input sequence.

We now describe the four key aspects of the HMM used by *CTrack* for grid sequencing: windowing, hidden states, emission score, and transition score.

#### *Windowing*

Because it is common for a single cell tower scan to miss some of the towers near the current location, we group fingerprints into *windows* rather than use raw fingerprints captured once per second. We aggregate the fingerprints seen over $W_{scan}$ seconds of scanning. to improve the reliability of the fingerprint. We chose $W_{scan} = 5$ seconds empirically: the phone typically sees all nearby cell towers within 3 scans, which takes about 5 seconds. In our evaluation, we show that windowing improves accuracy (Table 4.1 of this chapter).

#### *Hidden States*

The hidden states of our HMM are grid cells. Since there are an enormous number of grid cells in the entire road map, running the Viterbi dynamic programming algorithm over *all* the grid cells in the map is computationally intensive. It is necessary to intelligently identify a subset of grid cells that are relevant candidates for a given fingerprint and narrow down the search space of the Viterbi decoder. Recall that this was easy to do in *VTrack* where the radio fingerprints had already been converted to $(lat, lon)$ coordinates. *VTrack* could hence use a simple geographic approach to prune the state space of its Hidden Markov Model. This is less straightforward in *CTrack*.

*CTrack* uses the following heuristic for pruning its search space. Given an observed fingerprint $F$, a grid cell $G$ is a candidate hidden state for $F$ if there is at least one training fingerprint in $G$ that has at least one cell tower in common with $F$. Note that we might sometimes omit a valid possible hidden state $G$ if the training data for $G$ is sparse or non-existent. This is a problem because eliminating such a grid $G$ will also eliminate all paths that transition through road segments in $G$, incorrectly pruning the search space.

To overcome this problem, *CTrack* uses a simple *wireless propagation model* to predict the set of towers seen from cells that contain no training data. The model simply computes the centroid and

diameter of the set of all geographic locations from which each tower/base station is seen in the training data. The model draws a "virtual circle" with this centre and diameter and assumes that all cells in the circle see the tower in question.

Thanks to the propagation model, it is possible for *CTrack* to retain some flexibility in producing trajectories that pass through the occasional grid with little or no training data, but are otherwise highly likely conditioned on the input fingerprints.

### *Emission Score*

The emission score $E(F, G)$ is intended to be proportional to the likelihood that a fingerprint $F$ is observed from grid cell $G$. A larger emission score means that a cell is a more likely match for the observed fingerprint. *CTrack*'s emission score uses the following heuristic. We find $F_c$, the closest fingerprint to $F$ seen in training data for $G$. "Closest" is defined to be the value of $F_c$ that maximizes a pairwise emission score $E_P(F, F_c)$. The pairwise score is inspired by RADAR [69]. It captures both the number of matching towers, $M$, between two fingerprints, and the Euclidean distance $d_R$ in between the signal strength vectors of the matching towers:

$$E_P(F_1, F_2) = M\lambda_{match} + (d_R^{max} - d_R(F_1, F_2)) \tag{4.1}$$

where $\lambda_{match}$ is a weighting parameter and $d_R^{max} = 32$ is the maximum possible RSSI distance between two fingerprints. A higher number of matching towers and a lower value of $d_R$ both correspond to a higher emission score. The maximum value of the pairwise emission score is then *normalized* as described below, and assigned as the emission score for $F$.

As an example, consider the fingerprints {(ID=1, RSSI=3), (ID=2, RSSI=5)} and {(ID=1, RSSI=6), (ID=2, RSSI=4), (ID=3, RSSI=10)}. The distance between them would be $2\lambda_{match} + (32 - \frac{\sqrt{(3-6)^2+(5-4)^2}}{2})$. The weighting parameter affects how much weight is given to tower matches versus signal-strength matches: we chose $\lambda_{match} = 3$.

*CTrack* normalizes all emission scores to the range $(0, 1)$ to ensure that they are in the same range as the transition scores, which we discuss next.

### *Transition Scores*

The transition score in *CTrack* is given by:

$$T(G_1, G_2) = \begin{cases} \frac{1}{d(G_1,G_2)} & , G_1 \neq G_2 \\ 1 & , G_1 = G_2 \end{cases}$$

where $d(G_1, G_2)$ is the Manhattan distance between grid cells $G_1$ and $G_2$ represented as ordered pairs $(x_1, y_1)$ and $(x_2, y_2)$. This transition score is based on the intuition that, between successive time instants, the user either stayed in the same grid cell or moved to an adjacent grid cell. It is unlikely that jumps between non-adjacent cells occur, but we permit them with a small probability to handle gaps in input data. When implementing *CTrack* on cellular data, it turns out to be important to permit such a *lax* transition score because cellular data often has abrupt "jumps" or "transitions" and does not immediately react to user movement — a given fingerprint is seen for some time, and then a sudden jump takes place to a new fingerprint.

Figure 4-5(b) shows the output of the grid sequencing step for our running example. As we can see, sequencing removes a significant amount of noise from the input data. In our evaluation, we demonstrate that the sequencing step is critical (Figure 4-12).

### 4.5.4 Smoothing and Interpolation

The next stage of *CTrack* takes a grid sequence as input and converts it into a sequence of $(lat, lon)$ coordinates that are then processed by the *Segment Matching* stage. Note that this conversion does not significantly impact trajectory mapping accuracy, unlike the conversion used in "hard-decision" map-matching, where the fingerprints are converted to coordinates *before* doing the map-matching.

We describe the steps involved in smoothing and interpolation below.

#### *Centroid Computation*

For effective segment matching, we want to pick the most representative coordinate in a grid cell. If the grid cell has training points, we use the centroid of the training points seen inside the grid cell. If not, we output the geometric centre of the grid. We use this "centroid heuristic" because in many cases, all the training points seen in a grid lie on a particular road or sequence of roads. In these cases, the centroid has the advantage that it will also lie on a road segment and can subsequently be map-matched easily.

#### *Smoothing Filter*

Typically, centroids from grid sequencing have high frequency noise in the form of back-and-forth transitions between grids. Figure 4-5(b) illustrates this problem on our running example. To mitigate this, we apply a smoothing low-pass filter with a sliding window of size $W_{smooth}$ to the centroids calculated above. The filter computes and returns the centroid of centroids in each window. This filter helps us to accurately determine the overall direction of movement and filter out the high frequency noise. We chose the filter window size, $W_{smooth} = 10$ seconds, empirically.

#### *Interpolation*

Earlier, we windowed the input trace and grouped cellular scans over a longer window of $W_{scan}$ seconds. As a result, the smoothing filter produces only one point every $W_{scan}$ seconds. We linearly interpolate these points to obtain points sampled at a 1-second interval, and pass them as input to the *Segment Matching* step described in Section 4.5.5.

The reason for interpolation is that segment matching produces a continuous trajectory where each segment is mapped to at least one input point. As in *VTrack*, the minimum frequency of input to the segment matcher is one that ensures that even the smallest segment has at least one point, which is approximately once a second or more, much as in *VTrack*.

Figure 4-5(c) shows the example drive after smoothing and interpolation. This output is free of back-and-forth transitions and correctly fixes the direction of travel at each time instant. Table 4.1 in our evaluation quantifies the benefit of smoothing.

### 4.5.5 Segment Matching

Segment Matching in *CTrack* maps sequenced, smoothed grids from the previous stages to road segments on a map. It takes as input the sequence of points from the *Smoothing and Interpolation* phase, and binary (0/1) turn and movement hints from inertial sensors on the mobile phone, to determine the most likely sequence of segments traversed. We describe how movement and turn hints are extracted in Section 4.6.

For segment matching, *CTrack* uses a version of *VTrack* that is augmented to incorporate movement and turn hints as follows:

- The HMM hidden states are the set of possible triplets $\{S, H_M, H_T\}$, where $S$ is a road segment, $H_M \in \{0, 1\}$ is the current movement hint, and $H_T \in \{0, 1\}$ is the current turn hint.

- The emission score of a point $(lat, lon, H_M, H_T)$ from a state $(S, H'_M, H'_T)$ is zero if $H_M \neq H'_M$ or $H_T \neq H'_T$. Otherwise, we make it Gaussian just like in *VTrack*, proportional to $e^{-D^2}$, where $D$ is the distance of $(lat, lon)$ from road segment $S$.

- The transition score between two triplets $\{S^1, H^1_M, H^1_T\}$ and $\{S^2, H^2_M, H^2_T\}$ is defined as follows. It is 0 if segments $S^1$ and $S^2$ are not adjacent, disallowing a transition between them. This restriction ensures that the output of matching is a continuous trajectory. For all other cases, the base transition score is 1. We multiply this transition score with a *movement penalty*, $\lambda_{movement} (0 < \lambda_{movement} < 1)$, if $H^1_M = H^2_M = 0$ and $S_1 \neq S_2$, to penalize transitions to a different road when the device is not moving. We also multiply with a turn penalty, $\lambda_{turn} (0 < \lambda_{turn} < 1)$ if the transition represents a turn, but the sensor hints report no turn. We used $\lambda_{movement} = 0.1$ and $\lambda_{turn} = 0.1$. Our algorithm is not very sensitive to these values, since the penalties are multiplied together and a small enough value suffices to correct incorrect turn/movement patterns.

- Similar to *VTrack*, the HMM here also includes a *speed constraint* that disallows transitions out of a segment if sufficient time has not been spent on that segment. The maximum permitted speed can be calibrated depending on whether we are tracking a user on foot or in a vehicle.

The output of the segment matching stage is a set of segments, one per fingerprint in the interpolated trace. The output for the running example is shown in Figure 4-5(d).

When running online as part of the *CTrack* web service, the segment matcher takes turn hints and sequenced grids as input in each iteration and returns the current segment to an application querying the web service.

*Running Time*

As with *VTrack*, the run-time complexity of the entire *CTrack* algorithm, including all stages, is $O(mn)$, where $m$ is the number of input fingerprints and $n$ is the number of search states — which is the larger of the number of grid cells and road segments on the map. Our Java implementation of *CTrack* on a MacBook Pro with 2.33 GHz CPU and 3 GB RAM map-matched an hour-long trace in approximately two minutes, approximately 30 times faster than real time. It is straightforward to reduce the run time by more aggressively pruning the search space.

## 4.6 Sensor Hint Extraction

An important component of *CTrack* is the "sensor hint extraction layer" that processes raw phone accelerometer readings to infer information about whether the phone being tracked is moving or not, and processes orientation sensor readings from a compass or a gyroscope to heuristically infer vehicular turns. These hints are transmitted along with the GSM fingerprint to the server for map matching.

In this section, we provide some background on inertial sensors included on commodity smartphones, and what inertial sensor data on a smartphone represents. We then explain how *CTrack*

extracts useful hints from this data — as we shall see, this is a non-trivial problem. For more detail on how inertial sensors work, we refer readers to a more detailed explanation, with illustrations, in Chapter 5 of this dissertation (Section 5.3).

### 4.6.1 Inertial Sensors

Many modern smartphones including the latest generation of iPhone and Android phones are equipped with a 3-axis MEMS accelerometer and a 3-axis MEMS magnetometer (magnetic compass). Some are also equipped with a 3-axis gyroscope, such as the iPhone 4 and the Android Nexus S.

An accelerometer is capable of measuring a quantity proportional to the force experienced by the phone along three axes in space. The reported raw force value includes the effect of the earth's gravity. If we subtract out the gravity vector, the accelerometer measurement yields the actual acceleration experienced by the phone along three axes.

Similarly, a magnetometer uses measurements of magnetic field to estimate the position of magnetic north, which it can then correct to obtain the approximate orientation of the phone with respect to true (or magnetic) North. In our experiments with phones, we have found that the magnetometer works best at measuring absolute orientation of a phone when the plane of the phone is parallel to the earth's surface. However, as we shall see, the magnetometer is good at estimating *change in orientation* even if the plane of the phone is not strictly parallel to the earth's surface — it works across a range of orientations.

A magnetometer can be inaccurate when there is a significant amount of metal or electromagnetic interference in the vicinity of a phone. For this reason, newer phones include a 3-axis gyroscope that also measures orientation, but is robust to electromagnetic interference. A gyroscope does not directly measure orientation, but instead measures angular velocity around three axes: two in the plane of the phone and one perpendicular to this phone. This angular velocity can be integrated to obtain the instantaneous orientation of the phone at any instant. Chapter 5 describes this procedure in more detail.

### 4.6.2 Challenges

Our goal is to extract useful information from the inertial sensors to aid trajectory matching. The idea is that movement and direction of movement of a person/vehicle can be approximately inferred from inertial sensor data. There are two major challenges that must be overcome for this idea to work:

- The *movement of a phone does not always correspond to a movement of the user/vehicle carrying the phone*.

  For example, consider a user driving a vehicle with her phone in her pocket. As long as the phone is her pocket, the accelerometer on the phone accurately reflects the movement of the vehicle because the position of the phone is (approximately) fixed with respect to the vehicle. However, suppose the user now takes the phone out of her pocket to pick up a phone call. This movement will register on all the inertial sensors (accelerometer or compass/gyroscope), but will *not* reflect actual motion of the vehicle — it could be stopped at a traffic light for all we know.

- *The absolute orientation of a phone may not correspond to absolute orientation on a road map.* If a phone is in a user's pocket, for example, the readings from the magnetic compass

reflect only the phone's orientation with respect to magnetic north, which has little to do with the vehicle or user's direction of movement.

The first challenge is to accurately filter out inertial sensing data that does not come from times where the phone's position with respect to the user/vehicle being tracked is fixed. We refer to this process as "anomaly detection", and it is dealt with by a special component of *CTrack* that aims to remove these anomalies from raw sensor data.

The second challenge drives the design of how sensor hints are used in *CTrack*. *CTrack* only uses orientation values to measure the *change in orientation* or direction of movement of a user or vehicle, and *not* the absolute orientation. This works reasonably well even if the phone is in an arbitrary orientation, be it in a users' pocket or handbag, or the dashboard or coffee holder of a car, as long as anomaly detection filters out situations where this relative orientation/position changes abruptly.

Below we describe anomaly detection.

### 4.6.3 Anomaly Detection

"Anomaly detection" filters out periods when the user is lifting the phone, speaking on the phone, texting, waving the phone about, or otherwise using the phone.

We have found empirically in our experiments with the iPhone and the Android G1 and Nexus One phones that when driving with the phone at rest in a vehicle or in a pocket, the raw accelerometer magnitude tends to be smaller than 14 $\text{ms}^{-2}$. Hence, anomaly detection looks for spikes in the raw accelerometer magnitude that exceed a threshold of 14 $\text{ms}^{-2}$. Whenever we encounter such a spike, we ignore all accelerometer and compass data in the map-matching algorithm until the phone comes back to a state of rest (this can be detected using standard deviation of acceleration, as explained below). On more recent phones such as the iPhone 4, the in-built gyroscope gives the exact orientation of the phone which can be directly read to determine if the phone is on a flat surface/in a user's pocket.

Figure 4-6 illustrates anomaly detection real data. The driver had the phone in his pocket while driving until $t = 1767$; at this point, he took the phone out of his pocket, causing a sudden spike. The driver replaced the phone in his pocket a while later. We cannot immediately infer that it is safe to begin using sensor hints, but we do know that the phone came to a state of complete rest after $t = 1880$; at this point, we can start using the accelerometer and compass hints once again.

Having filtered out "anomalous periods", the hint extraction layer proceeds to process the left over periods, which we call "stable periods", to extract movement and turn hints, as explained in the sections below.

### 4.6.4 Movement Hint Extraction

The logical way to use raw accelerometer data to infer changes in position is to *integrate* it. However, this is challenging, as we explain below.

Recall that acceleration, being the second derivative of position, needs to be integrated twice to obtain displacement, or change in position. Integrating once yields the change in velocity over some time period, and then integrating the velocity again yields the displacement. Any measurement on an MEMS sensor is subject to noise. A simple, and reasonably accurate model of sensor error is to assume the raw acceleration values have a random zero-mean Gaussian error with a standard

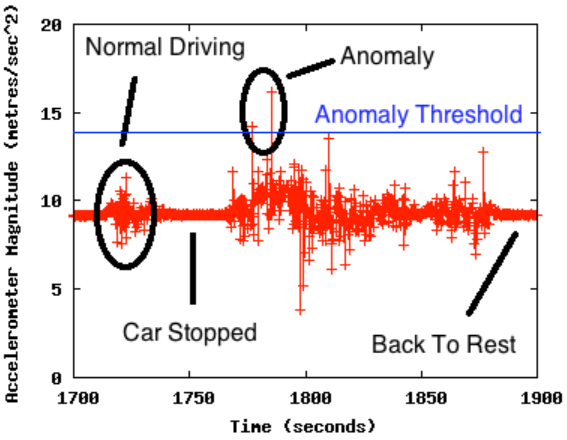



**Figure 4-6: Anomaly detection in accelerometer data.**

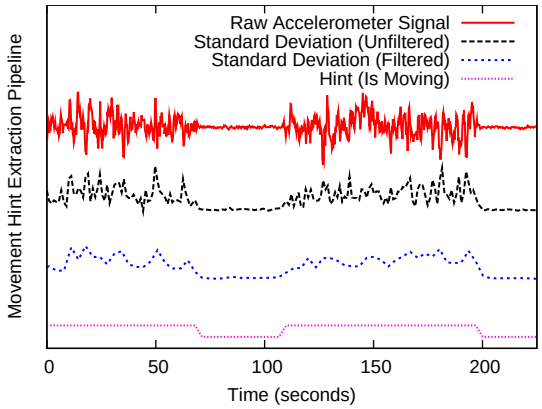



**Figure 4-7: Movement hint extraction from the accelerometer.**

deviation of $N_{acc}$. If this is the case, the mean *squared* error in velocity computed by integrating raw acceleration data will grow linearly with the number of observations integrated (or equivalently, linearly with time). This is analogous to the classic "drunkard's random walk", or Brownian motion in a straight line: if a person randomly moves 1 step backward or forward in each time step, the expected distance from the origin after N steps is proportional to $\sqrt{N}$.

The error in velocity is biased in one direction. Hence integrating the velocity with time causes the *drift*, or error in position to grow approximately as $O(t^{3/2})$, where $t$ is time (or number of observations being integrated). This is a fast growth rate, and in practice quickly results in integration results from accelerometry being unusable.

Because accelerometer data is noisy and difficult to integrate accurately without accumulating significant error drift, *CTrack* instead extracts a simple "static" or "moving" (0/1 hint) from the accelerometer, rather than integrating the accelerometer data to compute velocities or processing it in a more complex way.

In contrast to integration, as we and other researchers have observed [75, 5] it is relatively easy to detect movement with an accelerometer: within a stable (spike-free) period, *the accelerometer shows a significantly higher variance while moving than when stationary*.

Accordingly, *CTrack* uses standard deviation to compute a boolean (true/false) movement hint for each time slot. We divide the data into one-second slots and compute the standard deviation of the 3-axis magnitude of the acceleration in each slot. Directly thresholding standard deviation sometimes results in spurious detections when the vehicle is static and the signal exhibits a short-lived outlier. To fix this, we apply an exponential weighted moving average (EWMA) filter to the stream of standard deviations to remove short-lived outliers. We then apply a threshold $\sigma_{movement}$, on the standard deviation to label each time slot as "static" or "moving".

We used a subset of our driving data across multiple phones as training (where we know ground truth from GPS) to learn the optimal value of $\sigma_{movement}$. The best value turned out to be approximately 0.15 $ms^{-2}$, when the standard deviation is computed over a one-second window.

Figure 4-7 visually illustrates the steps of *CTrack*'s movement hint extraction algorithm on example accelerometer data from an Android G1 phone.

*CTrack* uses accelerometer data sampled at 20 Hz. A few (5-10) samples of accelerometer data are sufficient to detect movement using the technique described above. If it is sufficient to detect movement within a second, any sampling frequency higher than 5 Hz should work for movement hint extraction.

### 4.6.5 Turn Hint Extraction

As we have discussed earlier, the information provided by the orientation sensor on a smartphone depends on whether it comes from a magnetometer or gyroscope. A magnetic compass provides orientation of the plane of the phone with respect to geographic or magnetic north, while a gyroscope provides angular velocity about three axis which can be integrated to obtain accurate orientation of the phone with respect to a "world reference frame". A magnetometer is less accurate than a gyroscope to obtain orientation because it often suffers significant electromagnetic interference, usually from nearby metallic objects.

However, at the time *CTrack* was originally implemented, the magnetometer was the only way to get orientation information on smartphones since no smartphone came with a gyroscope. This has since changed, with many newer smartphones including gyroscopes. For this reason, the rest of the discussion in this chapter is focused mainly on using magnetometer orientation information for turn hints. The basic algorithm we describe is applicable to gyroscopes as well, but because a gyroscope can provide more accurate information, it is likely that the accuracy of *CTrack* can be significantly improved if using a gyroscope, by extracting *much richer* turn hint information than is possible with a magnetometer owing to its accuracy limitations. We refer the interested reader to the *iTrack* algorithm for indoor trajectory mapping discussed in the next chapter, which extracts very rich sensor hint information from the phone gyroscope.

The orientation sensor APIs on most smartphone platforms provide orientation about three axes. We are most interested in the axis that provides the relative rotation of the phone about an axis parallel to gravity (called "yaw" in the iPhone 4 and Android conventions).

As mentioned earlier, because a phone can be in any orientation in a handbag or pocket, *CTrack* does *not* use absolute orientation information. The key observation used instead is that *irrespective of how the phone is situated*, a true change in orientation manifests as a *persistent*, *significant*, and *steep* change in the value of the orientation sensor.

With magnetic compass data, the main challenge is that the orientation reported is noisy owing to interference from nearby metal, or because the compass goes out of calibration for some other

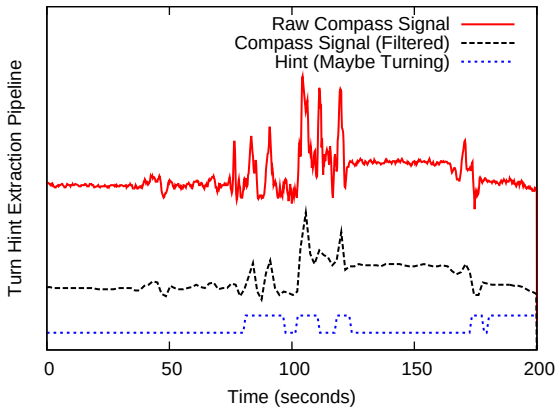



**Figure 4-8: Turn hint extraction from the magnetic compass.**

reason. *CTrack* solves this problem by applying a *median filter* with a 3-second window to the raw orientation values, which filters out non-persistent noise with considerable success. We note that the obvious approach of using a *mean*, or EWMA filter fails. A mean filter also successfully removes noise, but it tends to *blur* sharp transitions in the compass value, which is *exactly what we want to detect.*

After applying a median filter, *CTrack* finds transitions in the raw compass value with a magnitude exceeding at least 20 degrees and slope exceeding a minimum threshold (signifying a sharp change as opposed to a slow drift of the compass value). We fixed the slope threshold at 1.5 by experimentation.

Figure 4-8 illustrates a plot of the compass data with the sequence of processing steps required to generate a turn hint.

**Limitation.** We note that the above filtering approach is not perfect. In particular, it can (and often does) produce false positives. A true change in orientation can sometimes be produced by a phone sliding around within a pocket or a bag, or changing orientation for reasons other than the user or vehicle being tracked actually turning. However, we find that in practice, the filtering approach successfully eliminates *false negatives* — i.e., failing to detect a possible turn. For this reason, the segment matching HMM used in *CTrack* is designed specifically to penalize paths through the Viterbi decoder that take a turn when the reported sensor hint is "no turn", but *not* vice-versa.

## 4.7 Evaluation

This section evaluates the *CTrack* algorithm. We show that the trajectory matches produced by *CTrack* on cellular (GSM) data are:

- Accurate enough to be useful for various tracking and positioning applications,
- Superior to sub-sampled GPS in terms of the accuracy-energy tradeoff, and
- Significantly better than using only "hard" information, by reducing each cellular fingerprint to the closest location before matching.

We investigate how much each of the four techniques used in *CTrack* — sequencing, windowing, smoothing, and sensor hints — contribute to the gains in trajectory mapping accuracy.

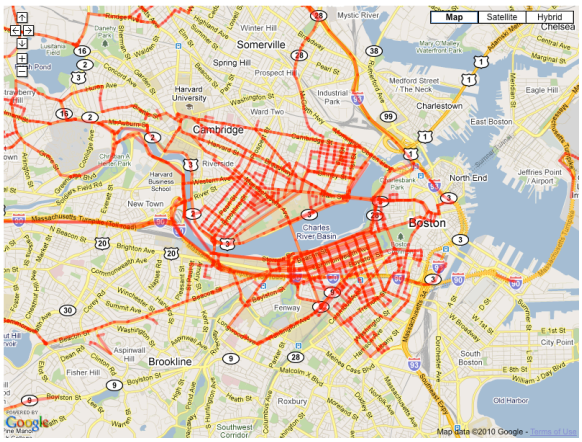



**Figure 4-9: Coverage map of driving data set.**

### 4.7.1 Method and Metrics

We evaluate *CTrack* on 126 hours of driving data in the Cambridge-Boston area, collected from 15 Android G1 phones and one Nexus One phone over a period of 4 months. We configured the *CTrack* phone library for the Android OS to continuously log the ground truth GPS location and the cell tower fingerprint every second, and the accelerometer and compass at 20 Hz. The data set covers 3,747 road segments, amounts to 1,718 km of driving, and 560 km of distinct road segments driven. The data set includes sightings of 857 distinct cell towers. Figure 4-9 shows a coverage map of the distinct road segments driven in our data set.

From 312 drives in all, we selected a subset of 53 drives verified manually to have high GPS accuracy as *test drives*, amounting to 109 distinct kilometres driven. We picked a limited subset as test drives to ensure each test drive was contained entirely within a small bounding box with dense training coverage. This is because evaluating the algorithm in areas of sparse coverage (which many of the other 259 drives venture into) could bias results in favour of *CTrack* by reducing the number of candidate paths to map-match to.

For each test drive, we perform *leave-one-out* evaluation of the map-matching algorithm: we train *CTrack* on all 311 drives excluding the test drive, and then map-match the test drive. We do this to ensure enough training data for each drive, and at the same time to keep the evaluation fair.

We compare *CTrack* to two other strategies in terms of energy and accuracy:

- *GPS k* gets one GPS sample every $k$ minutes ($k = 2, 4$), interpolates, and map-matches it using *VTrack* (Chapter 3).
- *Placelab-VTrack* computes the best geographic location for each time instant using Placelab's fingerprinting technique [57], and matches the locations using *VTrack*.

We use three metrics in our evaluation of accuracy: *precision*, *recall*, and *geographic error*. The "precision" and "recall" we use are similar to the metrics used in the micro-benchmarks for *VTrack* in Chapter 3. They are similar to conventional precision and recall, but take the order of matched segments in the trajectory into account. We say that a subset of segments in a trajectory $T_1$ that also appears in trajectory $T_2$ are *aligned* if those segments appear in $T_1$ in the same order in which they

appear in $T_2$. Given a ground truth sequence of segments $G$ and an output sequence $X$ to evaluate (produced by one of the algorithms), we run a dynamic program to find the maximum length of aligned segments between $G$ and $X$. We define:

$$Precision = \frac{Total\ length\ of\ aligned\ segments}{Total\ length\ of\ X} \tag{4.2}$$

$$Recall = \frac{Total\ length\ of\ aligned\ segments}{Total\ length\ of\ G} \tag{4.3}$$

We estimate the ground truth sequencing of segments by map-matching GPS data sampled every second with *VTrack*, and manually fixing a few minor flaws in the results.

In one sense, for applications like traffic monitoring, we care more about precision than recall because we want to minimize incorrect output, but maintaining a minimum level of recall is important because it would be trivial (but useless) for an algorithm to output no segments and achieve perfect precision.

**Geographic error.** Precision and recall are relevant to applications that care about obtaining information at a segment-level, such as traffic monitoring. However, applications such as visualization or finding if the route a vehicle took was one of $k$ pre-defined paths (e.g. [64]) do not need to know the exact road segments traversed, but may want to identify the broad contours of the route followed. In such applications, mistaking a road for a nearby parallel road may be acceptable in some cases.

To quantify this notion, we compute a third metric, *geographic error*, which captures the spatial distance between the ground truth and the matched output. We compute the maximum alignment between the ground truth trajectory $G$ and output trajectory $X$ using dynamic programming. This alignment matches each segment $S$ of $X$ to either the same segment $S$ on $G$ (if *CTrack* matched that segment correctly) or to a segment $S_{wrong} \in G$ (if matched incorrectly).

We define the *segment geographic error* to be the distance between $S$ and $S_{wrong}$ for incorrect segments, and 0 for correctly matched segments. The mean segment geographic error over all segments in $X$ is the *overall geographic error*.

A small value of the geographic error implies that the algorithm finds a trajectory very close to the original trajectory in terms of distance. A small overall geographic error means that most segments are matched correctly (contributing zero error), or that incorrectly matched segments are close to corresponding segments on the ground truth (rather than completely off), or both.

### 4.7.2 Key Findings

The key findings of our evaluation are:

- *CTrack* on cellular data has 75% precision and 80% recall in both the mean and median, and a median geographic error of 44.7 meters. We discuss what these numbers mean in the context of real applications below.
- *CTrack* has 2.5× better precision and 3.5× smaller geographic error than *Placelab+VTrack*.

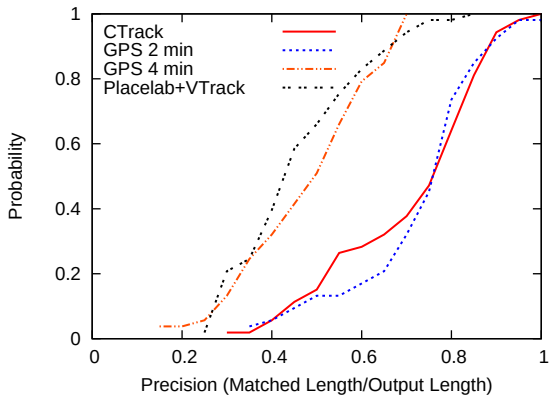



**Figure 4-10: CDF of precision: *CTrack* is better than *Placelab + VTrack*.**

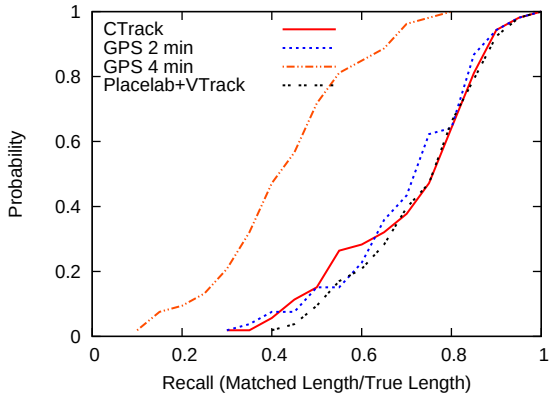



**Figure 4-11: CDF of recall: comparison.**

- *CTrack* is equivalent in precision to map-matching GPS sub-sampled every 2 minutes while consuming over 2.5× less energy. It also reduces error $(1 - precision)$ by a factor of over 2× compared to sub-sampling GPS every 4 minutes, consuming a similar amount of energy. *CTrack* is 6× better than continuous WiFi sampling in terms of battery lifetime on the Android platform, with somewhat lower precision. Going by the "segment error rate" results from *VTrack*, WiFi has a precision of about 85%, 10% more than *CTrack*.

- The first step of *CTrack*, grid sequencing, is critical. Without grid sequencing, *CTrack* effectively reduces to computing a $(lat, lon)$ estimate from the best fingerprint match, and ignoring a variety of location matches for that fingerprint. The median precision when not using sequencing is only 50%, similar to the accuracy of *VTrack* when run on cellular positioning data from Placelab. See Section 4.7.5 for more detail.

- We can extract movement and turn hints from raw sensor data with approximately 75% precision and recall. These hints improve accuracy by removing spurious "loops" and "turns" in the output. Using sensor hints improves the precision of trajectory mapping by 6% and the recall by 3%. See Section 4.7.6 for more detail.

### 4.7.3 Accuracy Results

The questions we aim to answer in this section are: is overall trajectory matching accuracy with cellular data and *CTrack* adequate for the kind of applications we are interested in? How does *CTrack* stack up against the other methods (in particular *VTrack* applied to cellular data, or to sub-sampled GPS data) in terms of accuracy and energy consumption?

Figure 4-10 shows a CDF of the map-matching precision for *CTrack*, *GPS k* (for $k = 2, 4$ minutes) and *Placelab+VTrack*. We highlight some important points from the figure:

- *CTrack* has a median precision of 75%, much higher than the both the energy-equivalent strategy of sub-sampling GPS every 4 minutes (48%), and *Placelab+VTrack* (42%).
- In effect, *CTrack* has over 2× lower error ($1 - precision$) than sub-sampling GPS every 4 minutes, and over 2.5× lower error than map-matching cellular localization estimates output by the Placelab method.
- Also, *CTrack* has equivalent precision to map-matching GPS sub-sampled every two minutes, while reducing energy consumption by approximately 2.5× compared to this approach (Figure 2-1, Chapter 2).

Figure 4-11 shows a CDF of the recall. All the strategies except *GPS 4* min are equivalent in terms of recall. Sub-sampling GPS every four minutes has poor recall (median only 41%) because a four-minute sampling interval misses significant turns in our input drives and finds the wrong path. The fact that *Placelab+VTrack* has identical recall shows that simple cellular localization and hard-decision map-matching do manage to recover a significant part of the input drive. However, converting cellular fingerprints to coordinates before map-matching them results in significant noise and long-lived outliers, and hence produces a large number of incorrect segments when map-matched directly, resulting in low precision.

### 4.7.4 What Does 75% Precision Mean?

To understand what 75% precision might mean in terms of a an actual application, we refer readers to the results from Chapter 3 on *VTrack*, which study the relationship between map-matching accuracy and the accuracy of two end-to-end applications: traffic delay monitoring and traffic hot-spot detection. In Chapter 3, we found that sub-sampled GPS and WiFi localization, which have median precision of the order of 85% (corresponding to a "Segment Error Rate" of 15% in the terminology used there) were usable for accurate traffic delay estimation. Our results for cellular (75%) are only somewhat worse, and while not directly comparable, they suggest a significant portion of travel time data from *CTrack* could be useful.

For applications such as route visualization, or those that aggregate statistics over paths (e.g., to compute histograms over which of $k$ possible routes is taken), or those that simply show a user's location on a map, getting most segments right with a low overall error is likely sufficient. The median geographic error when using *CTrack* is quite low—just 45 meters—suggesting *CTrack* would have sufficient accuracy for such applications. In contrast, the median geographic error of the Placelab+VTrack approach is 156 meters, over 3.5× worse than *CTrack*.

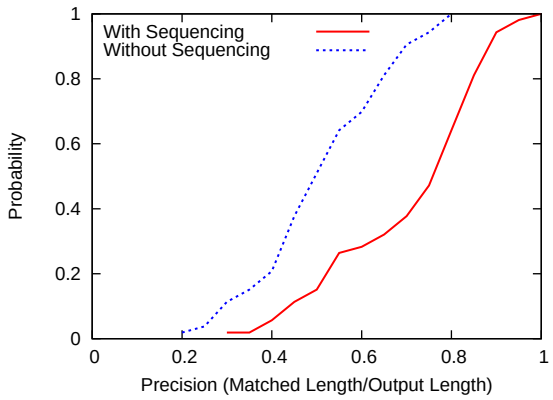



**Figure 4-12: Precision with and without grid sequencing.**

### 4.7.5 Benefit of Sequencing

This section elaborates on the key technical contribution of *CTrack*; the idea that the first pass of grid sequencing *before* converting fingerprints to geographic locations is crucial to achieving good matching accuracy. We provide experimental evidence supporting this idea. We also show that windowing and smoothing help improve matching accuracy, though to a lesser extent.

Figure 4-12 is a CDF that compares the precision of *CTrack* with and without the first pass of grid sequencing. This figure shows that sequencing is critical to achieving reasonable accuracy: without sequencing, the median precision drops from 75% to 50%. The reason is that running *CTrack* without sequencing amounts to reducing each fingerprint to its best match in the training database, and running *VTrack* on it.

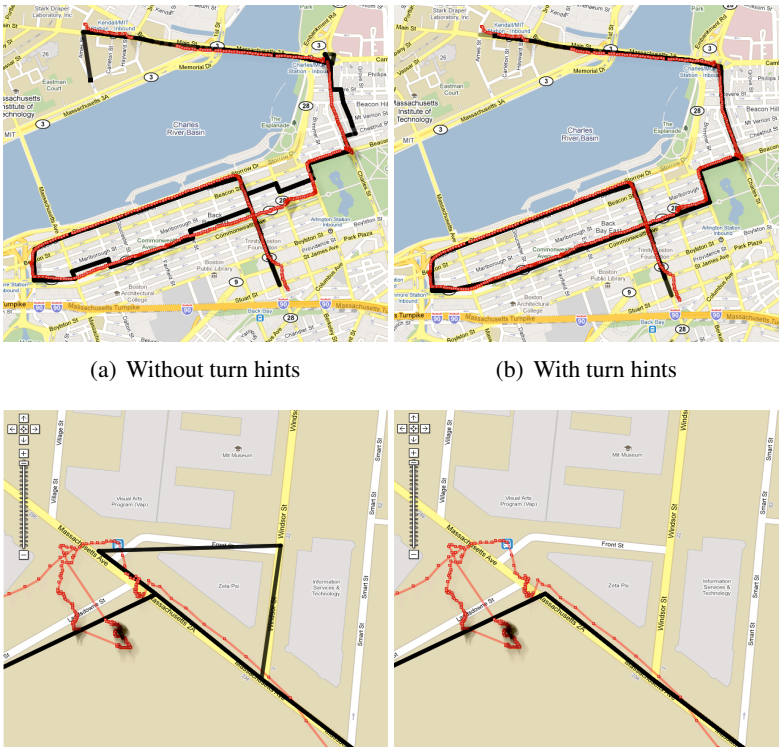
**Windowing and Smoothing** Table 4.1 shows the precision and recall of *CTrack* with and without windowing and smoothing, two other heuristics used in *CTrack*. We see that each of these features improves the precision by approximately 10%, which is a noticeable amount. The recall does not improve because the algorithm without windowing/smoothing is good enough to identify most of the segments driven: the heuristics mainly help eliminate loops in the output, which arise from the back-and-forth grid transitions output by the first pass HMM.

| | With | | Without | |
|---|---|---|---|---|
| | **Prec.** | **Recall** | **Prec.** | **Recall** |
| Windowing | 75.4% | 80.3% | 65.6% | 82.3% |
| Smoothing | 75.4% | 80.3% | 66.5% | 82.5% |

**Table 4.1: Windowing and smoothing improve median trajectory matching precision.**

### 4.7.6 Do Sensor Hints Help?

Figure 4-13 illustrates by example how turn hints extracted from the phone compass help in trajectory matching. Without using turn hints (Figure 4-13(a)), our algorithm finds the overall path quite accurately but includes several spurious turns and kinks, owing to errors in cellular localization. After including turn hints in the HMM, the false turns and kinks disappear (Figure 4-13(b)).



(a) Without turn hints (b) With turn hints

(c) Without movement hints (d) With movement hints

**Figure 4-13: Sensor hints from the compass and accelerometer aid map-matching. Red points show ground truth and the black line is the matched trajectory.**

In Figure 4-13(c), the driver stopped at a gas station to refuel, which can be seen from the cluster of ground-truth GPS points. Before using movement hints, errors from cellular localization were spread out, causing the map-matching to introduce a loop not present in the ground truth (Figure 4-13(c)). After incorporating movement hints, the speed constraint in *CTrack*'s HMM eliminates this loop because it detects that the car would not have had sufficient time to complete the loop (Figure 4-13(d)).

**Limitation.** We note a limitation of the movement hint. Stop detection works because the phone was placed on the dashboard: if it had been in the driver's pocket during refueling, the movement hints would not have helped had the driver gotten out of the car and been moving about, since that portion would have been filtered out.

Figure 4-14 is a CDF that compares the precision of *CTrack* with and without sensor hints (both movement and turn). This figure shows that sensor hints improve the median precision of matching by approximately 6%. While this may not seem huge, there exist several trajectories for which the hints do help significantly, suggesting that using them is a good idea when available. In our experience, the main benefit of the hints is in eliminating the several "kinks" and spurious turns in the matched trajectory, which the quantitative metrics don't adequately capture. We expect that more accurate turn hints, such as from a gyroscope (as opposed to a magnetometer) should yield greater benefit.

We now drill down into how accurately our approach is able to extract individual movement and turn

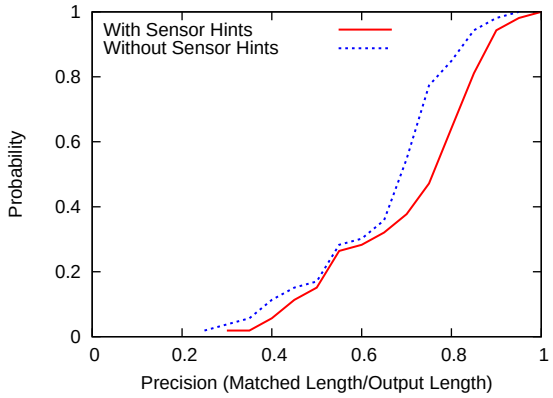



**Figure 4-14: Precision with and without sensor hints.**

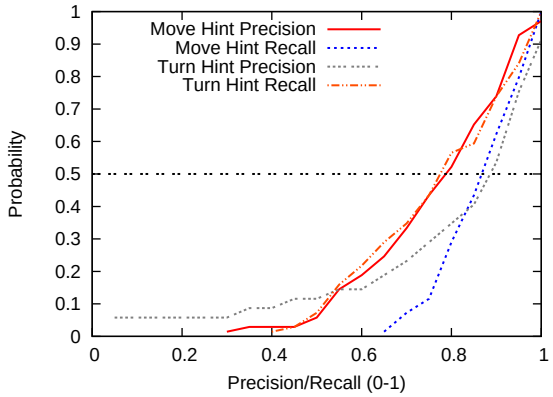



**Figure 4-15: CDF showing precision/recall for hint extraction.**

hints. We used the ground truth GPS for all of our test drives to compute the status of the device in each slot (i.e. moving or stopped, turning or not turning). We then computed precision and recall metrics, defined as follows. For motion hints, the precision is the fraction of stops we report that are actually stops, and the recall is the fraction of stopped time slots that we report as stopped. We compute the precision for stop detection because a movement hint of 0 is what impacts the transition score in our HMM (Section 4.5.5). Similarly, for turn hints, the precision is the fraction of "no-turn" zones we report that actually do not contain a turn, and the recall is the fraction of "no-turn" zones we detect as "no-turn" zones.

Figure 4-15 shows that the precision and recall of motion and turn hint extraction all exceed 75%. We do not claim that *CTrack*'s algorithms for hint extraction are the best possible; our broader point is to show reasonably accurate hint extraction is feasible and helps trajectory matching.

### 4.7.7 How Much Training Data?

An important aspect of deploying a cellular location system based on *CTrack* is that it requires effort to build a training database of cell towers by driving around the geographic area of interest. This section seeks to answer the question of *how much training data* is required to bootstrap a system like *CTrack*.

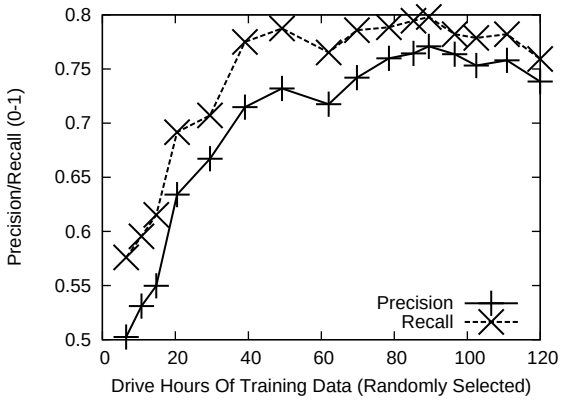



**Figure 4-16: Precision/recall as a function of the amount of training data.**

To quantify the amount of training data essential to achieving good trajectory mapping accuracy with *CTrack*, we picked a pool of test drives at random, amounting to 5% of our data set (8 hours of data), and designated the remaining 95% as the training pool. We picked subsets of the training pool of increasing size, i.e., first using fewer drives for training, then using more. In each run, the training subset was used to train *CTrack* and then evaluated on the test pool. Figure 4-16 shows the mean precision and recall of *CTrack* on the test pool as a function of the number of drive hours of training data used to train the system. The accuracy is poor for very small training pools, as expected, but encouragingly, it quickly increases as more training data is available. The algorithm performs almost as accurately with 40 hours of training data as with 120, suggesting that 40 hours of training is sufficient for our data set.

The 40-hour number, of course, is specific to the geographic area we covered in and around Boston, and to the test pool. To gain more general insight, we measure the *drive count* for each road segment in the test pool, defined as the number of times the segment is traversed by any drive in the training pool. Figure 4-17 shows the distribution of *test segment drive counts* corresponding to 40 hours of training data. While the mean drive count is approximately 3, this does not mean each road segment on the map needs to be driven thrice to achieve good accuracy. As the graph shows, about 60% of the test segments were not traversed even once in the training pool, but we can still map-match many of these segments correctly. The reason is that they lie in the same grid cell as some nearby segment that was driven in the training pool. This result is promising because it suggests that training does not have to cover every road segment on the map to achieve acceptable accuracy.

### 4.7.8 How Long Is Training Data Valid?

A related concern with both WiFi and cellular localization techniques that rely on building a training database is that as the set of access points or cell towers in an area changes, the training database slowly becomes invalid, requiring the training effort to be re-undertaken periodically (e.g., by driving through the area again).

We have not explicitly quantified the time for which training (wardriving) data remains valid. Our qualitative experience has been that it decays slowly and remains usable over months to years, at least for cellular data. For example, we have evaluated *CTrack* on driving data as recent as February 2011 using training data collected no later than July 2010, and some data collected as early as late 2009. We have found no marked degradation in accuracy.

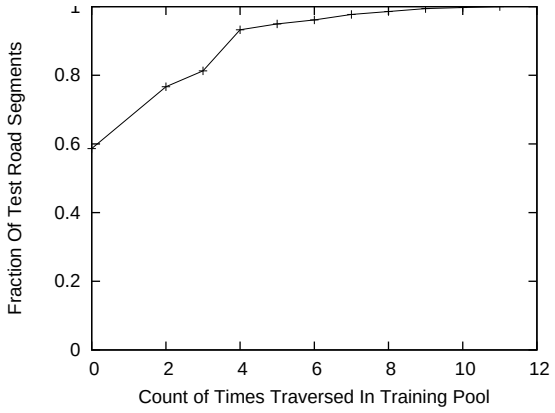



**Figure 4-17: CDF of traversal counts for each road segment, with 40 hrs of training data.**

### 4.7.9 A Confidence Metric?

For accuracy sensitive applications, a reasonable trade-off would be to use a *filtering* step to throw out tracks, or portions of tracks, where the *CTrack* algorithm has low confidence in its output. This has the effect of sacrificing some of the "recall" for substantially better "precision". Recall the "bad zone detector" used in *VTrack*: can we develop something similar for *CTrack*?

We have spent some time investigating whether a confidence metric could be used to filter out drives on which *CTrack* does poorly. Overall, our results have been quite weak, unlike in the *VTrack* case. Our failure to find a confidence metric has led us to conjecture (though we do not have conclusive proof) that the problem of finding a good confidence metric for radio localization is as hard as localization itself. The rough intuition is that if it were possible for the algorithm to know exactly on which inputs it does poorly, it would not do as poorly on those inputs.

However, we have found two predictors that are *weakly* correlated with map-matching accuracy:

- The 90th percentile distance of smoothed grids from the segments they are matched to, *before* the second pass HMM used in *CTrack*.
- The mean difference (over all points $P$) in emission score between the segment that $P$ is matched to in the output, and the segment closest to $P$.

The intuition in both cases is that a point far away from the road segment it is matched to, or closer to a different road segment, implies lower confidence in the match. When applying these confidence filters to our output drives, we currently improve the median precision from 75% to 86%, but lose substantially in terms of recall, whose median reduces from 80% to 35%. One interesting question for future work is whether machine learning techniques like *boosting* [81] could be used to combine these weak confidence predictors into a stronger predictor of confidence.

### 4.7.10 Comparison To WiFi Localization

As we have seen earlier, continuously sampling WiFi is much more energy-intensive than GSM (by over 6x on an Android G1 phone). Sub-sampling WiFi by duty-cycling the WiFi radio is one way to reduce this energy requirement, which makes sub-sampled WiFi an interesting alternative to using cellular localization with *CTrack*.

We ran a simple experiment to compare trajectory matching using *CTrack* on GSM fingerprints to trajectory matching with sub-sampled WiFi. Unfortunately, at the time of performing the experiments for *CTrack*, we did not have access to a training WiFi database. Hence, we only have access to "hard decision" WiFi information for this experiment. We used the Android network location API which scans for WiFi periodically and uses Google's wardriving database to look up the location corresponding to a WiFi scan, as a proxy for WiFi localization.

In our experiment, we duty-cycled the WiFi radio manually every 90 seconds, and obtained a new "network location" fix each time the radio was switched on. We chose a value of 90 seconds based on a battery drain experiment similar to that in Section 2.2.4, which found that sub-sampling WiFi every 90 seconds is equivalent to CTrack in terms of battery life (the WiFi radio takes about 10 seconds on average to turn on and off, so this corresponds to a duty cycle of about 10% for WiFi). We also continuously sampled GPS for ground truth, and GSM for *CTrack*. We ran *CTrack* on the GSM data and applied *VTrack* [7] to map-match the WiFi data. While we do not have a large corpus of evaluation data for this experiment, Table 4.2 below reports the precision/recall results for three test drives in different areas of Boston, amounting to over an hour of driving data. The results are too small a sample set to be conclusive, but suggest that the approaches could be comparable in terms of accuracy.

We note that *CTrack* can also be applied to soft information from WiFi localization. We believe it is likely to improve accuracy over the hard decision *VTrack* approach, though we have not implemented or evaluated this.

| | **CTrack** | | **Sub-Sampled WiFi** | |
|---|---|---|---|---|
| | **Prec.** | **Recall** | **Prec.** | **Recall** |
| Drive 1 | 64.7% | 68.6% | 30.6% | 18.3% |
| Drive 2 | 57% | 60.1% | 42.8% | 48.5% |
| Drive 3 | 50.1% | 65.8% | 61.6% | 64.2% |

**Table 4.2: *CTrack* on cellular vs *VTrack* on 10% duty-cycled WiFi.**

## 4.8 Related Work

Placelab performed a comprehensive study of GSM localization and used a fingerprinting scheme for cellular localization [57]. RADAR used a similar fingerprinting heuristic for indoor WiFi localizations [69], and the map-matching emission score used in *CTrack* is inspired by these methods. However, neither Placelab nor RADAR address the problem of trajectory matching, and are concerned more with the accuracy of individual localization estimates, rather than finding the optimal sequencing of estimates. As shown by the results in this dissertation, the use of soft information in the sequencing step is critical: applying a map-matching algorithm directly to Placelab-style location estimates results in significantly worse accuracy (by a factor of over $2\times$) compared to *CTrack*.

Letchner et al. [44] and our previous work on *VTrack*, presented earlier in this dissertation, also both use HMMs for map-matching. These algorithms do not use sensor hints. Moreover, as has been explained, these previous algorithms use and process $(lat, lon)$ coordinates as input and use a Gaussian noise model for emissions, and are hence inaccurate for map-matching cellular fingerprints.

CompAcc [38] proposes to use smartphone compasses and accelerometers to find the best match for a walking trail by computing directional "path signatures" for these trails. They do not use cell towers. However, from our understanding, the paper uses absolute values of compass readings. This

approach did not work in our experiments, because the absolute orientation of a phone can be quite different depending on whether it is in a driver's pocket, on a flat surface, or held in a person's hand. For this reason, we chose to use boolean turn hints instead, which are more robust and can be accurately computed regardless of changes in the phone's initial orientation or position.

For extracting motion hints and detecting walking and driving using the accelerometer, we use algorithms similar to those in [75, 5, 51].

Some previous papers [37, 45, 60] have proposed energy-efficient localization schemes that reduce reliance on continuously sampling GPS by using a more energy-efficient sensor, such as the accelerometer, to trigger sampling GPS. RAPS [45] also uses cell towers to "blacklist" areas where GPS accuracy is low and hence GPS should be switched off, to save energy. However, none of these papers address trajectory matching or propose a GPS-free, accurate solution for map-matching.

Skyhook [78] and Navizon [65] are two commercial providers for WiFi and Cellular localization, providing databases and APIs that allow programmers to submit WiFi access point(s) or cell tower(s) and look up the nearest location. However, to the best of our knowledge, they do not use any form of sequencing or map-matching, and focus on providing the best static localization estimate.

## 4.9 Conclusion

This chapter described *CTrack*, an algorithm that uses soft information to improve the accuracy of map-matching radio localization data.

Applying *CTrack* to cellular signal data yields an energy-efficient, GPS-free system for trajectory mapping using cellular tower fingerprints alone. On the Android platform, our *CTrack* implementation uses close to zero extra energy while achieving good mapping accuracy, making it a good way to distribute collaborative trajectory-based applications like traffic monitoring to a huge number of users without any associated energy consumption or battery drain concerns. A GPS-free approach to trajectory matching also opens up the possibility of providing more fine-grained location services on the world's most popular, cheapest phones that do not have GPS, but that do have GSM connectivity.

*CTrack* shows the promise of using sensor hints from inertial sensors to improve localization in the absence of GPS, and shows how such hints can be integrated into a probabilistic model for trajectory mapping. While the impact of sensor hints in *CTrack* is relatively small, sensor hints prove to be much more powerful in the context of *indoor trajectory mapping*, which will be the subject of the next chapter of this dissertation. The *iTrack* indoor trajectory mapping system we present in Chapter 5 uses many of the sensor hint ideas from *CTrack*. *iTrack* uses a probabilistic model that *fuses* inertial sensor information with WiFi localization measurements indoors to reduce training effort and improve accuracy.

# Chapter 5

# Indoor Trajectory Mapping

Chapters 3 and 4 focused primarily on energy efficient trajectory mapping for mobile devices in an *outdoor* setting. There, the main technical challenge was the high energy consumption of GPS, which necessitated using lower-energy WiFi and cellular localization.

Indoors, the challenge is that GPS simply does not work. This is because the GPS signal-to-noise ratio is extremely low inside most buildings.

People spend most of their time indoors, creating a tremendous opportunity for fine-grained indoor positioning and tracking to enable an exciting range of new services and applications. Examples include locating missing objects at home, tracking or locating key personnel such as doctors in a hospital or employees in an office, navigation and search in large indoor spaces such as malls and museums, assisting visually impaired or disabled people navigating indoor spaces, and location analytics — understanding where people spend time and how they use an indoor space.

There exist techniques for accurate indoor localization to within a few centimetres that use *specialized hardware*, such as ultrasound and radio-ranging systems (Cricket [62] and Active Bat [3]), laser ranging (LIDAR) systems [56], and foot-mounted inertial sensors [29, 67]. However, these are often expensive to deploy and do not work on commodity mobile phones.

For this reason, we turn to WiFi localization [69, 58, 46, 70] and cellular localization [8], which we investigated in the outdoor context previously. These can both be used indoors to localize within a few metres, and hold promise because they work on commodity mobile phones. However, they face a significant deployment challenge, as we describe below.

## 5.1 The Training Challenge

The biggest challenge with indoor WiFi localization techniques today is the requirement of *extensive manual training* to associate radio fingerprints to ground truth locations in the building. Because there is no ready ground truth reference indoors (unlike GPS outdoors), the training procedure for WiFi localization is cumbersome. It requires dedicated personnel or volunteers to manually visit hundreds or even thousands of locations in a building, mark the location they are at accurately on a map, and collect training data at each location. This process is painful, expensive and time-consuming. [69, 58, 70, 46]. In our own experience, even just dragging the cursor on a smartphone touchscreen to the correct location, or panning a zoom window of a map to locate where to mark points is cumbersome to do for a few dozen points, let alone hundreds or thousands.

Manual annotation is additionally subject to human error which in turn can cause errors in localization. The OIL project at MIT [46], which relies on crowd-sourcing indoor location from volunteers, has documented many examples of such errors.

## 5.2 *iTrack* And Contributions

*iTrack* is a system we have built for fine-grained indoor trajectory mapping on mobile smartphones that *fuses* information from inertial sensors (accelerometer and gyroscope) with WiFi fingerprints.

*iTrack* makes two key contributions. First, *iTrack* can recover the trajectory of a mobile phone when held by a user walking steadily with phone in his/her hand or pants pocket highly accurately, to within *less than a metre*. To do this, it uses inertial sensors to detect periods of steady walking, and to extract the approximate shape and size of each user walk from the gyroscope and accelerometer respectively (we define "shape" and "size" precisely later). Each walk is matched to the contours of a building floorplan using a *particle filter*. A particle filter uses Monte Carlo simulation to explore a large number of paths with shape and size close to the measured shape and size extracted from the sensor data. The filter narrows down a candidate path by eliminating particles (paths) that cross a wall or obstacle in the floorplan. The filter also progressively fuses in WiFi signal strength information as it becomes available to improve the quality of tracks it extracts.

Second, *iTrack* provides a novel, simplified approach to training a WiFi localization system in an indoor space by simply *walking around the space to map it*, significantly reducing human effort compared to the manual annotation approach. Training proceeds in two stages:

- The first stage initializes the algorithm with "seed WiFi data" from a small number of walks where a volunteer or dedicated trainer specifies his/her starting position approximately on a map. Unlike full-fledged manual training, the trainer only specifies one point per walk on the map, and only needs to specify it approximately. About 5-10 walks (amounting to only 10 minutes of training) suffice to seed an entire floor of the MIT CSAIL building, which is approximately 2,500 square metres in size.
- Once seed data has been collected, *iTrack* crowd-sources more data from users of the indoor space. Users can contribute more training data by simply downloading an application on to their smartphones and running it in the background. This process is completely automated and can rapidly collect a large amount of data as users walk around. It requires *no extra effort or annotation from the user* beyond running the application on their phone.

Our training procedure is a huge improvement over previous techniques requiring point by point manual input from a human to indicate where he/she is, or requiring multiple devices to be carefully measured and placed on a dense grid covering the floorplan to collect training data [69, 58].

### 5.2.1 Implementation And Evaluation

We have implemented *iTrack* on the iPhone 4, which includes a 3-axis gyroscope. We have also tested the algorithms used by *iTrack* on gyroscope data from an Android Nexus S phone, but have not yet built a full-fledged Android application.

We evaluate *iTrack* on 50 test walks collected using an iPhone 4 on the 9th floor of the MIT Stata Center. We use tape markings made on the ground to accurately estimate ground truth for each

walk. We find that given only 4 walks of seed training data (which took $\approx$ 5 minutes to collect), *iTrack* can extract accurate trajectories from 80% (40 out of 50) of the walks. The walks extracted have a median error of approximately 3.1 feet, or less than a metre.

We also found that WiFi localization data learned from 97 walks crowd-sourced with *iTrack* could localize a user to within 5.4 metres on average on test data, comparable to the expected accuracy from manual training approaches.

### 5.2.2 Rest of this chapter

The rest of this chapter is organized as follows. Section 5.3 provides background on inertial phone sensors and the raw data they provide. In particular, we illustrate raw data collected from the accelerometer and gyroscope on the iPhone 4 and explain what the values mean. Section 5.4 provides an overview of traditional inertial navigation. This explains how to integrate accelerometer and gyroscope data to obtain a precise estimate of a device's current position and orientation. Section 5.4.2 explains why the conventional approach does not apply directly to mobile phones. We also explain why modifications to the conventional approach used to deal with pedestrian foot-mounted sensors do not work for mobile phones.

Section 5.5 describes the *iTrack* algorithm, and each stage involved extracting an accurate indoor trajectory from inertial sensing data. Section 5.6 describes how the tracks extracted from *iTrack* can be used for rapidly learning a WiFi training database for indoor localization, while being iteratively used to improve *iTrack*'s own estimates. Section 5.7 describes our iPhone implementation. Section 5.8 evaluates *iTrack*. Section 5.9 describes related work, and Section 5.10 concludes this chapter.

This chapter focuses mainly on strategies to improve the *accuracy* of indoor trajectory mapping. We do not investigate energy consumption, which would be an interesting direction for future exploration.

## 5.3 Inertial Phone Sensors

### 5.3.1 Accelerometer

Smartphones in the market as of 2011 are almost all equipped with a three-axis accelerometer, which can measure the effective acceleration along three axes in space. These accelerometers use MEMS (micro-electro-mechanical sensing) technology. An accelerometer cannot measure free-fall acceleration, but rather measures the *weight* experienced by a test mass lying in the frame of reference of the accelerometer device. For example, a phone held on a user's hand will measure an acceleration of $g$, the acceleration due to gravity because this is proportional to the weight experienced by the accelerometer inside.

An MEMS accelerometer in a phone today typically consists of a micro-scale cantilever beam with a *proof mass* suspended from it. Under the influence of external acceleration, the proof mass deviates from its normal position. The deviation can be measured to obtain an estimate of the net external acceleration applied to the accelerometer casing, along one particular axis. Modern phone accelerometers combine two such devices in one plane with one device out of plane to measure acceleration along three axes. The iPhone, for example, uses an LIS302DL three-axis MEMS accelerometer.

Figure 5-1 shows the data provided by the iPhone accelerometer, as documented in Apple's Core Motion API [23]. The device measures the acceleration $a_x$, $a_y$ and $a_z$ along three axes defined

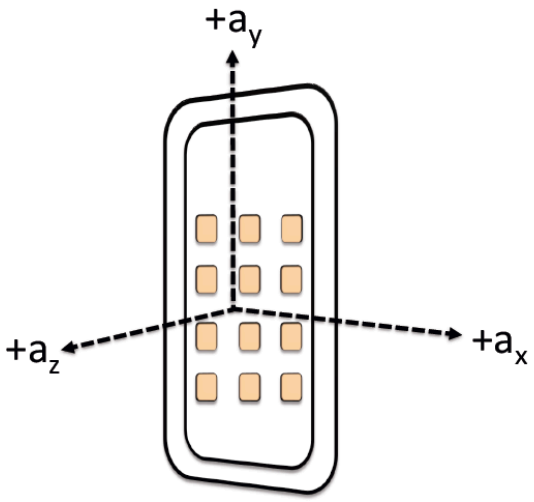



**Figure 5-1: What a smartphone accelerometer measures.**

with respect to the phone's frame of reference. In the figure, $a_z$ is the acceleration along an axis perpendicular to the plane of the phone, and $a_x$ and $a_y$ are accelerations along axes within the plane of the phone. The net acceleration due to gravity is *included* in the acceleration vector. Its components along the x, y and z axes depend on the absolute orientation of the phone, which cannot be determined using the accelerometer alone.

A typical smartphone accelerometer can sample accelerometer data at up to 50-100 Hz. Our iPhone implementation samples accelerometer data at 100 Hz.

### 5.3.2 Gyroscope

A gyroscope measures the angular velocity of rotation of an object about three axes in space. Modern MEMS gyroscopes also consist of a proof mass similar to MEMS accelerometers. Instead of being suspended from a cantilever as in the case of an accelerometer, the mass is made to vibrate continuously by applying a drive signal to a set of drive capacitor plates. When a user rotates the phone, the vibrating mass is displaced along all three axes by *Coriolis* forces. The MEMS gyroscope measures the deviation of the proof mass along all three axes to obtain an accurate estimate of the angular velocity of rotation.

Figure 5-2 shows the data provided by the iPhone gyroscope, as documented in Apple's Core Motion API. The device measures the angular velocity $\omega_x$, $\omega_y$ and $\omega_z$ about three axes defined with respect to the phone's frame of reference. In the figure, $\omega_z$ is the angular velocity of rotation about an axis perpendicular to the plane of the phone, i.e., the angular velocity of rotation of the phone in the X-Y plane. Similarly, $\omega_x$ and $\omega_y$ represent the angular velocity of the phone's rotation in the Y-Z and Z-X planes respectively.

A typical phone gyroscope can sample gyroscope data at 100 Hz or more. Our implementation of *iTrack* uses 100 Hz gyroscope data.

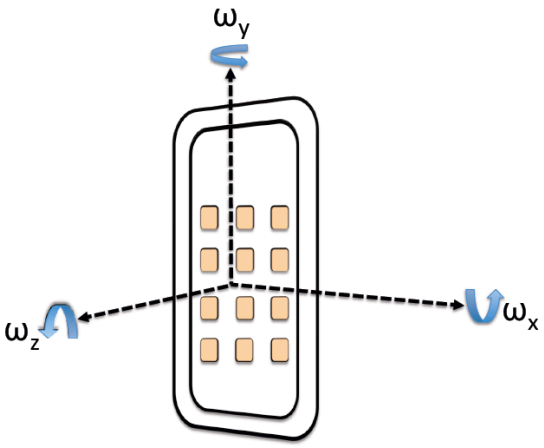



**Figure 5-2: What a smartphone gyroscope measures.**

## 5.4 Inertial Navigation

Conventional inertial navigation systems *combine* data from an accelerometer and a gyroscope to estimate the absolute position of a device, assuming its initial position, velocity and orientation are all known. These systems are in widespread use in aerospace systems, for tracking satellites (e.g., they are used to keep track of GPS satellites as they orbit the earth), industrial tracking applications and in robotics.

This section describes how inertial navigation works at a high level. We note from our discussion in the previous section that a gyroscope does not directly measure the absolute orientation of a phone. Rather, it measures angular velocity, which represents the rate of change of orientation. Given a known initial orientation, it is possible to integrate angular velocity measurements from a gyroscope to obtain absolute orientation at any time instant (described in Section 5.4.1, *Attitude Computation*).

By combining this orientation data with acceleration data from the accelerometer, it is possible to subtract out gravity, and indirectly compute the instantaneous *user acceleration* of the phone along three axes fixed in space. This in turn can be integrated twice to obtain an estimate of the user's position (described in Section 5.4.1, *Position Computation*).

The following section derives the equations used by inertial navigation systems to compute the position and orientation using integration.

### 5.4.1 Inertial Navigation Equations

The derivation presented in this section is based on material in Chapter 3 of *Strapdown Inertial Navigation Technology*, by David H. Titterton and John L. Weston [26].

We consider the problem of tracking the position and orientation of a device over time, given a sequence of instantaneous acceleration and angular velocity measurements of the form:

- $t, a_x(t), a_y(t), a_z(t)$ from the accelerometer.
- $t, \omega_x(t), \omega_y(t), \omega_z(t)$ from the gyroscope.

All the instantaneous measurements are with respect to the phone's frame of reference at any time instant $t$. Since the phone is continuously rotating, its frame of reference is also continuously changing. This means that the acceleration and angular velocity measurements need to be first converted to a fixed, or *inertial* reference frame before integrating them. The goal of integration is to find the absolute orientation, or *attitude* of the phone at any instant of time.

*Attitude Computation*

Assume a fixed (i.e., non-rotating) reference frame with three axes X, Y and Z in which the gravity vector points in the negative Z direction, and the X and Y axes are arbitrarily chosen. To represent the 3-D orientation of the phone at any instant of time, we use a representation called the *direction cosine* representation. The direction cosine matrix of the phone at time $t$, denoted by $C_{phone}(t)$, is a $3 \times 3$ matrix. The columns of this matrix represent unit vectors along the X, Y, and Z axes of the phone (as shown in Figure 5-1) as expressed in the fixed frame:

$$C_{phone}(t) = \begin{vmatrix} c_{11}(t) & c_{12}(t) & c_{13}(t) \\ c_{21}(t) & c_{22}(t) & c_{23}(t) \\ c_{31}(t) & c_{32}(t) & c_{33}(t) \end{vmatrix} \tag{5.1}$$

In the above matrix, the column vector $c_{11}(t)\ \hat{i} + c_{21}(t)\ \hat{j} + c_{31}(t)\ \hat{k}$ represents the unit vector along the phone's X axis (an axis parallel to the base of the phone, in the plane of the phone).

Given that the direction cosine matrix of the phone is $C_{phone}(t)$ at time t, any vector $\vec{v}_{phone}(t)$ at time $t$ defined with respect to the phone's reference frame, can be transformed to the ground reference frame by pre-multiplying it with the direction cosine matrix:

$$\vec{v}_{ground}(t) = C_{phone}(t)\ \vec{v}_{phone}(t) \tag{5.2}$$

Our goal is to relate the instantaneous angular velocity from the gyroscope, $\vec{\omega}_{phone}(t)$ to the change in the direction cosine matrix. To do this, we write out an expression that yields the rate of change of the direction cosine matrix with time:

$$\dot{C}_{phone}(t) = \lim_{\delta t \to 0} \frac{C_{phone}(t + \delta t) - C_{phone}(t)}{\delta t} \tag{5.3}$$

The term $C_{phone}(t + \delta t)$ can be written as the product of two direction cosine matrices as follows:

$$C_{phone}(t + \delta t) = C_{phone}(t)\ A(t) \tag{5.4}$$

where $A(t)$ is a direction cosine matrix representing the (small) rotation of the phone's reference frame from time $t$ to time $t + \delta t$.

For small times $\delta t$ and hence small angles of rotation, the rotation $A(t)$ can be approximated as follows (for a proof, see [26]):

$$A(t) = [I + \delta t\ \Omega_{phone}(t)] \tag{5.5}$$

where $I$ is the identity matrix, and $\Omega_{phone}(t)$ is the *skew symmetric matrix* form of the angular velocity vector $\vec{\omega}(t)$:

$$\Omega_{phone}(t) = \begin{vmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{vmatrix} \tag{5.6}$$

Substituting the expression for $A(t)$ above in the limit equation yields:

$$\dot{C}_{phone}(t) = C_{phone}(t)\, \Omega_{phone}(t) \tag{5.7}$$

where $\Omega_{phone}(t)$ is the skew-symmetric matrix representation of the angular velocity vector $\vec{\omega}_(t)$.

This differential equation can be solved numerically to yield the instantaneous value of the direction cosine matrix, $C_{phone}(t)$ at any time instant $t$, and hence the instantaneous orientation of the phone at any time instant.

*Position Computation*

Given the instantaneous attitude direction cosine matrix $C_{phone}(t)$, we can use the following equation, analogous to equation 5.2 presented earlier to transform the phone accelerometer measurements to the ground reference frame:

$$\vec{a}_{ground}(t) = C_{phone}(t)\, \vec{a}_{phone}(t) \tag{5.8}$$

The effective user acceleration can then be computed by subtracting the effect of the earth's gravity, $\vec{g}$, from the acceleration vector expressed in the ground reference frame:

$$\vec{a}_{user}(t) = \vec{a}_{ground}(t) - \vec{g} \tag{5.9}$$

Suppose the phone has a known initial position $\vec{x}(0)$ and a known initial velocity $\vec{v}(0)$. The numerical integration equations to find velocity and position at each time instant are as follows:

$$\vec{v}(t + \delta t) = \vec{v}(t) + \vec{a}_{user}(t)\, \delta t \tag{5.10}$$

$$\vec{x}(t + \delta t) = \vec{x}(t) + \vec{v}(t)\, \delta t \tag{5.11}$$

In principle, one could use a numerical integration technique such as Simpsons' rule, or Runge-Kutta integration, to solve these equations and obtain the position of the mobile phone $\vec{x}(t)$ at any time instant $t$. However, as we shall see in the next section, this does not work in practice.

### 5.4.2 Challenge: Inertial Drift

Given a phone with a three-axis accelerometer and gyroscope, and a known initial position and velocity — for example, from a known time instant when the phone was static, it should in theory

be possible to directly solve the inertial navigation equations stated above to obtain the precise position of the phone at any time instant.

However, in practice, as with any real-world sensor, the acceleration and angular velocity measurements from inertial MEMS sensors are not perfectly accurate. As we shall see shortly, the measurement error is exacerbated when sensor data is used as input to higher-order integration, which is the case in the inertial navigation equations described above. This results in a problem known as *drift* in inertial navigation parlance. Drift is a phenomenon where the estimated trajectory of a device deviates from the true trajectory with a deviation that increases with time, often rapidly or non-linearly.

Both kinds of inertial measurements can experience both random noise and systematic bias, as well as more complex kinds of bias such as dynamically evolving bias. In this work, we focus on two types of errors: systematic errors and random noise. We describe the impact of both kinds of errors below.

### *Systematic Bias*

The systematic bias of a sensor is a systematic error that is added to each measurement made by the sensor, and *always* has the same sign (positive or negative). It is easy to correct for because if the value of the bias is known, it can simply be subtracted from each sensor measurement.

It is easy to measure and correct for the systematic bias of both MEMS accelerometers and gyroscopes. A phone that is static — for example, placed stationary on a flat desk, should have a user acceleration vector that is zero. The long-term average of the measured user acceleration from the accelerometer during such a static period, after subtracting gravity, yields the bias of the accelerometer along all three axes.

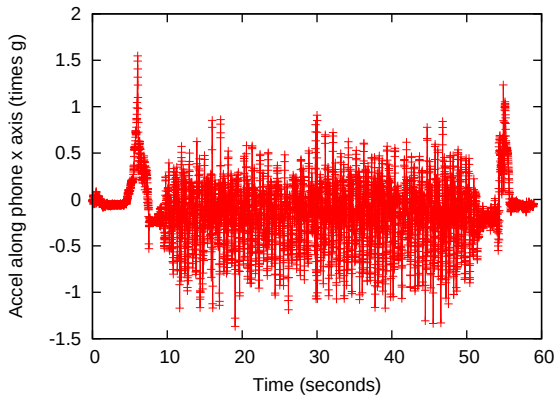
Similarly, a phone at rest should have zero angular velocity. The long-term average of the angular velocity measured by the gyroscope when the phone is static yields the gyroscope bias.

### *Random Noise*

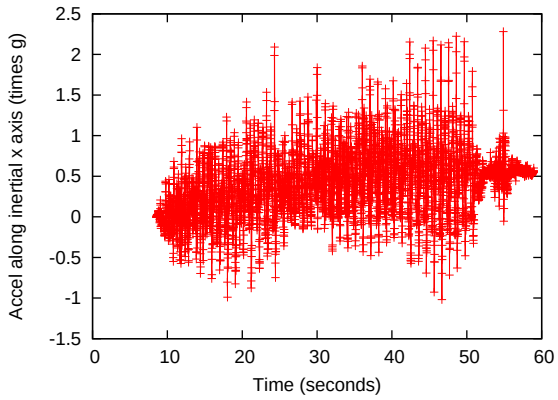
Random noise is usually thermal in nature and does *not* always have the same sign. It is therefore harder to correct than systematic bias. Assuming all systematic bias has been corrected, the remaining error comes from an error distribution that has zero mean but a non-zero variance.

Recall the discussion in Section 4.6.4 of Chapter 4. Assuming raw acceleration values have a random zero-mean Gaussian distributed error with a standard deviation of $N_{acc}$, we showed in that discussion that the error in position, obtained by twice integrating the acceleration, grows approximately as $O(t^{3/2})$ with time $t$. In the inertial navigation case, a key point to note is that the instantaneous acceleration vector is obtained indirectly by transforming *measured* acceleration using the current orientation of the phone. The current orientation *itself is subject to error from integrating the raw gyroscope data*.

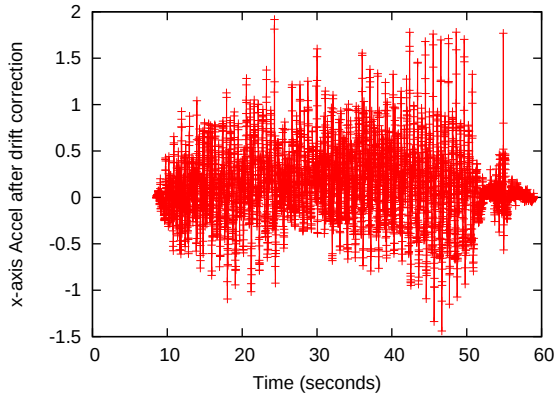
If the raw angular velocity $\vec{\omega}$ measured by the gyroscope is assumed to have zero-mean Gaussian noise with standard deviation $N_{gyro}$, the change in the direction cosine matrix at any time instant will also have Gaussian distributed error. Analogous to the walking drunkard discussed in Section 4.6.4, the orientation obtained by integrating (summing) gyroscope data will have mean squared error proportional to $t$, the time period of integration. This in turn means the transformed acceleration values from equation 5.8 have zero-mean error distributed with a variance proportional to $\sqrt{t}$.



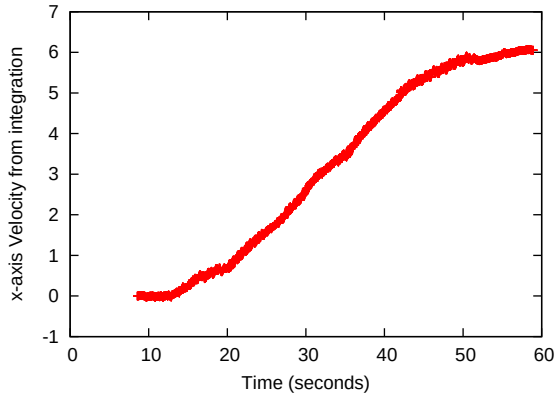

(a) Raw x-axis acceleration in smartphone reference frame.




(b) x-axis acceleration in inertial frame, with bias.




(c) x-axis acceleration after bias removal.




(d) x-axis velocity from integrating unbiased acceleration.

**Figure 5-3: Drift when integrating smartphone accelerometer data.**

If integrating the acceleration twice to obtain position, the expected error in position can be shown to grow as $O(t^{2.5})$, *even after correcting for systematic bias*. If we cannot correct for systematic bias, the growth rate of error is even higher, of the order of $O(t^3)$.

### 5.4.3 Correcting For Drift

Sensor drift is an extremely serious problem for most inertial navigation systems. In each application domain, inertial systems have evolved specialized tricks and techniques to periodically correct for drift and halt the growth of error in the estimated position. For example, GPS satellites periodically use knowledge about the direction of the earth's gravity vector to correct accumulated error in their direction cosine matrix, and thereby arrest some of the drift.

In the context of indoor localization, $O(t^{2.5})$ or $O(t^3)$ is too high a growth rate for position estimates from direct integration to be usable. Simply integrating raw accelerometer data results in estimates with so high an error that they are unusable for tracking a mobile device indoors.

*Illustration*

Figure 5-3 illustrates drift on real accelerometer data collected from an iPhone in a user's pocket when walking indoors. The first pane, Figure 5-3(a) shows the measured raw value of acceleration along the X-axis. This value is always with respect to the phone's reference frame. Since the phone is continuously changing orientation in the user's pants pocket, this acceleration needs to be transformed to a global inertial frame using orientation from the gyroscope, as we have discussed earlier. In this particular trace, the user started in a stationary position and immediately started walking. The walking ends towards the very end of the trace.

The transformed acceleration obtained from equation 5.8 is shown in the second pane, Figure 5-3(b). As we can see, the transformed acceleration is already drifting upwards steadily and suffers significant bias. A big part of this drift is due to drift in the absolute device orientation computed by integrating the gyroscope using the equations described earlier. Even a small error in the computed direction cosine matrix, $C_{phone}(t)$, is exacerbated because the error in the matrix is multiplied by $\vec{g}$, the acceleration due to gravity, when transforming the acceleration to the ground frame using equation 5.8.

It is possible to correct this accumulated drift in acceleration somewhat by explicitly considering each walking period separately and factoring out a linearly growing drift error term, $d = \alpha\ t$ for some constant $\alpha$. Figure 5-3(c) shows the result of such a linear drift correction term applied to the transformed acceleration, for the best possible value of the constant $\alpha$.

*However, even integrating this acceleration results in velocity drift owing to random noise.* Figure 5-3(d) shows the estimated velocity of the mobile device along the X-axis obtained by numerically integrating the "unbiased" acceleration from Figure 5-3(c). As we see, the velocity quickly drifts upwards. The velocity drift is proportional to $O(\sqrt{t})$ because it comes from zero-mean random noise in the acceleration, as we discussed earlier. The velocities output by this procedure are themselves not accurate enough to be useful. A further level of integration to find displacement would produce even larger errors that grow faster, and would clearly be unusable for trajectory mapping.

Researchers have previously looked into the drift problem in the context of pedestrian indoor tracking using foot-mounted inertial sensors [67, 29]. These approaches deal with the problem of drift by using a technique called *zero-velocity updates* (ZUPTs). The idea is that the inertial sensor mounted on a person's foot experiences zero velocity and zero angular velocity whenever the person's foot is on the ground during his/her stride.

Graphs of angular velocities measured by such a foot-mounted gyroscope show a flat "plateau" region with zero or close to zero angular velocity, which can be easily detected and classified as a step event. Accordingly, the velocity estimated from inertial navigation can be corrected to zero at each such point, once for each stride. This correction can be back-propagated to correct the orientation matrix using a Kalman filter, as discussed in [67]. The authors of that work show that using this technique can reduce the growth in position drift to $O(t)$, and can accurately estimate the length of each individual stride.

Unfortunately, these existing approaches do not work for mobile phone tracking, because a phone is not mounted on a user's foot. A phone is typically held in a user's hand or pants pocket, or handbag, where it has some wiggle room. Hence, it never experiences zero angular velocity or zero velocity, or even velocity close to zero. In fact, integration drift error is compounded when inertial data comes from a mobile phone that can be swinging up and down in a users' pants pocket or handbag, or from a phone oscillating gently up and down or sideways in a user's hand when walking.

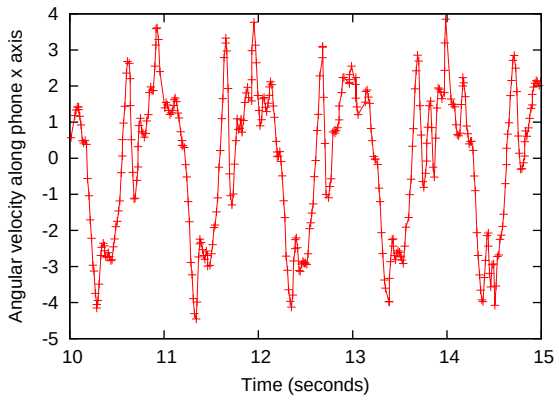



**Figure 5-4: Angular velocity of an iPhone in a user's pocket when walking.**

Figure 5-4 shows raw angular velocity data from the iPhone gyroscope from the same trace used for illustration in Figure 5-3. As we see, the angular velocity does not experience any flat region close to zero. Rather, it varies continuously during all parts of the user's stride.

### *Approach Used in iTrack*

A key novel contribution of *iTrack* is how it deals with and corrects inertial sensor drift on a mobile phone . *iTrack* does not integrate acceleration data outright. *iTrack* instead uses acceleration data to specifically detect *steady walking*, and distinguish it from other movements such as one time movement, talking, picking up the phone and other phone use. Wherever it identifies a steady walking period, *iTrack* uses the accelerometer data to perform *step-counting* in the user's walking periods, which, as we show in the next section, can be done more accurately than integrating noisy acceleration samples.

In contrast to the approach used for acceleration data, *iTrack* does integrate data from the gyroscope to estimate the approximate shape of a user's walk. This does not suffer drift error because *iTrack* uses only *changes* in the gyroscope values rather than the absolute orientation information to determine a phone's trajectory.

The shape and size of the walking trajectory (estimated from step count and gyroscope data) are used as a guide to find the best match to a known building floorplan.

## 5.5 The *iTrack* System

Figure 5-5 shows the architecture of *iTrack*. The system consists of a phone application and a server-side trajectory matching service. The phone application continuously samples WiFi fingerprints, gyroscope and accelerometer data at the maximum possible frequency (100 Hz for gyro and accelerometer on the iPhone). The accelerometer signal is used in conjunction with the gyroscope to detect periods of steady walking, using a *walking detector*. All the sensor data in walking periods is transmitted to the *iTrack* server, which matches it to a known building floorplan to find user trajectories. Currently, trajectory mapping is implemented as an offline process on the server. It would be interesting to explore performing trajectory mapping on the phone itself as part of future work.

In addition to sampling and sending sensor data back to the server for trajectory mapping, *iTrack* also uses observed WiFi fingerprints to continuously localize the phone. The *localization engine*

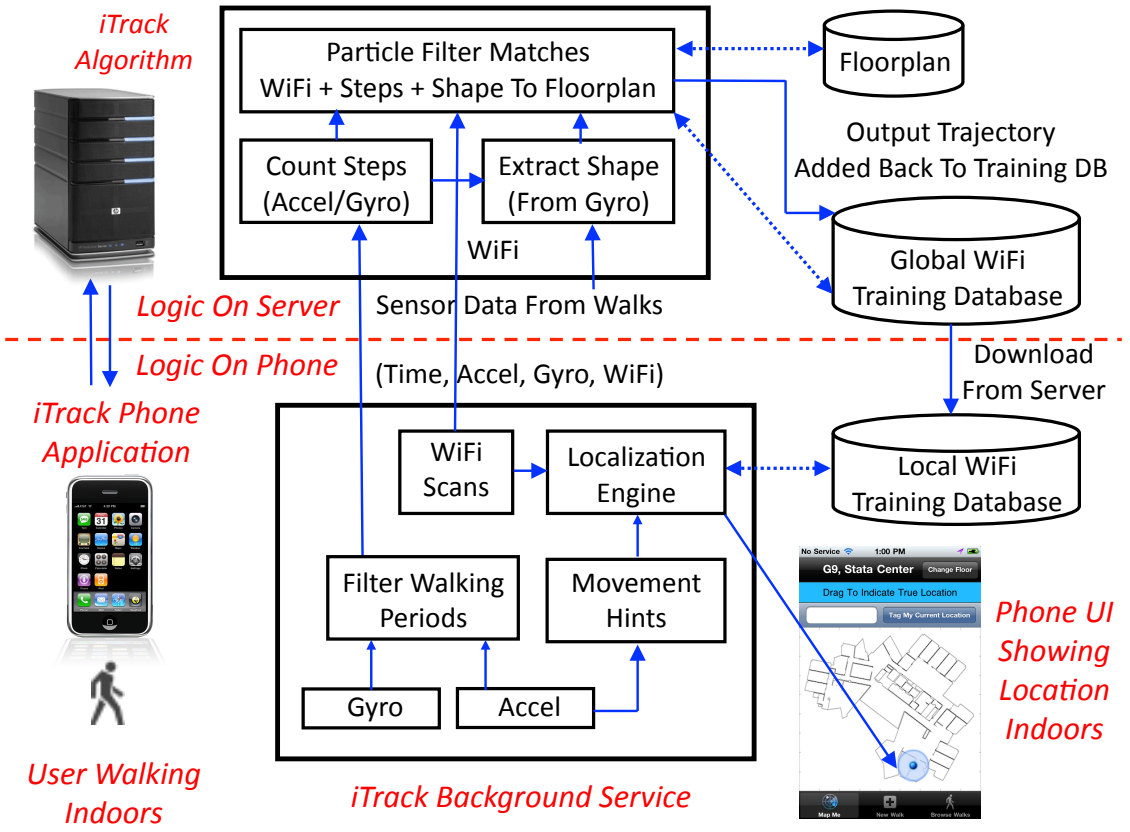



**Figure 5-5: *iTrack* system architecture.**

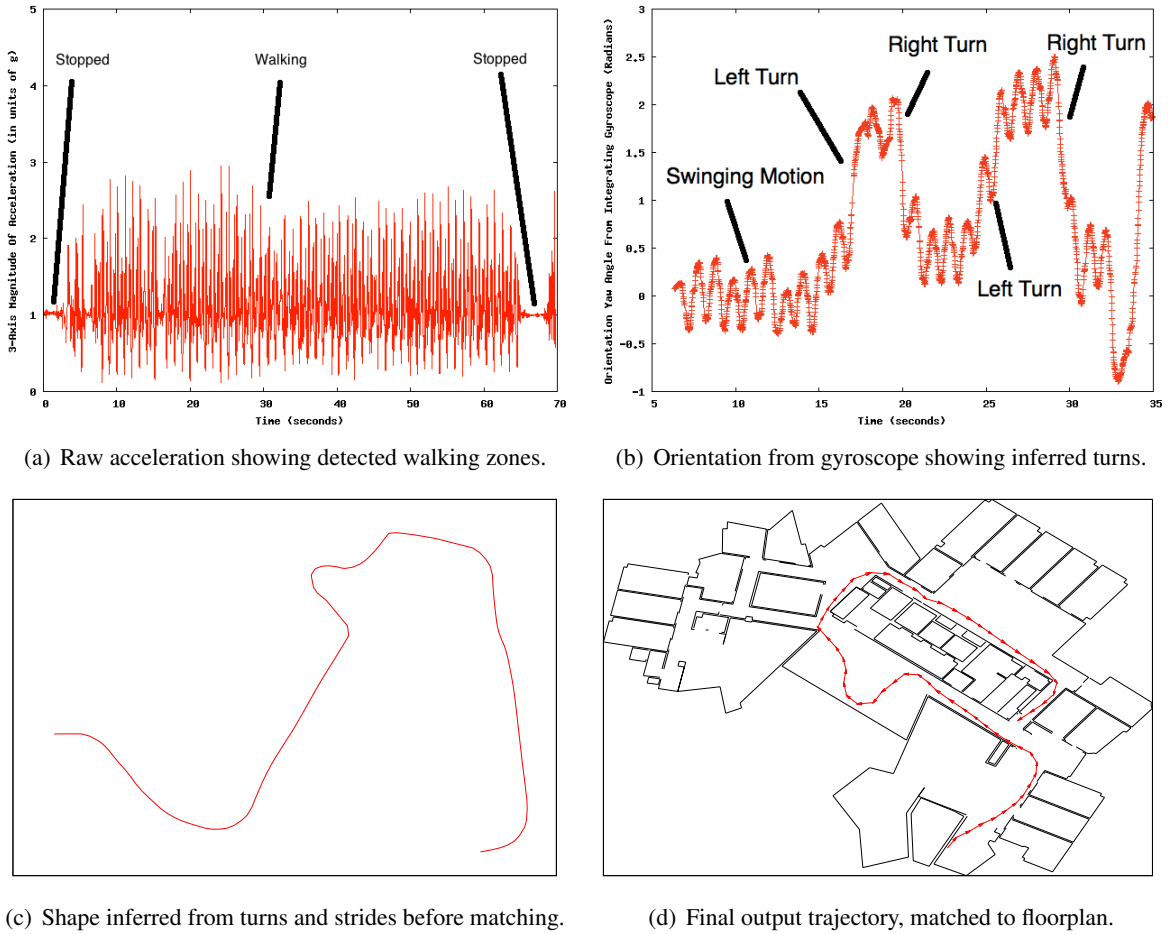
shown in the figure looks up fingerprints in a local WiFi training database to find the approximate location of the phone, and displays this location on an indoor map to the end user. The local training database is an on-phone cache of a global training database that is built up over time by *iTrack*. The localization engine also uses movement hints from the accelerometer to do better localization. It uses a simplified Viterbi algorithm similar to *CTrack*. We do not discuss the localization engine in this dissertation.

The task of the *iTrack server* is to extract accurate trajectories from WiFi fingerprints, accelerometer and gyroscope data in time periods when the user is known to be walking steadily with the phone either in his/her hand or pocket. The algorithm on the server consists of a *walking detector*, a *step counter*, a *shape extractor* and a *particle filter*. The step counter counts the number of steps in a walk using acceleration and/or gyroscope data. The shape extractor uses gyroscope data to estimate the approximate shape of the walk, i.e., where the significant turns in a user's trajectory lie.

The particle filter is an important component of the algorithm. Given a floorplan of the building, it uses Monte Carlo simulation to simulate paths with similar shape and size to the estimated shape and size, and find the most likely such path through the floorplan. The "size" of any stretch is obtained approximately from the step count using known bounds on human stride length. The Monte Carlo simulation helps eliminate candidate paths (or particles) that intersect walls or violate other constraints imposed by the floorplan. The particle filter also optionally fuses in previously collected WiFi training data to improve the accuracy of its output trajectories. Our filter is inspired by particle filters used in robotics, and to track pedestrians with foot-mounted sensors [67, 29].

However, as we shall see, there are some key differences from previous approaches to make the filter work with data from mobile phones.

The output trajectories produced by the particle filter are accurate to within less than a metre. In most cases, these trajectories can be joined with WiFi scanning data collected during the walk to generate new, accurate training data for WiFi localization. Training data is contributed back automatically to the global training database, which grows over time.




(a) Raw acceleration showing detected walking zones.

(b) Orientation from gyroscope showing inferred turns.

(c) Shape inferred from turns and strides before matching.

(d) Final output trajectory, matched to floorplan.

**Figure 5-6: Illustration of steps in *iTrack*.**

Figure 5-6 visually illustrates the steps to run *iTrack* on an example walk on the 9th floor of the MIT Computer Science department building. Figure 5-6(a) illustrates the first step, which identifies walking zones and extracts step counts from each walking zone in the accelerometer data. Figure 5-6(b) illustrates the process of shape extraction from gyroscope data. The figure shows the orientation computed by integrating gyroscope data, rather than raw angular velocities (which were shown earlier in Figure 5-4). As the plot shows, it is possible to infer the magnitude and direction of turns in a user's walk from this. Figure 5-6(c) shows an outline of the shape obtained by combining the turn information from the gyroscope with the step count information from the accelerometer. This shape is approximate and needs to be matched to the floorplan to find the most likely path taken by the user. The last step of *iTrack* is the particle filter, which takes the approximate shape and optionally, WiFi signal strength signatures as input, and matches the shape to a known building floorplan. The algorithm requires information about the locations of walls/obstacles (constraints)

and the absolute size of the floorplan to do this matching.

Figure 5-6(d) shows the output produced by the particle filter on our example walk. As we shall show, the trajectories produced by *iTrack* are accurate to within less than a metre.

The following sections describe the design of the individual components of *iTrack* in more detail: *walking detection*, *step counting*, *shape extraction* and *particle filtering*.

### 5.5.1 Walking Detection

The goal of walking detection is to identify zones where the user is walking steadily, so that we can extract trajectories for each of the walks made by the user. Walking detection takes raw acceleration from the accelerometer and angular velocity data from the gyroscope as input. It outputs a sequence of triplets $\langle t_{begin}, t_{end}, mode \rangle$, where $t_{begin}$ and $t_{end}$ represent the beginning and end of a time interval when the user is detected to be walking steadily, and $mode$ is an annotation for the interval that indicates if the phone was exclusively in the user's hand or pocket, or neither during the interval. Our current implementation of *iTrack* is able to extract accurate trajectory data from steady paced walks where the phone is in exactly one of the two poses: hand or pocket.

We model a phone as always being in one of three states: *stopped*, *moving* (but not walking steadily) and *steady walking*. Data from both stopped and steady walking zones is useful. In the stopped case, it can be used to learn how long a user spent at a particular location, or to learn what WiFi signatures occur at a particular location. In the steady walking case, it can be used to map the users' walk and/or collect WiFi training data for points in the walk.

In contrast, periods where the phone is moving, but the user is not walking steadily, usually represent the user using the phone in some way, such as to text, play a game, or make a call. These periods can also sometimes represent the user moving or walking slowly or irregularly, with frequent stops, so as to be undetectable as "steady walking" by the walking detector. We do not currently use the sensor data from these periods for mapping indoor trajectories because it is hard to distinguish slow or interrupted walks from random movements of the phone. Gyroscope data in such a period could reflect the user picking up or texting on the phone rather than changes in the orientation of the phone due to a user walking.

*iTrack* aims to accurately filter out and discard sensor data from two kinds of periods:

- Periods where a user is moving the phone in some way but not walking steadily.
- Periods where the phone changes position relative to its user, even in the midst of a steady walk (e.g., picking up a phone call).

We tackle this problem by dividing it into a number of simpler sub-problems that we describe in turn below.

#### *Static vs Moving*

The first sub-problem is identifying whether a phone is moving or at rest. *iTrack* uses the movement hint extractor developed for *CTrack* (described in Chapter 4, Section 4.6.4) to distinguish a phone at rest from a phone that is not.

*Anomaly Detection*

As discussed here and in the previous chapter on *CTrack*, it is essential to filter out "anomalies" where the phone is not at rest relative to its user to ensure accelerometer and gyroscope data accurately reflects turns and movements along a user's walk. We use the same procedure described there (spike detection) to carry out anomaly detection. It should also be possible to use application state e.g., whether a call is active or what applications are running, to improve anomaly detection accuracy by filtering out some motions that don't represent walking such as gaming, typing, or talking on the phone. We have not explored this in our current implementation, but believe it would be a useful extension to *iTrack*.

*Hand vs Pocket*

Assuming the pose of a phone is steady and anomalies or sudden transitions between poses have been filtered out, we present a procedure to determine the *pose* of the phone in any time window. We distinguish two specific poses: the phone being in the user's hand, and the phone being in the user's pocket. This problem is important because all the subsequent stages of *iTrack* (walking detection, step counting, and shape extraction) use different algorithms for the hand and pocket cases.

*iTrack* uses orientation computed by the gyroscope to distinguish a phone held flat in a user's hand while walking from a phone held in a user's pocket while walking. To do this, we use two key observations:

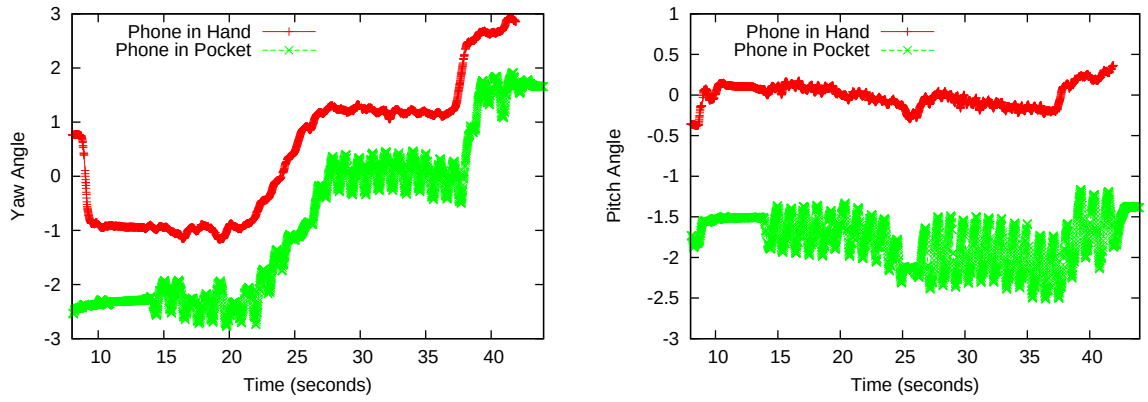
- The Euler *pitch* angle has magnitude close to zero when a phone is held flat in a user's hand, but has non-zero magnitude (usually close to +1 or -1) when a phone is in a vertical orientation in a user's pocket.
- The Euler *yaw* angle computed by integrating gyroscope data shows a marked, periodic up and down swing with high standard deviation in a user's pocket when the user is walking, and a *much* lower standard deviation in a user's hand when he/she is walking.

Recall from Section 5.4 that the output of integrating raw gyroscope data is a direction cosine matrix, $C_{phone}(t)$ at each time instant $t$. An alternative representation of 3-D orientation is the *Euler angle representation*: given any direction cosine or rotation matrix, it can be transformed to three Euler angles, called *roll*, *pitch* and *yaw*, often denoted by $\alpha$, $\beta$, and $\gamma$ respectively, that uniquely characterize a 3-D rotation in space [31].

It is possible to show that a change in yaw ($\gamma$) represents a rotation about the z-axis (parallel to gravity), which occurs whenever a user lifts her leg to take a stride. Similarly, a zero value of pitch ($\beta \approx 0$) represents a phone being flat in the $x$-$y$ plane, most likely on a desk or on a user's hand.

Figure 5-7 illustrates our observations. The panes of the figure plot the yaw angle $\gamma$ and the pitch $\beta$ obtained by solving the inertial navigation equation 5.7. The traces were collected by giving a user two iPhones, and asking him to walk with one iPhone in his hand and the other in his left pants pocket. Both yaw and pitch show a greater up-and-down swing with the phone in the pocket, but as it turns out, only yaw exhibits this swinging behaviour consistently across all the traces we have collected. Similarly, a pitch close to zero indicates a phone held flat, usually in a user's hand if the phone is known to be moving.

*iTrack* windows its input trace into chunks of size a few seconds (over which this variance of yaw can be measured noticeably) and uses simple cutoffs on the values of two metrics to decide whether




(a) Yaw angle $\gamma$ from hand and pocket, same walk.

(b) Pitch angle $\beta$ from hand and pocket, same walk.

**Figure 5-7: Distinguishing phone-in-hand and phone-in-pocket using Euler angles.**

each window of data comes from a user's hand or pocket. If the absolute value of pitch $|\beta| < 0.5$ or the median standard deviation of yaw $\sigma_\gamma < 0.1$, the window is determined to come from a user's hand, otherwise it is determined to come from a user's pocket. The values of these cutoffs were calibrated using a set of representative training walks.

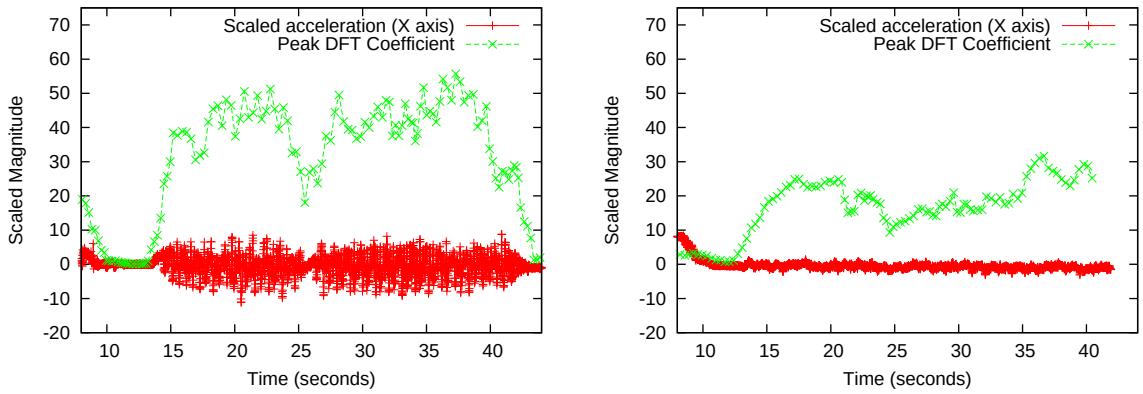
We note that it is possible that the yaw can have standard deviation $\sigma_{gamma} < 0.1$ in some time window if the phone is in the user's pocket, but the user is not walking steadily. Since we only process data from steady walking, it is acceptable for the pose detector to mislabel such time windows as coming from a user's hand. These time windows will be thrown out in any case in the next step (the walking detector) because they do not represent walking.

### *Walking vs Not Walking*

Detecting walking is a trickier problem than detecting movement. To distinguish walking from periods of movement without walking, we use a *frequency domain* detection approach proposed in [75, 5], but modified to take into account the pose of the phone determined in the previous step (hand or pocket).

The key idea of frequency domain walking detection is that walking manifests as a specific kind of periodicity in the accelerometer signal. A phone experiences a periodic up-and-down swing in a user's pants pocket or handbag when walking, or a gentle up-down or sideways motion even when a user walks with a phone in his/her hand. Both of these register a marked peak on a Discrete Fourier Transform (DFT). Such a peak is not found when a phone experiences a one-time (non-periodic) movement, such as a user picking up a phone call. It is possible to deliberately fool a walking filter based on an FFT by simply waving a phone up and down while at rest, but we believe this kind of adversarial behaviour should be rare in practice.

Figure 5-8 illustrates walking detection on data from an iPhone 4 accelerometer. The first pane, Figure 5-8(a) shows data from a trace collected with a phone in a user's pocket when walking, and the second, Figure 5-8(b) shows data from a phone in a user's hand when walking. The figures show the raw X-axis acceleration, as well as a plot of the peak DFT coefficient computed over a sliding window of 256 samples, as a function of time. The two magnitudes do not have the same units, but we have normalized their magnitudes to show them together on the same graph.




(a) X-Axis accel and peak DFT coefficient, phone in pocket. (b) X-Axis accel and peak DFT coefficient, phone in hand.

**Figure 5-8: Walking detection using fourier transforms.**

We can see that the walking zone is easy to visually identify in both situations, and corresponds to an increase in the magnitude of the peak DFT coefficient. Previous work in which the author of this dissertation was a co-author [5] quantitatively shows that the FFT peak power approach has high walking detection accuracy in terms of both precision (over 99%) and recall (over 97%).

The periodic motion is more pronounced in the pocket case because the phone swings up and down more regularly. It is less pronounced in the phone-in-hand case and corresponds to a smaller rise in the peak DFT coefficient. This suggests that the cutoff on DFT coefficient to identify a walking zone should be lower in the phone-in-hand case than the phone-in-pocket case.

To achieve this, we first *distinguish* the phone and hand cases explicitly as described in the previous section. We use a cutoff of 25 on the DFT coefficient for the phone-in-pocket case and a cutoff of 15 for the phone-in-hand case in our implementation of walking detection.

*Limitations*

Detecting walks when a phone in a user's hand is fundamentally unreliable, because a lower cutoff on DFT coefficient means a higher likelihood of *false positives*, where we identify a non-walking zone as a walking zone. Whether false positives are acceptable or not depends on the application. If using the walking data to crowd-source WiFi training in a building, we want a high cutoff to completely eliminate false positives and use only walks we are very confident about. If using the data mainly to understand a user's approximate movement pattern, false positives may be more tolerable. Fortunately, most false positives are short-lived and can be easily distinguished from longer walks by using a cutoff on walking time.

### 5.5.2 Step Counting

The step counting phase takes as input accelerometer and gyroscope data from walking zones detected by the walking detector. The output of step counting is a sequence of $\langle t_1, t_2 \rangle$ pairs for each walking zone indicating the beginning and end time of each stride in the walk. The total number of such pairs gives the number of strides in the walk.

The method used by *iTrack* for step-counting depends on whether the phone is in the user's pocket or held in her hand. In the pocket case, the periodic up-and-down swing of the phone is well-marked

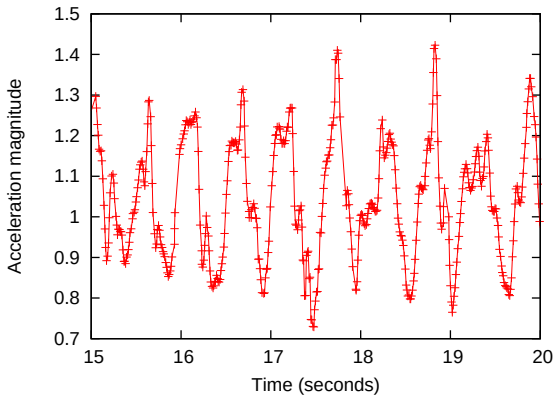



**Figure 5-9: Peaks and valleys in acceleration, from phone held in user's hand.**

on a plot of the yaw Euler angle, as shown in Figure 5-7(a) from the previous section. This swing can be used to be differentiate and count individual steps accurately.

In the phone in hand case, the individual steps are *not visible* on an orientation plot from the gyroscope, but the steps are somewhat discernible from a plot of raw acceleration. Figure 5-9 zooms in to a plot of the magnitude of acceleration over a smaller time scale. The *prominent* peaks and valleys in the acceleration signal approximately correspond to individual steps taken by the user.

In either case, *iTrack* needs to find the major peaks and valleys in a time domain signal — the yaw in the pocket case, and the acceleration magnitude in the hand case. Finding peaks and valleys turns out to be surprisingly challenging, because fluctuations in the value of yaw or acceleration magnitude commonly manifest as false peaks. Figure 5-9, for example, illustrates this problem: the acceleration magnitude often has multiple peaks per stride. It is non-trivial to design an algorithm to count the number of strides, even though the strides can be easily distinguished by eye.

One approach is to use a smoothing (low-pass) filter to eliminate some of the small high-frequency fluctuations. However, we have found that a low-pass filter ends up eliminating some genuine steps irrespective of how it is tuned, which is undesirable.

Hence, rather than using a low-pass filter, *iTrack* uses an iterative heuristic to find the relevant peaks and valleys. The heuristic we use operates in multiple passes. The first pass identifies *all* peaks and valleys (taking wrap-around of orientation into account if operating on yaw angle data). The peaks and valleys extracted in the first pass include fluctuations. The second pass examines the peaks and valleys found in the first pass, and *eliminates* a subset of valleys and peaks that are likely to be from fluctuations. Specifically, we eliminate peaks and valleys that satisfy the following condition(s):

- We eliminate any peak $P$ that has a nearby *higher* peak $P'$ (within some time interval) such that *there is no valley between $P$ and $P'$*. The intuition is that such a peak is not the main peak in its stride with high probability.
- Analogous to the above, we eliminate any valley $V$ that has a nearby lower valley $V'$ (within some time interval) such that there is no peak between $V$ and $V'$.

We repeat the above process iteratively. In practice, we have found that three passes are sufficient to remove all the fluctuations and retain only the major peaks and valleys, one per stride.

| True Step Count | Estimated From Pocket | Estimated From Hand |
|---|---|---|
| 78 | 80 | 90 |
| 32 | 34 | 30 |
| 29 | 30 | 36 |
| 49 | 50 | 56 |
| 64 | 66 | 68 |

**Table 5.1: Step counts from pocket are more accurate than from hand.**

*How Well Does Step Counting Work?*

Table 5.1 shows the accuracy of step counting on 5 walks of different lengths. Each walk was collected with two phones: one in the pants pocket and one held flat in the hand. The user counted out the number of steps loudly while walking to record the true number of steps walked.

The mean estimation error is 3.5% when the phone is in the pants pocket and over 11% when the phone is in the user's hand The peak-finding approach is extremely accurate for a phone in a user's pants pocket, almost always finding the exact number of steps walked to within one or two. However, the approach is not as accurate for the phone-in-hand case since the up-and-down or sideways swings of the phone are not as well-marked and regular. For this reason, the estimated step count inevitably has some error in the phone-in-hand case.

However, the approximate step count even from the hand case is still accurate enough to provide useful information to the next step of *iTrack*, the particle filter. The particle filter in *iTrack* is a probabilistic model that assumes estimated stride counts have error distributed according to some distribution. As we shall see in Section 5.8.10, by using a carefully chosen *non-Gaussian* error distribution for stride length, the particle filter can correct for errors in stride count and still find the correct trajectory walked by the user. This is analogous to how the higher-layer Markov Model in *VTrack* (Chapter 3) deals with erroneous WiFi data obtained from a lower-layer centroid localization algorithm.

### 5.5.3 Shape Extraction

The goal of the shape extractor is to determine the shape of a user's trajectory as precisely as possible from gyroscope data. The shape extractor takes raw gyroscope data as well as the output of step counting as input. The output of the shape extractor is a set of triplets of the form $\langle t_1, t_2, \delta\theta \rangle$ where $t_1$ and $t_2$ are the begin and end times of each stride (same as the output of step counting) and $\delta\theta$ is the *change* in walking direction from the previous stride to the current stride.

To understand how shape extraction works in *iTrack*, recall that the output of integrating the angular velocity from the gyroscope is a direction cosine matrix, $C_{phone}(t)$. The columns of this matrix represent unit vectors along the axes of the phone's reference frame at time $t$. This matrix can be converted to a different representation in terms of three Euler angles: yaw ($\gamma$), pitch ($\beta$) and roll ($\alpha$) [31]. Among these, the yaw represents rotation of the phone's reference frame about the $z$ axis (parallel to gravity), and is therefore the most relevant from the point of view of determining walking direction in the $x$-$y$ plane.

When the phone is held in a user's hand, shape extraction is easy (assuming, as before, that we have filtered out cases where the user uses the phone for a call or fiddles around with it) because the change in the walking direction corresponds to the change in the orientation of the phone in the $x$-$y$

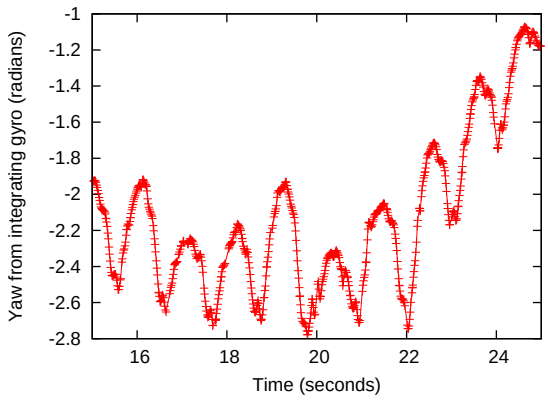



**Figure 5-10: Variation in yaw within each stride, phone in pocket.**

plane, which is the yaw. Therefore, we simply compute the change in yaw from each stride to the next to find the change in walking direction $\delta\theta$ from each stride to the next.

The shape extraction problem is challenging when a phone is in a user's pocket. This is because a phone in a pocket executes a complex rotation about all three axes in 3-D space when the user lifts her leg to start a stride or pivots on her leg while stepping forward with another leg. The relationship between the absolute yaw value of the phone and the direction of the user's walk is non-trivial. In fact, it is *not even fixed: the orientation of the phone relative to the walking direction varies continuously during a person's stride*. We refer the reader to Figure 5-10, which shows a zoomed in plot of yaw angle from integrating gyroscope data. The figure shows this up-and-down variation in yaw within each stride.

The figure, however, shows that the *baseline level* of the yaw (mid-point between peaks and valleys) tracks the turns made by a user quite well. In other words, although the 3-D orientation of the phone relative to user walking direction is unknown, and is never constant, the *change in the baseline level of yaw over multiple strides tracks the change in walking direction very precisely*. For example, observe the transition (increase) in the baseline yaw at approximately $t = 22$ seconds in the figure. This corresponds to a left turn when using the iPhone's convention for reporting angular velocity.

*iTrack* uses this observation to find the shape of a user's trajectory. We use the same peak finding algorithm discussed in the previous section for step counting. Having identified a sequence of peaks and valleys, the algorithm computes the midpoint of each $\langle peak, valley \rangle$ pair. It then computes the change from each midpoint to the next to find the change in walking direction $\delta\theta$ from each stride to the next.

### *How Well Does Shape Extraction Work?*

We have seen that inertial drift is a major problem with integrating raw gyroscope data. *iTrack*'s shape extraction algorithm avoids the drift problem because it only estimates and uses *change* in orientation, rather than absolute orientation. Hence, the estimation error is confined to turn angles, and the error is upper-bounded by the relatively small integration drift error over the short time scale of each turn.

In practice, we have found that shape extraction is extremely accurate for a phone held flat in a user's hand. When a phone is in a user's pocket, however, we have found that raw gyroscope data

on the iPhone sometimes experiences outages lasting for a relatively long time (up to 1 second or more), which sometimes lead to significant errors in estimating turn angle.

It is difficult to provide a quantitative evaluation of shape extraction error because our ground truth approach relies on a human approximately following a marked path (Section 5.8.2), which is not accurate enough to measure the true turn angle precisely. Qualitatively, we have found that when a user is walking straight without changing his direction, *iTrack*'s shape extraction works perfectly and detects that the direction has not changed. When the user does make a significant turn, it is always detected, but the turn angle may sometimes be estimated incorrectly by integration.

This observation influences the design of the particle filter described in the next section. Previous approaches that use particle filters for inertial tracking have used a Gaussian model of error in orientation. As we have described earlier, this does not accurately reflect the underlying process. We instead use a model with a fixed probability of error when a user makes a significant turn, and a smaller probability of error when the user's walking direction is not changing significantly. We show that this model improves the *survival rate* of the particle filter.

### 5.5.4 Particle Filtering

The key step of *iTrack* is a particle filter algorithm that matches the approximate shape and size extracted by the shape extractor and step counter to a floorplan of the building to find the most likely trajectory taken by a user. The particle filter takes the following as input:

- A sequence of $\langle t_1, t_2, \delta\theta \rangle$ triplets indicating the beginning time, end time and change in direction for each stride, output by the shape extractor (see previous section).
- The geometry of the floor the user is known to be on (we discuss how we can find which floorplan to use later), consisting of the geometry of all walls and other constraints.

Optionally, the particle filter can also use the following inputs to improve accuracy:

- A sequence of observed WiFi signatures at times along the walk, of the form $\langle t, w \rangle$.
- A training database of WiFi signatures and known points in the floorplan they were observed from, of the form $\langle x, y, w \rangle$. These could have been learned over time from previous walks collected by the system.

The output of the particle filter is:

- The most likely trajectory taken by the user, as a sequence $\langle t, x, y \rangle$, if it succeeds in finding a feasible trajectory (it may not in all cases, as we shall see).

The key intuition behind the particle filter is that the walls and other obstacles in the floorplan *constrain* the set of possible trajectories the user could have taken. This, in combination with the constraint that the trajectory follows a particular shape and has (approximately) a particular length should in theory suffice to uniquely localize the user within the floorplan, and determine the exact trajectory.

However, this assumption is not true for short walks with few or no turns, and in floorplans with significant intra-plan symmetry. For example, if the floorplan has a box-like layout with identical corridors, knowing the shape and size of a walk does not suffice to uniquely determine its location. In such cases, *iTrack* can either use WiFi localization with its existing training database if available to disambiguate different trajectories, or it can uniquely identify the trajectory if the approximate starting point and/or direction of the walk is known.

We therefore use a hybrid approach, as discussed in the next section. We first collect a few "seed walks" with known start position (input manually on a map) to initialize the WiFi training database on each floor of the building we want to map. Subsequently, users carrying around phones can contribute data in the background without having to mark their locations on a map. We show in the evaluation that *iTrack* is able to use a small amount of seed WiFi training data, of the order of 4-5 walks to localize subsequent walks to within less than a metre. Collecting 4-5 walks of seed data with *iTrack* requires only 5-10 minutes of training effort on a $\approx$ 10,000 sq. ft. floor.

### *How Does iTrack's Particle Filter Differ From Previous Approaches?*

The use of particle filters for indoor localization and tracking is not novel. Robots have used particle filters for localization and mapping (SLAM) indoors. However, they have usually relied on more accurate sensors such as laser range finders. More recently, Woodman and Harle [67] used a particle filtering technique similar to *iTrack* to track pedestrian movement indoors with foot-mounted inertial sensors. *iTrack*'s particle filter differs in three key respects:

- *iTrack*'s particle filter uses a mixture model for stride length to correct for errors in stride count estimation, which are quite common especially with phones held in a user's hand. In contrast, [67] uses a Gaussian model for stride length. This works well for foot-mounted inertial sensors because the length of a pedestrian's stride can be accurately estimated from such a sensor, but as we show, it *does not work well for phones*.
- *iTrack*'s particle filter uses WiFi measurements throughout the filter to improve tracking accuracy. Previous techniques have mainly used WiFi for initialization rather than throughout the course of the filter.
- Walks collected by *iTrack* are fed back to the training database to improve the quality of WiFi localization estimates. This means *iTrack* can be run iteratively with the same training data to further improve matching accuracy, and so on. This iterative process yields an improvement in accuracy and is a novel contribution of our system.

The rest of this section is organized as follows. We first explain what a particle filter is and how it works. We also explain how it relates to a Hidden Markov Model — which was used in *VTrack* and *CTrack* — and why, unlike in *VTrack* and *CTrack*, a particle filter is a more appropriate tool for *iTrack* than an HMM. We then describe the particle filter used in *iTrack*.

### *What is a Particle Filter?*

A particle filter is a Monte Carlo simulation technique that can approximate the *posterior* probability distribution of a continuous-domain Markov process. The easiest way to understand particle filters is to understand how they relate to (and differ from) HMMs. As we have seen in Chapters 3 and 4, an HMM consists of a discrete set of underlying hidden states and observables. Each observable has

a probability of being emitted from each hidden state and there is a transition probability governing transitions between states. Given a sequence of observables, the Viterbi algorithm finds the most likely sequence of (discrete) hidden states.

HMMs naturally generalize to a *continuous* state space, where the hidden state at each time step is a continuous variable rather than one of a set of discrete possibilities. An indoor trajectory of a person walking through a building is an example of such a sequence of continuous-domain hidden states, where each unknown is a location coordinate $\langle x, y \rangle$ on the floorplan. In the continuous domain, one can no longer use the Viterbi algorithm to solve for the most likely sequence of hidden states, since there are an infinite number of possibilities. Finding the most likely trajectory is not analytically tractable in most cases.

Particle filters instead use Monte Carlo simulation to approximate the solution to this problem. The idea is to approximate the *posterior* probability distribution over the space of hidden states by a set of weighted *particles*. In the indoor trajectory mapping context, for example, each particle represents a candidate trajectory through the floorplan. At each step of the filter, each particle advances to a new hidden state (new location in the indoor positioning context), which is randomly sampled from a *proposal distribution*.

The weights of particles are updated after each simulation step to factor in the probability of seeing the observable (sensor data) at that time step from the newly simulated hidden state. Particles with extremely low or zero weight — in the indoor tracking context, paths that violate constraints in the floorplan, are discarded at the end of each step.

At the end of the simulation, what remains is a set of weighted particles that approximates the true posterior distribution over the sequence of hidden states. The most likely sequence of hidden states is simply chosen to be the particle with maximum weight.

### *Formalism*

Given:

- A sequence of observations $o_1, o_2, ..., o_n$.
- The probability density function of seeing an observable $o$ from a given hidden state $h$, $P_{emission}(o|h)$.
- A density function governing transitions between hidden states, $P_{transition}(h_{i+1}|h_i)$.

The sequence of observations $o_1, o_2, ..., o_n$ corresponds to some true (unknown) sequence of hidden states $h_1, h_2, ..., h_n$. The goal of a particle filter is to estimate the posterior probability distribution $P(h_1, h_2, ..., h_n|o_1, o_2, ..., o_n)$, and use this to find the most likely sequence of hidden states.

The particle filter approximates the posterior probability distribution at each step by a set of $M$ particles with associated weights:

$$\langle p^1, w^1 \rangle, \langle p^2, w^2 \rangle, ... \langle p^M, w^M \rangle. \tag{5.12}$$

The particles are initialized according to a prior probability distribution $P_{prior}(h_0)$ over the initial value of the hidden state $h_0$.

At each step $i$, each particle $p^j$ stores a sequence of hidden states up to step $i$: $h_1^j, h_2^j, ..., h_{i-1}^j$. The next hidden state, $h_i^j$ is sampled from a *proposal* distribution $P_{proposal}(h_i|h_{i-1})$. The proposal distribution is usually the same as the transition probability distribution (though it need not be).

At each step $i$ of the Monte Carlo simulation, the weight $w^j$ of each particle $j(1 \leq j \leq M)$ is updated using Bayes theorem:

$$w_i^j = w_{i-1}^j \times \frac{p(o_i|h_i^j)\, p(h_i^j|h_{i-1}^j)}{P_{proposal}(h_i^j|h_{0..i-1}^j, o_{0..i})} \tag{5.13}$$

If the proposal distribution used is the transition probability distribution (as is usually the case), this simplifies to:

$$w_i^j = \begin{cases} w_{i-1}^j \times p(o_i|h_i^j), & P_{proposal}(h_i^j|h_{0..i-1}^j, o_{0..i}) \neq 0 \\ 0, & \text{otherwise} \end{cases} \tag{5.14}$$

The above weight update rule can be interpreted intuitively as follows. If the newly simulated hidden state for a particle has zero probability — for example, if it violates a constraint, such as crossing a wall, then the particle's weight should be set to zero, killing the particle. Otherwise, the particle's weight should be multiplied by the emission probability for the new observation from the newly simulated hidden state. Unlike in the Viterbi decoding algorithm for a HMM, we do not multiply by the transition probability because it is already implicitly accounted for in the way we sample new hidden states from the proposal distribution.

**Resampling.** As the importance weights of a particle filter are updated according to equation 5.14, the importance weights of all but a few of the particles tend to approach zero or become zero. This happens in indoor trajectory mapping because most simulated particles start with the wrong initial position or orientation and quickly hit walls or violate other floorplan constraints. This leads to a problem called *degeneracy*: if only a small number of surviving particles are left behind after the first few steps of the simulation, they cannot satisfactorily explore the state space on subsequent steps. For this reason, most particle filters include a *resampling* step to ensure the number of surviving valid particles remains above a threshold.

We use a standard technique from the particle filtering literature called sequential importance resampling (SIR). SIR works by computing an estimate of the effective number of particles at the end of each step:

$$M_{eff} = \frac{1}{\sum_{j=1}^{M} (w_j^{norm})^2} \tag{5.15}$$

where $w_j^{norm}$ is the normalized importance weight of particle $j$, obtained from the following equation:

$$w_j^{norm} = \frac{w_j}{\sum_{j=1}^{M} w_j} \tag{5.16}$$

If the effective number of particles $M_{eff}$ is less than a given threshold $M_{thresh}$, then the filter performs resampling. Resampling works by re-drawing M particles from the current particle set

with probabilities proportional to their importance weights. The M new particles are used to replace the current set of particles, and all their importance weights are set to $\frac{1}{M}$.

*Particle Filter Used In iTrack*

We first describe the simplest version of the particle filter used in ***iTrack*** that does not take WiFi signal strength observations into account (we later show how we can modify the filter to do this). The basic filter is a slightly modified particle filter with the following components:

- The "steps" of the Monte Carlo simulation are individual strides made by the user, and extracted by the step counting algorithm described earlier.
- The unknown hidden state $h_i$ at the end of step $i$ is a triplet $\langle x_i, y_i, \theta_i \rangle$ where $x_i$ and $y_i$ are coordinates of the user's location on the floorplan and $\theta_i$ is the *absolute* walking direction of the user ($\theta = 0$ represents due East if the floorplan is oriented N-S).
- The observable at each step $o_i$ is $\delta\theta$, the change in angle from the previous stride to the next stride, obtained from the output of ***iTrack***'s shape extractor.

Rather than using an explicit emission probability distribution for the observable $\delta\theta$, we fold the observable into the proposal distribution for the particle filter and set the emission probability to be equal to 1 always. The proposal distribution we use generates the new hidden state of the particle filter at step $i$ given the old hidden state at step $i-1$ using the following rules:

$$\theta_i = \theta_{i-1} + \delta\theta + E_{\delta\theta} \tag{5.17}$$

$$x_i = x_{i-1} + l\ cos(\theta_i) \tag{5.18}$$

$$y_i = y_{i-1} + l\ sin(\theta_i) \tag{5.19}$$

Importantly, particles where the transition from $\langle x_i, y_i \rangle$ to $\langle x_{i+1}, y_{i+1} \rangle$ crosses a wall or obstacle in the known floorplan are assigned zero weight and killed.

In the above rules:

- $E_{\delta\theta}$ is a perturbation error term that models the error in extracting turn angles from the gyroscope. This term is drawn from an *angle error distribution* we describe below.
- $l$ is a stride length sampled from a *stride length distribution*.

The angle error distribution models the error in extracting a turn angle from the phone gyroscope. Each simulated particle is assigned a slightly different random turn angle which is a perturbation of the measured turn angle from the gyroscope.

Similarly, the stride length distribution models the uncertainty in how long each stride is. Each simulated particle uses a different random stride length at each time step. Some particles end up

modeling stretches of walk where the user takes short strides, and others explore paths with longer strides.

In a collective sense, the simulated uncertainty in turn angle and stride length help explore the space of possible trajectories thoroughly and discover the true trajectory of the end user, correcting for estimation errors made by the step counting or shape extraction phases. Particle trajectories that simulate the wrong stride lengths or angles end up crossing walls or violating constraints, and are assigned zero weight. Particles that do not violate the floorplan constraints are automatically given higher weight.

**Resampling.** In the *iTrack* case, some weights are zero and the other importance weights are all equal. In this case, the effective weight for resampling $M_{eff}$ reduces to a simple count of the number of surviving particles. The resampling step reduces to:

- Discarding particles that have not survived.
- Making duplicate copies of the surviving particles randomly so that the total particle count remains at $M$.

Our particle filter implementation uses $M = 100,000$ and $M_{eff} = 10,000$. Both these values reflect a trade-off between how thoroughly we need to explore the state space, and how computationally expensive the particle filter ends up being. Larger values correspond to better exploration and higher accuracy, but slower running times, and vice versa. We describe more details of this tradeoff and our choice of parameters in Section 5.7.2.

### *Choice Of Error Models and Survival Rate*

The particle filter as described above may not always produce a valid trajectory. Since the number of particles that can be simulated, $M$, is finite owing to computational constraints, if *all* the particles being simulated end up violating a floorplan constraint by crossing a wall or obstacle at the same time instant during the trajectory, the filter ends up producing no output at all.

The usual reason for a failure to survive is an error in the sensor measurements that is unlikely to occur under the model being used for angles or stride length. This produces an incorrect shape which cannot be corrected by the filter unless an order of magnitude more particles are simulated (which is computationally infeasible).

The choice of angle error distribution and stride length distribution are therefore important to ensure a good survival rate. Unlike previous approaches, *iTrack* uses non-Gaussian models for both angle error and stride length, which we show to have better survival rate than Gaussian models since they are more faithful to the underlying sensor data from a phone (Section 5.8).

### *Angle Error Model*

As we have discussed earlier, the shape extractor usually does a good job of detecting when a person's walking direction is not changing much, and also of detecting turns. However, it sometimes computes the turn angle incorrectly owing to gaps in the gyroscope data that lead to errors in numerical integration. For this reason, rather than using a simple Gaussian model of error in $\delta\theta$, we use a mixture model that simulates two types of error in the angle:

- A Gaussian error with small variance $\sigma_{small}$ that we add to each $\delta\theta$ sample from the gyroscope with probability $p_1$.
- A Gaussian error with larger variance $\sigma_{large}$ that we add *only to samples that represent significant turns*, with some probability $p_2$. A significant turn is defined to be one that exceeds a certain cutoff ($\delta\theta \geq \theta_{turn}$).

The first type of error simulates small thermal or other noise, while the second captures integration errors that are made sometimes when computing the change in orientation over a significant turn.

We used $\sigma_{small} = \frac{\pi}{100}$, $\sigma_{large} = \frac{\pi}{10}$, and $\theta_{turn} = \frac{\pi}{10}$, calibrated from tuning experiments. A reasonably wide range of values appear to work for these constants.

We show in our evaluation (Section 5-20(b)) that using a mixture model improves the *survival rate* of our particle filter. Compared to a Gaussian model for angle that has a 68% survival rate, the mixture model we use has an 82% survival rate.

### *Stride Length Model*

*Why, the height of a man, in nine cases out of ten, can be told from the length of his stride.*

- Sherlock Holmes to Dr. Watson, *A Study in Scarlet* (1887)

Similar to angle error, *iTrack* uses a mixture model for stride length rather than a simple Gaussian model. The reason for this is that a person's stride is usually multi-modal: the stride length varies depending on how fast he/she is walking. The stride length is usually longer when a person is walking faster and shorter when a person is walking slower. A Gaussian model for stride length cannot capture this because a sequence of short or long strides is always extremely unlikely (in fact, exponentially unlikely) in a Gaussian model, *irrespective of its standard deviation*.

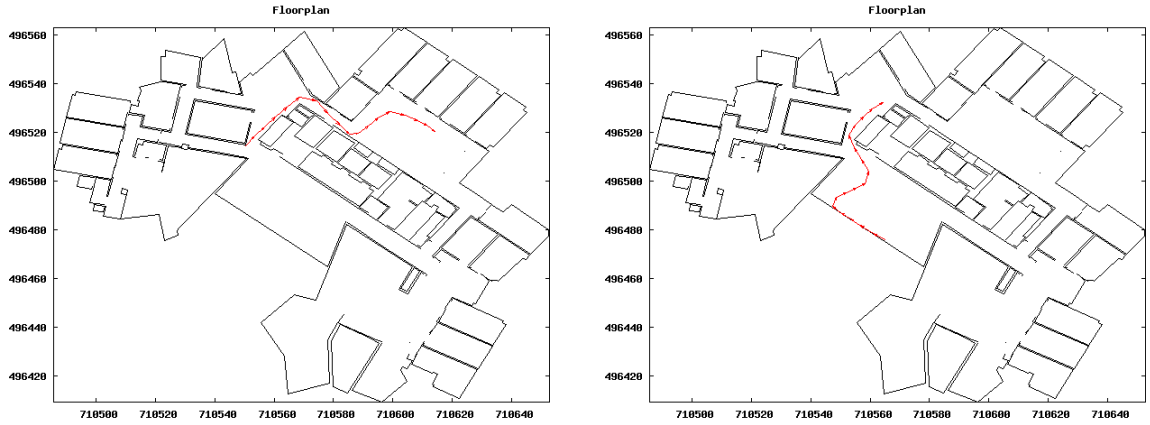
For example, in situations where a person traverses a stretch or a corridor slowly using a sequence of short strides (quite common in our experience), a Gaussian model will inevitably fail to find even a single particle that survives the passage through the corridor.

For this reason, we use a mixture model which keeps track of the previous stride length, and either *transitions* to a different stride length on a new stride with a transition probability $P_t = 0.5$, or remains at the same stride length with probability $1 - P_t$. Each new stride length is drawn from a Gaussian with mean equal to approximately 3 feet (the average stride length of a human). As suggested by the quotation at the beginning of this section, the stride length model can be improved if the height of the user is known. However, we do not rely on this since this requires extra calibration.

We show in our evaluation that using a mixture model for stride length significantly improves the survival rate over using a Gaussian model from 64% to 82%.

### *Bootstrapping the Filter*

We find in our evaluation that the simple particle filter (as described above) is inaccurate, producing the wrong output for 30% of its input trajectories. This happens because the initial position and orientation of the user are unknown, and there are multiple particles that have non-zero weight, all with different trajectories. With short or medium length walks, the same shape can be a valid path anywhere within a large floorplan owing to symmetry in the building plan. This is a serious problem




(a) First candidate particle (close to truth).

(b) Second candidate particle (incorrect).

**Figure 5-11: Example showing ambiguity in the output of the particle filter.**

because in practice, many walks taken by a user are likely to be short or medium length walks e.g., a quick stroll to the lounge, a meeting room, or a vending machine in an office building, or a walk between close by sections of a store or museum.

Figure 5-11 illustrates the ambiguity in the output of the particle filter with an example from a short walking trace. The output of the filter consists of many particles with equal weight, of which the figure shows two. Figure 5-11(a) shows a particle that is close to the true trajectory walked by the user. Figure 5-11(b) shows a second particle whose shape is *identical* to the first particle, but is the incorrect output. Without knowledge of initial position or orientation or other external information, it is impossible to distinguish between the two traces from inertial sensing data alone.

*iTrack* uses two main techniques to eliminate ambiguity in matching short and medium length walks:

- If some "seed WiFi" information is available, it integrates WiFi localization probabilities into the particle filter to disambiguate output particles.
- It can use a *dedicated training phase* where the user explicitly inputs his/her initial position on a map of the floorplan, by marking it on a mobile device. However, unlike previous crowd-sourcing schemes [46, 70], it is sufficient for the user to mark the starting location — the rest of the walk can be deduced automatically using our particle filtering algorithm.

In practice, our system uses a combination of both the above techniques. For each floor of each building we want to cover, we use a dedicated training phase where a single person or group of people contribute walks to the system. The user marks initial position information explicitly on a map before he/she begins walking. We show a screenshot of our iPhone user interface for training in Section 5.7.

Once a small number of such "seed walks" have been collected spanning the entire floor, the WiFi data from the walks can be used to map subsequent walks accurately without knowing their initial position. When this happens, the dedicated training phase is complete. From here on, as we show in our evaluation, any walk the user takes with the phone in his/her hand or pocket can be tracked accurately irrespective of its length.

This is a significant improvement over previous crowd-sourcing techniques because it requires only a limited number of manual training points on each floor being mapped.

The next sections describe how *iTrack*'s particle filter incorporates initial position information and WiFi localization information, respectively.

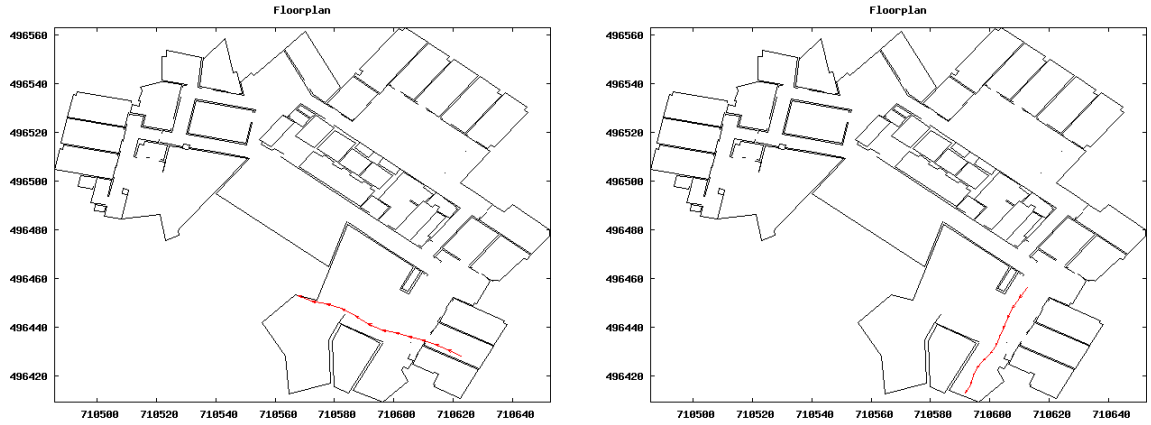
### *Incorporating Initial Position*

It is possible to incorporate approximate knowledge of initial position by constraining the prior distribution $P_{prior}(h_0)$. Suppose a user marks her initial location on the floorplan to be $\langle x_0, y_0 \rangle$. The particles used in our particle filter are initialized with values of $h_0$ randomly drawn from a uniform distribution within a circle with centre $(x_0, y_0)$ and radius $R$. $R$ is a radius of uncertainty to allow for some error in the user input. We use $R = 10$ feet in our implementation.

We show quantitatively in our evaluation that knowledge of initial position significantly improves the accuracy and recall of trajectory matching indoors.

**Initial Orientation.** In addition to incorporating initial position, we have implemented an improvement to *iTrack* that can use the magnetic compass to find initial orientation and use it to improve the accuracy of trajectory mapping. This is used in dedicated training mode when gathering seed WiFi data.

The training procedure requires users to hold a phone flat in their hand held forward *in the direction they are going to be walking in*. The system detects when the phone is flat in the person's hand using the techniques we have described earlier, and measures the absolute orientation of the phone relative to geographic north. This orientation is used to initialize the initial orientation $\theta_0$ of particles in the prior. Orientation from the magnetic compass tends to be quite inaccurate, especially indoors near metallic objects. Hence, we sample the initial orientation of particles uniformly from a (relatively large) radius of uncertainty, $\theta_0 \pm \frac{\pi}{5}$, just as in the case of initial position.




(a) Initial position known, orientation unknown.

(b) Initial position and orientation known.

**Figure 5-12: Knowledge of initial direction helps improve trajectory matching.**

Figure 5-12 shows an example of a short walk on which knowledge of the initial orientation improves trajectory mapping. Figure 5-12(a) shows the output of our particle filter when given the approximate initial position (from WiFi localization), *but not the initial orientation*. Figure 5-12(b) shows the filter output when given both the initial position and walking direction (from compass).

This is also the true trajectory walked by the user. Without knowing the initial walking direction, we see that it is hard for the algorithm to prefer either of the trajectory in Figure 5-12(a) or Figure 5-12(b), because the initial location obtained from WiFi localization is approximate and cannot distinguish the two starting points.

*Incorporating WiFi Information*

If a seed WiFi training database is available, *iTrack* can use WiFi location information to significantly improve the accuracy of trajectory mapping. WiFi helps eliminate ambiguity between candidate trajectories in different parts of the floorplan space that have similar shape.

We incorporate WiFi location by modifying the particle filter to include an emission probability distribution for WiFi. Given a new observation $o_i$ with a WiFi fingerprint $F_i$, we modify the particle weight update rule to:

$$w_i^j = w_{i-1}^j \times E(F_i|x_i^j, y_i^j) \tag{5.20}$$

Here, $E(F|x, y)$ is an emission score proportional to the probability of seeing wifi signature $F$ from location $(x, y)$ in the floorplan. We use the gridding approach proposed in *CTrack* to evaluate this emission score. We divide the floorplan area into equal-sized grids, and find the grid $G$ containing coordinate $(x, y)$. We set $E(F|x, y)$ to be identically equal to $E(F|G)$, the the emission score in *CTrack* (Section 4.5.3, Chapter 4).

We also modify the SIR resampling step of the particle filter to incorporate the weights from WiFi observations and compute the true effective weight using equation 5.15.

We show in our evaluation that incorporating WiFi localization information significantly improves the accuracy of *iTrack*, and a small amount of seed WiFi information is in fact necessary for the system to work correctly.

## 5.6 Simplifying Training With *iTrack*

One of the objectives of *iTrack* is to significantly reduce the effort to gather training data in a new building or floorplan, and to enable users to easily contribute crowd-sourcing data with little effort. To achieve this, *iTrack* uses a novel *iterative* algorithm to maximize the accuracy of WiFi training data collected and to collect the highest quality training data from a set of user walks. The algorithm proceeds in three phases: *dedicated training*, and two passes of *crowd-sourcing*, as we explain.

### 5.6.1 Dedicated Training

In this phase, a dedicated training person or volunteer collects a few seed walks using a well-defined procedure to collect a small amount of "seed" WiFi training data for each floorplan.

Before each walk, if necessary, the person explicitly chooses which floorplan to train for from a set of nearby candidate floorplans, found using network location (such as obtained from the Skyhook or Google API) or last known GPS location. He then marks initial location on a map of the floor while holding the phone in the direction he intends to start walking. He puts the phone in his pocket, and starts walking. When done, he takes the phone out of his pocket and indicates that the walk is complete, upon which the walk is sent back to the *iTrack* server.

The seed walks should cover the important regions of the floorplan but do not have to comprehensively cover every location. For example, one person could collect the seed data for the 9th floor in the MIT Stata Center in a few minutes.

### 5.6.2 Crowd-sourcing Phase 1

End users download and install the *iTrack* application on their phones to view their approximate location on the map, and to contribute data. Importantly, once dedicated training is complete, *end users do not have to take any effort to contribute data — they simply need to run the iTrack application on their phones*. This is a big improvement over having to explicitly mark or indicate location on a map.

The system continuously runs in the background as shown in Figure 5-5, detecting walks made by the user. Each new walk is uploaded to the *iTrack* server.

Having collected a certain number of walks from users, the system runs an initial pass of the particle filter over all the walks, with the goal of extracting more WiFi data. This first pass of the filter uses the seed WiFi data from dedicated training.

A key optimization we use is to add WiFi data back from the first pass to the training database, and then *re-run iTrack over all the walks* using the augmented database. This constitutes Phase 2.

### 5.6.3 Crowd-sourcing Phase 2

Phase 2 of crowd-sourcing runs a second pass of the particle filter over *all the walks* collected by the system, using the augmented WiFi training database from the first pass as input. When matching a walk $W$ in the second pass, we selectively use only the WiFi training data that did *not* come from the first pass of running *iTrack* over $W$. The idea of this "leave-out" technique is to avoid reinforcing errors made in the first pass. If the first pass produced a wrong trajectory for $W$, this incorrect data gets added to the WiFi training database and further increases the likelihood that the second pass will also produce the same (or very close by), but incorrect output. Our approach ensures this does not happen.

We show in our evaluation that using a second pass of iteration helps improve the accuracy of trajectory matching, especially in the tail (not so much in the median). Some of the walks that do poorest on the first pass improve substantially in the second pass. We use only two passes because we have not found additional passes of the algorithm to yield substantial benefit.

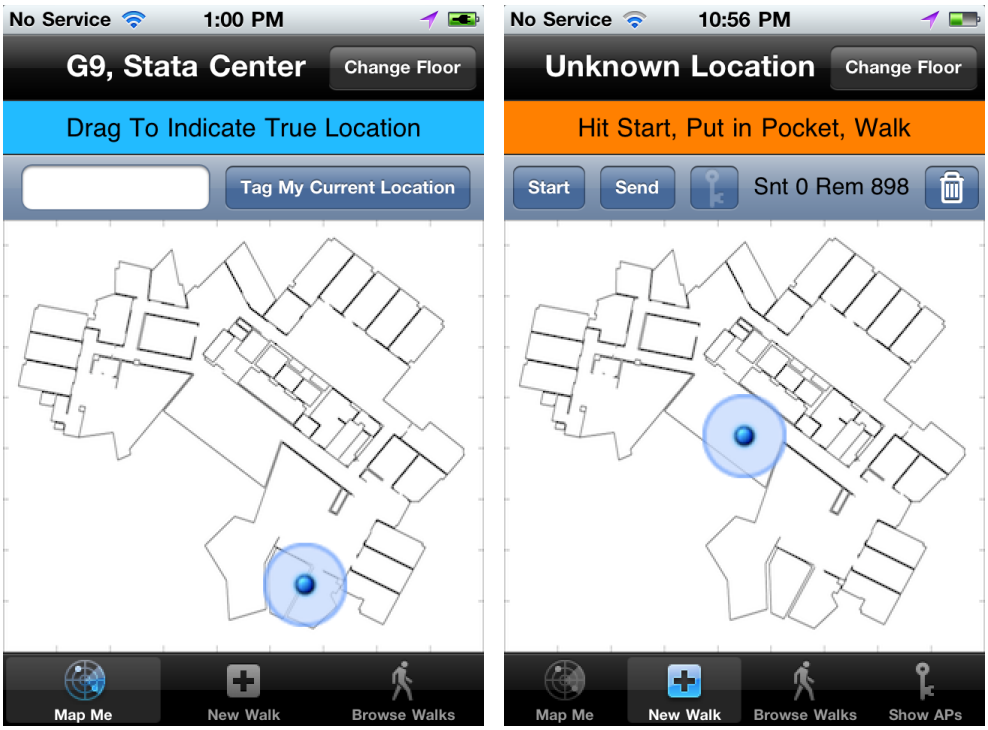
## 5.7 Implementation

We have implemented *iTrack* as an iPhone 4 application. The application samples the in-built accelerometer, gyroscope, and magnetic compass, and sends back data over the internet to a central server which runs a particle filter to perform trajectory matching, as shown in Figure 5-5. The application also makes it easy to carry out the training procedure described in the previous section.

### 5.7.1 iPhone App

The application, shown in Figure 5-13 has four tabs:

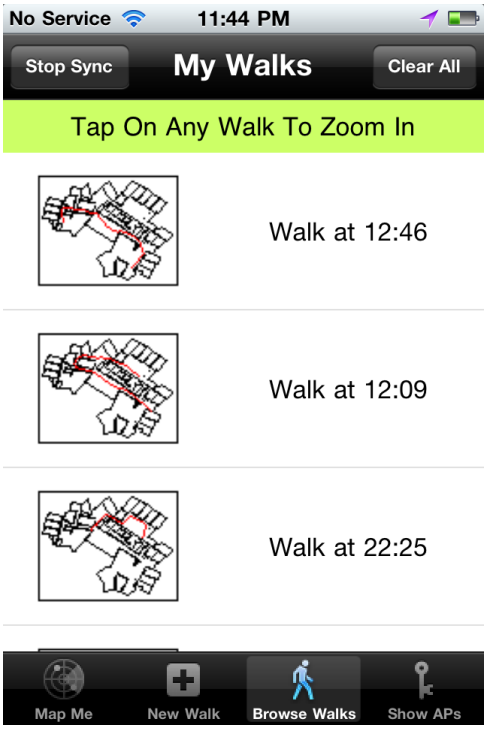
- A *Map Me* tab that shows the user their approximate (current) position on the floorplan using WiFi localization. Figure 5-13(a) shows a screenshot of this tab.
- A *New Walk* tab that allows a user to contribute a training walk in the dedicated training phase. Figure 5-13(b) shows a screenshot of this tab. It is almost identical to the *Map Me* tab, except that the blue dot in the centre can be moved around by the user to indicate her initial position to the system before walking. The user hits Start, puts the phone in her pocket, walks, and then hits Stop to send the data from the walk back to the central server.




(a) Tab showing current location.




(b) Tab to contribute training walks.




(c) Tab showing previous training walks.


(d) Tab showing nearby WiFi access points.

**Figure 5-13: *iTrack* application for the iPhone.**

- A *Browse Walks* tab that shows the user their previous walks. Each time the user collects a training walk, it shows up in this tab once the system has finished map-matching it. It also provides a UI to indicate whether or not the walk found was correct, to obtain user feedback. Figure 5-13(c) shows a screenshot of this tab.
- A *Show APs* tab shows nearby access points and their signal strengths. Figure 5-13(d) shows a screenshot.

### 5.7.2 Run Time of Map-Matching

The running time of *iTrack* is dominated by the run time of the particle filter since the steps to extract strides and angles are comparatively very fast, and linear in the size of the input sensor data.

The running time of the particle filter is upper-bounded by $O(MN)$, where $M$ is the number of particles used for simulation and $N$ is the number of strides in the user's walk. In practice, if $M'$ is the threshold on effective weight for resampling, the running time is closer to $O(M'N)$ because approximately $M'$ particles survive on average at each time step. It is possible to implement further optimizations that have been proposed in the particle filtering literature — for example, to track the convergence of the particle and adaptively reduce the number of particles simulated when the filter converges. We have not done so in our initial implementation.

We have implemented the *iTrack* particle filter in C++. The filter runs as a server side process that continuously listens for input from new walks. When a walk is complete on the phone side, the phone sends back inertial and WiFi data from the walk to the server, triggering the particle filter to run on the server side. We do not focus on server side scalability issues in our implementation (and this dissertation).

Our filter implementation uses $M = 100,000$ particles and a cutoff of $M' = 10,000$ to trigger SIR resampling. We have found this number to be necessary to achieve acceptable accuracy in our experiments. More particles give slightly more benefit, but incur excessive computational cost. As it stands, our C++ implementation runs approximately in real time on a 2.33 GHz Macbook Pro laptop with 3GB RAM, taking about a minute to match data from a minute of walking. While collecting training data, the trajectory matching can run in the background, so a user need not wait for it to complete before walking again.

### 5.7.3 Ideas For Performance Optimization

Taking a minute to match one minute of data is on the slow side if trying to scale iTrack to thousands of buildings or users. We are investigating the following performance optimizations in future work:

- Reducing the number of particles used, $M$, if the initial position and orientation are known more accurately (e.g. in steady state when a substantial WiFi training database has been built).
- Using known techniques from the particle filtering literature like adaptive resampling, which reduces the number of particles when the variance of the particle cloud falls below some threshold.
- Grouping together stretches of walk without significant turns, where the user walks straight, to reduce the effective number of steps of simulation. The most likely sub-trajectory within each such stride group can be solved for analytically. This is similar to Rao-Blackwellization, a state space factorization technique commonly used in the particle filtering literature [2].

- Parallelizing the particle filter by running each particle in parallel (though some coordination is required for periodic resampling).

## 5.8 Evaluation

We evaluate *iTrack* using a data set consisting of:

- 50 walks with known ground truth trajectories, collected on the 9th floor of the MIT Stata Center. Most of the walks were done with phone in the user's pocket.
- 97 walks without recorded ground truth information. These walks are mainly used to collect a dense data set of WiFi training data for comparison. These walks were collected from phones both in a user's hand and pocket.

We use tape markings made manually on the floor to determine precise ground truth for the walks. The markings are accurate to within less than a centimetre, but the accuracy of the ground truth itself depends on how faithfully the user followed the marked tape. We believe it should be well within a foot.

The key questions we answer in our evaluation are:

- What is the absolute accuracy of *iTrack* when mapping a person's trajectory indoors? How does this compare to WiFi localization without a gyroscope?
- How much seed WiFi training data is required for *iTrack* to work well? How much time and effort does it take to collect this seed data?
- How well does *iTrack* work in dedicated training mode when initial position and/or orientation are known?
- When used for crowd-sourcing, to what extent does *iTrack* help reduce manual training effort compared to marking points manually on a map?
- Are the specific models of angle and stride error used in *iTrack* better than using simple Gaussian models?

We do *not* answer the following (arguably important) questions in this dissertation:

- How much does the accuracy of *iTrack* depend on the layout, shape and size of the building floorplan? One would expect *iTrack* to perform better in floorplans with tighter corridors and more turns, and worse in open spaces where there is a lot of ambiguity, and a given shape can map to many different legal trajectories.
- How much does the accuracy of *iTrack* depend on the density and number of WiFi access points (relative to floorplan area)?

The above questions are interesting, and we plan to evaluate them going forward as we deploy *iTrack* on more floors and in different buildings.

### 5.8.1 Key Findings

We summarize the key findings of our evaluation below:

- *iTrack* is extremely accurate at mapping trajectories indoors given a small amount of seed WiFi training data collecting using our dedicated process. In our experiments, approximately 5 minutes worth of seed training data (4 training walks) resulted in a mean trajectory mapping error of 3.1 feet on our test data set. An error of 3.1 feet is 5-6$\times$ more accurate than using just WiFi localization in our building, which has a mean localization error of 5.4 metres.
- The recall (survival rate) of the algorithm is approximately 82%.
- The mixture models we use for stride length and angle error yield a significant improvement over the Gaussian models used in previous work in terms of recall. They improve the survival rate from 64% to 82% and 68% to 82% respectively.
- The dedicated training procedure we use is accurate and quick, with mapping errors smaller than 3 feet and a survival rate of 88% when using start position and walking direction information to initialize the particle filter.
- Using multiple passes of iteration for crowd-sourcing has significant accuracy benefits. The reduction in median error is not very big (from 3.5 feet to 3.1 feet), but the second pass reduces *tail* error significantly, correcting some of the worst mistakes in the first pass.

The rest of this evaluation is organized as follows. Section 5.8.2 describes the ground truth setup we use, and Section 5.8.3 describes the procedure we use for collecting walking data. Section 5.8.4 explains the evaluation metrics we use. The subsequent sections are devoted to answering the key questions posed earlier, starting with evaluating the absolute accuracy of *iTrack* (Section 5.8.5).

### 5.8.2 Ground Truth

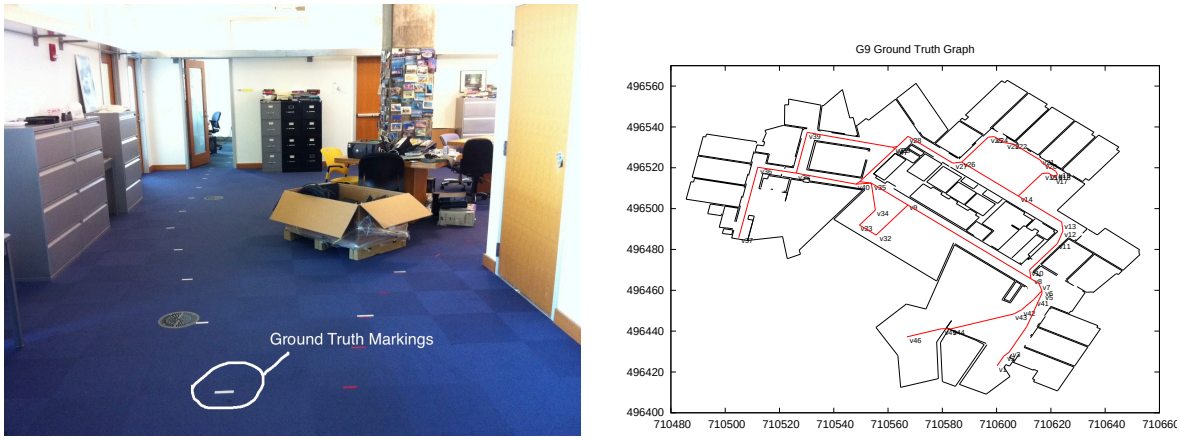
Recall that we used a cleaned up form of GPS as ground truth when evaluating both *VTrack* and *CTrack*. Since GPS does not work indoors, indoor location ground truth is inherently harder to obtain and it is difficult to obtain a large amount of ground truth data.

We used a relatively "low-tech" approach to obtaining ground truth in which we manually placed tape markings on the 9th floor of the MIT Stata Center. The carpet is laid out in uniform square grids, making it relatively easy to place these markings at regular intervals. Figure 5-14(a) illustrates a section of the floor instrumented with these tape markings.

We placed about 150 such markings on the ground, of which 46 were designated as *vertices*. They satisfy the property that any of the markings lies on a line segment joining two vertices. Thus, all the markings lie on the edges of a *ground truth graph*. Figure 5-14(b) shows this graph on a floorplan of the building.

We manually measured the exact $(x, y)$ coordinates of each of the 46 vertices of the graph using measuring tape to compute the distance from walls and/or other vertices in the graph, and known coordinates of the walls obtained from the floorplan architectural data.

The 50 "labeled" walks we used for evaluation were all collected by deliberately walking only along the contours defined by the markings. As soon as each walk was completed, a human being recorded




(a) Photograph of ground truth setup.

(b) Ground truth graph.

**Figure 5-14: Ground truth setup for evaluating *iTrack*.**

the sequence of vertices traversed in the walk. For each labeled walk, we are now guaranteed that the true trajectory must traverse exactly the sequence of vertices recorded in the ground truth log. Each walk has a steady pace and does not include stops or interruptions. However, some walks have sharp or sudden turns, including U-turns.

However, since we did not record any timing information for the walks, we do not know exactly *when* these vertices are crossed in the course of the walk. To recover this information very accurately, we use the (known) fact that the gyroscope data from the walks can precisely pinpoint the times of major turns in each walk. We ran a constrained version of our particle filter that *only explores particles that traverse the known sequence of vertices in the known order*. This helps align the turn times from the gyroscope exactly to the sequence of vertices and obtain accurate crossing times for each vertex. We manually verified that this was indeed the true crossing time for *each vertex in each of the 50 walks*. While this was a painstaking process that took about a day of repetitive work, we did this to avoid biasing the ground truth by mistakes made by the constrained particle filter. The result is a highly clean, reliable set of 50 ground truth walks, including timing information, that we can trust to be accurate.

### 5.8.3 Walk Data Collection

We collect each of the 50 ground truth walks using the following procedure. We go to a randomly chosen vertex in the ground truth graph (any vertex in the set $v_1, v_2, ..., v_{46}$ in Figure 5-14(b)). The user holds the phone outstretched in his palm pointed towards the initial direction of walking and marks his approximate location on the phone UI shown in Figure 5-13(b). He then hits a "start" button to start logging inertial sensor and WiFi data, and places the phone in his left or right pocket, or in his hand (most of our walks were collected from users' pants pockets). He walks along the contours defined by the markings on the floor and finally stops at some end vertex. When done, the user takes the phone out of his pocket and hits a "Stop" button to send data back to the *iTrack* server.

While we recorded the starting position and orientation of each walk during data collection for convenience, some of our experiments do not use this information. We evaluate the performance of *iTrack* both with and without knowledge of the initial position and walking direction, as the following sections describe.

The next section defines the evaluation metrics we use throughout the rest of the evaluation.

### 5.8.4 Evaluation Metrics

We use two key metrics in our evaluation: *localization error* and *survival rate*.

*Localization Error*

Assume we have a ground truth trajectory $G$ consisting of a sequence of the form:

$\langle t_1^G, x_1^G, y_1^G \rangle$, $\langle t_2^G, x_2^G, y_2^G \rangle$, ..., $\langle t_m^G, x_m^G, y_M^G \rangle$,

and assume we are trying to evaluate a strategy that produces an output trajectory $S$:

$\langle t_1^S, x_1^S, y_1^S \rangle$, $\langle t_2^S, x_2^S, y_2^S \rangle$, ..., $\langle t_m^S, x_m^S, y_N^S \rangle$,

Note that that it is possible that $N \neq M$.

We compute the localization error for each output location sample $\langle t_i^S, x_i^S, y_i^S \rangle$ by linearly interpolating the available ground truth locations to find the true location $\langle x^G, y^G \rangle$ at time $t_i^S$. The localization error in this sample is given by:

$$Err_i = \sqrt{(x_i^S - x^G)^2 + (y_i^S - y^G)^2} \tag{5.21}$$

The experiments in our evaluation show both mean values of the error $Err_i$ over entire trajectories, as well as probability distributions of this error.

*Survival Rate*

As mentioned in Section 5.5.4, the particle filter in *iTrack* does not always succeed in producing an output trajectory on all of its runs. When errors in the input sensor data are too extreme to be corrected by the filter, it sometimes happens that all the particles being simulated hit walls and die, leaving no valid trajectory. If computational power or latency is not a constraint, it is possible to re-run the simulation with a larger number of particles (we normally use 100,000 particles) but this too may fail to produce valid output if the error in shape is too extreme.

For this reason, we measure a second metric: the *survival rate* of a strategy. This refers to the proportion of walks that produce some answer when run through our standard particle filter with 100,000 particles (chosen for its reasonable running time). A low survival rate indicates low recall accuracy. For example, a 50% survival rate would mean that 50% of the walks fail to produce any trajectory match when run through the particle filter.

### 5.8.5 Absolute Accuracy of iTrack

Figure 5-15 illustrates a CDF of the mean localization error over the test walks when using *iTrack*. The *iTrack* algorithm in this experiment was seeded with WiFi training data from 4 walks. The 4 walks were randomly selected from the set of 97 walks collected without ground truth, and did not overlap with the test set of 50 used for evaluation. The 4 walks amount to all of 150 WiFi samples, and required only 3-4 minutes of training time for one person to collect using the iPhone app. The particle filter in this experiment was not given the initial position or walking direction of the user.

The error shown in the graph was computed by finding the mean localization error over all the output location samples in a walk. The CDF for *iTrack* was computed across 41 of the 50 test walks (9 failed to produce valid output as described below). As a reference point for comparison, the figure

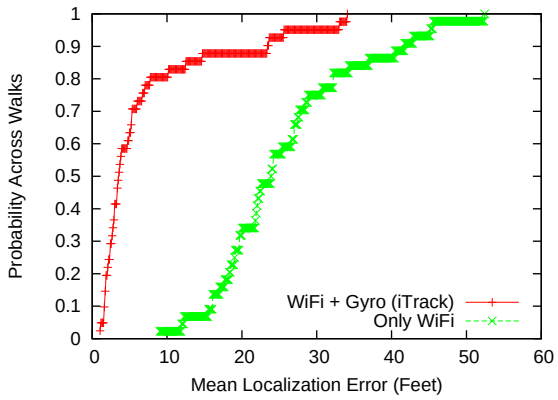



**Figure 5-15: Absolute accuracy of *iTrack* compared to WiFi localization.**

also shows a CDF of the mean localization error when using the seed WiFi data to find the user's track (with a *CTrack*-like algorithm).

As can be seen, using inertial data from the gyroscope results in significantly more accurate tracks than using WiFi localization for trajectory mapping. The median error with *iTrack* is 3.5 feet, 5-6× smaller than the median error with only WiFi, which is of the order of 5.4 metres. *iTrack* also yields significant benefits in the tail, reducing the 80th percentile error by a factor of over 3× and the 90th percentile error by over 1.5× compared to WiFi.

**Survival Rate.** We found that *iTrack* seeded with the 4 walks selected above had a survival rate of 82%, with 41 of the 50 test trajectories producing some output. 9 of the test walks failed to produce any output.

### 5.8.6 Impact On Training

In this section, we show that *iTrack* is nearly an order of magnitude faster and easier at building a WiFi training map for indoor localization than using manual training. It is important to be able to localize or track a phone using WiFi alone because not all devices have gyroscopes, and a phone is not always guaranteed to be fixed in a user's pocket or hand when being localized in real-life, in which case the gyroscope cannot be used to track it.

The benefits from *iTrack* over manual training are two-fold:

- A medium to long walk can cover a large number of points in one go, eliminating repeated requests to the user to mark her location on a UI.
- It is potentially less error-prone than manual training, since the accuracy of training coordinates does not depend on a user marking the location correctly on the UI.

#### *Time And Effort Advantage*

We performed a simple experiment to compare the time required to collect a single training walk in *iTrack* to collecting multiple points along the walk manually. The walk we performed took a total time of 41 seconds in *iTrack* — including hitting a "start" button on the phone UI, walking with the phone, and hitting a "stop" button to send back data after the walk. We found that WiFi

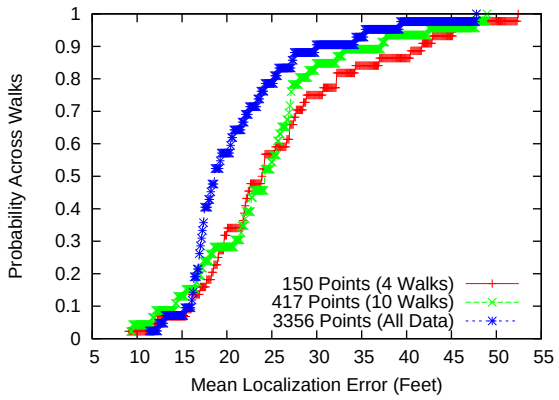



**Figure 5-16: More WiFi training data reduces localization error.**

scanning along the path collected 28 WiFi training points (this is smaller than the 41 second walk time because scans must be separated by more than one second to yield meaningful, non-cached results).

We also collected 28 training points along the walk using manual training. To do this, the user progressively advances to a new point on the walk by taking a stride or two. He/she then drags the blue circle shown in Figure 5-13(a) to the approximate position of the point manually, hits a "log" button and stops momentarily (to allow time for a WiFi scan to occur at that position). He/she then advances to the next point and repeats the procedure.

The total time for manual training on the walk we collected was 215 seconds, which is about 5.2× more than training with *iTrack*. The time savings are even more for longer walks. Moreover, the savings of 5.2× are just for the seed data, and do not even factor in *iTrack*'s powerful ability to *crowd-source* data in the background once seed data has been collected. If factoring in crowd-sourcing, *iTrack* would be orders of magnitude quicker than manual training and make it possible to collect more data than ever possible with manual training.

Collecting additional data is important, because can significantly improve the accuracy of WiFi localization. To illustrate this, Figure 5-16 shows the accuracy of WiFi localization (implemented using a RADAR-like approach [69]) for different amounts of training data collected on our test floorplan. We see that there is a significant reduction in both the median and higher quantiles of localization error as more WiFi training data is available to the system.

*Accuracy Advantage*

Both we and researchers on the OIL project at MIT [46] have found manual training to be error prone, in addition to being slow. The OIL project has even evolved techniques to mitigate some of this error. We could not evaluate this benefit quantitatively because we did not have access to a large corpus of manual training input from users.

### 5.8.7 How Much Seed Data Do We Need?

The discussion in Section 5.8.5 shows that *iTrack* can produce high quality trajectories for most input walks when seeded with a small amount of training data. We performed a simple experiment to quantify the impact of seed training data and to understand how much seed data is really required to achieve good mapping accuracy.

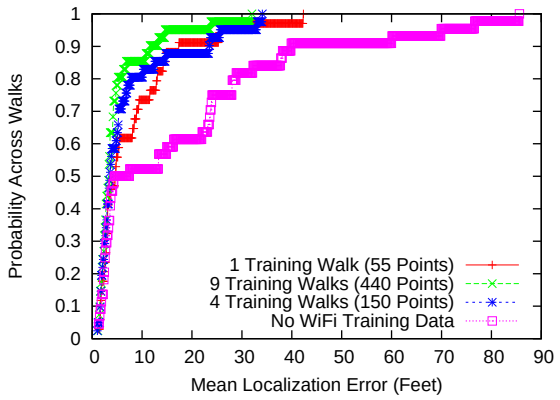



**Figure 5-17: Accuracy of *iTrack* for different amounts of seed data.**

Figure 5-17 shows the same CDF as shown in the absolute accuracy experiment in the previous subsection, but for varying amounts of seed WiFi training data — including the extreme of using no WiFi data at all and relying purely on inertial (accelerometer and gyroscope) data. As in the previous experiment, the seed data was drawn from one or more walks selected from the unlabeled pool of 97 walks.

The figure shows that it is essential to seed the algorithm with *some* WiFi data. Although the median error of *iTrack* without any seed WiFi data is close to the median error of the strategies with some seed WiFi data, the tail is considerably worse. The 90th percentile error without seed WiFi data can be as bad as 80 feet. Visual inspection also reveals that the output trajectories produced for these walks are completely or partially wrong.

We also see that the 80th and 90th percentile error drop dramatically even if adding seed WiFi data from just one walk (albeit one that covered a significant part of the floorplan). The errors drop further as more seed data is available. The graph indicates that 4-5 seed walks should be sufficient to ensure the median error is within 3.1 feet, which we believe to be good enough accuracy for many applications indoors (within room level or better). Using 9 training walks yields an improvement over 4 walks, but perhaps not enough to justify the extra manual training effort.

Table 5.2 shows the survival rate of the particle filter for the same scenarios shown in the CDF, as a function of amount of seed WiFi training data. Somewhat paradoxically, survival rate is highest when there is no seed WiFi data (88%). This can be explained by the fact that in the absence of any seed WiFi data, the system operates without any constraints whatsoever and so has greater leeway to find some particle that conforms to the measured shape, even if the shape has errors. Even one walk of training data significantly constraints the search space of the particle filter, and causes erroneous shapes to fail to survive the Monte Carlo simulation.

Interestingly, after the first walk, more training data improves the survival rate. We believe this is because more WiFi training data helps fix location more precisely, and allows more particles to explore a smaller area, making it more statistically likely that *some* particle survives the simulation.

The 4-5 number is obviously dependent on the particular floorplan we have evaluated on. This requirement is likely to vary depending on the floorplan geometry. It is likely to be more for a floorplan with larger area where 4-5 walks may not span all the major areas of a floor, and likely to be smaller in a smaller area where 2 or 3 walks may be enough.

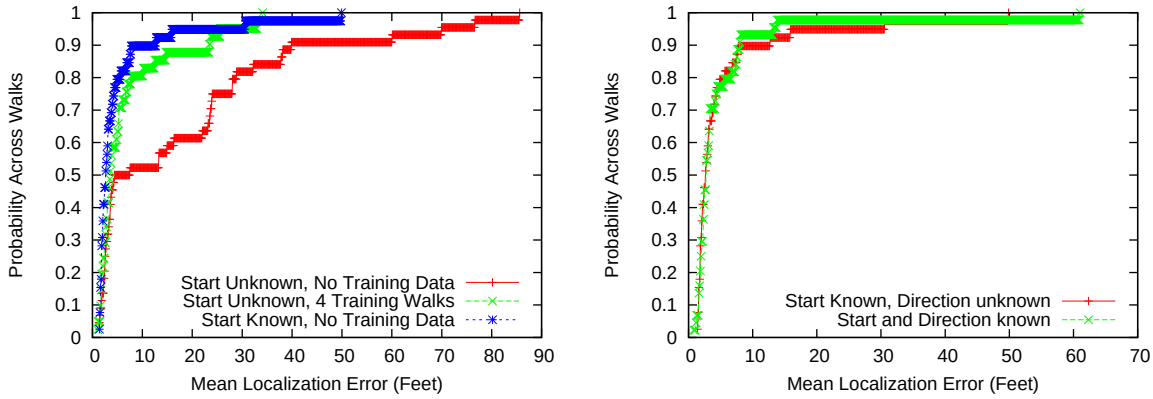
| Seed Training | Survival Rate |
|---|---|
| None | 88% |
| 1 Walk (55 Points) | 68% |
| 4 Walks (150 Points) | 82% |
| 9 Walks (440 Points) | 82% |

**Table 5.2: Survival rate of *iTrack* for different amounts of seed data.**

The next section measures the accuracy of *iTrack* when using our dedicated training procedure to collect the 4-5 seed WiFi walks.

### 5.8.8 How Accurate Is Dedicated Training?

As we have seen, *iTrack* requires a few seed walks to be collected as dedicated training data before it can crowd-source data without any user input. The walks used for dedicated training are all collected by asking the user to specify their approximate initial position. The system also infers approximate initial walking direction using the in-built phone compass as we have described earlier. Some of the initial positions may have human error but they are all approximately correct. In this experiment, the algorithm does not use any seed WiFi data in the training phase — though in practice, it could conceivably begin using its own WiFi data as soon as it matches a single walk.




(a) iTrack is very accurate if start position is known.

(b) Knowing initial direction helps a little.

**Figure 5-18: Knowing initial position and/or orientation improves accuracy.**

Figure 5-18(a) shows a CDF of the mean localization error over our test walks when the algorithm knows the approximate initial position. The graph also shows two of the CDFs presented earlier as references for comparison — CDF of error when running the algorithm both without knowledge of initial position and without seed WiFi data, and CDF of error when running the algorithm with seed WiFi data, but without knowledge of initial position.

The figure shows that that median, 80th and 90th percentile of localization error are all within 1 metre, comparable or smaller than the case where we use 4 seed WiFi training walks. This is because knowledge of the initial position fixes the approximate region of the floorplan where the user has walked and eliminates the structural ambiguity owing to the same shape being permissible in different parts of the floorplan.

We now look at whether knowledge of initial walking direction is beneficial. This requires a little extra effort because a trainer needs to consciously remember to keep the phone flat in his/her hand

| Algorithm Configuration | Survival Rate |
|---|---|
| Start Known, No Training | 78% |
| Start+Angle Known, No Training | 88% |

**Table 5.3: Survival rate of *iTrack* for different initialization configurations.**

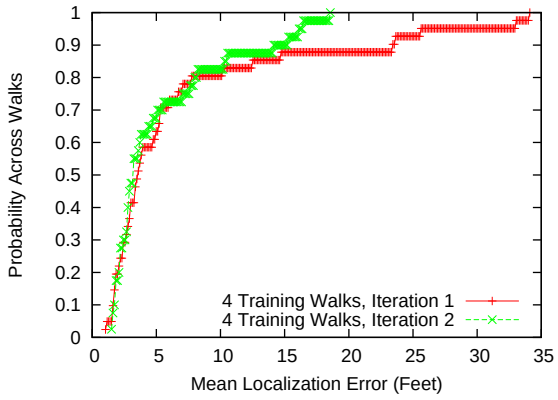



**Figure 5-19: Iterative training can improve accuracy of *iTrack*.**

towards the direction of walking while marking initial location on the map. Figure 5-18(b) shows a CDF of the mean localization error over the test walks with and without knowledge of the initial orientation. The orientation information helps overall. Interestingly, a close look at the graph will reveal that one of the walks in the tail is actually *hurt* by using initial orientation information: the magnetic compass happened to produce an incorrect heading on this walk, confusing the algorithm.

**Survival Rates.** Table 5.3 shows the survival rates with knowledge of initial position and orientation. This shows that with knowledge of initial position and orientation, the survival rate is high enough (close to 80%) to make training not too much a hassle. If the survival rate were lower, then collecting 4-5 valid seed walks would require walking much more than 4-5 times.

To summarize, the CDF of error shows that knowledge of initial orientation helps, but just the win in terms of lower error is not big enough in quantitative terms to justify complicating the training procedure. However, the improvement in survival rate is seen to be significant. Hence, we have adopted this optimization in our implementation of *iTrack*.

### 5.8.9 Impact Of Iteration

This section evaluates the benefit of using multiple passes of iteration to incorporate WiFi training data in the form of a feedback loop to improve the *iTrack* algorithm, as described in Section 5.6.

Figure 5-19 shows a CDF of the mean localization error across our test walks for the first pass and second pass of *iTrack*. The first pass uses only 4 seed WiFi walks to map the trajectory. The output of the first pass is incorporated into the training database and used in the second pass as feedback. The figure shows that while iteration does not improve the median accuracy by much, it helps significantly in the tail, helping correct some of the worst mistakes in mapping made by the first pass of *iTrack*.

The survival rate of the second pass was 80%, compared to a survival rate of 82% on the first iteration (one trajectory that survived the first pass — probably an incorrectly matched one — failed

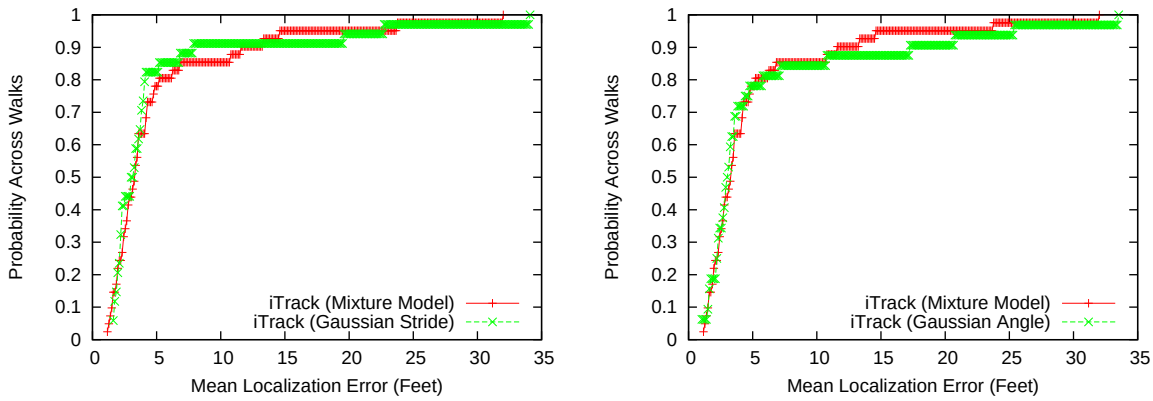
| Algorithm Configuration | Survival Rate |
|---|---|
| Regular *iTrack* | 82% |
| Gaussian Stride Model | 68% |
| Gaussian Angle Error Model | 64% |

**Table 5.4: Survival rate of *iTrack* for different error models.**

to survive the second pass). We did not find additional passes of the algorithm to yield substantial benefit.

### 5.8.10 Impact Of Non-Gaussian Models

In this section, we drill down into the choice of stride length and angle error models used in *iTrack*'s particle filter. We elaborate on one of the key contributions of *iTrack*, showing that the non-Gaussian models used by *iTrack* are a significant improvement over the Gaussian models used in previous work [67]. While the non-Gaussian models do not reduce the localization error of walks output by the system, they increase the survival rate and hence the recall of the system significantly, making it possible to map more walks correctly.




(a) Impact of stride length model.

(b) Impact of angle error model.

**Figure 5-20: Angle and stride length models do not affect localization error.**

Figure 5-20(a) shows a CDF of localization error using *iTrack*'s non-Gaussian stride model compared to a version of *iTrack* that uses a Gaussian stride model (assuming 9 training walks). Similarly, Figure 5-20(b) compares *iTrack*'s non-Gaussian angle model compared to a Gaussian model. The figures show that the choice of model has little or no influence on the localization error.

However, Table 5.4 shows that the choice of error model has a significant impact on the survival rate: the proportion of walks for which the particle filter finds one or more surviving valid particles. The non-Gaussian models we use both significantly increase the survival rate compared to a Gaussian model.

The explanation behind these results is that the walks that users perform tend to fall into one of two categories:

- Walks with little or no sensor error, whose true shape is almost exactly the observed shape from the gyroscope and whose true stride lengths match the average stride length of a human well (neither too slow nor fast).

| Technique | 99th Pctile. Error (metres) |
|---|---|
| WiFi Only | 18.9 |
| *iTrack*, Start Known | 12.5 |
| *iTrack*, 4 Seed Walks | 15.6 |
| *iTrack*, 9 Seed Walks | 12.9 |
| *iTrack*, 9 Seed Walks + Iteration | 9.6 |

**Table 5.5: Worst case (99th) percentile localization errors of different approaches.**

- Walks with significant gyroscope sensor error, or portions with slow or fast walking, and hence a sequence of strides much smaller or longer than average.

The walks in the first category are easy to match and the particle filter always manage to find the true match irrespective of the choice of proposal distribution used to explore its state space. Even if the choice of proposal distribution is Gaussian, the deviations of both angle and stride from mean are small enough that some subset of particle(s) in the Monte Carlo simulation experience these deviations, and converge to a valid answer without violating a constraint in the floorplan.

The walks in the second category are the ones that are problematic for a Gaussian error model. Consider a walk where the user walked a sequence of steps that are much shorter than her average stride length to traverse a corridor or other constrained region of the floorplan. A Gaussian model for length will require a huge number of particles (exponential in the deviation from the mean) to encounter this possibility in a Monte Carlo simulation, and hence will never be able to converge to a valid path without always violating a wall or other constraint. The non-Gaussian model we use *does* explore such sequences and can direct the exploration of the particle filter (via periodic resampling) to the more fruitful parts of the search space that have a chance of satisfying the constraint and staying within the corridor.

A similar logic applies to errors in turn angle.

### 5.8.11 Robustness In The 99th Percentile

To gain a better understanding of the worst case errors with *iTrack*, we compared the *99th percentile error* of localization estimates for different configurations of *iTrack*, as well as for trajectory mapping using simple WiFi localization alone. Since we have only 50 walks, the number we report is the 99th percentile of point localization error across all the raw data points from the walks (rather than across the averaged errors over entire walks).

Table 5.5 shows the results. We see that with sufficient seed data and if using the iterative training procedure (Section 5.6), the 99th percentile error is smaller than 10 metres. This shows that *iTrack* is pretty robust even in the worst case.

## 5.9 Related Work

Indoor positioning and tracking are widely explored problems, with a large number of previous solution efforts spanning specialized ultrasound-based tracking systems, wireless and cellular location systems, inertial sensors, laser range finders, and many more.

The most closely related previous work to *iTrack* is a class of algorithms for pedestrian tracking with foot-mounted inertial sensors, such as the NavShoe [29], FootSLAM [72] and work from

the University of Cambridge using an XSens Mtx inertial measurement unit [67]. Like *iTrack*, all of these techniques use accelerometer and gyroscope data in conjunction with particle filtering to determine a user's trajectory indoors. *iTrack* differs from this work in three main ways:

- First, these systems all use a specialized technique called zero-velocity updates (ZUPTs) to control the drift from inertial integration. This technique is peculiar to foot-mounted sensors which experience a point of zero velocity when a user's foot rests on the ground during her stride, and *is not applicable to mobile phones*. *iTrack* uses a different approach to dealing with drift based on step counting, and finding peaks and valleys in gyroscope data.
- Second, most of these previous techniques assume a Gaussian error model for stride length and angle in the particle filter they use. While this may work for specialized foot mounted IMUs like the Xsens Mtx, it does not work for a mobile phone jiggling around in a user's pocket. As we have shown, using a non-Gaussian model for stride length and angle error results in significantly improved recall for our particle filter.
- *iTrack* integrates WiFi localization information to improve its trajectory mapping accuracy. While some foot-tracking systems such as [67] have also proposed to use WiFi for initialization, it is not used throughout the mapping process, and in the iterative fashion that *iTrack* uses WiFi. As we have shown, a small amount of seed WiFi data is essential to achieving acceptable accuracy, and *iTrack*'s multi-pass iterative approach is effective at bootstrapping accurate indoor location from this data.

Inertial navigation itself is an old idea, having been used in robotics and in satellite tracking systems for well over four decades. For an excellent description of the core inertial navigation logic used in *iTrack* and a more rigorous derivation of the inertial navigation equations, see [26].

WiFi and cellular localization indoors is a well-studied topic. RADAR [69] was one of the pioneering systems in this field, proposing to locate a wireless receiver by finding the closest matching fingerprint(s) in a training database and computing their centroid. The Placelab project [57] used a similar approach indoors using a specialized GSM receiver than can receive *wide* GSM fingerprints. The Horus system [58] uses a probabilistic algorithm to improve the accuracy of WiFi localization compared to the RADAR or Placelab approaches. A number of commercial efforts and startups also exist offering indoor WiFi localization solutions including AlphaTrek [12], a commercial spin-off of Horus, and Ekahau [30]. *iTrack* achieves less a metre median tracking accuracy, similar to Horus [58], but with significantly simplified training compared to Horus.

*iTrack* falls into a class of indoor localization systems that aim to reduce training effort by using crowd-sourcing. Previous work in this area includes Redpin [70], Oil [46] and EZ [49]. Redpin and Oil both include user interfaces that allow a user to mark her location on an indoor map, and then scan for WiFi fingerprints to build a training database from user input. Oil includes an optimization to allow the user to indicate that he/she is static at a given location, so that it can obtain extended amounts of WiFi training data at the same location. This is a useful optimization, and we have automated it in *iTrack* by using the accelerometer to detect that a person is not moving.

The major difference between *iTrack* and previous WiFi crowd-sourcing systems such as Redpin and Oil is that *iTrack requires much less manual user input than these systems*. The training process on each floor requires a few controlled walks to collect seed training data, which can be done within 5-10 minutes. Thereafter, as we have shown, the system can crowd-source walks from users walking

around within the space automatically whenever the phone is in the appropriate position (hand or pocket), without any manual user input to indicate when they are walking, the starting point of the walk or any other information. This enables *iTrack* to gather a large amount of WiFi training data extremely quickly and easily compared to other crowd-sourcing systems.

The EZ crowd-sourcing system [49] also aims to crowd-source WiFi localization data from users walking around within a space without manual user input, and thereby alleviate the training problem for indoor localization. The system requires a small amount of seed training data in the form of GPS points obtained occasionally when a user walking the space is near a window or entrance to the building, and uses the rest of the (unlabeled) WiFi data only to establish constraints on wireless propagation. The system produces localization estimates that are less accurate than standard WiFi localization with more training data (e.g., RADAR or Horus), and as such, is likely to be less accurate for trajectory mapping than *iTrack*. Moreover, the availability of GPS locks indoors is rare. We have never been able to obtain a GPS lock inside our building, for example.

Ultrasound location systems such as Cricket [62], specialized radio ranging systems such as Active Bat [3] and laser range finder-based systems such as the MIT Intelligent Wheelchair [80] can measure the location of an object very precisely indoors, to within a few centimetres. The Cricket and Active Bat systems use the time difference of arrival between directed ultrasound or wireless signals to position the receiver, much as GPS works outdoors. Laser range finders use laser beams to measure their distance to the nearest obstacle in all directions, and use a particle filter to solve for the most likely trajectory of an object given a pre-built training map or a floorplan that is to scale. While all these systems are more accurate than (or comparable in accuracy to) *iTrack*, unlike *iTrack*, they require specialized, expensive hardware and do not work with commodity smartphones.

## 5.10 Conclusion

This chapter described *iTrack*, a system for easy and accurate trajectory mapping indoors where GPS does not work. *iTrack* uses a probabilistic particle filter model to fuse inertial sensors and WiFi on a mobile phone to localize a user's trajectory to within less than a metre indoors. *iTrack* is easy to train compared to previous WiFi localization systems, requiring a small amount of seed WiFi data for each floor of interest that can be collected within 5-10 minutes using a simple training procedure, assuming pre-existing availability of a digital floorplan. Once the seed WiFi data is available, *iTrack* uses a novel technique to crowd-source walks from users walking around in the space, *without any manual user input* unlike previous crowd-sourcing approaches. We have implemented *iTrack* on the iPhone 4 and found that it can map a user's walk to within a metre of accuracy, with minimal training effort.

The key novel technical contributions of *iTrack* we described were:

- The use of step-counting and peak finding to extract shape data without experiencing drift due to integration error.
- The use of non-Gaussian angle and stride length models to improve particle filter survival rate and hence recall, for constrained environments.
- The use of multiple passes of iteration by incorporating WiFi data from the first pass of running *iTrack* as training data for a second pass, to establish a positive feedback effect.

There is significant room for future work. The less than one metre tracking accuracy holds only for trajectories specifically known to be extracted when a user is walking steadily with her phone in her hand or pocket, or other similarly fixed configurations. *iTrack* is not currently robust to arbitrary phone use, or to real-world user movement with arbitrary stops and interruptions. This leaves open the following question for future work:

*Can we continuously localize a user accurately to within less than a metre or better indoors when the phone is in an arbitrary orientation, or not experiencing a steady walking motion?*

This is a challenging research problem because a phone experiencing typical use experiences complex movements — a user may flip it from hand to hand, play games with it, lift it from his pocket, or take a call. Future algorithms will need a generalized, robust way to detect and filter out, or even use inertial sensor data from these time periods without affecting localization accuracy.

Other important areas for future work include developing ways of running the particle filter on the phone, optimizing energy consumption and performance, and studying the accuracy of *iTrack*-like approaches for a wider variety of building layouts and WiFi access point densities.

# Chapter 6

# Conclusion

This dissertation has described three different, but closely related systems that can determine the trajectory of a mobile phone from sensor data:

- *VTrack*, which works over geographic coordinates extracted from sparsely sampled GPS and noisy WiFi localization.
- *CTrack*, which operates directly on cellular tower sightings and their signal strengths.
- *iTrack*, which fuses data from the accelerometer and gyroscope of a mobile phone with WiFi access point signatures indoors.

A common thread in all of these systems is the use of probabilistic models to extract *accurate information from inaccurate and/or infrequently sampled raw data*. In particular, the systems we have presented have used two kinds of (closely related) models:

- **Hidden Markov Models**, which model a trajectory as a sequence of hidden states and use dynamic programming to solve for the most likely sequence of states, and
- **Particle filters**, which also model a trajectory as a sequence of hidden states, but in a continuous state space. Since an exact solution is not tractable, unlike HMMs, they use Monte Carlo simulation to approximately solve for the most likely state sequence.

Probabilistic models have proved to be a powerful tool in all three of the systems and have been the key to recovering useful information from the raw data, which on its own is often wrong, or not accurate enough for the applications of interest.

We believe a "one-liner" take-away lesson from this dissertation research is that *a careful choice of underlying model is critical*: a machine learning tool such as a HMM or a particle filter is only as accurate as the underlying probabilistic model used in it. It is important to select the state space, observation space, emission probability distribution and transition probability distributions carefully to get these models to work in practice, and a simple or naive choice can often result in poor accuracy or even the model completely failing to work.

Examples of careful choice of model surface in all the systems we have presented in this dissertation, some of which we recap below:

- *VTrack*'s use of $\epsilon$-transition probabilities along all outgoing segments from an intersection, rather than the simple solution of assigning a probability of $\frac{1}{n}$ to each of $n$ outgoing segments. This choice was crucial to avoid biasing in favour of paths with fewer outgoing intersections, and was non-obvious before we discovered this bias.

- *CTrack*'s use of cell towers and their signal strengths as the observations of choice, rather than using geographic coordinates obtained from these fingerprints as the inputs to the Hidden Markov Model. As we showed, this choice reduced trajectory matching error by $2\times$.

- *iTrack*'s use of non-Gaussian models for stride length and angle error resulted in much higher survival rate and recall than using a Gaussian model, because a Gaussian proposal distribution could not model the common case where a *sequence of strides* are all shorter than average, or one user has shorter strides than another owing to height differences.

There are many more examples the reader can find if he/she reads this dissertation in more depth.

In each of these cases, rather than the algorithms used for the HMM or particle filter themselves being important, it was the *choice of model that was crucial.* The simple or straightforward choice of model resulted in poor accuracy in each of the three cases above.

## 6.1 Future Work

We conclude this dissertation with an outline of directions for future work.

### 6.1.1 VTrack and CTrack

We describe two directions that we think are interesting for future work:

- **Using Inertial Sensors:** Given that mobile smartphones on the market are increasingly equipped with accurate gyroscopes, it would be interesting to apply the lessons learned from *iTrack* to build an extremely accurate, yet energy-efficient outdoor tracking and navigation system. In particular, is it possible to integrate gyroscope data to obtain the approximate shape of a person's trajectory outdoors, and fuse with cellular localization, thereby significantly reducing the trajectory matching and point localization error of *CTrack* from the current levels of 75% and 45 metres respectively? We think this could be a good "fit of opposites" because cellular localization is good at a macro scale but poor at obtaining a trajectory right at a micro scale. In contrast, inertial sensors are good at a micro scale but not so good at localizing a device on a macro scale.

- **On-Phone Algorithms:** This dissertation has proposed map-matching algorithms that run on a server, mainly owing to computational constraints. Is it possible to come up with lightweight versions of the HMM algorithms that can be run continuously on a phone without hogging on-phone CPU and energy? Running map-matching on the phone would be useful for situations where there is no internet connectivity, or where the extra latency of communicating with the cloud is prohibitive, or where privacy is a significant concern.

### 6.1.2 iTrack and Indoor Localization

We believe that *iTrack*, for all its sophistication, barely scratches the surface of what is possible to make indoor localization more accurate and easy. We mentioned the following big technical challenge at the end of the last chapter:

*Can we use inertial sensors to continuously localize a user accurately to within less than a metre or better indoors when the phone is in an arbitrary orientation?*

The key challenge, as mentioned earlier, is to develop new robust algorithms to detect and filter out data from time periods where a user is using a phone unpredictably, such as flipping the phone around or taking a call.

*Given the increasing accuracy and sophistication of MEMS inertial sensor technology, we believe it is a question of when, not if this will happen.* Solving this research problem would enable much more fine-grained applications of indoor positioning at a wide scale simply not possible on mobile phones today — such as knowing where a consumer is in a store at rack level, where a tourist is in a museum at the level of an individual exhibit, or where a doctor is in a hospital at the level of a patient bed.

On that optimistic note, we conclude this dissertation.

# Bibliography


[1] The Mobile Millenium Project. http://traffic.berkeley.edu.

[2] A. Doucet, J. de Freitas, K. Murphy and S. Russel. Rao-Blackwellized Particle Filtering For Dynamic Bayesian Networks. In *UAI*, 2000.

[3] A. Harter, A. Hopper, P. Steggles, A. Ward and P. Webster. The Anatomy Of a Context-Aware Application. In *Wireless Networks*, 2002.

[4] A. J. Viterbi. Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm. In *IEEE Transactions on Information Theory*, 1967.

[5] A. Thiagarajan, J. Biagioni, T. Gerlich and J. Eriksson. Cooperative Transit Tracking Using GPS-enabled Smartphones. In *Sensys*, 2010.

[6] A. Thiagarajan, L. Ravindranath, H. Balakrishnan, S. Madden and L. Girod. Accurate, Low-Energy Trajectory Mapping For Mobile Devices. In *NSDI*, 2011.

[7] A. Thiagarajan, L. Ravindranath, K. LaCurts, S. Madden, H. Balakrishnan, S. Toledo and J. Eriksson. VTrack: Accurate, Energy-Aware Road Traffic Delay Estimation Using Mobile Phones. In *Sensys*, 2009.

[8] A. Varshavsky, E. de Lara, J. Hightower, A. LaMarca and V. Otsason. GSM Indoor Localization. *Pervasive Mobile Computing*, 3:698–720, December 2007.

[9] Traffic In the United States: ABC Survey. http://abcnews.go.com/Technology/Traffic/story?id=485098&page=1.

[10] American Community Survey. http://www.census.gov/newsroom/releases/archives/american_community_survey_acs/cb07-cn06.html.

[11] Analog Devices AD9864 Datasheet: GSM RF Front End and Digitizing Subsystem. http://www.analog.com/static/imported-files/data_sheets/AD9864.pdf.

[12] AlphaTrek Inc. http://www.alphatrek.com.

[13] Analog Devices, Inc. *ADXL330: Small, Low Power, 3-Axis +/-3 g iMEMS Accelerometer (Data Sheet)*, 2007. http://www.analog.com/static/imported-files/data_sheets/ADXL330.pdf.

[14] B. Hoh, M. Gruteser, H. Xiong and A. Alrabady. Enhancing Security and Privacy in Traffic-monitoring Systems. *IEEE Pervasive Computing*, 5(4):38–46, 2006.

[15] B. Hoh, M. Gruteser, H. Xiong and A. Alrabady. Preserving Privacy in GPS Traces via Uncertainty-Aware Path Cloaking. In *CCS*, 2007.

[16] B. Hoh, M. Gruteser, R. Herring, J. Ban, D. Work, J. Herrera, A. Bayen, M. Annavaram and Q. Jacobson. Virtual Trip Lines for Distributed Privacy-Preserving Traffic Monitoring. In *Mobisys*, 2008.

[17] B. Hull, V. Bychkovsky, Y. Zhang, K. Chen, M. Goraczko, A.K. Miu, E. Shih, H. Balakrishnan and S. Madden. CarTel: A Distributed Mobile Sensor Computing System. In *Sensys*, 2006.

[18] B. N. Waber, D. O. Olguin, T. Kim, A. Mohan, K. Ara and A. Pentland. Organizational Engineering Using Sociometric Badges. In *Netsci*, 2007.

[19] GPS and Mobile Handsets. `http://www.berginsight.com/ReportPDF/ProductSheet/bi-gps4-ps.pdf`.

[20] Bureau of Transportation Statistics. `http://www.bts.gov`.

[21] C.G. Claudel, A. Hofleitner, N. Mignerey and A.M. Bayen. Guaranteed Bounds on Highway Travel Times using Probe and Fixed Data. In *88th TRB Annual Meeting Compendium of Papers*, 2009.

[22] C.G. Claudel and A.M. Bayen. Guaranteed Bounds for Traffic Flow Parameters Estimation using Mixed Lagrangian-Eulerian Sensing. In *Allerton*, 2008.

[23] Event Handling Guide for iOS: Motion Events. `http://developer.apple.com/library/iOS/#documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/MotionEvents/MotionEvents.html#//apple_ref/doc/uid/TP40009541-CH4-SW1`.

[24] D. Pfoser, S. Brakatsoulas, P. Brosch, M. Umlauft, N. Tryfona and G. Tsironis. Dynamic Travel Time Provision for Road Networks. In *International Conference on Advances in Geographic Information Systems*, 2008.

[25] D. Sperling and D. Gordon. *Two Billion Cars: Driving Toward Sustainability*. Oxford University Press, 2009.

[26] D. Titterton and J. Weston. *Strapdown Inertial Navigation Technology*. 2005.

[27] Digifit. `http://www.digifit.com`.

[28] D.J. Turner, S. Savage and A.C. Snoeren. On the Empirical Performance of Self-Calibrating WiFi Location Systems. In *IEEE Conference on Local Computer Networks (LCN)*, 2011.

[29] E. Foxlin. Pedestrian Tracking with Shoe-mounted Inertial Sensors. In *IEEE Computer Graphics and Applications*, 2005.

[30] Ekahau Inc. `http://www.ekahau.com`.

[31] Euler Angles. `http://mathworld.wolfram.com/EulerAngles.html`.

[32] F.B. Abdesslem, A. Phillips and T. Henderson. Less is more: Energy-efficient Mobile Sensing with SenseLess. In *Mobiheld*, 2009.

[33] F.V. Diggelen. GPS Accuracy: Lies, Damn Lies, and Statistics. In *GPS World*, 1998.

[34] Getting a fix with your Garmin. `http://www.gpsinformation.org/dale/gpsfix.htm`.

[35] Information On Human Exposure To Radiofrequency Fields From Cellular and PCS Radio Transmitters. `http://www.fcc.gov/oet/rfsafety/cellpcs.html`.

[36] B. Hummel. Map Matching for Vehicle Guidance. In *Dynamic and Mobile GIS: Investigating Space and Time*, 2006.

[37] I. Constandache, I, S. Gaonkar, M. Sayler, R.R. Choudhury and L. Cox. EnLoc: Energy-Efficient Localization for Mobile Phones. In *INFOCOM*, 2009.

[38] I. Constandache, R.R. Choudhury and I. Rhee. CompAcc: Using Mobile Phone Compasses and Accelerometers for Localization. In *INFOCOM*, 2010.

[39] iCartel. `http://icartel.net/icartel-docs/`.

[40] Inrix Inc. `http://www.inrix.com/`.

[41] Apple Q&A On Location Data. `http://www.apple.com/pr/library/2011/04/27Apple-Q-A-on-Location-Data.html`.

[42] J. Eriksson, L. Girod, B. Hull, R. Newton, S. Madden and H. Balakrishnan. The Pothole Patrol: using a Mobile Sensor Network for Road Surface Monitoring. In *Mobisys*, 2008.

[43] J. Krumm. Inference Attacks on Location Tracks. In *Pervasive*, 2007.

[44] J. Krumm, J. Letchner and E. Horvitz. Map Matching with Travel Time Constraints. In *SAE World Congress*, 2007.

[45] J. Paek, J. Kim and R. Govindan. Energy-Efficient Rate-adaptive GPS-based Positioning for Smartphones. In *Mobisys*, 2010.

[46] J. Park, B. Charrow, J. Battat, D. Curtis, E. Minkov, J. Hicks, S. Teller and J. Ledlie. Growing an Organic Indoor Location System. In *Mobisys*, 2010.

[47] J. Proakis. *Digital Communications, 4th Edition*. McGraw Hill, 2001.

[48] J. Yoon, B. Noble and M. Liu. Surface Street Traffic Estimation. In *Mobisys*, 2007.

[49] K. Chintalapudi, A.P. Iyer and V. Padmanabhan. Indoor Localization Without the Pain. In *Mobicom*, 2010.

[50] K. Lin, A. Kansal, D. Lymberopoulos and F. Zhao. Energy-Accuracy Trade-off for Continuous Mobile Device Location. In *Mobisys*, 2010.

[51] L. Ravindranath, C. Newport, H. Balakrishnan and S. Madden. Improving Wireless Network Performance Using Sensor Hints. In *NSDI*, 2011.

[52] L. Sweeney. k-anonymity: A Model for Protecting Privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5), 2002.

[53] M. Gruteser and B. Hoh. On the Anonymity of Periodic Location Samples. In *Pervasive*, 2005.

[54] M. Gruteser and D. Grunwald. Anonymous Usage of Location-based Services through Spatial and Temporal Cloaking. In *Mobisys*, 2003.

[55] M. Mun, S. Reddy, K. Shilton, N. Yau, J. Burke, D. Estrin, M. Hansen, E. Howard and R. West. PEIR, The Personal Environmental Impact Report, as a Platform for Participatory Sensing Systems Research, 2009.

[56] M. Quigley, D. Stavens, A. Coates and S. Thrun. Sub-meter Indoor Localization in Unmodified Environments with Inexpensive Sensors. In *IEEE International Conference on Intelligent Robots and Systems*, 2010.

[57] M. Y. Chen, T. Sohn, D. Chmelev, D. Haehnel, J. Hightower, J. Hughes, A. Lamarca, F. Potter, I. Smith and A. Varshavsky. Practical Metropolitan-scale Positioning for GSM Phones. In *UbiComp*, 2006.

[58] M. Youssef and A. Agrawala. The Horus Location Determination System. *Wireless Networks*, 14:357–374, June 2008.

[59] MapMyRun. `http://www.mapmyrun.com`.

[60] M.B. Kjaergaard, J. Langdal, T. Godsk and T. Toftkjaer. EnTracked: Energy-Efficient Robust Position Tracking for Mobile Devices. In *Mobisys*, 2009.

[61] Meraki Inc. `http://www.meraki.com`.

[62] N. B. Priyantha, A. Chakraborthy and H. Balakrishnan. The Cricket Location-Support System. In *Mobicom*.

[63] N. Balasubramanian, A. Balasubramanian and A. Venkataramani. Energy Consumption in Mobile Phones: A Measurement Study and Implications for Network Applications. In *IMC*, 2009.

[64] N. Malviya, S. Madden and A. Bhattacharya. A Continuous Query System For Dynamic Route Planning. In *ICDE*, 2011.

[65] Navizon Inc. `http://www.navizon.com`.

[66] NAVTEQ Data. `http://navteq.com/about/data.html`.

[67] O. Woodman and R. Harle. Pedestrian Localisation For Indoor Environments. In *Ubicomp*, 2008.

[68] OpenStreetMap. `http://www.openstreetmap.org`.

[69] P. Bahl and V. Padmanabhan. RADAR: An In-Building RF-based User Location and Tracking System. In *INFOCOM*, 2000.

[70] P. Bolliger. Redpin: Adaptive, Zero-Configuration Indoor Localization Through User Collaboration. In *1st ACM International Workshop On Mobile Entity Localization and Tracking in GPSless Environments*, 2008.

[71] P. Mohan, V. Padmanabhan and R. Ramjee. Nericell: Rich Monitoring of Road and Traffic Conditions using Mobile Smartphones. In *Sensys*, 2008.

[72] P. Robertson, M. Angermann and B. Krach. Simultaneous Localization and Mapping for Pedestrians using only Foot-Mounted Inertial Sensors. In *Ubicomp*, 2009.

[73] PNI Corporation. *MicroMag3 3-Axis Magnetic Sensor Module*. `http://www.sparkfun.com/datasheets/Sensors/MicroMag3%20Data%20Sheet.pdf`.

[74] Reduced Cost and Complexity Expand Application Potential for Healthcare. `http://www.radianse.com/press-tipping-062304.html`.

[75] S. Reddy, M. Mun, J. Burke, D. Estrin, M. Hansen, and M. Srivastava. Using Mobile Phones to Determine Transportation Modes. *Transactions on Sensor Networks*, 6(2), 2010.

[76] S. Brakatsoulas, D. Pfoser, R. Salas and C. Wenk. On Map-Matching Vehicle Tracking Data. In *VLDB*, 2005.

[77] S. Gaonkar, J. Li, R.R. Choudhury, L. Cox and A. Schmidt. Micro-Blog: Sharing and Querying Content through Mobile Phones and Social Participation. In *Mobisys*, 2008.

[78] Skyhook. `http://www.skyhookwireless.com`.

[79] Telit GE865 Datasheet. `http://www.telit.com/module/infopool/download.php?id=1666`.

[80] The MIT Intelligent Wheelchair Project. `http://rvsn.csail.mit.edu/wheelchair`.

[81] Y. Freund and R.E. Schapire. A Decision Theoretic Generalization of Online Learning and an Application to Boosting. In *EuroCOLT*, 1995.

[82] Y. Wang, J. Lin, M. Annavaram, Q. Jacobson, J. Hong, B. Krishnamachari and N. Sadeh. A Framework of Energy Efficient Mobile Sensing for Automatic User State Recognition. In *Mobisys*, 2009.