Juha Vuokko

Accuracy in Tracking of Location of Mobile Device

Metropolia University of Applied Sciences

Bachelor of Engineering

Mobile Solutions

Bachelor's Thesis

27 October 2020

Author Title Number of Pages Date	Juha Vuokko Accuracy in Tracking of Location of Mobile Device 28 pages + 2 appendices 27 October 2020
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Professional Major	Mobile Solutions
Instructor	Patrick Ausderau, Senior Lecturer

When collecting locations to track walking path, collected data can be inaccurate. If tracking paths are an important feature of application, this reduces the sense of quality of the application. This thesis studies, how errors in tracked paths can be corrected.

In this thesis the accuracy of locations given by mobile device is studied as well as how this accuracy can be increased using filtering algorithms. An Android application using Kotlin is developed for this purpose. The application has a service that can run as foreground service collecting locations. The application has a live map that shows recent locations. These locations are stored in the SQLite-database using Room. After saving all the locations in the database, this data can be viewed. The map used is OpenStreetMap.

Four different algorithms are compared. Three of these can be manually tuned by changing variables. The algorithms include Kalman filter, Ramer-Douglas-Peucker, removal of most inaccurate locations and running average. The algorithms are compared visually to find the best solution used in walking route mapping for different projects. The algorithms can be compared for one route and they can be drawn in a map together with accuracies and bearings.

The solution does not need to be complicated, an algorithm that removes the most inaccurate points works with random errors in pathway that are not too plentiful. If the locations are collected frequently, the running average can smooth the path. This algorithm could be used even when collecting the locations and save just the averages to the database. This could be developed in a later project.

Keywords	Android, Kotlin, GPS, map application

Tekijä Otsikko Sivumäärä Aika	Juha Vuokko Mobiililaitteille tallennettujen sijaintitietojen tarkkuus 28 sivua + 2 liitettä 27.10.2020
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Mobile Solutions
Ohjaaja	Lehtori Patrick Ausderau

Kerättäessä mobiililaitteella sijaintitietoja kulkureitiltä kerätyissä sijainneissa esiintyy epätarkkuuksia. Jos kulkureitin tallentaminen on olennainen osa sovelluksen toimintaa, liian epämääräisesti tallentunut reitti laskee sovelluksen laadun tuntua huomattavasti. Insinöörityössä tutkittiin, miten sijaintitiedoista muodostetun reitin oikeellisuuteen voi vaikuttaa.

Työssä selvitettiin mobiililaitteiden tallentamien sijaintitietojen tarkkuutta ja sitä, miten tätä tarkkuutta voidaan parantaa käyttämällä suodatusalgoritmeja. Tätä varten kehitettiin Android-laitteelle sovellus käyttäen ohjelmointikielenä Kotlinia. Sovellus käyttää taustapalvelua, jonka tarkoitus on kerätä sijaintitiedot. Kerätyt sijaintitiedot tallennetaan SQLite-tietokantaan käyttäen Room-tietokantakirjastoa. Tietokantaan kerättyjä sijainteja voi tarkastella omassa karttanäkymässään. Karttapohjana toimii OpenStreetMap.

Työssä vertailtiin neljää eri algoritmia, joista kolmea voi hienosäätää sovelluksessa. Käytetyt algoritmit olivat Ramer-Douglas-Peucker, Kalman-suodin, epätarkimpien sijaintien poistaminen ja juokseva keskiarvo. Näitä algoritmeja vertailtiin sovelluksessa visuaalisesti, jotta saatiin selville, millä algoritmilla reitti näyttää parhaiten noudattavan todellista reittiä. Samassa karttanäkymässä voidaan esittää niin alkuperäinen reitti kuin myös valituilla algoritmeilla muokatut reitit. Tässä näkymässä voidaan esittää sijaintien tarkkuus ja niiden yhteydessä tallennettu kompassisuunta.

Vertailussa ilmeni, että sijaintitietojen tarkkuuta saadaan parannettua käyttämällä algoritmeja. Kaikissa algoritmeissa on puolensa. Epätarkimpien sijaintien poistaminen voi riittää hyvien tuloksien saamiseen. Jatkossa täytyy tutkia vielä juoksevan keskiarvon käyttöä jo sijaintitietojen keräysvaiheessa.

Avainsanat	Android, Kotlin, GPS, karttasovellus

Contents

List of Abbreviations

A-GPS	Assisted Global Positioning System
API	Application Programming Interface
BDS	BeiDou Navigation Satellite System
GLONASS	Global Navigation System (Russian)
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
CSV	Comma Separated Values
HAL	Hardware Abstraction Layer
Json	JavaScript Object Notation
KTX	Kotlin extensions
MSA	Mobile Station Assisted
NPS	Network Positioning System
SoC	System on chip
USB	Universal Serial Bus
WLAN	Wireless Local Area Network

1 Introduction

When tracking a walking path with mobile device, the accuracy of obtained coordinates varies as demonstrated in figure 1. The drawn paths do not always follow the road that was walked on. The goal of this thesis is to study the accuracy of the locations and how to improve it. One way of improving the accuracy is filtering the most inaccurate Global Positioning system (GPS)-locations away. The focus of this thesis will be in finding and comparing filtering algorithms. Other methods than filtering to adjust the pathway do exist. With some algorithm new estimates of locations are calculated using measured locations together with additional information such as speed and bearings.

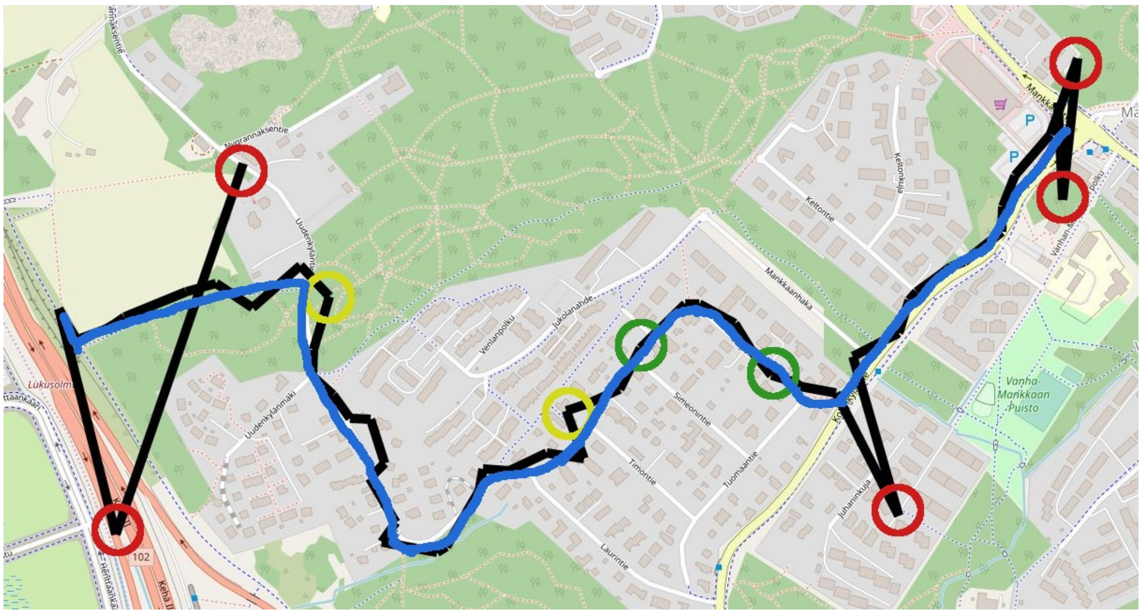


Figure 1. Map base is OpenStreetMap. The blue drawn line represents the actual path taken while the black path is obtained by GPS. The red circles show the worst deviations from path, the yellow ones show slight deviations while the green one show locations accurate enough.

In an application specially made for tracking path walked, sudden peaks in tracked path are not tolerable. One solution could be to let the user manually trim the walking route. This is also completely doable and possible solution to the problem. The possibility to modify walking route leads to problems too. This can be considered as extra inconvenience, and this makes it possible for user to change the route in a way that it does not correspond the actual route. That is why different kind of solutions are tested and the different algorithms for correcting inaccuracies in path are compared. This makes it possible to develop better applications using location data.

To study the accuracy of tracking, the application to Android devices is made using Kotlin. In this application one can track walking routes and different correction algorithms can be studied visually. The earlier application was used as basis in designing and implementing the new application. The basic concepts of how to record locations and how to store them into database are the same, but the layouts and functionalities differ. The application uses local database and all calculations are conducted locally. This removes the need for an external server and makes the application easier to test while moving outdoors. For user of applications the local database is a better option, because in that way the user's private information stays local and under user's control.

Programming is conducted using Android Studio as integrated development environment. Testing of code is carried out both with emulator and real Android device. Location data will be gathered by using the program in real device while walking or cycling. Git is used for version control. The code is stored in GitHub repository[1]. As this is a personal project there is no company made rules about what can be published and what not and that is the reason GitHub repository is made public.

[1] https://github.com/juhavuo/TrackingAccuracy

2 Technologies

2.1 Android and Kotlin

Android is an open source operating system mainly for mobile devices. It is owned by Open Handset Alliance and Google has a major role in its development. [1] Android Inc was founded in 2003 in Palo Alto, California and it was originally going to work with operating systems of digital cameras. Soon they moved to developing operating systems to mobile phones, because the market for digital cameras was declining. Google bought Android in year 2005. The first Android phone was launched in 2008. It was T-Mobile G1 with a physical keyboard. [2]

The Android architecture is shown in schematics in figure 2. It displays selected details of Android software stack. Android has Linux Kernel in its core that has power management and drivers such as driver for Bluetooth and camera. Linux kernel takes care of security features, process management, memory management and multitasking. [3] This Linux Kernel is heavily modified by Google, system on chip (SoC) manufacturer and manufacturer of Android device. The basis Linux kernel is not typically getting any updates and it stays the same from the start of the development of new device. This means that a new Android device can have a two-year old Linux kernel, because all the modifications made by manufacturers the different devices have different kind of kernels. [4]

Hardware Abstraction Layer (HAL) handles communication between software and hardware. In different devices different setups of hardware made by various companies can be found. In HAL there are interfaces for these different components such as camera, Universal Serial Bus (USB) and Bluetooth. Vendors can design and implement drivers and HAL just as they like as long as the interface is following specifications. [5]

In native C/C++ libraries there are libraries such as SQLite, that is responsible for database. Because the SQL database is readily available the application uses SQLite with Room Library. Other examples of these native libraries are WebKit for browser support and FreeType for font support. [6] Most of these native libraries are open source libraries. [2]

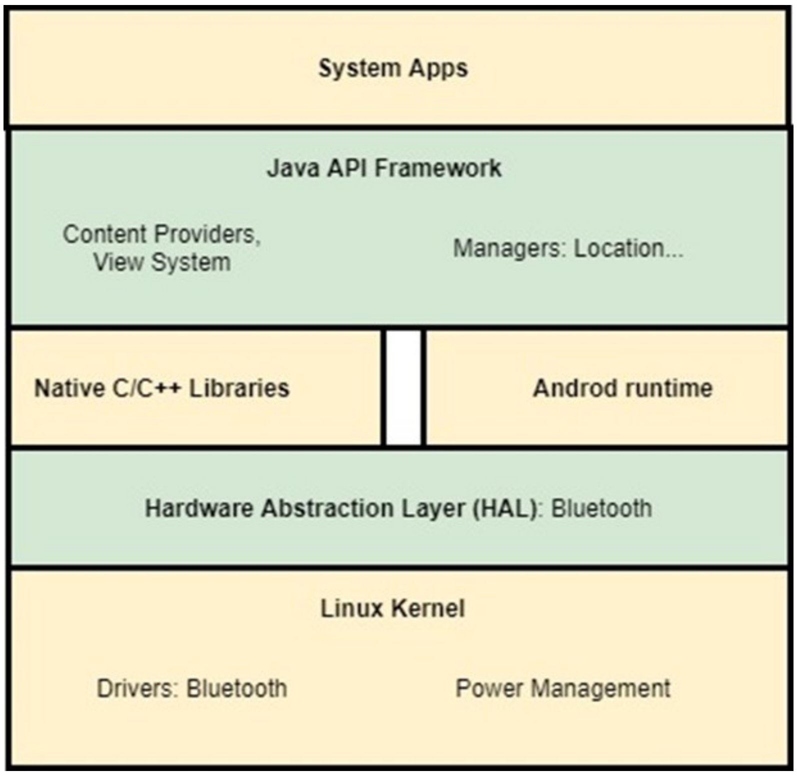


Figure 2. Android Software Stack[8]

Java API framework is what is used to build Java applications. It is written in Java. This framework includes View System that can be used to build user interfaces, Resource Manager for non-code resources such as layout files, Notification Manager for alerts is shown in status bar. It also includes Activity Manager and Content Providers. Activity manager starts the application when an icon of that application is clicked. It takes care of activity lifecycle management, activity stacks and creating of threads for activities. [7] Content Providers can be used to share data between applications. System apps are core apps that are included in Android. These are no different than applications downloaded by user of Android device. These applications can be used separately but they can be utilised as providers in an application to be developed. [8]

The application to be developed for this thesis is going to use AndroidX namespace. This removes problems with updating libraries. In earlier projects updating one library typically leads to errors and a situation, where the application would not load, because all Support Libraries were not using the same version. AndroidX packages can be updated independently without worries. AndroidX-packages needed for project are appcompat, core, recyclerview, preference and libraries for possible unit testing.[9]

Kotlin as a programming language dates back to year 2011. First it was used only by the company that developed it, JetBrains. Kotlin was published a year later with Apache open source license. Kotlin become an officially supported language for Android in 2017. [10] In May 2019 Google announced that Kotlin will become the primary language for Android instead of Java. It means that many new features for Android come to Kotlin first and it is also recommended to use Kotlin in an Android project. Kotlin was already popular at that time: more than 50 percent of professional developers make Android applications using Kotlin. [11]

Reason for JetBrains starting to develop Kotlin was that they needed a more modern language that was completely compatible with Java. This compatibility with Java was important, because they have old Java projects that they wanted to continue. Other requirements for the language besides interoperability with Java was that it must be fast, and its code compact and expressive. [10] This interoperability means that Java code can be called from Kotlin, so Java libraries can be used directly with Kotlin, like ArrayList from java.util. Kotlin can use its own types instead of types of Java. These are mapped to Kotlin types. These include Object in Java that corresponds to Any in Kotlin. Instead of java.lang.String Kotlin has own kotlin.String. Kotlin differs from Java in that it does not use primitive types directly as Java does, but as objects. This means that all Java types map to Kotlin types such as int to kotlin.Int. [12]

Because Kotlin is the primary language for Android, Kotlin extensions (KTX) is included in Android libraries. This affects the existing Android Application programming interfaces (API) by bringing the Kotlin features such as extension functions and lambdas to these APIs. Android KTX has multiple packages. Examples of these are Fragment KTX and Collections KTX. [13]

2.2 Global Positioning System

GPS system has at least 24 available satellites orbiting round the earth 20000 kilometers above the sea level. They all have synchronized atomic clocks and they are sending information that is read by receivers. A receiver also gets the synchronized time readings. The location coordinates are obtained by calculating how much time it took for radio waves to travel between satellites and receiver. As radio waves travel at speed of light the distance can be calculated. One needs distance measurements from multiple satellites to be able to measure location precisely. [14] Satellites send new time stamp signal in 30 second intervals [15].

The altitude measured by GPS system is not from ocean level but reference ellipsoid that is approximation of Earth's surface. The difference between these altitudes can be tens of meters. It depends on the model, what value of altitude the GPS receivers are given. Some models directly give the GPS altitude, some calculate the estimate of altitude from ocean surface. [16]

Scientists at Johns Hopkins University were able to track satellite Sputnik's path by studying signal it sent in 1958. This led to the idea of using a satellite to track down their own position. The first system of this kind was Transit that was operational in 1964. It was able to track position in two dimensions and with 25-meter accuracy. Before that the project for tracking position in three dimensions was studied. This was under project name 621B. After series of tests and studies the first operational GPS satellite was launched in 1978. [17] During this study period they had several problems to solve. Atomic clocks had been available since 1950s, but they were not directly suitable for conditions in satellites orbiting the Earth. The problematic issues were radiation and changes of temperature. Furthermore, the locations of satellites in orbit must be able to predict for times they are out of sight of upload stations in United States. Multiple phenomena must be considered, such as general and special relativity, solar radiation and Earth tides, when satellites are orbiting the Earth. One thing to take into consideration was also the durability of satellites. At all the times there were to be 24 operational satellites. The satellites were designed in a way that they would be operational for as long as possible. The average age reached by first 10 satellites was 7.6 years. [18]

A typical Android device uses Assisted Global Positioning System (A-GPS) as a receiver for radio waves sent by Global Navigation Satellite System (GNSS) satellites. Different GNSS systems other than GPS exist. These systems include Galileo from Europe, Global Navigation Satellite system (Глобальная навигационная спутниковая система, GLONASS) from Russia and BeiDou Navigation Satellite System (北斗卫星导航系统, BDS) from China [19]. A-GPS is not only relying on GPS satellites, it also uses cellular location data. The location can be calculated by triangulating using locations of three or more cell towers. Using solely GPS in continuous location tracking takes a lot of battery power and when GPS data is used, it can take up to one minute to get the location. GPS receiver uses cellular location data as raw location and finetunes that with position information obtained from satellites. [20] An Android device not only gets the location data from A-GPS but also from Network Positioning System (NPS). NPS uses signal strengths of transmitters in nearby area whether it is wireless local area network (WLAN) or cellular. [21]

A-GPS can be Mobile Station Based. In this case data is obtained from a remote server such as time and course location and this data is used together with GPS signal to calculate the position. Another possibility is Mobile Station Assisted (MSA) where a mobile device sends data obtained from GPS satellite to a server and the server sends back the calculated location information. [21] Android devices contain a built-in GPS chip that does the communications with the satellites. This chip is controlled by GPS driver. [22]

GPS locations can be far off for multiple reasons. The accuracy of GPS is depending on how many satellites is within reach of GPS receiver. Pure GPS receiver needs at least three satellite readings to be able to give a location. It takes seven or eight satellites to get accuracies of 10 meters, if the amount is less than this, accuracy is poorer. Different kind of obstacles can prevent for satellite signals to reach the receiver. These can be buildings, trees or even clothing. To get good accuracy GPS receiver needs open skies above it. Signals can be bounced by walls of tall buildings. This can be a source of inaccuracy. [23] Movement speed of receiver affects the accuracy of GPS measurements. When moving faster, the receiver has to make new connections to satellites, and this causes jumpiness in locations [15]. To get altitude readings from GPS, data at least from four different satellites is needed and for good accuracy one of them needs to be above the receiver, so altitude readings are typically inaccurate without any alternative source of data. [24]

2.3 Ramer-Douglas-Peucker and Kalman filter algorithms

With Ramer-Douglas-Peucker algorithm paths can be simplified and clean noise out of them. Algorithm is explained in figure 3. First the end points are selected, then a point farthest away from the line connecting these points is searched. These end points are always kept. The distance from this point to the line connecting start and end points is compared to epsilon value, which is a threshold distance that can be given freely. If it is larger, the new segments are created, from that point to start point and to the end point. In second part of figure 3, the new line from point to start point is shown. Again, the distance from line is compared, but this time it is shorter. This means that all points between these points are removed as is shown in third part. This iterative process is repeated until there are no points to select between the endpoints. This process is done to whole route. [25]

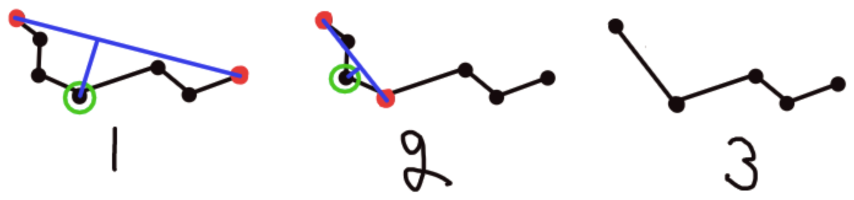


Figure 3. Ramen-Douglas-Pecker explanation

Kalman filter can be used in systems with noise. Using it is iterative process. It has two stages. They are prediction and update. Using Kalman filter starts with building a model. From the initial estimate prediction is calculated, from prediction the correction and after that it just loops between prediction and correction. General form of Kalman equations are presented in formulas 3 and 4.

$$x_k = Ax_{k-1} + Bu_k + w_{k-1} \tag{3}$$

$$z_k = Hx_k + v_k \tag{4}$$

In equations 3 and 4 A, B and H are typically matrices, but they also can reduce to single number. x is marking the signal values, u is for control signal, which can be nonextinct, w is noise as well as v. [26]

3 Practical work

3.1 Application basics

Practical work is started with making the basis of measuring application. At first locations must be able to be recorded and stored to database. Locations and their inaccuracies must also be able to show in a view. In this stage it would be preferable if numerical data of locations and their accuracies could be transferred from mobile device to laptop to further analysis.

When recording walking route, it would be preferable to put the mobile phone to pocket and focus on walking or jogging rather than carry the device in hand all the time. On the other hand, it would be interesting to see mapping process live while walking. There can even be a difference in accuracy of measured locations whether the mobile device is kept in pocket or it is kept in hand out in the open. To get most accurate location measurements, device must have open sky above it.

To be able to track locations, when device is in pocket, service running in the foreground must be used. Service must bind to activity in activity's `onStart`()-method and unbind in activity's `onStop()`-method to make the live view work. `onStart()`-method is called, when activity becomes visible to the user. From this point activity needs to get updates from the service to draw the locations to map. `onStop()` is being called, when activity is no longer visible to the user: At this stage activity needs no more updates from service, so service can unbind.

The problem that needs to be solved is how location data can be sent from service to the activity. Service is bound to activity when activity is visible. For at least the first version uses interface in service class called `CallbackForService` and `LiveMappingActivity` implements it. Service takes reference of that activity class in a nullable variable, when activity loads and makes it null, when activity goes to `onStop()`. This prevents from using the activity variable when service is not bound to that activity. When service is not bound, it means that activity is not visible and there might not be that activity class running. Using reference to that activity, when it is not running, could crash the application.

In the live map only raw locations are shown without any additional information such as accuracies and bearings. No algorithms are used to location data at this stage. In the live map recorded locations can be shown using markers, which is readily usable class in osmdroid-package. Markers can have icon set by user. In map view, in which the gathered data is compared to location calculated using filtering algorithms, the routes are shown using polyline. In polyline the locations of the route are connected with lines. Accuracies of measured values can be shown using polygons. In the polygon class there is a static method `pointsAsCircle()` that takes center of geopoint and accuracy in meters as parameters and gives circle's circumference as list of geopoints. The polygon can be constructed from these geopoints and drawn on the map.

Bearings can be shown in a map too. A bearing is a direction of traveling and it does not depend on orientation of device. It is an angle between 0° and 360°, where north is zero and east is 90°. These bearings can be drawn by using polylines with two locations. The one is the measured GPS location and the other one must be calculated. It can be calculated in the same way as cartesian coordinates are calculated from polar coordinates. The difference in rotation directions and the zero points must be taken into consideration.

3.2 Classes and activities used in the project

All activities of the project application are shown in figure 4. `MainActivity` has simple layout. Buttons are main component of layout. By pressing new route button, `Start-MapingActivity` opens, where name and description to the route must be added, before pressing start-button. Pressing start button leads to `TrackingMapActivity`. Service for location tracking starts immediately and binds to that activity. Pressing back-button leads to `MainActivity`. This is done by overriding function `onBackPressed()`. If service is still running, first button shows text return to live map and pressing that leads straight to `TrackingMapActivity`. This is possible, because in `LocationService`-class companion object contains a Boolean variable `isServiceStarted`. This variable can be read all the time from the `MainActivity` and it describes the state of the service, is it running or not.

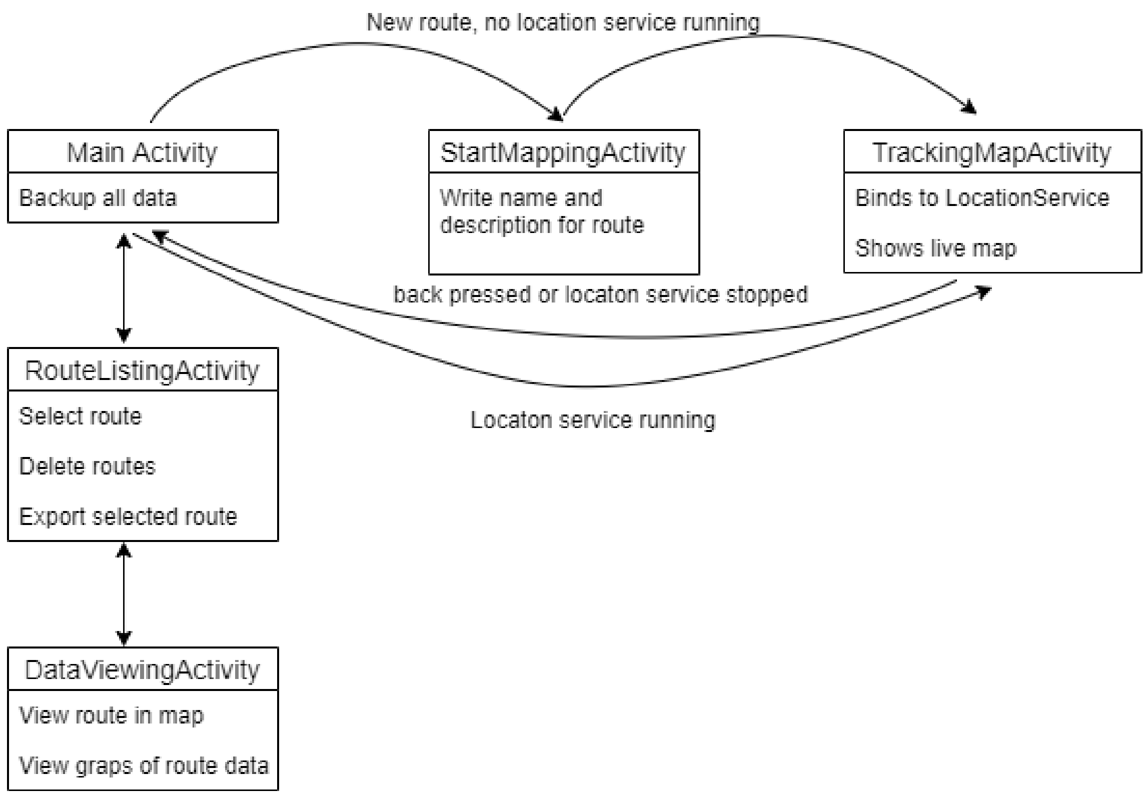


Figure 4. Activities of application and their relations

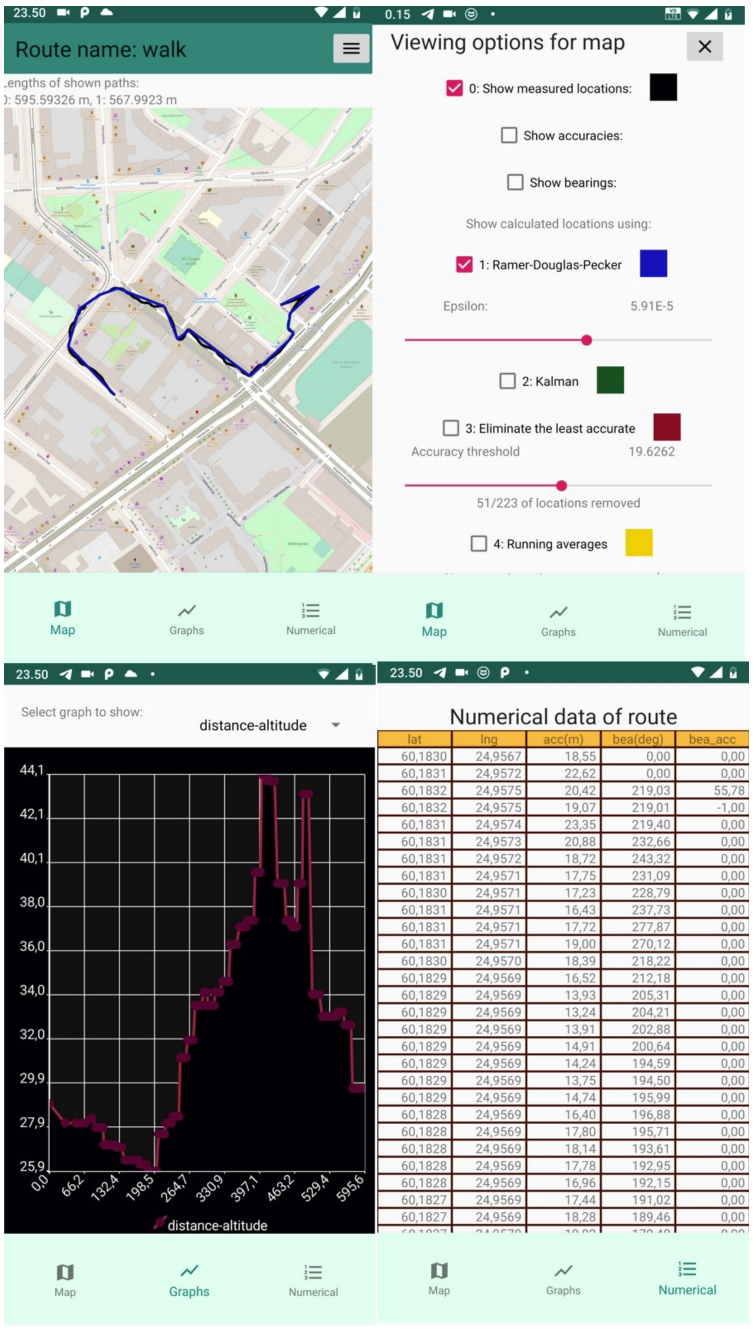
From `MainActivity` can be accessed to activity that lists tracked routes. The list is shown using `RecyclerView`. This `RecyclerView` needs separate `Adapter`-class, where the logic of view is placed. Order of the routes can be selected by choosing sort by date or sort by name from drop down list. This sorting can be accomplished by using Kotlin's `sortWith()`-function for collections. This `sortWith()`-function can take comparator as parameter. Objects can be sorted out using this comparator. It can be given, by which objects field the comparison is done. An example of this is shown in code of listing 1. In function `organizeByName()` the names are compared as lower case strings and the compared feature is, which string comes first in alphabetic order.

```
fun organizeByName(alphabetic: Boolean){
        items.sortWith(kotlin.Comparator<Route>{ p0,p1->
            when{
                p0.name.toLowerCase(Locale.ROOT) > p1.name.toLowerCase(
                Locale.ROOT) -> 1
                p0.name.toLowerCase(Locale.ROOT) == p1.name.toLowerCase(
                Locale.ROOT) -> 0
                else -> -1
            }
        })
        if(!alphabetic)
            items.reverse()
 }
```

Listing 1. The sorting algorithm for ordering routes with their names either alphabetically or in reversed order depending on Boolean parameter `reverseAlpabetic`.

Tapping of one route in list leads to `DataViewingActivity`, that uses fragments. The screenshots of these fragments are shown in figure 5. The first is `MapFragment`, where data of route can be viewed in map. In this fragment the lengths of all routes, the measured one as well as calculated ones, are shown. The length of the route is obtained by calculating all the distances between adjacent locations and summing these distances. The distance between two locations can be obtained by using `distanceBetween()`-function in Android's Location class. It takes latitudes and longitudes of two locations without need to use Location objects themselves.

The second one is `GraphFragment`, where data is shown in different graphs. The graph to show in view can be selected from drop down list. `GraphFragment` currently contains three different graphs to choose from. These graphs are made from measured location data. These are unfiltered and are shown just as they are with all the uncertainties. The graphs are distance-time, speed-time and altitude-distance graphs. These graphs are made using Androidplot library, which is under Apache 2.0 license.



Lengths of shown paths:
0: 595.59326 m, 1: 567.9923 m

Viewing options for map

[✓] 0: Show measured locations: ⬛

☐ Show accuracies:

☐ Show bearings:

Show calculated locations using:

[✓] 1: Ramer-Douglas-Pecker 🟦

Epsilon: 5.91E-5

☐ 2: Kalman 🟩

☐ 3: Eliminate the least accurate 🟥

Accuracy threshold 19.6262

51/223 of locations removed

☐ 4: Running averages 🟨

Select graph to show: distance-altitude

Numerical data of route

lat	lng	acc(m)	bea(deg)	bea_acc
60,1830	24,9567	18,55	0,00	0,00
60,1831	24,9572	22,62	0,00	0,00
60,1832	24,9575	20,42	219,03	55,78
60,1832	24,9575	19,07	219,01	-1,00
60,1831	24,9574	23,35	219,40	0,00
60,1831	24,9573	20,88	232,66	0,00
60,1831	24,9572	18,72	243,32	0,00
60,1831	24,9571	17,75	231,09	0,00
60,1830	24,9571	17,23	228,79	0,00
60,1831	24,9571	16,43	237,73	0,00
60,1831	24,9571	17,72	277,87	0,00
60,1831	24,9571	19,00	270,12	0,00
60,1830	24,9570	18,39	218,22	0,00
60,1829	24,9569	16,52	212,18	0,00
60,1829	24,9569	13,93	205,31	0,00
60,1829	24,9569	13,24	204,21	0,00
60,1829	24,9569	13,91	202,88	0,00
60,1829	24,9569	14,91	200,64	0,00
60,1829	24,9569	14,24	194,59	0,00
60,1829	24,9569	13,75	194,50	0,00
60,1829	24,9569	14,74	195,99	0,00
60,1828	24,9569	16,40	196,88	0,00
60,1828	24,9569	17,80	195,71	0,00
60,1828	24,9569	18,14	193,61	0,00
60,1828	24,9569	17,78	192,95	0,00
60,1828	24,9569	16,96	192,15	0,00
60,1827	24,9569	17,44	191,02	0,00
60,1827	24,9569	18,28	189,46	0,00

Figure 5. Screenshots of fragments of DataViewingActivity

The third one is `NumericalFragment`, that shows data in numerical format. This currently lists the main info of measured locations, which entails bearings and accuracies related to the coordinates. Fragment uses `RecyclerView` and the rows are divided to equally wide portions to simulate data sheet. This is not suitable as universal solution, because the cell size depends on the dimensions of Android device used. For a more general solution cells with predefined widths need to be inside scroll view. This way the widths stay always as intended.

The fourth fragment, `MenuFragment`, is related to `MapFragment`. It shows the menu of viewing options for the map. Here one can select, which calculated routes are shown on a map together with a measured route as well as what kind of other information is shown on the map, such as accuracies and bearings related to measured locations. The viewing references are stored in shared preferences. Class `MapPreferencesHandler` is handling the shared preferences. It stores and fetches the values of the preferences. The algorithms for removing the inaccuracies in locations is in a separate class named `DataAnalyzer`.

3.3 Database structure

Location data is stored locally in SQL-database using Room. Database has currently two tables. These tables are shown in figure 6. One lists all routes and the other lists measured locations of these routes. From the locations table all the locations of one route can be queried by using id of that route. If additional information such as heart rate can be obtained, then it will be stored in a separate table.

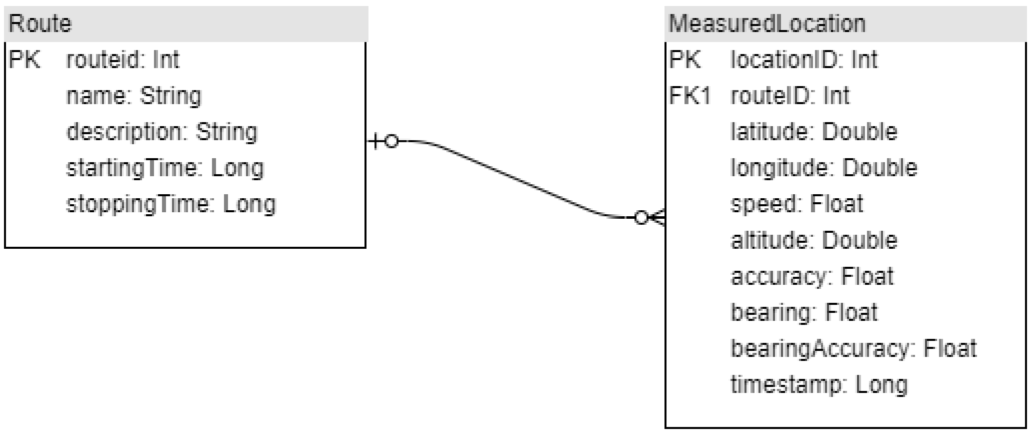


Figure 6. Room database structure

The Route table entries has stored ID of the route, name and description given to the route and starting and stopping times. These times are taken from devices own time. `MeasuredLocation` table entries have location IDs, latitudes, longitudes, timestamps related to location data and ID of route as foreign key. Database contains additional data such as accuracy, altitude, speed, bearing and accuracy related to bearing. The altitude that GPS gives is not measured from sea level but is from refence ellipsoid. At first this value is used directly. In later stages this can change.

3.4 Working with asynchronous data

Loading data from database takes some time. The exact time, when the data is ready to use, is not certain beforehand. This leads to problems, when an activity uses fragments and the first fragment needs to show that data from database immediately it is loaded. This happens with `DataViewingActivity` and `MapFragment`. The ID of tapped route is passed with intent from `RoutesListingActivity` to `DataViewingActivity` and with this ID all related measured locations are fetched from database. These locations are needed in `DataAnalyzer`-class to use different filters with this data to obtain more accurate route. The original measured locations and results of calculations are shown in `MapFragment` as soon as everything is ready. The fragment life cycle related to activities life cycle can cause issues. The components of fragment can be used after its `onCreateView()`-method. Calling the component before this leads to crash.

Operations that may take long time such as database operations must be done in separate thread than main thread. That is because blocking main thread that handles user interface would lead to unresponsiveness of application. Multithreading is carried out by writing the code inside Thread objects. It must be noticed that code immediately after the launch of second thread can happen before all code from that started thread is executed. If in that other thread the biggest route ID is fetched and to this id one is added to get the ID of new route, these operations are not executed, when the code in main thread, that comes right after, is executed. If this id is put in intent and other activity is launched, the id is not going to be the right one and this leads to problems, when that ID is used in database operations. Solution is simply to put the sending of that intent to database handling thread. This ensures, that the ID is right.

3.5 Using filtering algorithms

Multiple ready-made examples for filtering algorithms and other solutions are available with codes. Packages are ready to be downloaded and used in the project. These give an easy starting point in application development. At first it is possible to take the code as it is and then study, how well those algorithms work and analyse the results. After this early study possible adjustments can be made to the algorithms or other kind of solutions can be searched, if the already used algorithms totally fail.

The first filtering algorithm is Ramer-Douglas-Pecker. The code for it is in appendix 1. The first version of this uses code from Rosettacode[2] The epsilon value that defines how much details are omitted, is how this algorithm can be adjusted. The seek bar is added for adjustments in `MenuFragment.` The first solution is to save epsilon value to the shared preferences although most suitable value differs from route to route. The second algorithm is Kalman filter and it’s code in in appendix 2. The first implementation of Kalman filter is based on an example code in Stackoverflow and it is originally written with Java. It has been modified to Kotlin.[3] For Kalman filter separate class `KalmanFilter.kt is used` This class takes care of iterations needed to calculate new locations.

[2] https://rosettacode.org/wiki/Ramer-Douglas-Peucker_line_simplification#Kotlin
[3] https://stackoverflow.com/questions/1134579/smooth-gps-data

```
fun getRemainingLocations(threshold_accuracy: Float): ArrayList<GeoPoint> {
        val geoPoints: ArrayList<GeoPoint> = ArrayList()
      for (mlocation in measuredLocations) {
            if (mlocation.accuracy <= threshold_accuracy) {
                  geoPoints.add(GeoPoint(mlocation.latitude,
                  mlocation.longitude))
            }
      }
      return geoPoints
}
```

Listing 2. Algorithm for removing least accurate locations. Accuracy is in meters, `measured-Locations` is ArrayList of `MeasuredLocation` objects.

In the third tested algorithm the most inaccurate measured locations are deleted. The code for the algorithm can be seen in listing 2. The threshold can be changed from one extreme to the other. The extreme values of accuracies of certain route must be searched from all the locations of that route and use these values to programmatically adjust the extremes of slider in a way that slider minimum value represents the smallest radius and maximum the largest radius of the accuracy. The fourth one is moving average. In this algorithm the new values are calculated by counting averages of latitudes and longitudes separately from selected amount of adjacent locations. The code for this can be seen in listing 3. The averages are calculated from two to ten different locations. It can be selected if those averages are weighted using reciprocal of accuracy value.

```
private fun movingAverage(list: List<Double>, amount: Int): ArrayList<Double>
{
      val averages: ArrayList<Double> = ArrayList()
      for (index in amount..list.size) {
            averages.add(list.subList(index - amount, index).average())
      }
      return averages
}

private fun movingAverageWithWeigths(list: List<Double>, amount: Int,
        weights: List<Float>): ArrayList<Double> {
      val averages: ArrayList<Double> = ArrayList()
      for (i in amount..list.size) {
            var weighedSum: Double = 0.0
            var sumOfWeights: Float = 0f
            for (j in i - amount until i) {
                  weighedSum += weights[j] * list[j]
                  sumOfWeights += weights[j]
            }
            averages.add(weighedSum / sumOfWeights)
      }
      return averages
}
```

Listing 3. Algorithms for both calculating moving average and moving average with weights. List is whole list of latitudes or longitudes and the amount is from how many numbers the average is calculated. Weights are given as a list `movingAverageWithWeights()`-function.

4 Discussion

4.1 Filter technics and algorithms

The effect of sky access in locations can be seen in figure 7. When carrying a phone in a bag or in pocket the path has inaccuracies and path differs from actual walking route along the streets. When carrying phone in hand, the path follows streets much more precisely, but there can still be parts, where locations are systematically wrong. Walking with a phone in hand all the time becomes tedious. A better option is to use some other device tied to wrist that records locations.

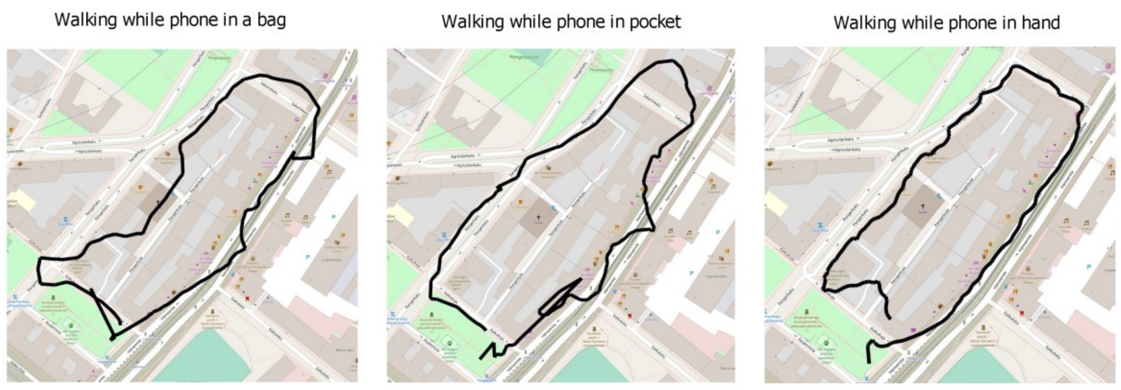
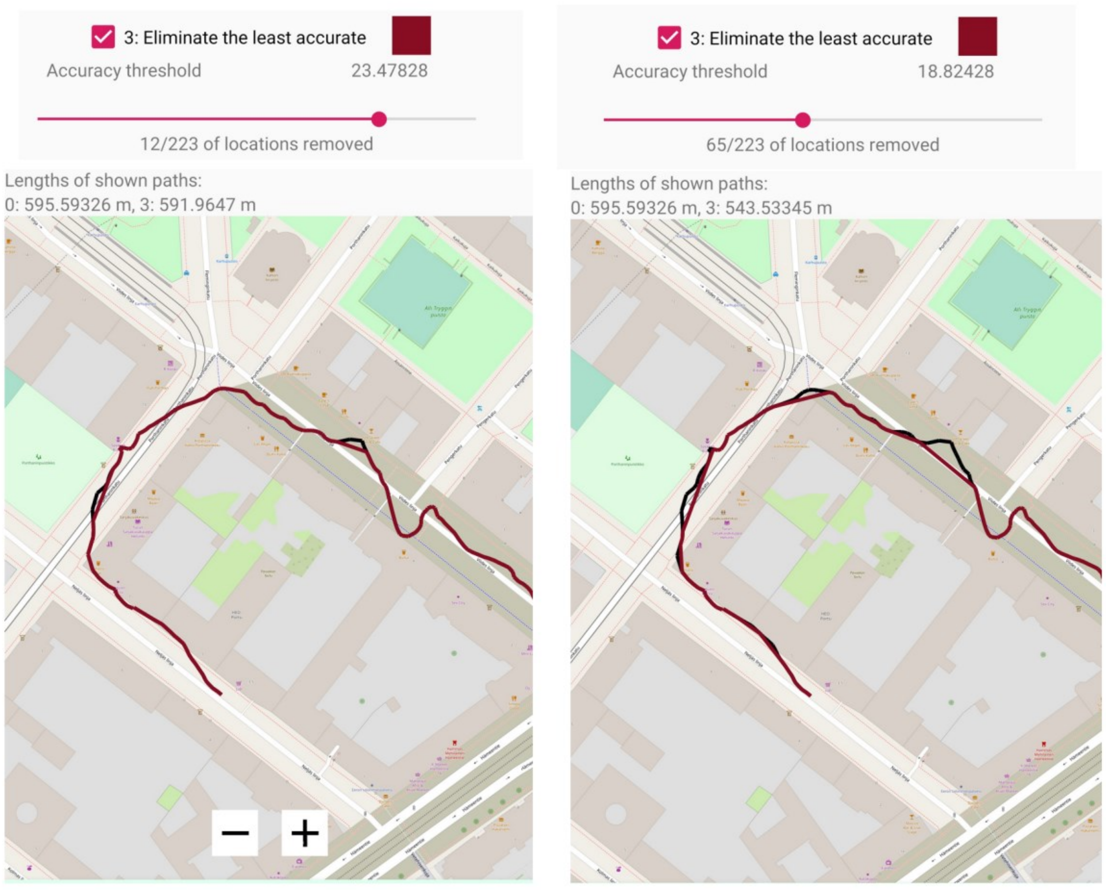

Walking while phone in a bag
Walking while phone in pocket
Walking while phone in hand

Figure 7. Same route walked multiple times.

Simple filtering methods can be very effective, if they are manually tuned. Even removing the most inaccurate locations can increase accuracy of a path significantly. If this is carried out automatically, this can lead to drastic changes, unless it is done carefully. Iteration would be possible to use to be able to find optimal threshold accuracy and eliminate all more inaccurate locations. The total length is one parameter to investigate. Total length must not change too drastically. It must be noticed that all inaccurate locations can be in same area and by removing these locations the path can change noticeable although the total length of route changes only a little. The full route could be divided into smaller sections and measure the change in route length of these smaller segments separately. This method can remove locations from start or end and not leaving first and last location in place. This can lead to drastic shortening of route. On the other hand, the first locations can be very inaccurate. It could be reasonable to remove the first few locations anyway, if the location recording speed is high. The last location would still be beneficial to keep no matter what.

The example of the removal of the least accurate locations is shown in figure 8. Here locations can be removed quite drastically and still the route stays nearly the same although the overall length of the route drops also significantly. This is mainly due to inaccuracies in the beginning of the route, which is not shown here.


✓ 3: Eliminate the least accurate
Accuracy threshold 23.47828
12/223 of locations removed
Lengths of shown paths:
0: 595.59326 m, 3: 591.9647 m

✓ 3: Eliminate the least accurate
Accuracy threshold 18.82428
65/223 of locations removed
Lengths of shown paths:
0: 595.59326 m, 3: 543.53345 m

Figure 8. Effects of eliminating least accurate values.

By using moving averages, the path can be smoothed as shown in figure 9. This does not remove the effects of the most inaccurate measurements. They change the averages noticeably and as a result there is still a deviation from the pathway left. The more points are used to calculate one average the more measured locations are needed. The weighting has very little effect on accuracy of route. Another way of using means is also possible. If the locations are measured very frequently, it could be possible to take averages and store these average values to database.

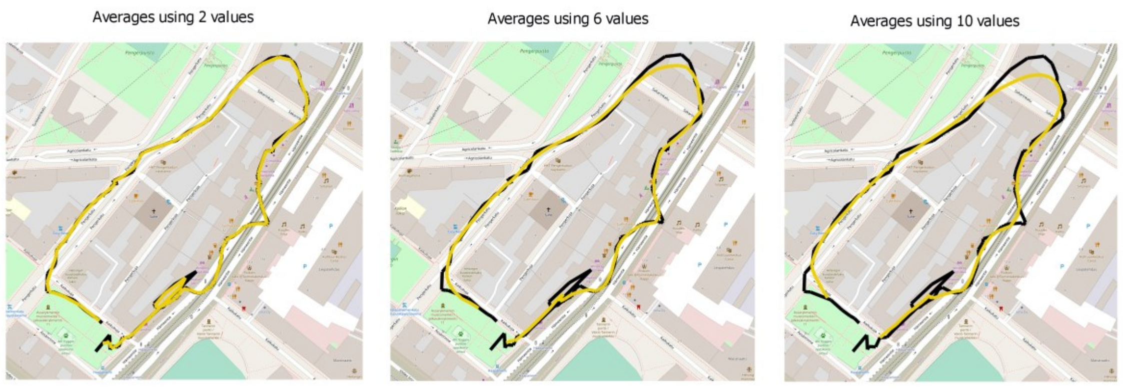


Figure 9. Effects of using running averages. Averages using 2, 6 and 10 values are compared.

Ramer-Douglas-Pecker algorithm is one way of smoothing the pathway. If extreme values are wrong, also the corrected path will be wrong, but it still can remove excess zigzagging from routes. The variable epsilon defines how much smoothing is done. The effects of different values of epsilon can be seen in figure 10. If the epsilon is too small, the algorithm has no effect, if it is too large, the calculated route cuts through corners.

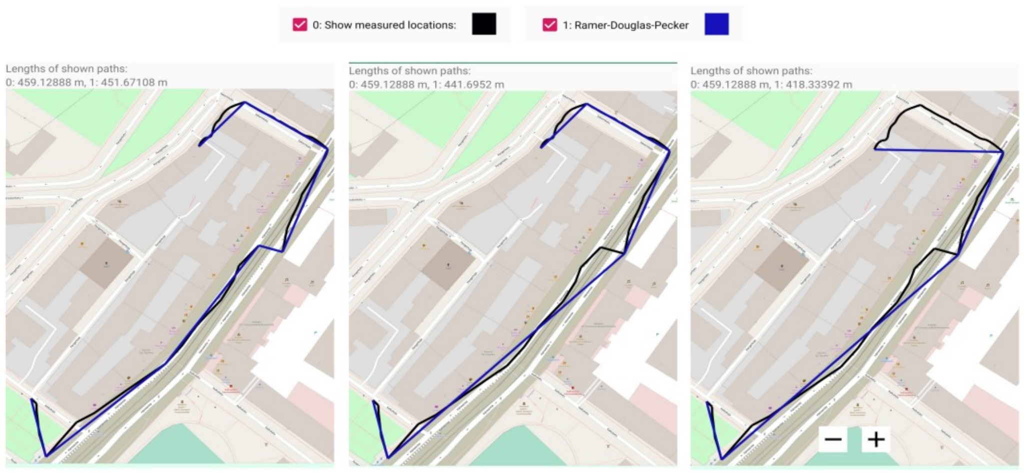

0: Show measured locations:
1: Ramer-Douglas-Pecker
Lengths of shown paths:
0: 459.12888 m, 1: 451.67108 m
Lengths of shown paths:
0: 459.12888 m, 1: 441.6952 m
Lengths of shown paths:
0: 459.12888 m, 1: 418.33392 m

Figure 10. The significance of epsilon value to Ramer-Douglas-Pecker algorithm. The epsilon is the smallest on the left side and the biggest on the right side

Kalman filtering differs from the other filtering methods in that it does not remove inaccurate values. It rather calculates the more probable positions for these locations. Kalman filter tried in this study did not use data from external source for verification of locations, but it has some effects as is shown in figure 11. Kalman filter can smooth the pathway but it can also cut too much from corners.

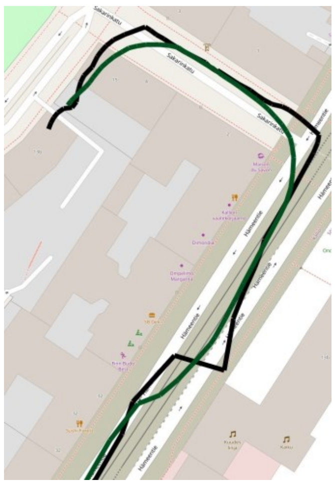


Figure 11. Effects of Kalman filtering, black is the original path and green is the filtered one.

Filtering methods can also be combined. First the most inaccurate locations could be removed and after this some other kind of filtering to further reduce the inaccuracies in route can be used.

Bearings can be used in faster speeds in some kind location verification algorithm. Bearings are more accurate at faster speeds, but they seem to give somewhat reasonable results in walking speeds as can be seen in figure 12. This shows that at walking speeds bearings follow walking path lazily. When walking straight they show to walking direction, but when turning they gradually change to new walking direction. When staying at the same location, the values of bearings get random values. The bearing accuracies could be utilised in location verification calculations. This is only for driving speeds or cycling speeds. They are not available in normal walking speeds; they all are recorded as zeros instead. Testing the bearing accuracies in faster speeds is possible future study subject.

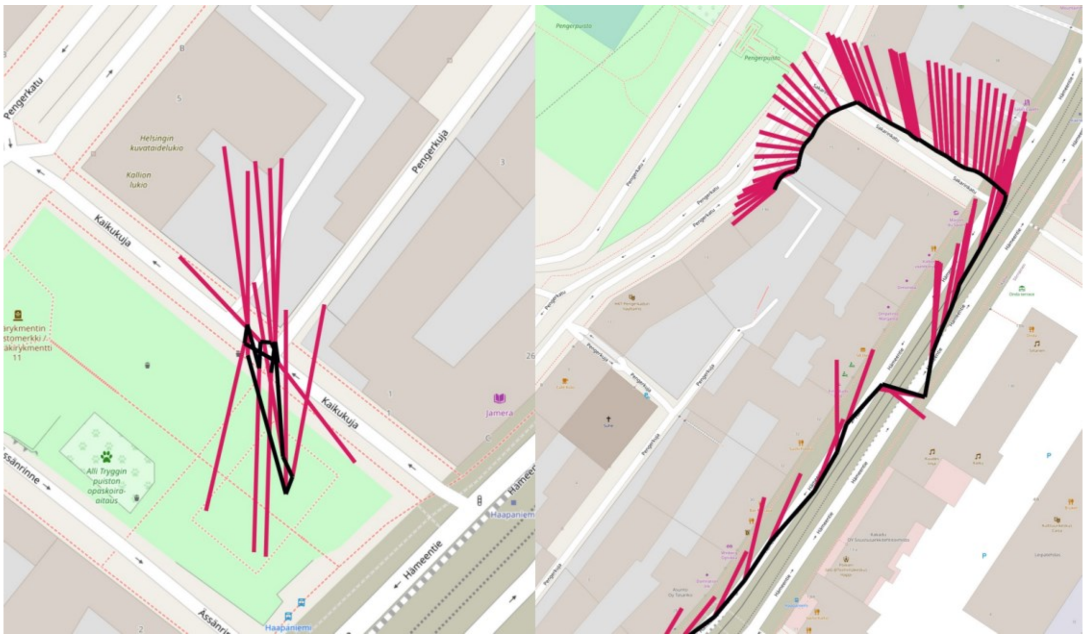


Figure 12. Bearings are shown here with red lines. The one on the left is recorded while sitting on a park bench. The one on the right is recorded while walking.

There are still multiple ideas that can be tested. From location data the extras could be saved in database in String form and their content could be analyzed. These extras include how much satellites are used to calculate the location and other technical details. The amount of satellites compared to accuracy could be interesting to study.

Another possible solution would be to match the route to the features of the map. Road coordinates could be found for certain maps, but it is not a universally feasible solution. If the moving is only done in vehicle such as a car, then this could work out, but movement on foot does not need to follow roads: It can be wandering in the forest or in the park and these paths cannot combine with certain map features.

4.2 Known bugs and further development ideas

Bugs are resolved as they come by. For now, all major bugs have been resolved. One of the worst bugs was the bug that prevents collecting location information, when service for that was unbound from the `TrackingMapActivity`. This was because erroneously the request for updates was inside if-block that was executed only when service was bounded.

In the current state only one route can be investigated at the same time. It would be interesting to combine results of walking same route multiple times. This way it would be easy to analyze how accuracies depend on location and how much there is random variation. `RouteListingActivity` needs to be changed in a way, that multiple routes can be selected. This can be acquired by using checkboxes for each routes and separate button for moving to `DataViewingActivity`. In `MapFragment` different routes would need differ visually from each other. Now different colors are used for measured routes and routes formed using different algorithms. For different routes different kind of line types could be used, like continuous, dashed or dotted lines. Different line thicknesses could be one option. Lines could made partially transparent, then lines top of each other could be recognized better.

`GraphFragment` could show multiple lines in same time. Legend needs to be added, where could be possible to select, what of those routes to be compared are shown and what kind of line they are represented. In case of different routes graphs showing these routes together does not make sense, but it is up to user to interpret meanings. `NumericalFragment` needs redesigning in a way that it is possible to choose what numerical data is shown. This fragment could be redesigned completely, since viewing just the lists of numerical values of all route locations is not useful. All kind of averages and variances would give user much more relevant information.

More options are needed for viewing live locations. An option to show accuracies and bearings is needed. A menu for choosing which kind of information is shown could be added. Data in numerical format could be added, at least information of how many locations are already stored in database. The application can be used in different ways and that is why different options for viewing are needed. When using the application while sitting in a tram, it could be interesting to follow what kind of information is obtained but when walking it is best to focus on surroundings and not to the screen of the mobile device.

One option could be to add an easy way to take screenshots from both map showing live locations and map showing locations from database. Of course, other ways to take screenshots exist but this way screenshots could be tailored to include only certain features and the dimensions can be adjusted. What to include in a screenshot could be selected. The screenshot could get descriptive name. There could be an option to share these screenshots to social media or to Google drive.

In this stage data from database can be downloaded as JavaScript object notation (Json). For now, this is not used anywhere and to fully utilise data in Json, another program should be developed. It would be easier to use the data if it were in comma separated values (csv), because then the data could be inserted into spreadsheet such as Microsoft Excel. An option to make a backup of the whole database as it is and restore the database from that backup is needed. Database needs further development. For different routes optimal variables for different algorithms could fit to Route-table. First these values could be nulls and these could be saved when optimal values are found by visual inspection.

In the current state the description of the route is not used in any way and it can not be modified after it is first saved. There can be reasons for modification of the description after the locations of the route have been recorded, because something unexpected might happen during gathering of the location data. The walking route can end up being different than planned. The live data might show that the location data suddenly becomes completely inaccurate and wrong. These notes would be nice to add to the description as this would help sort the routes at the time of analyzing the data.

5 Conclusion

The application made for this project collects location data and saves it to the local database. This can be used to collect locations from a walking route. This collecting is done by service. This service can be bound to an activity that shows location on the map or it can run as foreground service. The already collected routes can be viewed in map and this original route data can be compared with routes obtained by using four different filtering algorithms.

The compared algorithms are Ramer-Douglas-Peucker, Kalman filter, removal of least accurate locations and moving average. Ramer-Douglas-Peucker does not depend on accuracies. It just searches the extremes and smoothens the paths between extreme locations. This means that pathways become straighter but if those extremes are erroneous, those errors will remain. Kalman filter iterates through the locations and can use external data for calculating the new estimates. The test algorithm used in this thesis did show that Kalman filter smoothens the pathway. This algorithm can be further developed to include a better way of estimating the correctness of locations. Kalman filter involves complicated mathematics and the developing of algorithm is hard. Other filtering methods are less complicated. Filtering can be done as simple as removing the most inaccurate locations. If the threshold is set correctly this can lead to a good enough solution. Running average smoothens the path and removes the separate values deviated from the path.

Representation of data is what needs to be developed further. The goal for this is to make comparison of algorithms better by increasing the options for the user of the application. This means the option to select multiple routes to compare them in the map and other means. The altitudes need filtering so that the most inaccurate ones are removed. Tracking of altitudes and representing the altitudes of a whole route in this application needs to be studied more to be able to use this altitude information in other applications in the future.

References

1. Gargenta, Marko & Nakamura, Masumi. 2014 Learning Android E-book. O'Reilly Media

2. Callaham, John, 2019. The history of Android OS: its name, origin and more. Webpage. Android Authority. <https://www.androidauthority.com/history-android-os-name-789433/> Visited: 5.6.2020

3. Dataflair Team, 2020. Android Architecture – 5 Components of Android Architecture. <https://data-flair.training/blogs/android-architecture/> Visited: 5.6.2020

4. Amadeo, Ron, 2019. Google outlines plans for mainline Linux kernel support in Android. Webpage. <https://arstechnica.com/gadgets/2019/11/google-outlines-plans-for-mainline-linux-kernel-support-in-android/> Visited 5.6.2020

5. Ye, Roger. 2017. Android System Programming . E-book. Packt Publishing

6. Android Architecture. Webpage. Javatpoint. <https://www.javatpoint.com/android-software-stack > Visited: 5.6.2020

7. Jain, Ayusch. Android Internals 101: How Android OS Starts You Application. Webpage. <https://www.droidcon.com/news-detail?content-id=/repository/collaboration/Groups/spaces/droidcon_hq/Documents/public/news/android-news/Android Internals 101 - How Android OS Starts You Application> Visited 30.7.2020

8. Platform Architecture. Webpage. googledevelopers. <https://developer.android.com/guide/platform > Visited: 5.6.2020

9. Android Jetpack, Webpage. googledevelopers. <https://developer.android.com/jetpack > Visited 5.6.2020

10. Tillu, Jay. 2019. Why JetBrains create Kotlin? The Inside story of Kotlin creation. Webpage. <https://dev.to/jay_tillu/why-jetbrains-create-kotlin-the-inside-story-of-kotlin-creation-1135> Visited: 5.6.2020

11. Lardinois, Frederic, 2019. Kotlin is now Google's preferred language for Android app development. Webpage. < https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred-language-for-android-app-development/?guccounter=1 > Visited: 5.6.2020

12. Kotlin. Calling Java code from Kotlin. Webpage < https://kotlinlang.org/docs/reference/java-interop.html > Visited: 5.6.2020

13. Meija, Arturo. 2018. Android KTX Tutorial: Getting Started. Webpage. < https://www.raywenderlich.com/5576-android-ktx-tutorial-getting-started > Visited: 5.6.2020

14. Doberstein, Dan. 2012. Fundamentals of GPS Receivers. E-book, Chapter 1. Springer-Verlag.

15. NexGenDesign. "Lost in Tracking" or why mobile GPS is inaccurate? <http://www.nexgendesign.com/lost-in-tracking-mobile-gps> Visited: 6.6.2020

16. ArcUser Online. 2003 Mean Sea Level, GPS, and the Geoid. Webpage. <https://www.esri.com/news/arcuser/0703/geoid2of3.html> Visited: 6.6.2020

17. GPS World Staff. 2010. Part 1: The Origins of GPS, and the Pioneers Who Launched the System. Webpage. <https://www.gpsworld.com/origins-gps-part-1/> Visited: 6.6.2020

18. GPS World Staff, 2010. Part 2: The Origins of GPS, Fighting to Survive. Webpage. <https://www.gpsworld.com/origins-gps-part-2-fighting-survive/> Visited: 6.6.2020

19. European Global Navigation Satellite Systems Agency. What is GNSS? Webpage. < https://www.gsa.europa.eu/european-gnss/what-gnss > Visited: 6.6.2020

20. Hidenbrand, Jerry, 2018. How does GPS work on my phone? Webpage. <https://www.androidcentral.com/how-does-gps-work-my-phone> Visited: 6.6.2020

21. Vallina-Rodriuez, Narseo, Crowcroft, Jon, Finamore, Alessndro, Grunenberger, Yan, Papagiannaki, Konstantina 2013. When assistance becomes dependence: characterizing the costs and inefficiencies of A-GPS, Webpage. <https://dl.acm.org/doi/10.1145/2557968.2557970> Visited: 6.6.2020

22. manhbt. 2018. Understanding Android GPS Architecture. Webpage. <https://manhbt.wordpress.com/2018/01/02/understanding-android-gps-archi-tecture/> Visited: 6.6.2020

23. Meg. 2019. Why is GPS data sometimes inaccurate? Webpage. <https://sup-port.strava.com/hc/en-us/articles/216917917-Why-is-GPS-data-sometimes-in-accurate> Visited: 6.6.2020

24. Singh, Ishveena, 2017. How accurate is the altimeter in a GPS watch? Webpage. <https://geoawesomeness.com/accurate-altimeter-gps-watch/> Visited: 6.6.2020

25. Bryan, Michael F., 2020. Line Simplification with Ramer–Douglas–Peucker. Webpage <http://adventures.michaelfbryan.com/posts/line-simplification/> Vis-ited: 6.6.2020

26. Esme, Bilgin, 2009. Kalman Filter For Dummies. Webpage. <http://bil-gin.esme.org/BitsAndBytes/KalmanFilterforDummies#> Visited: 6.6.2020

Code for Ramer-Douglas-Peucker algorithm

```
//ttps://rosettacode.org/wiki/Ramer-Douglas-Peucker_line_simplification#Kotlin
private fun perpendicularDistance(location: MeasuredLocation,
        startLocation: MeasuredLocation, endLocation: MeasuredLocation
): Double {
     var dx = endLocation.longitude - startLocation.longitude
     var dy = endLocation.latitude - startLocation.latitude

     val pointList = measuredLocations

     val mag = hypot(dx, dy)
     if (mag > 0.0) {
          dx /= mag
          dy /= mag
     }

     val pvx = location.longitude - startLocation.longitude
     val pvy = location.latitude - startLocation.latitude

     val pvdot = dx * pvx + dy * pvy

     val ax = pvx - pvdot * dx
     val ay = pvy - pvdot * dy

    return hypot(ax, ay)
}


fun ramerDouglasPeucker(pointList: List<MeasuredLocation>,epsilon: Double,
        out: MutableList<MeasuredLocation>
    ): Boolean {
     if (pointList.size < 2) {
          return false
     }

     // Obtain the point with the maximum distance from line between start
     //and end
     var dmax = 0.0
     var index = 0
     val end = pointList.size - 1
     for (i in 1 until end) {
          val d = perpendicularDistance(pointList[i], pointList[0],
                              pointList[end])
          if (d > dmax) {
                index = i; dmax = d
          }
     }

        // If max distance is greater than epsilon, recursively simplify
        if (dmax > epsilon) {
            val recResults1 = mutableListOf<MeasuredLocation>()
            val recResults2 = mutableListOf<MeasuredLocation>()
            val firstLine = pointList.take(index + 1)
            val lastLine = pointList.drop(index)
            ramerDouglasPeucker(firstLine, epsilon, recResults1)
            ramerDouglasPeucker(lastLine, epsilon, recResults2)
// formulate result list
            out.addAll(recResults1.take(recResults1.size - 1))
            out.addAll(recResults2)
            if (out.size < 2) {
                return false
            }
        } else {
            // Just return start and end points
            out.clear()
            out.add(pointList.first())
            out.add(pointList.last())
        }

        return true
    }

    /**
     * for algorithm 1: Ramer-Douglas-Pecker
     */
    fun getAlgorithm1GeoPoints(epsilon: Double): ArrayList<GeoPoint> {
        val locations: ArrayList<MeasuredLocation> = ArrayList()
        val success = ramerDouglasPeucker(measuredLocations, epsilon, loca-
tions)
        if (!success) {
            locations.clear()
        }
        return getMeasuredLocationsAsGeoPoints(locations)

}
```

Code for Kalman Filter

```
KalmanFilter.kt

package fi.metropolia.juhavuo.trackingaccuracy

import kotlin.math.pow
import kotlin.math.sqrt


//https://stackoverflow.com/questions/1134579/smooth-gps-data, Stochastical-
ly's aswer (in Java)
class KalmanFilter(speed: Float){

    private val minAccuracy = 1f

    private var meanSpeed = speed
    var lat = 0.0
    private set

    var lng = 0.0
    private set

    var variance = -1f
    private set

    var ts = 0L
    private set

    fun getAccuracy(): Float = sqrt(variance)

    fun setState(latitude: Double, longitude: Double, accuracy: Float,
timestamp: Long){
        lat = latitude
        lng = longitude
        variance = accuracy.pow(2)
        ts = timestamp
    }

    fun process(measured_lat: Double, measured_lng: Double, me_acc: Float,
timestamp: Long){
        var measured_acc = me_acc
        if(measured_acc<minAccuracy){
           measured_acc = minAccuracy
        }
        if(variance<0){
            setState(measured_lat,measured_lng,measured_acc,timestamp)
        }else{
            val timeBetween = timestamp - ts
            if(timeBetween> 0){
                variance += timeBetween*meanSpeed.pow(2)/1000
                ts = timestamp
            }
        }

        val k = variance/(variance+measured_acc.pow(2))
        lat += k*(measured_lat-lat)
        lng += k*(measured_lng-lng)
        variance *= (1 - k)
    }
```

In DataAnalyzer.kt class:

```
fun getKalmanFilteredGeoPoints(): ArrayList<GeoPoint> {
     val kalmanGeoPoints: ArrayList<GeoPoint> = ArrayList()
     var speed = getSpeedMeanValue()
     speed *= 1.2f //this can be changed, for better values
     if (speed < 3f) {
          speed = 3f
      }

     val kalmanFilter = KalmanFilter(speed)

     kalmanFilter.setState(
            measuredLocations[0].latitude,
            measuredLocations[0].longitude,
            measuredLocations[0].accuracy,
            measuredLocations[0].timestamp
     )
     for (index in 1 until measuredLocations.size) {
          kalmanFilter.process(
                measuredLocations[index].latitude
                , measuredLocations[index].longitude
                , measuredLocations[index].accuracy
                , measuredLocations[index].timestamp
            )

            kalmanGeoPoints.add(GeoPoint(kalmanFilter.lat, kalmanFilter.lng))
     }

     return kalmanGeoPoints
}
```