HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

Term project - Fall 2017

Ranck

Downhill Support App

Author:
Kurath Samuel

Supervisor:
Prof. Dr. Farhad Mehta

February 14, 2018

Abstract

The project Ranck aims to support downhill bikers during their risky rides along trails. For this purpose, the cyclist needs to be informed if he is too fast for the next curve or his speed is appropriate. Hence Ranck tries to realize this goal with the help of an application for smartphones and their built-in sensors. If the built-in sensors aren't accurate enough external ones would be considered. As a first approach, the GPS sensor and the accelerometer were fused with a Kalman filter. Unfortunately, this didn't lead to accurate enough measuring data, since the application didn't recognize speed changes within a reasonable time. To counteract this problem an external bike speed sensor from Garmin was considered and tested. The external sensor improved the measurement and provided the ability to detect fast speed changes. However, the determination of the exact position on the track is still missing. Consequently, an application that is able to handle the requirements of a downhill biker was not achieved. Nevertheless, the project resulted in a runnable prototype and solved different sub-problems that might be helpful for similar projects.

Declaration of Authorship

Place and date	Samuel Kurath

Contents

1. Introduction

1.1. Motivation

Sport without technical support is unthinkable anymore. The current unbelievable achievements of athletes are often related to improved analysis, measurement methods and the usage of technical gadgets. All top athletes use devices to track their training and performances. Even the amateur athletes wear multiple devices like a sports watch or smartphone during their sportive activities. For example, skijumping professionals use a differential global positioning system (DGPS) to get highly accurate measurements of their trajectories and the amateur sportsman uses the bikecomputer to monitor his pedalling rate and average speed over the past five kilometers.

In addition to the boost in performance, an other aspect is the security. Even the best tennis player wouldn't win a match with a broken arm. Hence, it is absolutely necessary to have as few injuries as possible.

1.2. Vision

The Ranck application aims to use a smartphone to help downhill bikers avoid crashes, by providing them information about the track, their current speed and a computed maximum speed for the next curve.

This goal should be reached through the use of the built-in sensors in the smartphone and tracks previously stored on the device. If external sensors are needed, the barrier to use the application rises. Only when the results of the built-in sensors aren't accurate enough, will the external sensors be considered.

Since it isn't certain that such an app can be realized, the project focuses on the best possible solution and tries to identify and analyse the related issues.

2. Challenges

With the big goal of a downhill support application in mind, I had to deal with a lot of interesting problems. These issues and the associated solutions are mentioned in the following section.

To give a brief overview of them, let us have a look at the questions that arose during the project.

- How can you determine if you are to fast for the next curve?
- What is a curve in a linestring of coordinates?
- Which is a reasonable spatial reference system?
- How can the acceleration values which are connected with to the device axis, be aligned with the chosen spatial reference system?
- How can multiple sensors be combined for more accurate measurements?
- Which external sensor could improve the Ranck application?
- What is relevant information for the biker and how to provide it?

2.1. Physical model

Problem To determine if the current speed of the downhill biker is too high for the next curve. For that purpose a model, which is able to describe the physical circumstances, is necessary.

Goal To define a reasonable physical model and identify the corresponding parameters and forces.

2.1.1. Approach

The chosen model is an approximation of the physical circumstance and should model the environment in an appropriate manner. To reduce the complexity of the model and avoid the need of external sensors, the physics of the bicycle like the slope of the steering wheel, weather circumstances, air resistance and more were ignored.

The parameters focused on are listed below. They are given by the defined tracks, added as a setting or are provided by the built-in sensors of the smartphone.

m:	Driver weight [kg]
v:	Velocity $[\frac{m}{s}]$
μ:	Friction coefficient
r:	Curve radius [m]
α:	Slope angle [°]
g:	Gravity acceleration $[\frac{m}{s^2}]$

Firgure 2.1 shows a bicycle in a banked curve and the relevant forces. The goal of the system is to let force $\vec{F_A}$ always be smaller or equal to the sum of the forces $\vec{F_B}$ and $\vec{F_R}$.

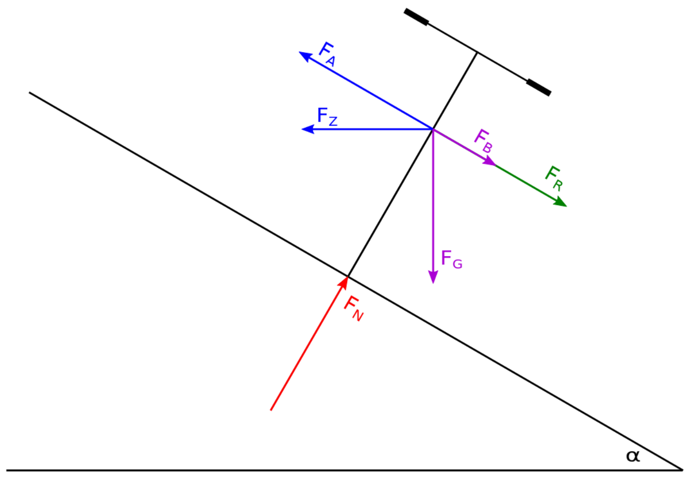


Figure 2.1.: *Physical model*

where:

$\vec{F_G}$: Gravity force
$\vec{F_N}$: Normal force
$\vec{F_R}$: Friction force
$\vec{F_Z}$: Centrifugal force
$\vec{F_B}$: Gravity force under angle α
$\vec{F_A}$: Centrifugal force under angle α

With the above listed parameters we are able to calculate the forces (2.1) and can determine the maximum possible speed (2.2).

$$\begin{aligned} \vec{F_R} &= \mu \frac{m \cdot g}{\cos \alpha} \\ \vec{F_Z} &= m \frac{v^2}{r} \\ \vec{F_G} &= m \cdot g \\ \vec{F_A} &= \vec{F_Z} \cdot \cos \alpha \\ \vec{F_B} &= \vec{F_G} \cdot \sin \alpha \end{aligned} \tag{2.1}$$

$$v_{max} = \sqrt{\frac{r \cdot g(\sin \alpha + \mu \cdot \cos \alpha)}{\cos \alpha - \mu \cdot \sin \alpha}} \tag{2.2}$$

Example To get a better grasp of what these equations represent, let us have a look at an example. The following are values for the parameters.

m $= 70\,\mathrm{kg}$
v $= 7\,\frac{\mathrm{m}}{\mathrm{s}} = 25.2\,\frac{\mathrm{km}}{\mathrm{h}}$
$\mu = 0.008$ (bicycle tire on rough paved road [Too18])
r $= 5\,\mathrm{m}$
$\alpha = 30°$
$g = 9.81\,\frac{\mathrm{m}}{\mathrm{s}^2}$

Applying these parameters results in the forces listed in (2.3).

$$\begin{aligned} \vec{F_Z} &= 70\,\mathrm{kg} \frac{(7\,\frac{\mathrm{m}}{\mathrm{s}})^2}{5\,\mathrm{m}} = 686\,\mathrm{N} \\ \vec{F_A} &= \vec{F_Z} \cdot \cos 30° = 594.1\,\mathrm{N} \\ \vec{F_R} &= 0.008 \frac{70\,\mathrm{kg} \cdot 9.81\,\frac{\mathrm{m}}{\mathrm{s}^2}}{\cos 30°} = 6.3\,\mathrm{N} \\ \vec{F_G} &= 70\,\mathrm{kg} \cdot 9.81\,\frac{\mathrm{m}}{\mathrm{s}^2} = 686.7\,\mathrm{N} \\ \vec{F_B} &= \vec{F_G} \cdot \sin 30° = 343.4\,\mathrm{N} \end{aligned} \tag{2.3}$$

As you can see in equation (2.4), force $\vec{F_A}$ is bigger than the sum of forces $\vec{F_B}$ and $\vec{F_R}$. Thus, the current speed in relation to our example parameters is too high for the curve.

$$\begin{gathered} \vec{F_A} >= \vec{F_B} + \vec{F_R} \\ 594.1\,\mathrm{N} >= 343.4\,\mathrm{N} + 6.3\,\mathrm{N} \end{gathered} \tag{2.4}$$

The related maximal speed could be determined and is shown in (2.5).

$$v_{max} = \sqrt{\frac{5\,\mathrm{m} \cdot 9.81\,\frac{\mathrm{m}}{\mathrm{s}^2}(\sin 30^\circ + 0.008 \cdot \cos 30^\circ)}{\cos 30^\circ - 0.008 \cdot \sin 30^\circ}} = 5.4\,\frac{\mathrm{m}}{\mathrm{s}} = 19.4\,\frac{\mathrm{km}}{\mathrm{h}} \tag{2.5}$$

The comparison between the calculated v_{max} and the current speed v consolidates this result as illustrated in (2.6).

$$\begin{gathered} v_{max} < v \\ 5.4\,\frac{\mathrm{m}}{\mathrm{s}} < 7\,\frac{\mathrm{m}}{\mathrm{s}} \end{gathered} \tag{2.6}$$

2.1.2. Conclusion

The chosen physical model fits the circumstances in an adequate way. It isn't too complex and is calculable in a suitable amount of time even on a smartphone with limited performance. More problematic are some important parameters like the slope of the curve, the friction coefficient or the skill of the driver. For example the slope of one curve isn't always the same and highly depends on the exact trail the biker takes. Further, the friction often changes due to weather or trail usage conditions. Hence, the Ranck application has an adjustable parameter to counteract these problems.

2.2. Curves

Problem The tracks appear in line strings of coordinates, there is no information about when a curve starts or ends, the radius, the direction or even what a curve is. Thus, we have to figure out how to split the tracks into reasonable parts and figure out how a curve could be represented.

Goal To build segments out of the tracks, calculate the radius of the curves, define their directions and find an adequate representation for the tracks and curves. Furthermore, the process should be repeatable and comprehensible.

2.2.1. Approach

To define the track segments, two processing steps were taken. First we calculate the radius and the direction for every point on the track [Fra17]. The radius is computable out of three points illustrated in figure 2.2. The direction is determined by the cross product of three points. The chosen format to represent these tracks is GeoJSON, since it is easy to understand, readable with every text editor and provides a property field to enhance the data in accordance to your wishes.

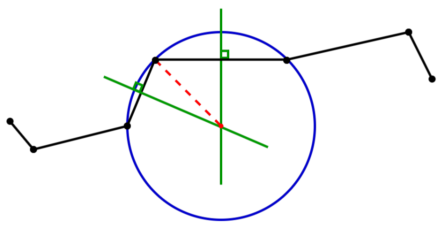


Figure 2.2.: *Radius out of three points*

The second step is to build segments from the points of the line string. The algorithm summarizes all points along the track until the radius leaves a certain range or the directions change. The ranges depend on the radius and are listed in table 2.1. After a change, a mean radius is calculated out of all points in the current segment. To store the gained information, a properties field is added to the GeoJSON file with the mean radius and the direction for every point of the track. Thus, if you go along the properties of the track, you know a new segment begins should the radius or the direction change.

Radius range [m]	Category	Name
0-10	1	small curve
10-40	2	curve
40-∞	3	straight

Table 2.1.: *Curve ranges*

Example To visualize the algorithm, have a look at figure 2.3. On the left-hand side, the raw track is represented and on the other side there is the processed track with a color for every segment that belongs together. The website geojson.io [Map17] provides a tool to create geometries represented as GeoJSON. This tool was used to generate the tracks for the Ranck application.

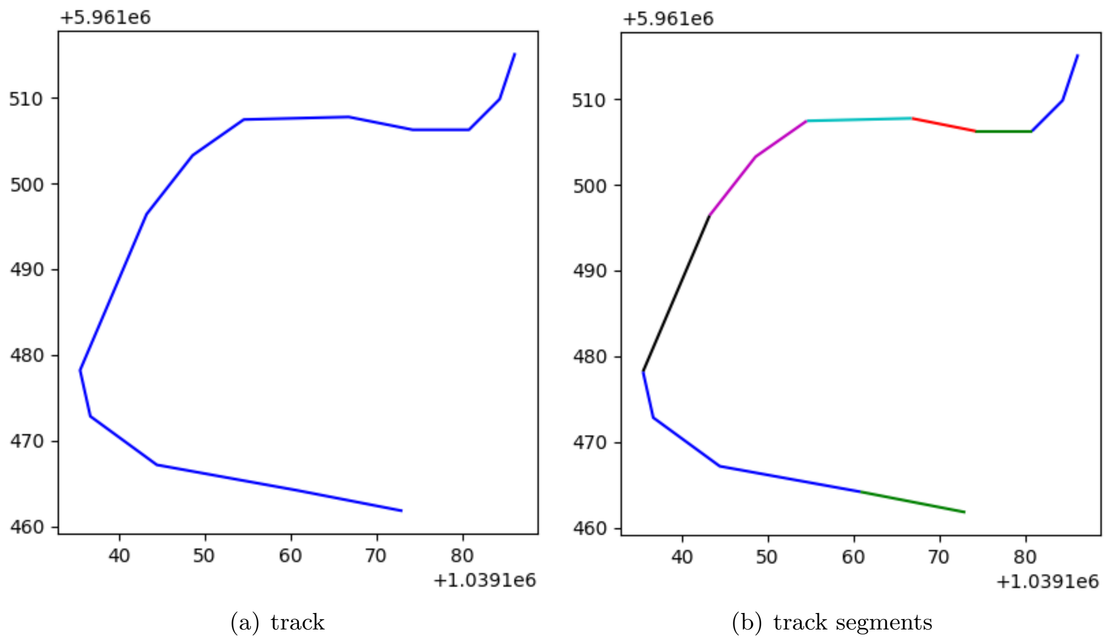


Figure 2.3.: *Track segment visualization*

Listing 2.1 shows the related line string in the GeoJSON format. As you can see, there are the additional properties directions and radiuses.

```
{
"features": [{
  "properties": {
    "directions": [
            -1.0, -1.0, -1.0, -1.0, -1.0, 1.0,
            1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0
      ],
    "radiuses": [
            11.679104489078362, 11.679104489078362,
            11.679104489078362, 7.612380836338962,
            35.98007719302654, 44.87616608734529,
            22.04574453395551, 22.04574453395551,
            54.83334336476179, 20.41688307583868,
            20.41688307583868, 20.41688307583868,
            1135.0356087700316
          ]
  },
  "geometry": {
    "type": "LineString",
    "coordinates": [
          [9.335168302059174, 47.11897340210018],
          [9.335152208805084, 47.11894054876184],
          [9.335120022296906, 47.118918646525],
          [9.335061013698578, 47.118918646525],
          [9.334993958473206, 47.118927772458136],
          [9.334883987903595, 47.11892594727162],
          [9.334830343723297, 47.11890039465411],
          [9.334782063961029, 47.11885841532722],
          [9.334712326526642, 47.11874707869123],
          [9.334723055362701, 47.11871422521314],
          [9.334792792797089, 47.118679546519814],
          [9.334940314292908, 47.118661294566884],
          [9.335050284862518, 47.118646693000024]
      ]
  },
  "type": "Feature"
}],
"type": "FeatureCollection"
}
```

Listing 2.1: *GeoJSON example*

Result The described process resulted in a repository on [**GitLab**] called track_processing [Kur18a]. The scope of application is not only restricted to the Ranck app. Other similar applications could benefit from these findings too.

2.2.2. Conclusion

The current approach to split the track into accurate segments fulfills the requirements. With the use of the GeoJSON format, a simple representation of the tracks is chosen with the possibility to add extended information. In addition there are several libraries that facilitate the handling of the format [Tob17] [Har17]. Creating the tracks by hand is time-consuming. To avoid this toil, further steps could include recording the tracks during the rides and automatically generating the processed tracks.

2.3. Spatial reference system

Problem The most common spatial reference system is the EPSG 4326 - WGS84 [IOG18] projection, known as latitude and longitude. It is also the served format of the Android API [Dev18a]. The unit in which longitude and latitude are given is degrees [°]. In Europe we use the metric system if we talk about speed or distances. Thus it would be helpful to use the metric system for the projection as well. This will enable an interaction with our physical model without any previous transformations.

Goal To use a spatial reference system with a metric system.

2.3.1. Approach

During the Rank project two different spatial reference systems were used. They are listed in the following subsection.

ECEF

The first choice of spatial reference system was the ECEF (earth-centered, earth-fixed), the corresponding EPSG identification number is 4978. ECEF is a three dimensional metric system with its origin in the center of the mass of the earth. The projection is illustrated in figure 2.4.

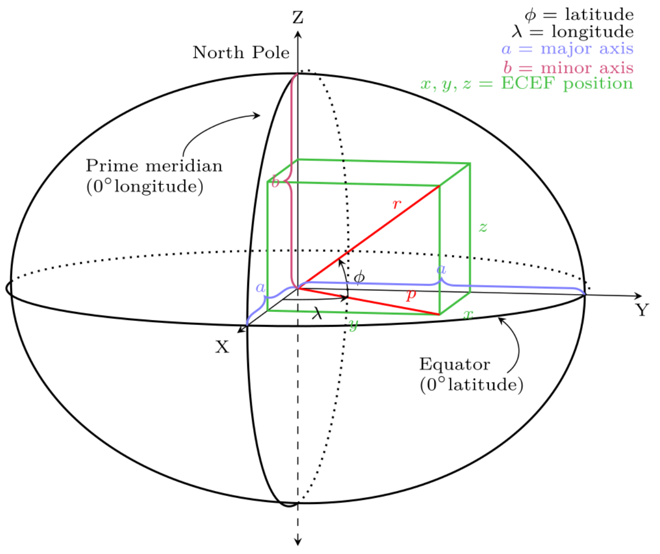


Figure 2.4.: *ECEF projection [Kri18]*

Web Mercator

Web Mercoator is a two dimensional metric spatial reference system (EPSG:3857 - WGS84). It's widely used in web mapping applications like Google Maps. The globe is projected onto a card as illustrated in figure 2.5.

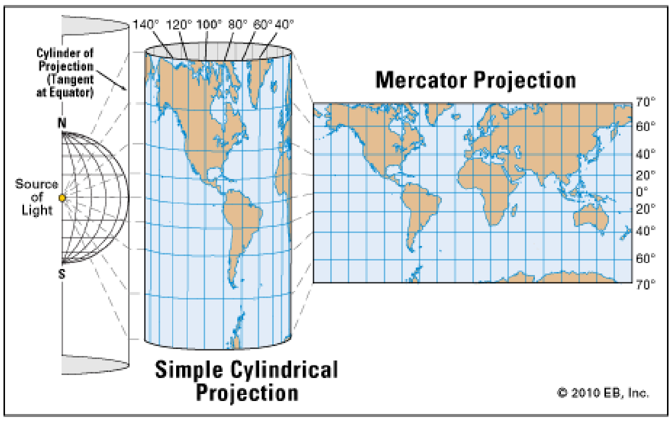


Figure 2.5.: *Mercator projection [TM18]*

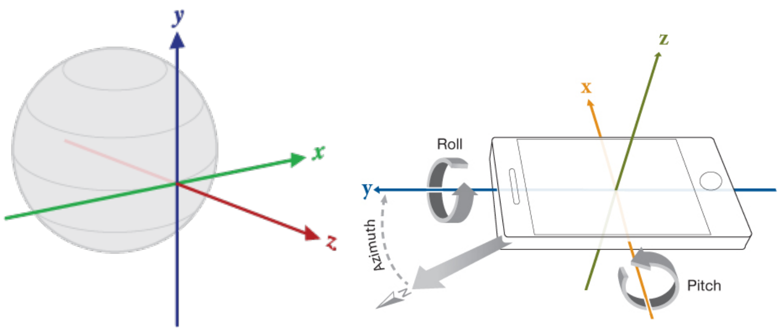
2.3.2. Conclusion

Unfortunately, PROJ.4 doesn't fully support ECEF, hence the projection from EPSG:4326 (latitude, longitude and altitude) had to be implemented on our own. And the altitude measurements of the device has scatter the most (up to 30 m between two measurements). This makes it rather impossible to calculate the exact speed and position. Thus, the altitude measurement is ignored and only the latitude and longitude values are consulted.

Decision The Ranck app uses the Web Mercator (EPSG:3857 - WGS84) spatial reference system. Because of the metric unit, the measured altitude values vary too much, the projection is widespread and supported by the PROJ.4 library [War18].

2.4. Alignment of the accelerometer

Problem The built-in accelerometer of a smartphone provides measurement values in the directions of the edges. If you want to combine the GPS values and the acceleration, you need to have the same measurement units and the same alignment of the data. Figure 2.6 show the two axis systems.



(a) Globe axis (b) Device axis (Azimuth, Pitch, Roll)

Figure 2.6.: *Axis systems [Tea17]*

Goal Align the acceleration measurements equal to the GPS values.

2.4.1. Approach

To align the acceleration values relative to the Web Mercator (EPSG:3857 - WGS84) projection, you can use the built-in geomagnetic field and to the gravity sensor. They enable us to determine the rotation matrix for the right alignment.

The units of the two sensors are micro tesla [µT] for the geomagnetic field and metre per second squared [$\frac{\text{m}}{\text{s}^2}$] for the gravity (along the device axises). Fortunately, we are only interested in the orientation of these forces and not their strength. Hence, we could normalize the raw values (2.7).

The cross product of the gravity and the magnetic north results in a vector pointing to the earth's east. The cross product of the east pointing vector and the gravity leads to the a vector pointing to the earth's north (2.8). This enables us to build the rotation matrix out of the gravity of the east pointing vector and the north pointing vector (2.9) [Mik17].

The mathematical equations are shown in 2.7.

$$\begin{aligned} G &= \frac{1}{|g|} g \\ M &= \frac{1}{|m|} m \end{aligned} \tag{2.7}$$

$$\begin{aligned} G \times M &= E \\ E \times G &= N \end{aligned} \tag{2.8}$$

$$R = \begin{bmatrix} x_E & y_E & z_E \\ x_N & y_N & z_N \\ x_G & y_G & z_G \end{bmatrix} \tag{2.9}$$

where:

g:	Raw values from the gravity sensor
m:	Raw values from the magnetic field sensor
G:	Normalized gravity vector
M:	Normalized magnetic field vector
E:	East vector
N:	North vector
R:	Rotation matrix

Figure 2.7.: *equations to determine the rotation matrix*

To get a better understanding of these equations, let's have a look at an example. Assume we have the following measurement values (2.10):

$$g = \begin{bmatrix} -5.76 \\ 7.17 \\ -3.42 \end{bmatrix}, m = \begin{bmatrix} 19.38 \\ -15.54 \\ 41.58 \end{bmatrix} \tag{2.10}$$

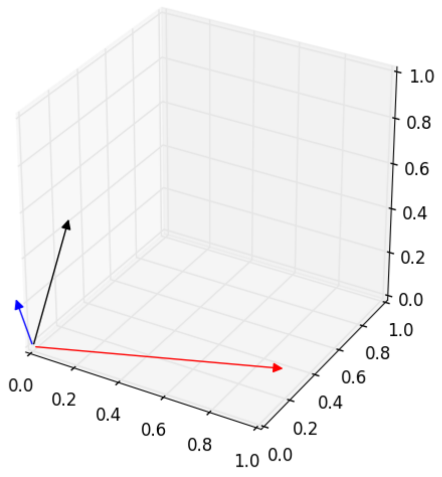
Now we are able to calculate the normalized vectors G and M (2.11).

$$\begin{aligned} G &= \frac{1}{\sqrt{-5.76^2 + 7.17^2 + -3.42^2}} \cdot \begin{bmatrix} -5.76 \\ 7.17 \\ -3.42 \end{bmatrix} = \begin{bmatrix} -0.59 \\ 0.73 \\ -0.35 \end{bmatrix} \\ M &= \frac{1}{\sqrt{19.38^2 + -15.54^2 + -41.58^2}} \cdot \begin{bmatrix} 19.38 \\ -15.54 \\ 41.58 \end{bmatrix} = \begin{bmatrix} 0.40 \\ -0.32 \\ 0.86 \end{bmatrix} \end{aligned} \tag{2.11}$$

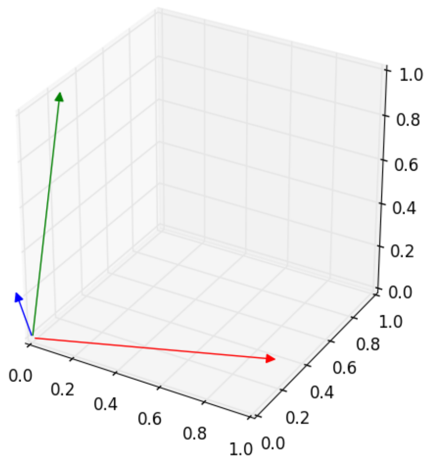
The corresponding vectors pointing to the earth's east and north can be computed as you can see at in equation (2.12).

$$\begin{aligned} E &= \begin{bmatrix} -0.59 \\ 0.73 \\ -0.35 \end{bmatrix} \times \begin{bmatrix} 0.40 \\ -0.32 \\ 0.86 \end{bmatrix} = \begin{bmatrix} 0.52 \\ 0.36 \\ -0.10 \end{bmatrix} \\ N &= \begin{bmatrix} 0.52 \\ 0.36 \\ -0.10 \end{bmatrix} \times \begin{bmatrix} -0.59 \\ 0.73 \\ -0.35 \end{bmatrix} = \begin{bmatrix} -0.05 \\ 0.24 \\ 0.59 \end{bmatrix} \end{aligned} \tag{2.12}$$

Figure 2.8 illustrates the determined vectors.



(a) Gravity (blue), Magnetic (black), East (red)



(b) East (red), Gravity (blue), north (green)

Figure 2.8.: *Cross products of the gravity and magnetic field sensor values resulting in the components for the rotation matrix*

2.4.2. Verification

To verify if the projection from the device axis aligned acceleration values to the Web Mercator (EPSG:3857 - WGS84) spatial reference system works as expected, two experiments were carried out.

Experiment 1

Construction The first experiment looks at the processed acceleration values (they should be relative to the Web Mercator projection). The acceleration in the z-axis that should result to the earth's gravity is particularly important. For that purpose, the smartphone is in a fix position (with indefinite rotation) illustrated in figure 2.9. Hence, the processed acceleration should approximately be $0.0\,\frac{m}{s^2}$ in the x-axis, $0.0\,\frac{m}{s^2}$ in the y-axis and $9.81\,\frac{m}{s^2}$ in the z-axis (gravity acceleration of the earth). Figure 2.9 represents the construction of the experiment.

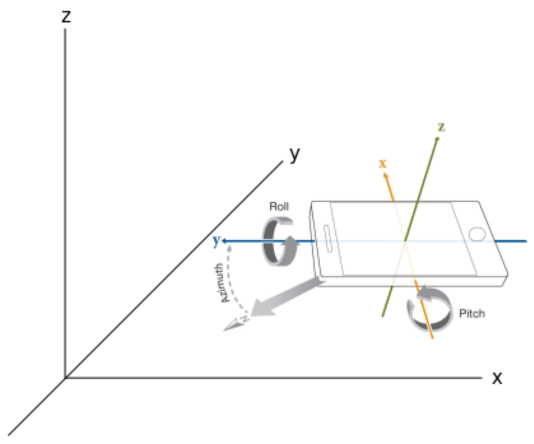


Figure 2.9.: *Verification experiment 1*

Results The tracked values of the raw acceleration and the processed ones verify the expected values. An excerpt of the measurements is shown in the table 2.2.

Raw acceleration			Processed acceleration		
x	y	z	x	x	z
-6.160	7.185	3.235	-0.021	-0.094	10.001
-6.153	7.204	3.220	-0.042	-0.106	10.006
-6.174	7.170	3.356	-0.023	-0.015	10.041
-6.174	7.173	3.416	-0.033	-0.069	10.062

Table 2.2.: *Measurements experiment 1*

Conclusion As you can see the values do have a certain shift related to measuring inaccuracies. To avoid this problem, an initial calibration should be done.

Experiment 2

Construction The second experiment observes the orientation of the acceleration of the x and y axises. The idea is to move the smartphone along a straight line, once in a vertical and once in a horizontal position, see figure 2.10. The expected values of the raw acceleration should differ in direction, but the processed values (relative to Web Mercator projection) should point in the same direction independently of the device orientation.

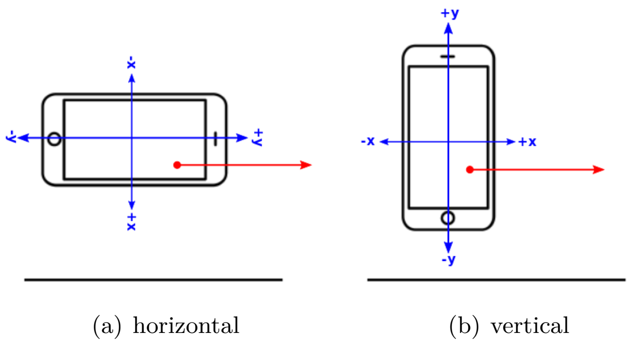


Figure 2.10.: *Experiment 2 visualization*

Results The logged values of the raw acceleration and processed ones verify the expected results. As you can see in figure 2.11 the processed values of both measurements (Vertical Acceleration Earth relative and Horizontal Acceleration Earth relative) point to the same direction, whereas the different directions are clearly recognizable between the Vertical acceleration and Vertical Acceleration Earth relative.

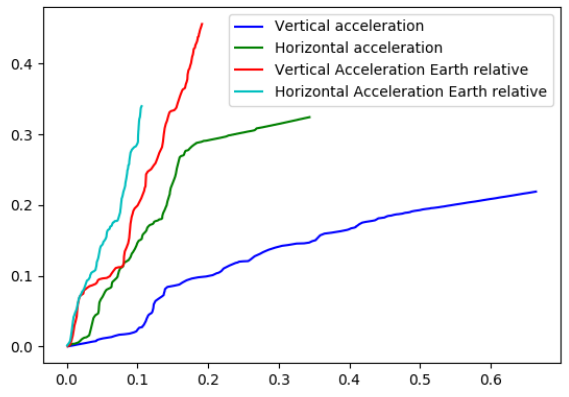


Figure 2.11.: *Verification experiment 2*

Conclusion The logged values verify the correctness of the reorienting of the acceleration, but also uncover some uncertainties in the measurement values.

2.4.3. Conclusion

The basic information to align the acceleration measurements relative to Web Mercator spatial reference system is projectable by the gravity and the magnetic field sensor of the smartphone. Furthermore, the Android SDK provides the functions to transform the values into the needed projection. Unfortunately, there are some pitfalls with that process like the sensor scattering [Kal17]. The magnetic sensor could be influenced by other magnetic fields. In addition, there is the problem of measuring the gravity when the device is in motion [17].

2.5. Sensor fusion

Problem A sticking point of the application is to determine the current position and the speed of the driver as exactly as possible. Since the GPS signal is not always accurate enough or even worse sometimes completely unavailable, the idea is to combine the multiple sensors of the smartphone.

Goal To use multiple sensors of the smarthphone to improve the measurement accuracy of the current position and the speed of the biker.

2.5.1. Approach

The chosen procedure in this project was to combine the accelerometer with the GPS sensor of the device. The sensor fusion is realized with a Kalman filter [Kal+60] [Sim01].

Kalman filter

The basic idea of a Kalman filter [RAH17] is to predict the future state of a linear system based on the previous ones. A linear system is a process described by a state equation (2.13) and an output equation (2.14).

$$x_k = F_{k-1}x_{k-1} + B_{k-1}u_{k-1} + w_{k-1} \tag{2.13}$$

$$y_k = H_k x_k + v_k \tag{2.14}$$

where:

x:	State vector
y:	Output vector
u:	Input vector
w:	Process noise vector
v:	Measurement noise vector
F:	State transition model matrix
B:	Control-input model matrix
H:	Observation model matrix

After the system is defined, we are able to predict the next measurement value with the following equation (2.16).

$$\hat{x}_{k|k-1} = F_{k-1}\hat{x}_{k-1} + B_{k-1}u_{k-1} \tag{2.15}$$

where:

$\hat{x}_{k|k-1}$: Predicted state vector

$\hat{x}_{k-1}$: Previous estimated state vector

And the state error covariance is determined by the equation (2.16)

$$P_{k|k-1} = F_{k-1} P_{k-1} F_{k-1}^T + Q_{k-1} \tag{2.16}$$

where:

$P_{k|k-1}$: Predicted state error covariance matrix
P_{k-1}: Previous estimated state error covariance matrix
Q: Process noise covariance matrix

With the predicted state error covariance matrix, we are able to calculate the Kalman gain (2.17).

$$K_k = P_{k|k-1} H_k^T (H_k P_{k|k-1} H_k^T + R_k)^{-1} \tag{2.17}$$

where:

H: Matrix to define the output equation
R: Measurement noise covariance
K: Kalman gain matrix

Finally, we can update the estimated state vector (2.18) and the state error covariance (2.19).

$$\hat{x}_k = \hat{x}_{k|k-1} + K_k (z_k - H_k \hat{x}_{k|k-1}) \tag{2.18}$$

$$P_k = (I - K_k H_k) P_{k|k-1} \tag{2.19}$$

where:

z_k: Measurement output
I: Identity matrix

Sensor fusion model

As already mentioned, we tried to combine the acceleration sensor and the GPS sensor to get a better position and velocity estimation. This is realized with a Kalman filter and the following section describes the used model.

State Vector x The state vector represents the values that will be estimated by the Kalman filter. In our case this is the position, the velocity and the acceleration.

$$x_k = \begin{bmatrix} x_p \\ y_p \\ x_v \\ y_v \\ x_a \\ x_a \end{bmatrix} \tag{2.20}$$

where:

x_p, y_p: Position
x_v, y_v: Velocity
x_a, y_a: Acceleration

State transition function The matrix F describes the state transition function which is related to Newton's laws of motion.

$$F = \begin{bmatrix} 1 & 0 & \Delta t & 0 & \frac{1}{2}\Delta t^2 & 0 \\ 0 & 1 & 0 & \Delta t & 0 & \frac{1}{2}\Delta t^2 \\ 0 & 0 & 1 & 0 & \Delta t & 0 \\ 0 & 0 & 0 & 1 & 0 & \Delta t \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.21}$$

where:

Δt: Time between filter steps

Process The process is described by the equation (2.22). The term Bu isn't relevant anymore because there is no control input.

$$x_k = F_{k-1} x_{k-1} \tag{2.22}$$

Measurment noise covariance R The noise covariance matrix R represents the squared noise of the acceleration and the position.

$$R = \begin{bmatrix} r_p & 0 & 0 & 0 \\ 0 & r_p & 0 & 0 \\ 0 & 0 & r_a & 0 \\ 0 & 0 & 0 & r_a \end{bmatrix} \tag{2.23}$$

where:

r_p: Squared acceleration noise
r_a: Squared position noise

Measurement matrix H The matrix H determines which states are measurable. In our case this is the acceleration and the position.

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.24}$$

State error covariance matrix P The matrix P defines the initial uncertainty and is going to be updated during the process.

$$P = \begin{bmatrix} r_p & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & r_p & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & r_v & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & r_v & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & r_a & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & r_a \end{bmatrix} \tag{2.25}$$

where:

r_v: Squared velocity noise

Process noise covariance matrix Q The matrix Q is related to the process noise. On our system, the acceleration could be influenced by potholes, wind, stones or other objects on the track. Thus, the process noise is $w_k = Ga_k$.

$$\begin{gathered} G = \begin{bmatrix} \frac{1}{2}\Delta t^2 & \frac{1}{2}\Delta t^2 & \Delta t & \Delta t & 1 & 1 \end{bmatrix} \\ Q = G^T G \sigma_a^2 \end{gathered} \tag{2.26}$$

where:

G Vector related to the motion
σ_a: Acceleration process noise

Example

To get an impression of what a Kalman filter does, imagine the following circumstances.

We measure the acceleration and position at a fix point and we don't move. Thus in a world with perfect measurement instruments we would get always the same position and no acceleration at all. Due to the fact that sensors are imperfect we have some inaccuracy in the values.

Assume the Kalman parameters are:

$$\begin{aligned} \Delta t &= 0.1 \\ r_p &= 10.0 \\ r_v &= 1.0 \\ r_a &= 0.25 \end{aligned} \tag{2.27}$$

The measured values are generated by a python script. The first three of them are listed in table 2.3.

Position x	-12.93	4.01	14.2
Position y	-28.15	-0.34	2.89
Acceleration x	0.87	-0.14	-0.24
Acceleration y	-0.36	0.65	-0.51

Table 2.3.: *Example measurements values*

Now we are able to set the initial values.

$$F = \begin{bmatrix} 1.0 & 0.0 & 0.1 & 0.0 & 0.005 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.1 & 0.0 & 0.005 \\ 0.0 & 0.0 & 1.0 & 0.0 & 0.1 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.1 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}, R = \begin{bmatrix} 100.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 100.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.25 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.25 \end{bmatrix}$$
$$P = \begin{bmatrix} 100.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 100.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.25 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.25 \end{bmatrix}, H = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$
$$Q = \begin{bmatrix} 2.5 \cdot 10^{-5} & 2.5 \cdot 10^{-5} & 0.0005 & 0.0005 & 0.005 & 0.005 \\ 2.5 \cdot 10^{-5} & 2.5 \cdot 10^{-5} & 0.0005 & 0.0005 & 0.005 & 0.005 \\ 0.0005 & 0.0005 & 0.01 & 0.01 & 0.1 & 0.1 \\ 0.0005 & 0.0005 & 0.01 & 0.01 & 0.1 & 0.1 \\ 0.005 & 0.005 & 0.1 & 0.1 & 1.0 & 1.0 \\ 0.005 & 0.005 & 0.1 & 0.1 & 1.0 & 1.0 \end{bmatrix} \tag{2.28}$$

After the initialization we can apply the Kalman filtering process at the measured values. This would result in the vector as you can see in (2.29) for the first three processing steps.

$$x_1 = \begin{bmatrix} -6.47 \\ -14.08 \\ 0.05 \\ -0.02 \\ 0.54 \\ -0.08 \end{bmatrix}, x_2 = \begin{bmatrix} -2.97 \\ -9.49 \\ 0.07 \\ 0.03 \\ 0.32 \\ 0.17 \end{bmatrix}, x_3 = \begin{bmatrix} 1.33 \\ -6.49 \\ 0.07 \\ 0.00 \\ -0.22 \\ -0.40 \end{bmatrix} \tag{2.29}$$

If we go ahead with this process for another 500 steps and observe the position, we can see that it goes towards zero in x and zero in y as we expected. Figure 2.12 shows the results of this process.

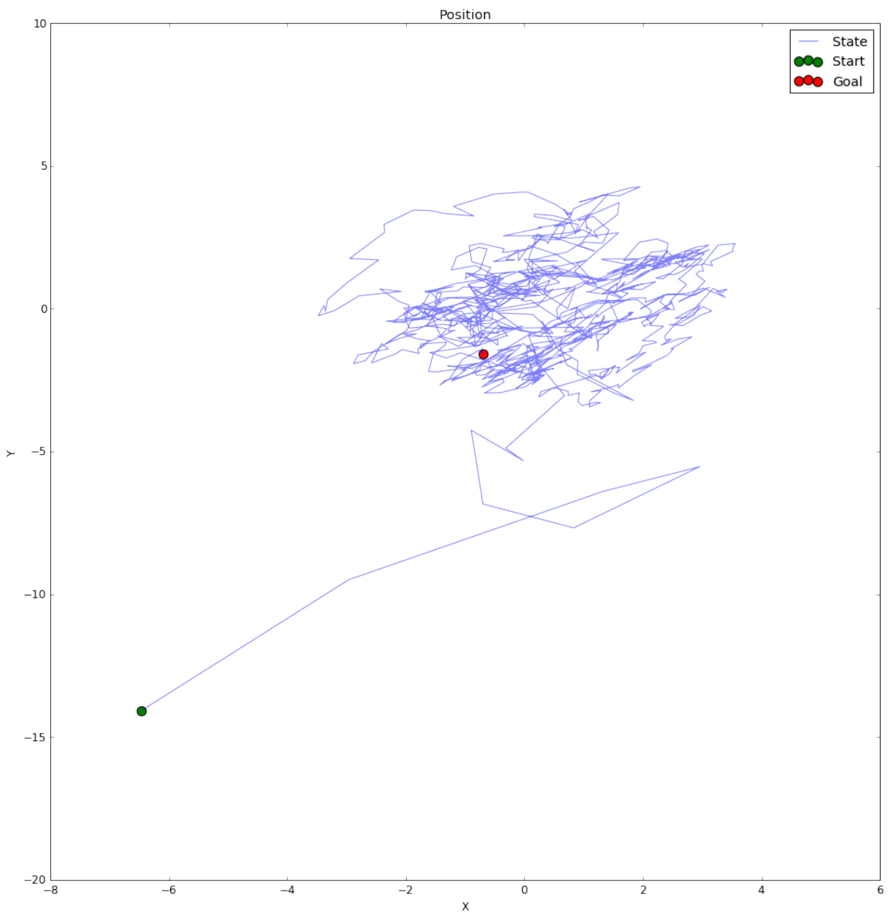


Figure 2.12.: *Verification experiment 1*

To reproduce the values and the gained result, use the script in D.1.

2.5.2. Conclusion

Using of a Kalman filter to enhance the measurement accuracy was an adequate approach. Since it allows the acceleration and the GPS values to fusion, it filters out measurement uncertainties and enables to refer to the underlying process.

Nevertheless, a Kalman filter does have some pitfalls too. For example, the modulation of a fitting process isn't always obvious. Time complexity raises in quadratic manner related to the number of parameters [Mon05]. To handle the noise parameters right you need a lot of experience and if the sensors are not accurate enough, a Kalman filter won't help either.

Hence, several attempts to enhance the measurement precision with the combination of the sensors and the usage of an Kalman filter didn't lead to sufficient results. Due to the fact that the sensors are not sensitive and accurate enough. Consequently, I decided to use an external bike speed sensor from Garmin.

2.6. External sensor

Problem The built-in sensors of the smartphone are not accurate enough to handle fast speed changes and to determine an adequate position.

Goal To improve the measurements to get sufficient position and speed values that will enable an suitable feedback for the biker.

Decision To improve the measurement accuracy, I used an external bike speed sensor from Garmin since they provide an API to the measurements directly from the smartphone.

2.6.1. Approach

The bike speed sensor from Garmin uses ANT+ technology to communicate with the smartphone. There is a instruction in how to access the sensor data and integrate it in your own Android application [Wir17e].

Installation

The basic steps to install and start using the ANT+ are:

1. Download the Android ANT+ SDK to communicate with ANT+ devices [Wir17d].
2. Add the SDK (antpluginlib_x-y-z.jar) to your Android Application libraries.
3. Install the ANT Radio Service App on your Android smartphone [Wir17c].
4. Install the ANT+ Plugins Service App on your Android smartphone [Wir17b].

Unfortunately, not all smartphones support the ANT+ protocol. To get an overview of all supporting smartphones, consider the list of compatible devices [Wir17f].

Development

The downloaded Android ANT+ SDK provides a document called "Creating ANT+ Android Applications". This document gives a brief overview and starting point of the ANT+ development with Android.

The procedure to use the ANT+ API is described as the following:

1. Search for devices
2. Request access to the plugin
3. Subscribe to events
4. Use device while monitoring the device connection state
5. Release the plugin communicator objects (called PCCs)

The documentation is not very detailed, luckily there are sample Android applications [Gal17] and API documentation [Wir17a]. To get more familiar with this procedure, let us take a look at some code snippets.

Search for devices

To get the devices you are interested in, you could use the class MultiDeviceSearch that takes the application context, a set of devices and a callback as parameters. An example of the device search is shown in listing 2.2. If a device is found, you have to ask for access as you can see in listing 2.3.

```
private void searchForDevices() {
        EnumSet<DeviceType> deviceTypes = EnumSet.of(DeviceType.BIKE_SPD);
        MultiDeviceSearch.SearchCallbacks searchCallbacks =
                                          new MultiDeviceSearch.SearchCallbacks() {

            @Override
            public void onSearchStarted(MultiDeviceSearch.RssiSupport rssiSupport) {
            }

            @Override
            public void onDeviceFound(MultiDeviceSearchResult multiDeviceSearchResult) {
                mMultiDeviceSearchResult = multiDeviceSearchResult;
                requestDeviceAccess(mMultiDeviceSearchResult);
            }

            @Override
            public void onSearchStopped(RequestAccessResult requestAccessResult) {
            }
        };
        MultiDeviceSearch _ =
                        new MultiDeviceSearch(context, deviceTypes, searchCallbacks);
    }
```

Listing 2.2: *Search for device listing*

Request access to the plugin

To get access, you could us the requestAccess method of the PCC object, concerned. Previously, you should check that you are using the sensor of your choice. An example is represented in listing 2.3.

```
    private void requestDeviceAccess(MultiDeviceSearchResult multiDeviceSearchResult) {
        boolean isBSC = multiDeviceSearchResult.getAntDeviceType()
                        .equals(DeviceType.BIKE_SPDCAD);
        AntPlusBikeSpeedDistancePcc.requestAccess(context,
                                    multiDeviceSearchResult.getAntDeviceNumber(), 0,
                                    isBSC, mResultReceiver, mDeviceStateChangeReceiver);
    }
```

Listing 2.3: *Request access listing*

Subscribe to events

The PCC objects allow to subscribe to events of the sensors. In listing 2.4 we use a AntPlusBikeSpeedDistancePcc related to our Bike Speed Sensor. This sensor needs the wheel circumference to calculate the speed. If a new value is provided by the sensor the onNewCaculatedSpeed method is called and you are able use the value for your needs.

```
    private void subscribeToEvents() {
        BigDecimal wheelCircumference = new BigDecimal(2.095) ; //average road tire
        AntPlusBikeSpeedDistancePcc.CalculatedSpeedReceiver speedReceiver =
            new AntPlusBikeSpeedDistancePcc.CalculatedSpeedReceiver(wheelCircumference)

        bsdPcc.subscribeCalculatedSpeedEvent(speedReceiver) {
            @Override
            public void onNewCalculatedSpeed(final long esTimestamp,
                                             final EnumSet<EventFlag> eventFlags,
                                             final BigDecimal calculatedSpeed) {
                setSpeed(calculatedSpeed.doubleValue());
            }
        });
    }
```

Listing 2.4: *Subscribe to events listing*

Use device while monitoring the device connection state

Device state changes could be handle by the DeviceStateChangeReceiver as displayed in listing 2.5. The handling of the measurements is done by a callback as shown in listing 2.4.

```
    private void monitorDeviceStateChanges(){
        mDeviceStateChangeReceiver = new AntPluginPcc.IDeviceStateChangeReceiver() {
            @Override
            public void onDeviceStateChange(final DeviceState newDeviceState) {
                Log.i(TAG, bsdPcc.getDeviceName() + ": " + newDeviceState);
                if (newDeviceState == DeviceState.DEAD) bsdPcc = null;
            }
        };
    }
```

Listing 2.5: *Monitoring the device connection state listing*

Release the plugin communicator objects

Finally, if you don't need the sensor data anymore, you could close the communication with the PccReleaseHandle. Illustrated in listing 2.6.

```
    private void destroy() {
        bsdReleaseHandle.close();
        if (bcReleaseHandle != null) {
            bcReleaseHandle.close();
        }
    }
```

Listing 2.6: *Release the plugin communicator objects listing*

2.6.2. Conclusion

The Garmin Bike speed sensor worked as expected and delivered accurate measurements. Due to the fact that the sensor only detects rotations and you have to set the circumference of the wheel on your own, measurement inaccuracies could appear. For example there could be more or less air in the tire. The documentation of the ANT API for developing an Android app was not very detailed. Furthermore, not all smartphones support the ANT protocol. Therefore, I had to change my development device from an Nexus X5 [Inc18] to a Samsung Galaxy S8 [SAM18]. Even with the adequate speed measurements, the exact position on the track is still missing.

It would be possible to use further external sensors to improve the measurement accuracy but this would have exceeded the scope of this project. Unfortunately, the more the external sensors are necessary the less likely it is that the Ranck app be used.

2.7. User interaction

Problem The section user interaction focuses on the interaction of the user with the Ranck application.

Goals The usage of the app should be self-explanatory. The features should be reduced to a minimum and the user should get the important information in an appropriate manner.

2.7.1. Approach

Mockups

To get a first impression of the look and feel of the Android application, I designed some mockups. They were created with the integrated designer of the Android Studio and Inkscape for image editing.

Initialization Figure 2.13 shows the initial screen. The progress bar should indicate that there is some work going on to get an acceptable GPS signal.

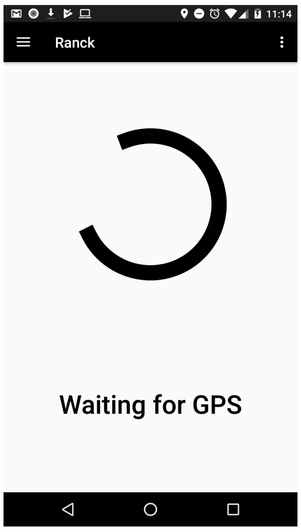
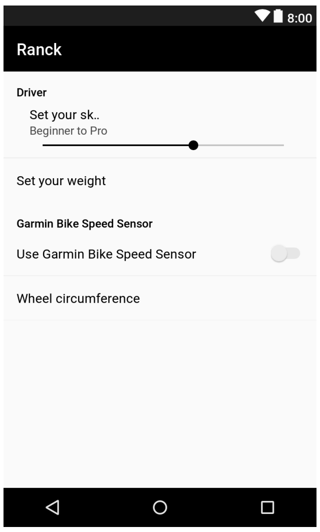


Figure 2.13.: *Waiting for GPS*

Settings The Ranck app has a settings screen to define the important preferences that are necessary to run the application. The goal was to use as few settings as possible in order to keep the configuration simple. The skill and the weight of the driver have to be set. Optionally, you could confirm the usage of a bike speed sensor and set the wheel circumference for that purpose.

Figure 2.14 illustrates the settings screen. The progress bar should indicate that there is some work going on to get an acceptable GPS signal.


Ranck

Driver

Set your sk..
Beginner to Pro

Set your weight

Garmin Bike Speed Sensor

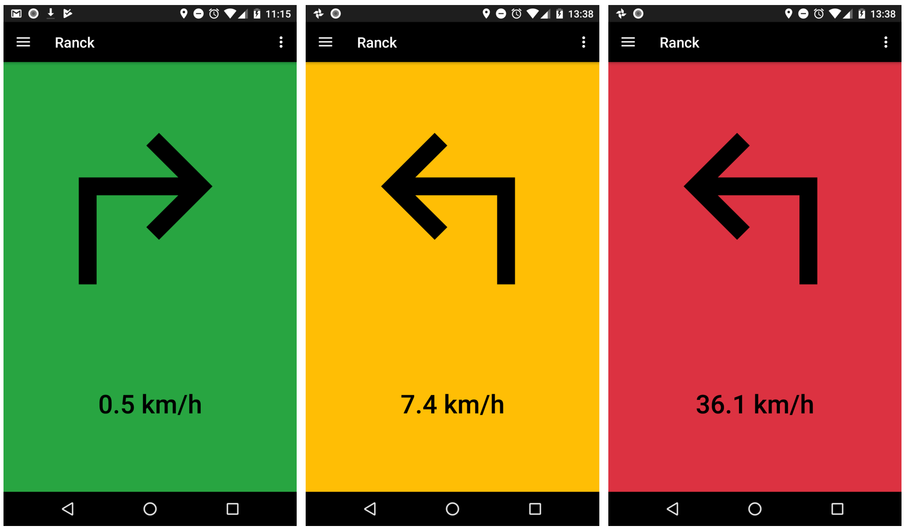
Use Garmin Bike Speed Sensor

Wheel circumference

Figure 2.14.: *Settings*

Biking

Visual Figure 2.15 shows the mockups for the application which are displayed during the biking phase. The main characteristics are the curve direction and the background colors. The color green indicates that everything is fine, orange signals that your velocity is critical and you should reduce it and finally if the background is red you are too fast for the next curve. The current speed is displayed at the bottom of the screen in $\frac{m}{s}$. The arrow indicates if the curve is left, right or straight and the angle of the arrow points out how tight the curve is.



(a) Green (b) Orange (c) Red

Figure 2.15.: *Mockups biking*

Sensing Despite the fact that the application might not be visible during the biking phase, the cyclist still likes some feedback. Hence, the Ranck app uses the vibration of the smartphone to warn the driver if he is too fast on the track.

User interface

The user interface (UI) uses the constraint layout of Android [Dev18b] and the images are vectorized. Thus, the UI is responsive and fits on devices with different resolutions.

2.7.2. Conclusion

The user interaction of the Rank application is kept as simple as possible. To fulfill that purpose, known signal colors from traffic lights were used. The arrows are self-explanatory and the information is kept to a minimum so that the cyclist is not distracted too much. If the smartphone is in a pocket during the ride, the vibration of the device is an accurate way to indicate that the driver is too fast.

3. Results

The project resulted in a collection of challenges and proposed solutions for them. The challenges related to the initial goals of the downhill supporting application are are listed in the chapter 2. Furthermore, a prototype of an Android application has been developed, which is able to determine your current speed and your position on the track as well as the sensor measurements allow. The app informs you if you are too fast for the next curve and displays the direction of the curve as well.

3.1. Procedure

The current section will give you a summary of the procedure of the project. The approach is strongly relate to the milestones as you can see in C.1.

The first tasks involved a rough planning of the project and the preparation of all the needed tools.

After the initial phase, the physical model was on the task board. The model serves as a basic module to determine if the current speed of the downhill biker is too high for the next curve or luckily not. If you are interested in the used parameters and the physics behind the model, have a look at section 2.1. The task resulted in an ready to use Java implementation of the physical model concerned.

Beside the sensor data, the tracks are an essential input of the application. The raw data of these tracks are GPS coordinates. Since we'd like to apply our physical model on these tracks, we first of all have to determine how a curve is represented by GPS points. For that purpose, we had to preprocess the track data as already mentioned in section 2.2. In addition, we had to choose an adequate spatial reference system as you can see in 2.3.

With the physical model and the preprocessed tracks, we were ready to build the shell of the Android application and stuck the components together. The missing parts were the position and the related speed determination, hence we could focus on these issues from now on.

The position and speed determination with the built-in sensors was a very difficult task and didn't lead to accurate enough results. The chosen approach is explained in detail in section 2.5. In short, we used a Kalman filter to fusion the acceleration and a GPS sensor to estimate the position and velocity as well as possible. The acceleration measurements had to be transformed from the axis system of the smartphone to the coordinate system of the earth. This is introduced in section 2.4.

The inadequate accuracy of the sensors lead to the usage of an external sensor. Fitting for our circumstances, was the bike speed sensor from Garmin. The integration of these sensor is listed in section 2.6. With the help of the external sensor, fast velocity changes could be detected and the position could be estimated if the GPS signal is lost. An exact determination of the position is still only possible to a certain degree and would need an additional, more accurate positioning sensor. Sadly, that was out of the scope of the Ranck project, thus the resulting prototype had to deal with the accuracy of the data from the smartphone and the bike speed sensor.

Finally, the work was documented and a promotional video of the prototype was recorded.

3.2. Conclusion

The project hosted several difficult and interesting problems. As we already expected before I started to work, the built-in sensors might not be accurate enough for reasonable results. Unfortunately, this issue confirms as true. The accuracy requirements in terms of position and speed determination such an application needs are only realizable with expensive external sensors. Even then it is an ambitious task. Nevertheless if you are confronted with problems which don't have out of the box solutions and the Ranck project had a lot of them, you learn the most. For example, I never had to deal with measurement uncertainties. That's why I first had to familiarize myself with the topic, test different approaches and verify the achieved results. Hence, I could apply a Kalman filter in practice and recycle some techniques I learnt during physic lessons. Since a lot of applications you are developing as a software engineer are largely abstract, it was a pleasure to deal with a program that has to interact with the environment. Next to the coding work, I had to go out and test the Ranck app. That was a lot of fun and I also uncoverd some missing bugs. The prototype combines the components and challenges into an Android application and additionally fulfils the requirement as well as the measurements of the sensors enable.

3.3. Outlook

The built-in sensors of smartphones aren't accurate enough and external sensors prevent the usage of an Application with requirements similar to the Ranck app. It could be a good idea to change the application requirement to the specific degree the built-in sensors are able to provide. For example, it is might not necessary to determine the maximal possible speed for every curve. It would be enough to set rough speed limits along the track to improve safety.

Glossary

ANT ANT and ANT + are proprietary wireless network standards for the 2.4 GHz ISM band. 25, 28

API An Application Programming Interface (API) provides an interface to an application from other parts of the application or even from an other software. 10, 25, 26, 28, 34

Azimuth Degrees of rotation around the z axis. The angle between the device's current compass direction and the magnetic north. 12

covariance Covariance provides a measure of the strength of the correlation between two or more sets of random variables. 19, 21

DGPS Differential Global Positioning System (DGPS) is an enhancement to Global Positioning System that provides improved location accuracy. 2

ECEF ECEF (earth-centered, earth-fixed) is a metric three dimensional spatial reference system. 10, 11

ECTS ECTS (European Credit Transfer and Accumulation System) credits are a european standard for comparing workload of higher education. 40

EPSG The EPSG (Geodetic Parameter Dataset) is a collection of definitions of coordinate reference systems and coordinate transformations. 10–12, 15, 38

Garmin Garmin Ltd. is a manufacturer of navigation receivers for satellite positioning and navigation. I, 24, 25, 28, 32, 38, 43, 45

GeoJSON GeoJSON is an open standard format designed for representing simple geographical features. 7–9, 38

GitLab GitLab is a web application for versioning software projects based on git. 37, 39

GPS The Global Positioning System (GPS) is a radionavigation system based in space. I, 12, 18, 20, 24, 29, 32, 33, 38–40

Kalman filter A Kalman filter tries to estimate the current state of a system based on previously done measurements and possible noises. I, 18, 20, 22–24, 32, 33, 38, 41, 42

PCC PCC are plug-in communicator objects of the ANT+ API. 25–27

Pitch Degrees of rotation around the x axis. The angle between a plane parallel to the device's screen and a plane parallel to the ground. 12

Roll Degrees of rotation around the y axis. The angle between a plane perpendicular to the device's screen and a plane perpendicular to the ground. 12

WGS84 World Geodetic System 1984 (WGS84) is a geodetic reference system that provides a unified basis for position information on earth and in near-earth space. 10–12, 15, 38

Appendices

A. Ranck Android app

The Code of the Ranck Android app is available on GitLab at the Ranck repository [Kur18b]. In addition to the application code, the repository provides the documentation as well.

The following section will give you an overview of the application architecture.

A.0.1. Package diagram

Beside the listed package on figure A.1, there are Android specific packages like the several resources which are not mentioned at this point since they are the basic ingredients of every Android application and don't vary a lot.

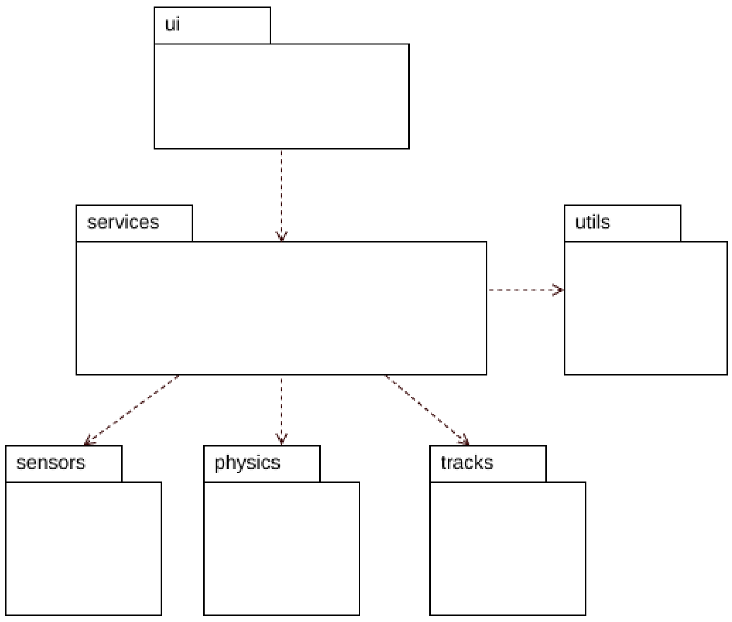


Figure A.1.: *Package Diagram*

ui

The ui package handles all interaction with the user, displays all necessary information for the user and is the main entry point for the application. Thus, it initializes the Ranck service.

services

The services package hosts the RanckService and RanckReceiver. The RanckService manages the interaction with the different sensors, the physical model and the tracks. To properly use the sensors, the RanckService is responsible for the sampling rate. The RanckReceiver handles the communication with the ui to provide up to date information.

sensors

The sensors package is responsible for the acceleration sensor, the location sensor and the bike speed sensor from Garmin. With the help of a Kalman filter, the measurement values are merged and the errors will be minimized.

physics

The physics package includes the physical model of the system, physical constants and determines if a driver is too fast for the next curve or not.

tracks

The tracks package provides a TrackReader that parses the GeoJSON track-files and provides Track and Curve classes for further usage.

utils

The package utils includes helper classes for the RanckService. There is a location estimator if no GPS signal is available and a location converter to transform the latitude/longitude coordinates (EPSG 4326 - WGS84) to the Web Mercoator projection (EPSG:3857 - WGS84) and vice versa.

B. Project Organization

The Ranck project was realized by Samuel Kurath with Prof. Dr. Farhad Mehta as advisor. During the project, there were several meetings about current state, planned tasks, problems that occurred and idea collection to solve them.

B.0.1. Milestones

For planing reasons, the project was split into the following milestones:

1. Project setup
2. Physical model
3. Extended GPS Tracks
4. Shell of App ready for velocity and position
5. External sensor usage
6. Accuracy and practicality verification
7. Finalize app and documentation

B.0.2. Tools

To organize the project, the tools listed in table B.1 were used.

Tool	Description
Landing Page	To get an overview of the project and fast access to all relevant project pages, a landing page was used [Kur18c].
GitLab	GitLab provided several tools for the project organization [Kur18b]: • Code repository • Documentation repository • Wiki • Project management • Time reporting

Table B.1.: *Used tools*

C. Time Report

The Ranck project started on 18. September 2017 and ended on 14. February 2018. It lasted a total of 22 weeks.

The time allotment structure can be seen in table C.1.

Milestone	Start	End	Number of days	Time spend [h]
Project setup	18.09.2017	26.09.2017	8	47,5
Physical model	27.09.2017	03.10.2017	6	28
Extended GPS Tracks	04.10.2017	17.10.2017	13	43
Shell of App ready for velocity and position	18.10.2017	14.11.2017	27	47,5
External sensor usage	15.11.2017	21.11.2017	6	26
Accuracy and practicality verification	22.11.2017	15.12.2017	23	57,5
Finalize app and documentation	16.12.2017	14.02.2018	60	140
Total				389.5

Table C.1.: *Working time*

The proposed working time for a PA (Projektarbeit / term project) is 360 hours. This is related to the 12 ECTS points you gain if you succeed.

D. Code snippets

D.1. Sensor fusion

The following script D.1 shows an example of a Kalman filter fusioning the acceleration and position measurements.

```python
from sympy import Matrix, eye
import numpy as np

number_of_steps = 500

# parameters
dt = 0.1

squared_acceleration_noise = 0.5 ** 2
squared_position_noise = 10.0 ** 2
squared_velocity_noise = 1.0 ** 2

# initialization
F = Matrix([[1.0, 0.0, dt, 0.0, 1 / 2.0 * dt ** 2, 0.0],
            [0.0, 1.0, 0.0, dt, 0.0, 1 / 2.0 * dt ** 2],
            [0.0, 0.0, 1.0, 0.0, dt, 0.0],
            [0.0, 0.0, 0.0, 1.0, 0.0, dt],
            [0.0, 0.0, 0.0, 0.0, 1.0, 0.0],
            [0.0, 0.0, 0.0, 0.0, 0.0, 1.0]]).reshape(6, 6)

R = Matrix([[squared_position_noise, 0.0, 0.0, 0.0],
            [0.0, squared_position_noise, 0.0, 0.0],
            [0.0, 0.0, squared_acceleration_noise, 0.0],
            [0.0, 0.0, 0.0, squared_acceleration_noise]]).reshape(4, 4)

P = Matrix([[squared_position_noise, 0., 0., 0., 0., 0.],
            [0., squared_position_noise, 0., 0., 0., 0.],
            [0., 0., squared_velocity_noise, 0., 0., 0.],
            [0., 0., 0., squared_velocity_noise, 0., 0.],
            [0., 0., 0., 0., squared_acceleration_noise, 0.],
            [0., 0., 0., 0., 0., squared_acceleration_noise]]).reshape(6, 6)

H = Matrix([[1.0, 0.0, 0.0, 0.0, 0.0, 0.0],
            [0.0, 1.0, 0.0, 0.0, 0.0, 0.0],
            [0.0, 0.0, 0.0, 0.0, 1.0, 0.0],
            [0.0, 0.0, 0.0, 0.0, 0.0, 1.0]]).reshape(4, 6)

acceleration_process_noise = 0.001
G = Matrix([1 / 2.0 * dt ** 2, 1 / 2.0 * dt ** 2, dt, dt, 1., 1.]).reshape(6, 1)
Q = G * G.T

I = eye(6)

x = Matrix([0., 0., 0., 0., 0., 0.]).reshape(6, 1)

# generate data
np.random.seed(seed=11)
np.set_printoptions(precision=2)
noise_acceleration = np.random.normal(0, 0.5, 2 * number_of_steps)
noise_position = np.random.normal(0, 10.0, 2 * number_of_steps)
```

```
positions = [(noise_position[i], noise_position[i + number_of_steps])
             for i in range(0, number_of_steps)]
accelerations = [(noise_acceleration[i], noise_acceleration[i + number_of_steps])
                 for i in range(0, number_of_steps)]

results = []

# kalman processing
for i in range(number_of_steps):
    z = Matrix([positions[i][0], positions[i][1],
                accelerations[i][0], accelerations[i][1]]).reshape(4, 1)
    x = F * x
    P = F * P * F.T + Q
    K = P * H.T * (H * P * H.T + R).inv()
    x = x + K * (z - H * x)
    P = (I - K * H) * P
    results.append(x)
```

Listing D.1: *Example Kalman filter*

D.2. Bike speed sensor

The listing D.2 shows a working example to interact with the Bike speed sensor from Garmin.

```java

import android.content.Context;
import android.content.SharedPreferences;
import android.preference.PreferenceManager;
import android.util.Log;

import com.dsi.ant.plugins.antplus.pcc.AntPlusBikeCadencePcc;
import com.dsi.ant.plugins.antplus.pcc.AntPlusBikeSpeedDistancePcc;
import com.dsi.ant.plugins.antplus.pcc.MultiDeviceSearch;
import com.dsi.ant.plugins.antplus.pcc.defines.DeviceState;
import com.dsi.ant.plugins.antplus.pcc.defines.DeviceType;
import com.dsi.ant.plugins.antplus.pcc.defines.EventFlag;
import com.dsi.ant.plugins.antplus.pcc.defines.RequestAccessResult;
import com.dsi.ant.plugins.antplus.pccbase.AntPluginPcc;
import com.dsi.ant.plugins.antplus.pccbase.PccReleaseHandle;
import com.dsi.ant.plugins.antplus.pccbase.MultiDeviceSearch.MultiDeviceSearchResult;
import com.murthy.ranck.R;

import java.math.BigDecimal;
import java.util.EnumSet;

public class BikeSpeedSensor {
    private final String TAG = this.getClass().getSimpleName();
    private SharedPreferences preferences;
    private Context context;
    private MultiDeviceSearchResult mMultiDeviceSearchResult;
    private AntPlusBikeSpeedDistancePcc bsdPcc = null;
    private PccReleaseHandle<AntPlusBikeSpeedDistancePcc> bsdReleaseHandle = null;
    private PccReleaseHandle<AntPlusBikeCadencePcc> bcReleaseHandle = null;
    private AntPluginPcc.IPluginAccessResultReceiver<AntPlusBikeSpeedDistancePcc>
            mResultReceiver;
    private AntPluginPcc.IDeviceStateChangeReceiver mDeviceStateChangeReceiver;

    private double mSpeed;
    private double distance;

    public BikeSpeedSensor(Context context) {
        this.context = context;
        preferences = PreferenceManager.getDefaultSharedPreferences(context);
        initReceiver();
        searchForDevices();
    }


    private void resetPcc() {
        //Release the old access if it exists
        if (bsdReleaseHandle != null) {
            bsdReleaseHandle.close();
        }
        if (bcReleaseHandle != null) {
            bcReleaseHandle.close();
        }
    }

    protected void destroy() {
        bsdReleaseHandle.close();
        if (bcReleaseHandle != null) {
            bcReleaseHandle.close();
        }
    }


    private void searchForDevices() {
        EnumSet<DeviceType> deviceTypes = EnumSet.of(DeviceType.BIKE_SPD);
        MultiDeviceSearch.SearchCallbacks searchCallbacks =
                                          new MultiDeviceSearch.SearchCallbacks() {
            @Override
            public void onSearchStarted(MultiDeviceSearch.RssiSupport rssiSupport) {
                Log.i(TAG, "Search stared");
            }

            @Override
            public void onDeviceFound(MultiDeviceSearchResult multiDeviceSearchResult) {
                mMultiDeviceSearchResult = multiDeviceSearchResult;
                requestDeviceAccess(mMultiDeviceSearchResult);
            }

            @Override
            public void onSearchStopped(RequestAccessResult requestAccessResult) {
                Log.i(TAG, "Search stopped");
            }
        };
        MultiDeviceSearch multiDeviceSearch =
                          new MultiDeviceSearch(context, deviceTypes, searchCallbacks);
    }

    private void requestDeviceAccess(MultiDeviceSearchResult multiDeviceSearchResult) {
        boolean isBSC = multiDeviceSearchResult.getAntDeviceType()
                        .equals(DeviceType.BIKE_SPDCAD);
        AntPlusBikeSpeedDistancePcc.requestAccess(context,
                                    multiDeviceSearchResult.getAntDeviceNumber(), 0,
                                    isBSC, mResultReceiver, mDeviceStateChangeReceiver);
    }

    private void subscribeToEvents() {
        BigDecimal wheelCircumference = new BigDecimal(2.095) ; //average road tire
        AntPlusBikeSpeedDistancePcc.CalculatedSpeedReceiver speedReceiver =
            new AntPlusBikeSpeedDistancePcc.CalculatedSpeedReceiver(wheelCircumference)

        bsdPcc.subscribeCalculatedSpeedEvent(speedReceiver) {
            @Override
            public void onNewCalculatedSpeed(final long esTimestamp,
                                             final EnumSet<EventFlag> eventFlags,
                                             final BigDecimal calculatedSpeed) {
                setSpeed(calculatedSpeed.doubleValue());
            }
        });
    }

    private void initReceiver() {
        mResultReceiver = new AntPluginPcc
                          .IPluginAccessResultReceiver<AntPlusBikeSpeedDistancePcc>() {
            @Override
            public void onResultReceived(AntPlusBikeSpeedDistancePcc result,
                                         RequestAccessResult resultCode,
                                         DeviceState initialDeviceState) {
                switch (resultCode) {
                    case SUCCESS:
                        bsdPcc = result;
                        Log.i(TAG, result.getDeviceName() + ": " + initialDeviceState);
                        subscribeToEvents();
                        break;
                    case CHANNEL_NOT_AVAILABLE:
                        Log.e(TAG, "Channel Not Available");
                        break;
                    case ADAPTER_NOT_DETECTED:
                        Log.e(TAG, "ANT Adapter Not Available.");
                        break;
                    case BAD_PARAMS:
                        // Note: Since we compose all the params ourself, we should
```

```
                            // never see this result
                            Log.e(TAG, "Bad request parameters.");
                            break;
                        case OTHER_FAILURE:
                            Log.e(TAG, "RequestAccess failed. See logcat for details.");
                            break;
                        case DEPENDENCY_NOT_INSTALLED:
                            Log.e(TAG, "Missing Dependency");
                            break;
                        case USER_CANCELLED:
                            break;
                        case UNRECOGNIZED:
                            Log.e(TAG, "Failed: UNRECOGNIZED. PluginLib Upgrade Required?");
                            break;
                        default:
                            Log.e(TAG, "Unrecognized result: " + resultCode);
                            break;
                    }
                }
            };

            // Receives state changes and shows it on the status display line
            monitorDeviceStateChanges();
        }

        private void monitorDeviceStateChanges(){
            mDeviceStateChangeReceiver = new AntPluginPcc.IDeviceStateChangeReceiver() {
                @Override
                public void onDeviceStateChange(final DeviceState newDeviceState) {
                    Log.i(TAG, bsdPcc.getDeviceName() + ": " + newDeviceState);
                    if (newDeviceState == DeviceState.DEAD) bsdPcc = null;
                }
            };
        }

        public double getSpeed() {
            return this.mSpeed;
        }

        private void setSpeed(double mSpeed) {
            this.mSpeed = mSpeed;
        }

        private void setDistance(double distance) {
            this.distance = distance;
        }
    }
```

Listing D.2: *Garmin Bike speed sensor listing*

Bibliography

[17] *getRotationMatrix*. Dec. 2017. URL: https://developer.android.com/reference/android/hardware/SensorManager.html#getRotationMatrix(float[],%20float[],%20float[],%20float[]).

[Dev18a] Android Developers. *Android Oreo*. Jan. 2018. URL: https://developer.android.com/index.html.

[Dev18b] Android Developers. *Build a Responsive UI with ConstraintLayout*. Jan. 2018. URL: https://developer.android.com/training/constraint-layout/index.html.

[Fra17] Adam Franco. *Technology*. Dec. 2017. URL: http://roadcurvature.com/technology/.

[Gal17] Shane Gallup. *Android ANT+ SDK Samples*. Dec. 2017. URL: https://github.com/ant-wireless/ANT-Android-SDKs/tree/master/ANT%2B_Android_SDK/Sample.

[Har17] Björn Harrtell. *jts2geojson*. Dec. 2017. URL: https://github.com/bjornharrtell/jts2geojson.

[Inc18] Google Inc. *5X*. Jan. 2018. URL: https://www.google.com/intl/de_ch/nexus/5x/.

[IOG18] IOGP. *About the EPSG Dataset*. Jan. 2018. URL: http://www.epsg.org/.

[Kal+60] Rudolph Emil Kalman et al. "A new approach to linear filtering and prediction problems". In: *Journal of basic Engineering* 82.1 (1960), pp. 35–45.

[Kal17] KalebKE. *FSensor*. Dec. 2017. URL: https://github.com/KalebKE/FSensor.

[Kri18] Krishnavedala. *ECEF*. Jan. 2018. URL: https://en.wikipedia.org/wiki/ECEF.

[Kur18a] Samuel Kurath. *Ranck*. Jan. 2018. URL: https://gitlab.com/Murthy10/track_processing.

[Kur18b] Samuel Kurath. *Ranck*. Jan. 2018. URL: https://gitlab.com/Murthy10/ranck.

[Kur18c] Samuel Kurath. *Ranck*. Jan. 2018. URL: https://samuelkurath.ch/ranck.

[Map17] Mapbox. *geojson.io*. Dec. 2017. URL: http://geojson.io.

[Mik17] Mike-Stanley. *Simple 3-axis and 6-axis Algorithms*. Dec. 2017. URL: https://github.com/memsindustrygroup/Open-Source-Sensor-Fusion/wiki/simple_algorithms.

[Mon05] Corey Montella. "The Kalman Filter and Related Algorithms: A Literature Review". In: (May 5).

[RAH17] Matthew Rhudy, Roger A Salguero, and Keaton Holappa. "A Kalman Filtering Tutorial for Undergraduate Students". In: 08 (Feb. 2017), pp. 01–18.

[SAM18] SAMSUNG. *Galaxy S8 / S8+*. Jan. 2018. URL: `http://www.samsung.com/ch/smartphones/galaxy-s8/`.

[Sim01] Dan Simon. "Kalman filtering". In: *Embedded systems programming* 14.6 (2001), pp. 72–79.

[Tea17] MathWorks Mobile Sensor Connectivity Team. *Capturing Azimuth, Pitch, and Roll Example*. Dec. 2017. URL: `https://www.mathworks.com/examples/matlab/community/20183-capturing-azimuth-pitch-and-roll-example`.

[TM18] James Talmage and Damon Maneice. *Mercator maps: Use and criticism*. Jan. 2018. URL: `https://kaiserscience.wordpress.com/earth-science/maps/mercator-maps-use-and-criticism/`.

[Tob17] Toblerity. *Shapely*. Dec. 2017. URL: `https://github.com/Toblerity/Shapely`.

[Too18] Engineering ToolBox. *Rolling Resistance*. Jan. 2018. URL: `https://www.engineeringtoolbox.com/rolling-friction-resistance-d_1303.html`.

[War18] Frank Warmerdam. *PROJ.4*. Jan. 2018. URL: `http://proj4.org/`.

[Wir17a] ANT Wireless. *ANDROID ANT+ API*. Dec. 2017. URL: `https://www.thisisant.com/APIassets/Android_ANT_plus_plugins_API/`.

[Wir17b] ANT Wireless. *ANT+ Plugins Service*. Dec. 2017. URL: `https://play.google.com/store/apps/details?id=com.dsi.ant.plugins.antplus`.

[Wir17c] ANT Wireless. *ANT Radio Service*. Dec. 2017. URL: `https://play.google.com/store/apps/details?id=com.dsi.ant.service.socket`.

[Wir17d] ANT Wireless. *DOWNLOADS: SOFTWARE AND DOCUMENTS*. Dec. 2017. URL: `https://www.thisisant.com/developer/resources/downloads`.

[Wir17e] ANT Wireless. *STARTING YOUR PROJECT*. Dec. 2017. URL: `https://www.thisisant.com/developer/ant/starting-your-project#75_tab`.

[Wir17f] ANT Wireless. *TOP BRANDS. COMPATIBLE DEVICES*. Dec. 2017. URL: `https://www.thisisant.com/directory`.