Smartphone-based tracking of rowing training

AQA Computer Science A-Level

Non-Examined Assessment

Timothy Langer

St Paul's School

Dr C. A. Harrison

25th March 2022

Contents

1 | Introduction

Rowing in its modern form developed in England at the start of the 18th century, with the first recorded race held in 1715. Nowadays, it is particularly popular in the UK and US. The Boat Race on television has over seven million viewers, with a further 250,000 attending in person.[1]

Two kinds of competitions exist: head races and regattas. A head race is one where competitors compete for the fastest time take to row a given distance. The outcome of the race is not clear until the race is over and times for every boat have been compiled. A regatta involves side-by-side racing, across several lanes, usually over a course of 2000m. Head races are typically of significantly longer distance (School's Head is roughly 7km, for exmaple) and so pacing is particularly important.

1.1 The rowing stroke

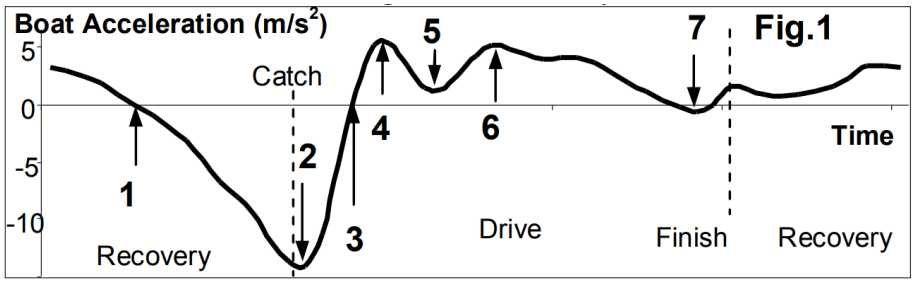

Figure 1.1: Typical pattern of boat acceleration during the stroke cycle [2]

Rowing (and/or sculling) involves the propulsion of a racing shell[1] through the water, using either one or two oars. Rowing is cyclic and repetitive in its nature; every stroke is alike. The rowing stroke consists of two phases: the *drive* and the *recovery*. During the drive, the athelete pushes with their legs, moving towards the bow of the boat, with the oars in the water, thus accelerating the boat. In the recovery, the athlete slides to the rear of the boat, while the boat slightly decelerates. Figure 1.1 shows a typical acceleration pattern of a single stroke.

This acceleration pattern repeats with little fluctuation; thus one can determine the period with which it repeats and calculate the number of strokes being taken per minute. By combining this data with GPS readings, one can also calculate the distance the boat travels per stroke.

[1] the long, light and narrow boat used for rowing

2 | Analysis

2.1 Identified issue

A critical part of training is the **stroke rate**, which measures the number of strokes taken per minute of rowing. A higher stroke rate, with the same technique and power application, allows for more overall power to be applied per minute and thus move the boat faster. However, achieving a higher rate is difficult as the stroke length tends to shorten to compensate, especially as the rower fatigues. As such, many workout pieces are capped at a specified stroke rate, and it is necessary for the strokeman[1] and the cox[2] to know the stroke rate of the boat and adapt the rowing stroke appropriately.

Another important part of training is the **split**, which measures the time taken, in minutes, to cover 500m; in essence, it is the speed of the boat. Although the speed of the boat will vary with conditions such as wind and current, over the length of the course relative changes to the split during a rowing piece are good indicators of performance.

2.2 Existing solutions

2.2.1 Simple solution

Average stroke rate can be measured by counting the number of strokes taken in a minute. All that is necessary is a stopwatch. This is a simple solution, but it is not very accurate. Additionally, the coxswain should be focusing on steering, not counting strokes!

The split of the boat can be calculated by manually measuring the time taken to cover a known distance of water. This solution makes it difficult to account for changes to the steering line, and does not allow for realtime feedback of the boat's speed.

2.2.2 Rowing stopwatches

The 1960s saw the emergence of specialised rowing stopwatches. Figure 2.1 shows a rowing stopwatch manufactured by Heuer-Leonidas. These had a 1/10th second resolution and were popular until the end of the 1980s. Repeated depressions of the crown at the same instant in each stroke provided a fully mechanical way of calculating the stroke rate.

2.2.3 Nielsen-Kellerman StrokeCoach and SpeedCoach

The first NK stroke rate meter was released in 1984, known as the Chronostroke. This was an electronic device that doubled as a stopwatch and allowed the cox or the coxswain to calculate the stroke rate. The

[1] the oarsman closest to the stern of the boat
[2] short for coxswain, the person steering the boat

Figure 2.1: A rowing stopwatch from the 1970-71 Heuer catalogue [3]

Figure 2.2: On the left is the NK SpeedCoach; on the right is a screenshot of the NK Logbook application synchronising 24 sessions off the SpeedCoach

1990s saw the first PaceCoach, which coupled with inboard and outboard sensors such as an impeller to calculate distance covered and real-time split.[4]

The NK SpeedCoach GPS (Model 2) was released in 2012 and is a popular performance monitor for rowing on the water. It displays realtime information including split, stroke rate and the time and distance rowed. The first model used a physical magnet installed in the boat to record stroke rate, however, the second edition uses an inbuilt accelerometer for this and is completely wireless.

An optional *Training Pack* is available with Model 2 that adds software features such as Bluetooth support for heart rate monitor and synchronising training data to a smartphone.

The basic edition of the second model retails for £399; the *Training Pack* is another £99.

Although the NK SpeedCoach is a popular rowing performance monitor, its price is certainly a barrier to entry.

In addition, getting recorded data off the device is difficult, as Bluetooth connection is not reliable and requires synchronisation of *all* the sessions on the stroke coach to obtain just the latest one.

2.3 General solution statement

The proposed solution, given the prevalence of smartphones in the modern world is a smartphone application that provides roughly equivalent core functionality to that of the commonplace NK SpeedCoach, including stroke rate, split and distance covered.

2.4 Identified end user

The end user is the coxswain or any of the rowers in the boat. In particular, the lower cost and availability of second-hand waterproof smartphones in comparison to the NK SpeedCoach would allow other rowers in the boat and not only the strokeman to be made aware of the rate. This solution would not be suitable for a coach in an adjacent boat, however, unlike a rowing stopwatch.

2.5 Interviews

2.5.1 Interview with a rower

An interview was conducted with the Captain of SPSBC, Sebastian Marsoner.
His answers have been paraphrased.

1. What would you expect from a stroke coach application?

   I would expect a stroke coach application to perform its most important task, determining the stroke rate, as accurately as possible. I have tried two different applications in an attempt to track my rowing, but neither of them could actually accurately tell me my stroke rate – it seemed implausible, and as such I purchased a Nielsen-Kellerman SpeedCoach.

2. What features do you use on the SpeedCoach?

   Usually I set it up to display my split, stroke rate, distance and heart rate.

3. Do you feel the SpeedCoach solution is lacking in any areas?

   Firstly, I would like to say that the SpeedCoach is great at what it does, although I am aware that it is rather expensive. Also, it has a rather limited memory and it is difficult to download the rowing sessions off the SpeedCoach. I would like to have a way to store all of my rowing sessions over time, so that I can compare a recent session with one from a previous year, for example, to better understand my progress.

2.5.2 Interview with a coxswain

An interview was conducted with the SPSBC 1st VIII cox, James Trotman.
His answers have been paraphrased.

1. What do you look for in a stroke coach?

   If I was to buy a stroke coach I would want it to

   - be **accurate** so that the stroke rate and splits can be acted upon in realtime,
   - be **cheaper** than currently available products such as the NK SpeedCoach,
   - be **easy to use** so that as much time as possible is dedicated to training,
   - integrate with other existing software to simplify the post-session review process.

2. You are familiar with the NK SpeedCoach and use it in your daily outings.
   Which of its metrics do you use?

   The NK SpeedCoach has a four-panel display, so usually I set it up to display a stopwatch, the stroke rate, split and distance covered. In a race I usually swap the distance covered for distance per stroke but even then I rarely use it [the distance per stroke metric]. As a cox I do not need to see my heart rate and so do not use this feature.

3. Is there any functionality that you feel the NK SpeedCoach is lacking?

   You can export rowing sessions off the SpeedCoaches, but it is not a simple task. Their app is buggy and the Bluetooth connection between my phone and the SpeedCoach is not always reliable. Also the synchronisation process requires one to download *all* the sessions off the SpeedCoach even if you are only interested in the latest one.

   This might be harder to implement, but it would also be amazing to have power curve analysis like on the RP3 on the water, and perhaps other metrics such as check[3] or ratio[4].

4. You have also examined the data produced by the BioRow in-boat telemetry system.
   Which of those additional metrics provided by BioRow you found useful?

   We found the graph of boat acceleration somewhat useful, as we were told that our boat acceleration at the catch is too shallow. However, it was not made clear what the optimal shape of the graph should look like. The *verticle angle + boat roll* metric once again indicated to us that the boat was not level during our session, but we could easily tell that this was the case even before the telemetry system was installed.

[3] the sudden dip in boat velocity at the point when the oar enters the water
[4] refers to the relationship between the amount of time spent on the drive and the recovery

2.6 Specific solution statement

A smartphone application is to be designed; one that will inform the user of the stroke rate and split and split of the boat in real time. It will also show the elapsed time and the distance covered during a rowing session. Rowing sessions can be recorded for later viewing.

Most people already have a smartphone with an accelerometer and a GPS sensor, which can be used to obtain data about the boat. In addition, since the tracking would be taking place on the phone itself, the need for synchronising sessions between a phone and a stroke coach would be eliminated.

2.6.1 Practical considerations

Although increasingly common, not all smartphones have a sufficient ingress protection rating to be comfortably taken into the boat. It is necessary to consider the requirement of a waterproof case, clamp or other holder that would allow the device to be affixed securely to the boat and prevent water damage in case of rain, waves or capsizing. One example would be a waterproof jogging armband: these are inexpensive, can be wrapped around a wing rigger [5] and are a low barrier to entry.

2.6.2 Device hardware

Almost all handheld Android smartphones have an accelerometer, since this is necessary for the screen auto-rotation feature, and many also feature either a gyroscope or a magnetometer.[5] The key component for this project is the accelerometer, however, the gyroscope and magnetometer could be used to improve the accuracy of the data.

The top 1822 most popular Android devices on Geekbench, a popular benchmarking tool, have an average performance of 1018, scoring slightly higher than a desktop Intel® Core™i3-8100 CPU. [6] However, we cannot rely on such performance. If this application is to appeal to rowing clubs as a viable alternative to the NK SpeedCoach, it cannot require an equally-priced smartphone to run! Therefore the core features of this app must be optimised to perform well on lower-end devices. Efficient algorithms reduce strain on the CPU and thus make the process more battery-efficient.

2.6.3 Programming languages and frameworks

There are three primary supported programming languages for developing Android apps: C++, Java and Kotlin. Kotlin is a modern statically typed programming language used by over 60% of professional Android developers. It is selected for this project due to the following advantages:

- Kotlin is cross-compatible with Java code, so although the language is relatively new, any libraries and frameworks developed for Java will work with Kotlin.
- Kotlin has many modern language features that draw inspiration from functional programming and allow for more concise code.
- Kotlin's coroutine functionality makes asynchronous programming simple and more efficient.

[5] modern version of an outrigger that spans across the middle from one saxboard to the other

- Kotlin makes handling nullability of variables and objects far easier, to help avoid Java's dreaded `NullPointerException`.

(From the *Android Developers* website [7])

2.6.4 Specification

A detailed specification is defined here that will be used to define the resulting application and the underlying technology.

The program **must**:

1. be an application installable on any modern[6] Android smartphone,
2. require minimal configuration and interaction from the end user, and ideally none at all,
3. be as easy to use and self-explanatory as possible for a new user,
4. calculate the following metrics, as accurately as possible:
   - the boat's stroke rate,
   - the boat's split,
   - the total distance rowed over the course of the session,
   - the elapsed total session time,
5. display the above metrics in a realtime and an easy-to-read manner,
6. collect data during a rowing session using nothing but the hardware on the smartphone,
7. provide start/stop functionality to record the rowing session for later review,
8. allow the user to export the recorded rowing session as a `FIT` activity file that can be imported into 3rd-party software, such as Strava, containing
   - the geolocation of the boat,
   - the speed of the boat,
   - the cadence, or stroke rate, of the rowers,
   - the estimated power generated by the rowers
9. be as battery-efficient as possible

[6] released within the last five years

3 | Design

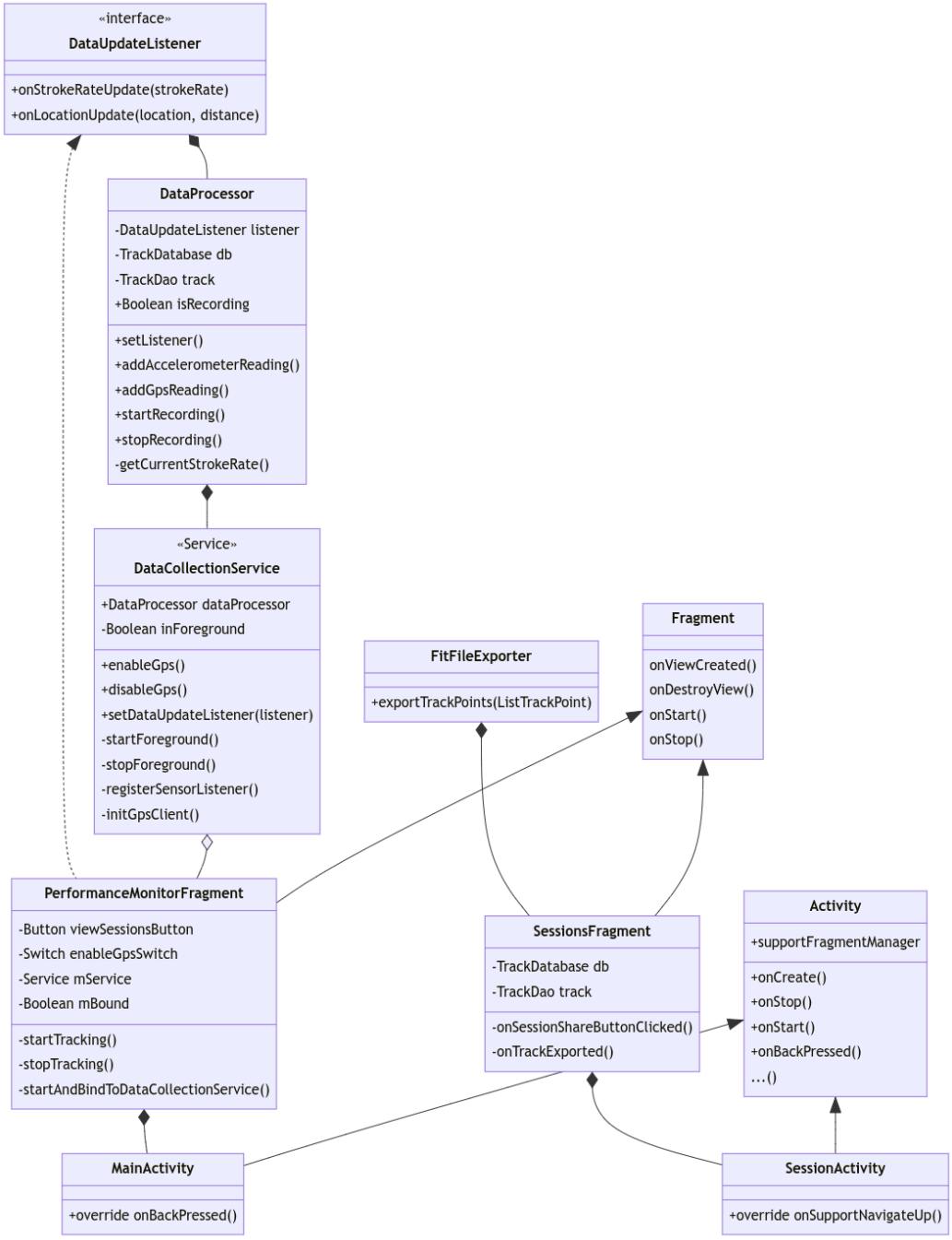

Figure 3.1: UML diagram overview of the project

3.1 Data collection

3.1.1 The data collection service

Data collection and aggregation is to be performed in a service separate from the data processor and any UI classes. This is to ensure that the data collection service is always running, and that the data processor is always ready to process the data. It allows for the tracking to be decoupled from the user interface, so that even if the user interface goes out of view (e.g. the user exits to the home screen) during a rowing session, the recording process is not interrupted.

The data collection process has the following primary objectives:

- Manage its own lifecycle to prevent the tracking process from being killed by the OS
- Handle registration of sensor listeners, which are used to receive hardware sensor data.
- Appropriately handle the permission checks and requests required by the GPS client
- Pass on accelerometer and GPS measurements to a data processing class

3.2 Processing incoming data

3.2.1 Stroke rate

As part of the Android software stack, a virtual sensor is available that filters out acceleration due to gravity from the raw signal produced by the accelerometer.[8]

The acceleration and linear acceleration sensors output a vector with x, y and z axes, which are oriented relative to the device, with the z-axis coming out of the screen of the device. Since the boat is travelling linearly and we do not need an accurate numeric value for the accleration, it is enough to calculate the magnitude of the linear acceleration

$$|\mathbf{a}| = \sqrt{x^2 + y^2 + z^2}$$

to obtain a repeating (with slight variation) pattern. Since we are taking the magnitude, the phone can be placed in any orientation and stroke rate detection can still be performed. Or, if the phone for example slips or changes position during the rowing session, stroke rate detection would not be interrupted.

The shape of the acceleration readings produced by the above equation should are assumed to be similar in pattern to the absolute value of the acceleration pattern shown in in figure 1.1.

Filtering noise from the accelerometer

> A low-pass filter is a filter that passes signals with a frequency lower than a selected cutoff frequency and attenuates signals with frequencies higher than the cutoff frequency. (...) Low-pass filters provide a smoother form of a signal, removing the short-term fluctuations and leaving the longer-term trend. [10]

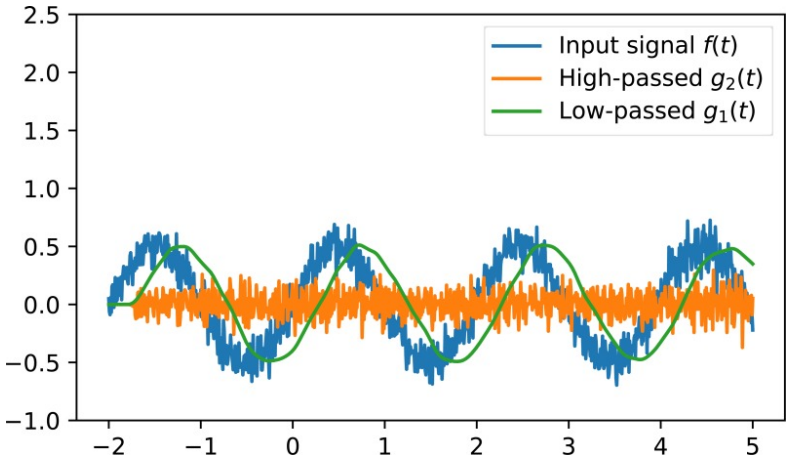

Figure 3.2: Sine signal with additive noise after processing with a low-pass filter [9]

Many smartphone accelerometer sensors are not best-in-class so have significant noise. The application is to use a low-pass filter to remove the noise from the accelerometer readings through a procedure called ramping. This is a very efficient and simple way of smoothing the acceleration readings. However, the downside to this method is a slight phase shift in the signal, as visible in figure 3.2. At the standard accelerometer sampling rate of 50Hz this is not significant enough to cause substantial delay.

The filter function f is defined recursivley below, where $\mathbf{a}_n$ is the last (nth) accelerometer reading as a three-dimensional vector and k is the filtering factor, a constant determined empirically.

$$f(\mathbf{a}_n) = k\mathbf{a}_n + (1-k)\, f(\mathbf{a}_{n-1}) \tag{3.1}$$

v1: Stroke rate by Fourier transform

To calculate the stroke rate from the acceleration pattern, a Fourier transform is proposed. The Fourier transform is a discrete transform that can be used to calculate the frequency of a signal. It works by decomposing the sampled repeating pattern into a series of sinusoids. The frequency of the sinusoid with the greatest amplitude is selected as the true stroke rate. More specifically, a Sliding Discrete Fourier Transform ($SDFT$) can be employed. The sliding DFT is a recursive algorithm to compute successive Fourier transforms of input data frames that are a single sample apart. It is an efficient $O(n)$ algorithm that can be used to calculate the frequency of a signal in real time.

> The idea behind the Sliding Discrete Fourier transform (SDFT) is to make use of the known values of $F_t(n)$, where $F_t(n)$ is the value in the nth bucket in the frequency domain, to calculate the value for the next window. In particular we assume that we are moving by one sample only.
>
> That means that in order to forget the oldest sample and accept a new sample, all that is needed is for the nth frequency amplitude is an addition, a subtraction and a complex rotation by $2\pi n/N$ radians. As we know that the samples are real this is a little simpler than it might appear.
>
> There remains the problem of starting, but if we assume that the signal was silent until the first sample, then the transform is also zero and so we can slide the samples in one at a time without creating an initial FFT [Fast Fourier Transform].

(from *Sliding is Smoother than Jumping* [11])

The only necessary initialisation parameter for the SDFT is the window size N, which determines the number of frequency bins available. The maximal possible identifiable frequency is half the sample frequency according to Nyquist's theoreom, so this is used to determine the number of frequency bins available. A DC[1] and Nyquist[2] bin are also included, so the number of available frequency bins is $\frac{N}{2}+2$.

Each frequency bin, in essence, is nothing more than a complex number. On each new sample these are altered appropriately. The sum of the real components of all the frequency bins is to be stored and recalculated on each addition of a new sample. This value is used to calculate the equivalent per-bin shift in value given the newly slid-in reading. The calculated shift in value is applied to each frequency bin in turn. [If there are n frequency bins, this would require n operations, which is what makes this algorithm $O(n)$.]

A function is also to be provided to calculate the dominant frequency at the given moment in time which would return the wavelength of the frequency bin with the greatest magnitude when represented in polar form. This wavelength, in turn, can be used to calculate the stroke rate.

Although very efficient, unfortunately on implementation this Discrete Fourier Transform method was found to be lacking; the reasons for this are discussed on page 26.

v2: Stroke rate by autocorrelation

Autocorrelation is a mathematical representation of the degree of similarity between a given time series and a lagged version of itself over successive time intervals. [12]

The autocorrelation algorithm is to slide a copy of the second half of the data over itself, scoring each offset with the correlation of the slid data and the previous signal.

The discrete autocorrelation R at lag τ for a discrete time signal X is given by

$$R(\tau) = \operatorname{Corr}\left(X_n,\, X_{n-\tau}\right)$$

Correlation of two data sets X and Y is given by the below formula (from the Further Maths syllabus!)

$$\operatorname{Corr}\left(X, Y\right) = \frac{\operatorname{Cov}\left[X, Y\right]}{\sqrt{\operatorname{Var}\left(X\right)\operatorname{Var}\left(Y\right)}}$$

Since in this case X and Y are the same underlying data, except with different lag, their variances must be the same. The denominator can be simplified to $\operatorname{Var}\left(X\right)$.

Expressing the covariance and variance as a summation, the below formula is obtained, where μ is the mean of the data.

$$R\left(\tau\right) = \frac{\sum\left(X_i - \mu\right)\left(X_{i-\tau} - \mu\right)}{\sum\left(X_i - \mu\right)^2} \tag{3.2}$$

R is to be calculated for every possible value of τ up to half the window size. The value of τ that produces the maximal output for R is selected as the most probable time period of the stroke rate, measured in a number of samples. By tracking the average accelerometer sampling rate as samples come in, the number of samples τ can be converted into a stroke rate.

[1] DC for direct current, the frequency bin for the zero-frequency sinusoid
[2] the frequency bin for the maximal frequency

3.3 Data storage

A SQL database is chosen to store the data. The database is to be used to store the the stroke rate, the location of the boat and the time of the readings. Each row in the database stores the mentioned quantities, as well as the speed of the boat at that instant, measured in metres per second. A separate column for speed allows for more accurate speed recordings than a simple $\frac{\text{distance}}{\text{time}}$ calculation. Each entry stores a unique point ID, as well as the ID of the session that it belongs to. The table called `records` is defined with the following schema:

```
CREATE TABLE IF NOT EXISTS records (
  pointId INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
  trackId INTEGER NOT NULL,
  timestamp INTEGER NOT NULL,
  stroke_rate REAL NOT NULL,
  latitude REAL,
  longitude REAL,
  speed REAL
)
```

The query below retrieves the a list of session IDs present in the database and their corresponding starting timestamps. This query will be performed when a list of previously recorded sessions is shown to the user.

```
  SELECT trackId, MIN(timestamp)
  FROM records
  GROUP BY trackId
  ORDER BY timestamp DESC
```

The query below retrieves all of the track points for a given session ID, in this example 123.

```
  SELECT *
  FROM records
  WHERE trackId == 123
  ORDER BY pointId ASC
```

3.3.1 SQL Libraries on Android

The Android Room library is part of the Android Jetpack collection of first-party libraries by Google. It is selected as the database library for this application due to the following convincing arguments:

> The Room persistence library provides an abstraction layer over SQLite to allow fluent database access while harnessing the full power of SQLite. In particular, Room provides the following benefits:
>
> - Compile-time verification of SQL queries.
> - Convenience annotations that minimize repetitive and error-prone boilerplate code.

- Streamlined database migration paths.
- Because of these considerations, we highly recommend that you use Room instead of using the SQLite APIs directly.

(extract from the official Android Developers Guide [13])

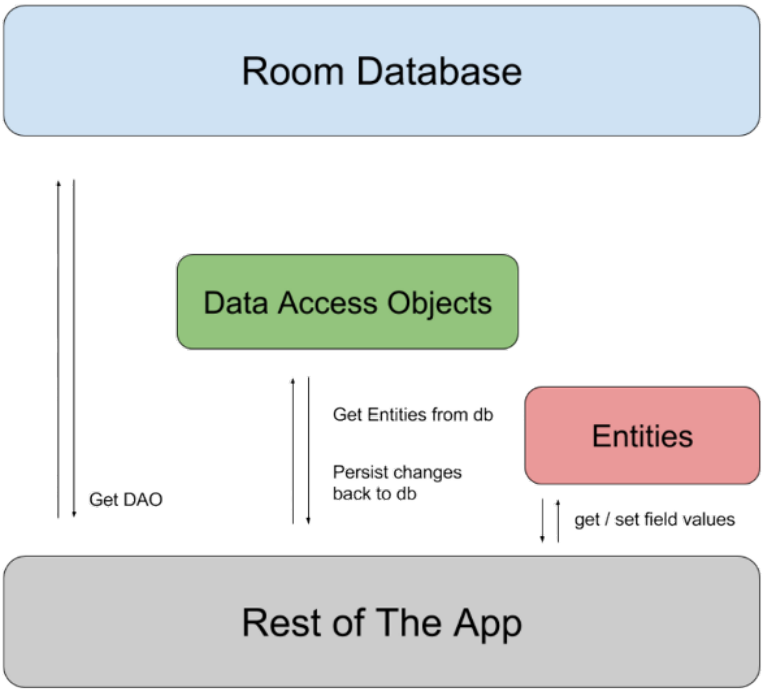

Figure 3.3: Diagram of Room library architecture [13]

As per the Room architecture (as shown in figure 3.3), creating a Room database requires three parts:

- the database itself
- the data access objects which provide queries for interacting with the database
- entities which define the tables and columns in the database

These are defined in separate classes, so three classes `TrackDatabase`, `TrackDao` and `TrackPoint` are to be created. The `TrackDao` class will use the SQL queries described earlier in this section. The `TrackPoint` class will be used to store the data in the database.

3.4 File export

The Flexible and Interoperable Data Transfer (FIT) protocol is designed specifically for the storing and sharing of data that originates from sport, fitness and health devices. The FIT protocol defines a set of data storage templates (FIT messages) that can be used to store information such as user profiles, activity data, courses, and workouts. It is specifically designed to be compact, interoperable and extensible.

The FIT file protocol was designed to provide:

- Interoperability of device data across various platforms
- Scalability from small embedded devices to cloud platforms
- Forward compatibility, allowing the protocol to grow and retain existing functionality
- Automated compatibility across platforms of different native endianness

After the initial sensor data is collected, the FIT protocol provides a consistent format allowing all devices in the subsequent chain to share and use the data.

(from the Garmin FIT SDK Overview [14])

When requested by the user, a given recorded rowing session should be exported as a FIT activity file. Garmin provides a FIT SDK in Java, which is cross-compatible with Kotlin. The FIT file type is chosen for its widespread support by sports software such as Strava. It is preferred to recording to, for example, a GPX file, since a FIT file has built-in support for additional metrics besides GPS, such as cadence and heart rate. Of course, a GPX file exporter could also be implemented but there is no such necessity if FIT export is available. There exists software such as FitCSVTool to convert a FIT activity file to a CSV file for analysis in spreadsheet software like Microsoft Excel.

3.5 User interface

3.5.1 Main performance screen

On fragment creation, the fragment is to perform a few initial user interface setup tasks (aside from actually inflating the layout of the UI components) including:

- requesting the screen to not turn off even after the usual dimming time
- requesting the device to stay oriented in portrait mode, so that boat roll does not cause unnecessary screen rotation that could distract the rower
- configuring callback functions for the "view sessions" button, the GPS status switch and start/stop recording button

In addition, this fragment is to be responsible for starting up, where necessary, the data collection service and binding to it while the user interface is active. Once bound, the fragment, which implements the listener interface, can set itself as a callback listener in order to receive updates when a new stroke rate or split becomes available.

3.5.2 Session history

A separate view, accessible via a *view sessions* button on the application home screen, takes the user to a screen where they can select a previously recorded session and export/share the session with another application on the device. The list of sessions is retrieved from the database and displayed in a list, with each row labelled with the session start time.

4 | Technical solution

4.1 Overview

- net.zeevox.nearow
  - data
    - Autocorrelator.kt
    - CircularDoubleBuffer.kt
    - DataProcessor.kt
  - db
    - Session.kt
    - TrackDao.kt
    - TrackPoint.kt
    - TrackDatabase.kt
  - input
    - DataCollectionService.kt
  - output
    - fragment
      - PerformanceMonitorFragment.kt
      - SessionsFragment.kt
    - recyclerview
      - SessionsListAdapter.kt
    - activity
      - MainActivity.kt
      - SessionsActivity.kt
  - utils
    - UnitConverter.kt

4.2 Data collection

The gathering and aggregation of collected data is handled by the `DataCollectionService`. This service is started when the application is started. It is run as service, and as such inherits from the Android `Service` class.

4.2.1 GPS

The `initGpsClient` function is called within the service's `onCreate` method. This function initializes the GPS client, which is used to obtain the current location of the boat. The GPS client is configured to

obtain a fix once a second, and to obtain the best possible accuracy. A `LocationCallback` is provided for the GPS client to call to notify the service when a new location is obtained. The callback passes the newly obtained location fix to the `DataProcessor`.

The `DataCollectionService` provides two functions to control whether GPS updates are to be requested. These functions, `enableGps` and `disableGps` are public and as such can be called by any class that binds to the running service. GPS collection is enabled by default, so the `enableGps` function is called in the service's `onCreate` method. The `disableGps` function is called in the service's `onDestroy` method so that the device ceases to request GPS location updates if/when the service is destroyed.

4.2.2 Accelerometer

The `DataCollectionService` implements the `SensorEventListener` interface, which allows it to receive sensor data from the Android hardware. The `onSensorChanged(event: SensorEvent)` method is called when a new sensor data is available. The `SensorEvent` object contains the sensor data, and in the case of the accleerometer, the `SensorEvent.values` array contains the acceleration readings as a three-dimensional vector. This array is passed to the `DataProcessor` for processing.

4.2.3 Service Management

Android has strict policies for long-running services with battery optimisation in mind. It is mandatory for the service to display a notification if the application interface is not visible to the user. This is why a significant portion of the `DataCollectionService` is dedicated to binding and unbinding from user interface classes and showing the service notification where appropriate. A private inner `NotificationUtils` class is responsible for actually creating the notifications that will be shown to the user to prevent cluttering the `DataCollectionService` class. Notifications are pushed by a coroutine task since this can take a non-neglible few milliseconds that can add up to a noticeable delay.

The code for the `DataCollectionService.kt` is displayed overleaf.

```kotlin
package net.zeevox.nearow.input

import ...

/**
* [DataCollectionService] is a foreground service that receives sensor and location updates and
* handles the lifecycle of the tracking process
*/
class DataCollectionService : Service(), SensorEventListener {

    /** [SensorManager] is a gateway to access device's hardware sensors */
    private lateinit var mSensorManager: SensorManager

    /**
     * [NotificationManagerCompat] is a wrapping library around [NotificationManager] Used to push
     * foreground service notifications, which are necessary to prevent the service from getting
     * killed
     */
    private lateinit var mNotificationManager: NotificationManagerCompat

    /**
     * Instance of [DataProcessor] to which we pass sensor and location updates for number-crunching
     */
    private lateinit var mDataProcessor: DataProcessor

    /** A direct reference to the [DataProcessor] currently in use by the [DataCollectionService] */
    val dataProcessor: DataProcessor
        get() = mDataProcessor

    /**
     * Whether this instance of the [DataCollectionService] is currently running as a foreground
     * service
     */
    private var inForeground: Boolean = false

    /** Contains parameters used by [FusedLocationProviderClient]. */
    private lateinit var mLocationRequest: LocationRequest

    /** Provides access to the Fused Location Provider API. */
    private lateinit var mFusedLocationClient: FusedLocationProviderClient

    /** Callback for changes in location. */
    private lateinit var mLocationCallback: LocationCallback

    companion object {
        const val NOTIFICATION_ID = 7652863

        // 20,000 us => ~50Hz sampling
        const val ACCELEROMETER_SAMPLING_DELAY = 20000

        /**
         * The desired interval for location updates. Inexact. Updates may be more or less frequent.
         */
        private const val UPDATE_INTERVAL_IN_MILLISECONDS: Long = 1000L

        /**
         * The fastest rate for active location updates. Updates will never be more frequent than
         * this value.
         */
        private const val FASTEST_UPDATE_INTERVAL_IN_MILLISECONDS: Long = 0L

        /** Logcat tag used for debugging */
        private val TAG = DataCollectionService::class.java.simpleName
    }
```

```
/** https://developer.android.com/guide/components/bound-services#Binder */
private val binder = LocalBinder()

/**
 * Class used for the client Binder. Because we know this service always runs in the same
 * process as its clients, we don't need to deal with IPC.
 */
inner class LocalBinder : Binder() {
    // Return this instance of LocalService so clients can call public methods
    fun getService(): DataCollectionService = this@DataCollectionService
}

override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
    return START_STICKY
}

override fun onCreate() {
    super.onCreate()

    Log.i(TAG, "Starting Nero data collection service")

    mNotificationManager = NotificationManagerCompat.from(this)

    // start the data processor before registering sensor and GPS
    // listeners so that it is ready to receive values as soon as
    // they start coming in.
    mDataProcessor =
        DataProcessor(applicationContext).also {
            // physical sensor data is not permission-protected so no need to check
            registerSensorListener()

            initGpsClient()

            // measuring GPS is neither always needed (e.g. erg) nor permitted by user
            // check that access has been granted to the user's geolocation before starting gps
            // collection
            if (isGpsPermissionGranted()) enableGps()
        }

    startService(Intent(applicationContext, DataCollectionService::class.java))
    startForeground()
}

/**
 * Called when a client comes to the foreground and binds with this service. The service should
 * cease to be a foreground service when that happens.
 */
override fun onBind(intent: Intent?): IBinder {
    Log.i(TAG, "Client bound to service")
    stopForeground()
    return binder
}

/**
 * Called when a client comes to the foreground and binds with this service. The service should
 * cease to be a foreground service when that happens.
 */
override fun onRebind(intent: Intent?) {
    Log.i(TAG, "Client rebound to service")
    stopForeground()
    super.onRebind(intent)
}
```

```
/**
 * Called when the last client unbinds from this service. If a track is being recorded,
 * make this service a foreground service.
 */
override fun onUnbind(intent: Intent?): Boolean {
    Log.i(TAG, "Last client unbound from service")

    if (mDataProcessor.isRecording) startForeground() else stopForeground()
    return true
}

/**
 * Switch the [DataCollectionService] to a foreground service so that sensor and location
 * updates can continue to be processed even though the application UI has gone out of view
 */
private fun startForeground() {
    Log.i(TAG, "Switching to foreground service")
    startForeground(NOTIFICATION_ID, NotificationUtils.getForegroundServiceNotification(this))

    // Pushing notifications on main thread is warned against by StrictMode
    CoroutineScope(Dispatchers.Default).launch {
        mNotificationManager.notify(
            NOTIFICATION_ID,
            NotificationUtils.getForegroundServiceNotification(this@DataCollectionService))
    }

    inForeground = true
}

/** Stop being a foreground service if the GUI comes back into view. */
private fun stopForeground() {
    Log.i(TAG, "Cancelling foreground service")
    stopForeground(true)
    inForeground = false
}

fun setDataUpdateListener(listener: DataProcessor.DataUpdateListener) =
    mDataProcessor.setListener(listener)

private fun registerSensorListener() {
    CoroutineScope(Dispatchers.IO).launch {
        mSensorManager = getSystemService(AppCompatActivity.SENSOR_SERVICE) as SensorManager
        mSensorManager.getDefaultSensor(Sensor.TYPE_LINEAR_ACCELERATION)?.also { accelerometer
            ->
            mSensorManager.registerListener(
                this@DataCollectionService, accelerometer, ACCELEROMETER_SAMPLING_DELAY)
        }
    }
}

private fun isGpsPermissionGranted(): Boolean {
    return ActivityCompat.checkSelfPermission(this, Manifest.permission.ACCESS_FINE_LOCATION) ==
        PackageManager.PERMISSION_GRANTED &&
        ActivityCompat.checkSelfPermission(this, Manifest.permission.ACCESS_COARSE_LOCATION) ==
            PackageManager.PERMISSION_GRANTED
}
```

```
    /**
                                                                                                    *
https://github.com/android/location-samples/blob/main/LocationUpdatesForegroundService/app/src/main/jav
a/com/google/android/gms/location/sample/locationupdatesforegroundservice/LocationUpdatesService.java
     */

    private fun initGpsClient() {
        mFusedLocationClient = LocationServices.getFusedLocationProviderClient(this)

        mLocationCallback =
            object : LocationCallback() {
                override fun onLocationResult(locationResult: LocationResult) {
                    super.onLocationResult(locationResult)
                    mDataProcessor.addGpsReading(locationResult.lastLocation)
                }
            }

        createLocationRequest()
    }

    fun enableGps() {
        try {
            mFusedLocationClient.requestLocationUpdates(
                mLocationRequest, mLocationCallback, Looper.getMainLooper())
        } catch (unlikely: SecurityException) {
            Log.e(TAG, "Lost location permission. Could not request updates.", unlikely)
        }
    }

    private fun createLocationRequest() {
        mLocationRequest =
            LocationRequest.create().apply {
                interval = UPDATE_INTERVAL_IN_MILLISECONDS
                fastestInterval = FASTEST_UPDATE_INTERVAL_IN_MILLISECONDS
                priority = LocationRequest.PRIORITY_HIGH_ACCURACY
            }
    }

    /** Stop requesting location updates */
    fun disableGps() {
        Log.i(TAG, "Requesting GPS location updates to stop")
        try {
            mFusedLocationClient.removeLocationUpdates(mLocationCallback)
        } catch (unlikely: SecurityException) {
            Log.e(TAG, "Lost location permission. Could not remove updates.", unlikely)
        }
    }

    /**
     * Called by the system to notify a Service that it is no longer used and is being removed. The
     * service should clean up any resources it holds (threads, registered receivers, etc) at this
     * point. Upon return, there will be no more calls in to this Service object and it is
     * effectively dead.
     */
    override fun onDestroy() {
        disableGps()
        super.onDestroy()
    }
```

```
    override fun onSensorChanged(event: SensorEvent?) {
        if (event == null) return

        when (event.sensor.type) {
            Sensor.TYPE_LINEAR_ACCELERATION -> mDataProcessor.addAccelerometerReading(event.values)
        }
    }

    override fun onAccuracyChanged(sensor: Sensor?, accuracy: Int) {
        // TODO accuracy handling?
        Log.w(TAG, "Unhandled ${sensor?.name} sensor accuracy change to $accuracy")
    }

    private class NotificationUtils private constructor() {
        companion object {
            private const val CHANNEL_ID = "tracking_channel"

            @RequiresApi(Build.VERSION_CODES.O)
            internal fun createServiceNotificationChannel(context: Context) {
                val notificationManager = NotificationManagerCompat.from(context)
                notificationManager.createNotificationChannel(
                    NotificationChannel(
                            CHANNEL_ID,
                            context.getString(R.string.notification_channel_tracking_service),
                            NotificationManager.IMPORTANCE_MIN)
                        .apply {
                            enableLights(false)
                            setSound(null, null)
                            enableVibration(false)
                            vibrationPattern = longArrayOf(0L)
                            setShowBadge(false)
                        })
            }

            internal fun getForegroundServiceNotification(context: Context): Notification {

                val notificationBuilder =
                    NotificationCompat.Builder(context, CHANNEL_ID)
                        .setAutoCancel(true)
                        .setDefaults(Notification.DEFAULT_ALL)
                        .setContentTitle(context.resources.getString(R.string.app_name))
                        .setContentText(
                            context.getString(R.string.notification_background_service_running))
                        .setWhen(System.currentTimeMillis())
                        .setOngoing(true)
                        .setVibrate(longArrayOf(0L))
                        .setSound(null)
                        .setSmallIcon(R.mipmap.ic_launcher_round)

                // Notifications channel required for Android 8.0+
                if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
                    createServiceNotificationChannel(context)
                    notificationBuilder.setChannelId(CHANNEL_ID)
                }

                return notificationBuilder.build()
            }
        }
    }
}
```

4.3 Data processing

4.3.1 `SlidingDFT.kt`

As specified in the design stage, the incoming acceleration readings are to undergo a Sliding Discrete Fourier Transform (SDFT) to find the dominant frequency.

```
/**
* Initialise a new sliding DFT processor
* @param sampleCount The number of samples to store in the buffer
* @param componentsPerSample The number of subdivisions of each frequency component (e.g. 10 for
* one decimal place)
*
* Initially based on Imagick-FT @see https://github.com/olavholten/imagick-FT
*/
class SlidingDFT(private val sampleCount: Int, private val componentsPerSample: Int) {

   // Nyquist Theorem, maximal possible identifiable frequency is half the sample frequency
   private val frequencyCount: Int = sampleCount / 2

   // Initialise an array of complex numbers to store the DFT results
   val sliderFrequencies: Array<Frequency> =
       (0..(frequencyCount + 1) * componentsPerSample)
           .map {
               Frequency(
                   frequencyCount, (it.toDouble() / componentsPerSample), componentsPerSample)
           }
           .toTypedArray()

   // Sum of real components of the frequency components
   private var realSum = 0.0
```

Figure 4.1

The `SlidingDFT` class is initialised with the number of successive samples to be used. The constructor and initialisation code is shown in figure 4.1. A later addition also includes a `componentsPerSample` parameter that allows for one to subdivide the frequency bins into a number of sub-bins to allow for greater frequency resolution at the expense of higher latency.

Figure 4.2 shows the code for the methods described on page 15. The `slide` method is called when a new sample is available. The `getMaximallyCorrelatedFrequency` method is called when the stroke rate is to be updated.

Within the `SlidingDFT` class an inner class called `Frequency` is defined, the code for which is displayed in figure 4.3. This class represents a single frequency component of the SDFT. As stated in the paper, each frequency component experiences a rotation of $2\pi n/N$ radians, where n is the wavelength and N is the window size. Since we halved the window size to obtain the total number of frequency bins, the factor of 2 cancels out. A correction is applied for DC and Nyquist frequency bins.

`Polar` and `Complex` are two data classes that represent a complex number in different forms. The two of them make use of a feature of Kotlin that I find rather appealing: infix functions. Both have an `into` function that converts the imaginary number from one representation into another. By using these `into` methods, we avoid recreating the objects when the imaginary number is changed from one representation to another in the `slide` function of the inner `Frequency` class, making the algorithm more efficient.

```
// Slide a new sample into the SDFT
fun slide(value: Double) {

    // calculate the equivalent change for a single frequency component
    val change: Double = (value - realSum) / sampleCount

    // slide each frequency component by the calculated value
    sliderFrequencies.map { it.slide(change) }

    // update the sum of the real components
    this.realSum = sliderFrequencies.sumOf { it.complex.real }
}

// the frequency component with the greatest amplitude is selected
fun getMaximallyCorrelatedFrequency(): Frequency? =
    sliderFrequencies.maxByOrNull { it.polar.magnitude }
```

Figure 4.2: `SlidingDFT` initialisation code

```
// class representing a single frequency component
inner class Frequency
constructor(
    // the total number of bins in the DFT (including DC and Nyquist)
    totalBinCount: Int,
    // the wavelength of this frequency component
    val wavelength: Double,
    // how many frequencies each wavelength is subdivided into
    componentsPerSample: Int
) {
    private val turnDegrees: Double = kotlin.math.PI / totalBinCount * wavelength

    var complex: Complex = Complex(0.0, 0.0)
    var polar: Polar = Polar(0.0, 0.0)

    private val multiplier: Double =
        (if (wavelength == 0.0 || wavelength == totalBinCount.toDouble()) 1.0 else 2.0) /
            componentsPerSample

    fun slide(change: Double) {
        complex += change * multiplier
        complex into polar
        polar.addPhase(turnDegrees)
        polar into complex
    }
}
```

Figure 4.3

Unfortunately in testing the sliding DFT proved to be inadequate for the task at hand. Although incredibly efficient (the most costly operation being the calculation of a sine and cosine) it was not able to accurately calculate the frequency of the signal. It was discovered only after some head-bashing firmly within the implementation stage that a fundamental downside of the Discrete Fourier Transform is that it is not possible to do DFT with low latency and fine frequency resolution at low frequencies.[15] The frequency resolution is limited by the sampling rate, which, for a modern smartphone is somewhere in the region of 50Hz. This is insufficient to find a reasonable compromise.

```kotlin
data class Polar(var magnitude: Double, var phase: Double) {
    fun addPhase(phaseTurn: Double) {
        phase += phaseTurn
    }

    infix fun into(complex: Complex) {
        complex.real = cos(phase) * magnitude
        complex.imaginary = sin(phase) * magnitude
    }
}


data class Complex(var real: Double, var imaginary: Double) {
    operator fun plus(otherReal: Double): Complex = Complex(real + otherReal, imaginary)

    infix fun into(polar: Polar) {
        polar.magnitude = sqrt(real * real + imaginary * imaginary)
        polar.phase = atan2(imaginary, real)
    }
}
```

Figure 4.4

By adding more frequency bins, a far higher resolution is achieved, with accuracy as far as one decimal place for the stroke rate. The downside to this is an incredibly increase in latency, with stroke rate persisting for up to half a minute (!) after device movement has ceased. Since the limiting factor is the sampling frequency, this would be a possible solution. And, with careful handling, a `SensorDirectChannel` can be accessed on Android that provides sensor sampling frequencies in excess of 500Hz, more than ten times as frequent as the smartphone's native sampling rate. However, this significantly increases power consumption for the device, both due to the increased accelerometer sampling rate and due to the increased overhead for processing more samples per second as well as the memory that would be required to store an adequately sized buffer for the readings.

As such, it was time to return back to the drawing board and a new solution was proposed.

4.3.2 `Autocorrelator.kt`

The `Autocorrelator` class provides utility methods for scoring the input readings according to their autocorrelation.

The method `getFrequencyScores(data)` shown in figure 4.5 implements the scoring algorithm described on page 15. In terms of the mathematical description of the algorithm, it returns a `DoubleArray` where the index is the offset τ and the value is the output of the autocorrelation function R. The `getFrequencyScores` method is called by the `DataProcessor` when the stroke rate is to be recalculated.

The mean is calculated using an inbuilt method provided by Kotlin's `Collection` interface. There are $\frac{N}{2}$ possible lag values, where N is the number of data points in the ring buffer. The correlation scores are calculated for each lag, or offset, and saved into a `DoubleArray`. The `DoubleArray`'s index represents the lag, which is recorded as a number of samples, and the value is the calculated correlation for this offset. This method assumes that the sampling rate remains fairly constant for its results to be reliable, and this is the case for any decent accelerometer.

```kotlin
/**
 * A copy of the right half of the array is incrementally slid over the whole array. Each
 * sequential offset is scored according to the correlation of the slid and fixed data
 * points.
 *
 * @param data a ring buffer of the data to be scored
 * @return an array of offset-correlation (index-value) mappings
 */
fun getFrequencyScores(data: CircularDoubleBuffer): DoubleArray {
    if (data.size < 10) return DoubleArray(0)

    val mean = data.average()
    val output = DoubleArray(data.size / 2)
    for (shift in output.indices) {
        var num = 0.0
        var den = 0.0
        for (index in data.indices) {
            val xim = data[index] - mean
            num += xim * (data[(index + shift) % data.size] - mean)
            den += xim * xim
        }
        output[shift] = num / den
    }

    return output
}
```

Figure 4.5

```kotlin
/**
 * Given an array of correlation scores stored in [frequencies] and a minimum frequency to
 * consider given by [minFreq], return the offset (index) of the best-correlated frequency.
 *
 * @param frequencies an array of offset correlations, e.g one given by [getFrequencyScores]
 * @param minFreq the minimum offset to consider, for eliminating DC and
 */
fun getBestFrequency(frequencies: DoubleArray, minFreq: Int): Int =
    frequencies.indices.drop(minFreq).maxByOrNull { frequencies[it] } ?: -1
```

Figure 4.6

The `getBestFrequency(correlations)` method shown in 4.6 selects the "best" frequency out of the provided autocorrelation array by choosing the one with the highest value. Stroke rates below a specified threshold are ignored. This is because, naturally, a tiny offset (e.g. 1 sample) will produce a near-perfect correlation. Since very low-rate stroke detection is not necessary these frequencies can be discounted.

4.3.3 `CircularDoubleBuffer.kt`

The `CircularDoubleBuffer` class provides an implementation of a circular buffer, also known as a ring buffer, for storing `Double` values. Its primary use in the context of this application is for storing recent acceleration readings. A circular buffer was chosen for its memory efficiency and fast append operations.

It fulfills the obligations of Kotlin's `Collection` interface.

```
/**
 * Add a single [Double] value to the end of the list, displacing the oldest recorded value.
 * @param value is a [Double] value to save to the tail of the ring buffer
 */
fun addLast(value: Double) {
    // record given reading
    buffer[pointer] = value

    // update circular buffer pointer
    pointer = (pointer + 1).mod(size)
}
```

Figure 4.7

A single append method is necessary: `addLast(double: Double)`, the code for which is shown in figure 4.7. This function stores the new value at the rear of the buffer, thus displacing (overwriting) the oldest value in the list which is no longer necessary for consideration.

The `pointer` variable always points to the next slot to be filled, so it is incremented only after the value has been stored. The Kotlin `mod` operator is used as opposed to the infix `%` operator, which is shorthand for `rem`. This is due to a subtlety in their functionality. The `mod` operator will always, as in mathematical modular arithmetic, return a value between 0 and $n-1$, where n is the value with respect to which modulo is taken. On the other hand, the `rem` operator can return negative values which could crash the application if it tries to access a negative index in the underlying `DoubleArray`.

```
override fun iterator(): Iterator<Double> =
    object : Iterator<Double> {
        private val index: AtomicInteger = AtomicInteger(0)

        /** Returns `true` if the iteration has more elements. */
        override fun hasNext(): Boolean = index.get() < size

        /** Returns the next element in the iteration. */
        override fun next(): Double = get(index.getAndIncrement())
    }
```

Figure 4.8

In addition, the `CircularDoubleBuffer` overrides the `iterator()` function of the parent `Collection` interface, as shown in figure 4.8. This allows programs that use the `CircularDoubleBuffer` to iterate through the its elements as any other Kotlin list with the `for (element in circularDoubleBuffer) {}` syntax.

An `AtomicInteger` was used instead of the normal `Int` class for thread-safe operations in a multithreaded context. This allows the `CircularDoubleBuffer` to be used from separate parts of the application without concern for concurrent access exceptions.

```kotlin
package net.zeevox.nearow.data

import java.util.concurrent.atomic.AtomicInteger

/**
* Initialise a new circular buffer (ring buffer / circular array) for storing [Double] values.
* @param _size integer specifying the capacity of the array
*/
class CircularDoubleBuffer(private val _size: Int) : Collection<Double> {

    /**
     * Secondary constructor that supports initialising the circular buffer with data given by a
     * user-provided function mapping indices to values.
     */
    constructor(_size: Int, function: (Int) -> Double) : this(_size) {
        (0 until size).forEach { index -> addLast(function(index)) }
    }

    /** Returns the size of the collection. */
    override val size: Int
        get() = _size

    private val buffer = DoubleArray(size)
    private var pointer = 0

    /**
     * Add a single [Double] value to the end of the list, displacing the oldest recorded value.
     * @param value is a [Double] value to save to the tail of the ring buffer
     */
    fun addLast(value: Double) {
        // record given reading
        buffer[pointer] = value

        // update circular buffer pointer
        pointer = (pointer + 1).mod(size)
    }

    /** Get the most recently added value */
    val head: Double
        get() = this[-1]

    /** Get the oldest value in the buffer */
    val tail: Double
        get() = this[0]

    /**
     * Returns a string representation of the circular array, with the newest value printed last.
     */
    override fun toString(): String =
        "[${
            (buffer.sliceArray(pointer until size)
                    + buffer.sliceArray(0 until pointer))
                .joinToString(", ")
        }]"

    operator fun get(index: Int): Double = buffer[(pointer + index).mod(size)]

    /** Checks if the specified element is contained in this collection. */
    override fun contains(element: Double): Boolean = buffer.any { it == element }

    /** Checks if all elements in the specified collection are contained in this collection. */
    override fun containsAll(elements: Collection<Double>): Boolean =
        elements.all { element -> buffer.any { it == element } }

    override fun iterator(): Iterator<Double> =
        object : Iterator<Double> {
            private val index: AtomicInteger = AtomicInteger(0)

            /** Returns `true` if the iteration has more elements. */
            override fun hasNext(): Boolean = index.get() < size

            /** Returns the next element in the iteration. */
            override fun next(): Double = get(index.getAndIncrement())
        }

    /** Returns `true` if the collection is empty (contains no elements), `false` otherwise. */
    override fun isEmpty(): Boolean = size > 0
}
```

4.4.1 `DataProcessor.kt`

The `DataProcessor` class handles the processing of incoming acceleration and location data. Accelerometer readings are smoothed and stroke rate is calculated; it tracks the total distance travelled over the course of the rowing session and informs any UI listeners of updates of any of the aforementioned metrics through callbacks. It is also responsible for persisting the calculated stroke rate, speed, and other characteristics to the application database.

The processing of the data is a computationally expensive process; since it would be unfavourable to slow down the user interface, most of the functionality of `DataProcessor` is executed on another thread. Instead of launching a full-on separate thread, we can use a functionality built into Kotlin called coroutines.

The Kotlin team defines coroutines as "lightweight threads". They are in essence tasks that can be executed by an actual thread. Kotlin coroutine jobs can be paused and resumed, passed between different actual threads and allow for easy handling of asynchronous execution. The context of the coroutine includes a coroutine dispatcher that determines what thread the corresponding coroutine uses for its execution.[16]

```
/**
 * Check the database for existing sessions. The new session ID is auto-incremented from the
 * last recorded session. In case this is the first recorded session, the ID returned is 1.
 */
private suspend fun getNewSessionId(): Int = coroutineScope {
    val sessionId = async(Dispatchers.IO) { (track.getLastSessionId() ?: 0) + 1 }
    sessionId.await()
}
```

Figure 4.9

For example, the `getNewSessionId` function shown in figure 4.9 gets the next sequentially availabe session ID. `Dispatchers.IO` is passed as an argument to the `async` function to specify that this command should be run on the applications I/O thread, as this function requires a SQL database query.

The stroke rate is calculated by an infinitely-running coroutine job that executes on the `Default` coroutine scope, i.e. a worker non-UI thread. The code for this task is shown in figure 4.10.

A call to the coroutine-provided function `ensureActive` checks that the associated coroutine scope is still being used. This way, if an unhandled exception is encountered on the main thread this infinite job would still terminate despite no explicit call to do so.

However, since the stroke rate is being calculated in a worker scope, the callback to inform the listener of an updated stroke rate value has to be posted to the main thread before it can be executed. This is because Android UI elements are protected in that they cannot be modified by code running on anything but the main UI thread.

The companion object of the `DataProcessor` sets various configurable compile-time constants. In theory these could be exposed to the user, but the average rower has no need to configure any of the parameters shown. Most of these numbers were either determined empirically through testing or by calculation.

Let me draw your attention to the `ACCEL_BUFFER_SECONDS` quantity that specifies how large of a buffer to store with acceleration readings. This was chosen to be 10s. By Nyquist's theorem, frequencies

```
/** Perform CPU-intensive tasks on the `Default` coroutine scope */
private val workerScope = CoroutineScope(Dispatchers.Default)

/** A constantly-running job that periodically recalculates stroke rate */
private val strokeRateCalculator: Job =
    workerScope.launch {
        // after a three-second stabilisation period
        delay(STROKE_RATE_INITIAL_DELAY)
        // recalculate stroke rate roughly once per second
        while (true) {
            // check that coroutine scope has not requested a shutdown
            ensureActive()

            getCurrentStrokeRate()

            // DataUpdateListener.onStrokeRateUpdate *must* be called on the UI thread, as it is
            // forbidden to alter UI elements from any other scope. As such, post the callback
            // onto the main thread. Ref. https://stackoverflow.com/a/56852228
            Handler(Looper.getMainLooper()).post {
                listener?.onStrokeRateUpdate(smoothedStrokeRate)
            }

            delay(STROKE_RATE_RECALCULATION_PERIOD)
        }
    }
```

Figure 4.10

with a wavelength of up to 5s can be detected. That results in a $60 \div 5 = 12$ spm minimum detection frequency. Stroke rates lower than 12 are incredibly rare and inefficient. Perhaps only in the case of a specific drill might the stroke rate drop that low. A typical paddling stroke rate could go as low as 15, which makes 12 a reasonable lower bound.

```
    /** Called when a new acceleration reading comes in, storing a damped value into the buffer */
    fun addAccelerometerReading(@Size(3) readings: FloatArray) {
        // saving of accel value is async so record timestamp now
        val timestamp = System.currentTimeMillis()

        // launch processing on worker coroutine scope
        workerScope.launch {
            // ramp-speed filtering https://stackoverflow.com/a/1736623
            val filtered =
                DoubleArray(3) {
                    readings[it] * FILTERING_FACTOR +
                        lastAccelReading[it] * CONJUGATE_FILTERING_FACTOR
                }

            // append the magnitude of the damped acceleration to the rear of the circular buffer
            accelReadings.addLast(magnitude(filtered))

            // store the corresponding timestamp as well
            timestamps.addLast(((timestamp - startTimestamp) / 1000L).toDouble())

            // save current readings into memory for when next readings come in
            System.arraycopy(filtered, 0, lastAccelReading, 0, 3)
        }
    }
```

Figure 4.11

The companion object aslo defines the filtering factor $k = 0.1$ talked about on page 14. The filtering constant is put to use by the `addAccelerometerReading` method (shown in figure 4.11), which is called when a new acceleration reading comes in. This function implements equation 3.1 to cal-

culate the new filtered/ramped/dampened acceleration reading, before storing its magnitude in a `CircularDoubleBuffer` of readings.

The function also stores the corresponding timestamp of when the acceleration reading came in. This too is stored in a separate `CircularDoubleBuffer`, and is used to determinte the actual sampling rate (as opposed to the requested sampling rate) of the accelerometer. This is necessary to turn a number representing the time period of a stroke as a number of samples into a stroke rate.

Lastly, the filtered acceleration value is saved into a triple array `lastAccelReading` so that the next iteration of the ramping function can use this value to smooth the next.

```
/**
 * Interface used to allow a class to update user-facing elements when new data are available.
 */
interface DataUpdateListener {
    /**
     * Called when stroke rate is recalculated
     * @param [strokeRate] estimated rate in strokes per minute
     */
    @UiThread fun onStrokeRateUpdate(strokeRate: Double)

    /**
     * Called when a new GPS fix is obtained
     * @param [location] [Location] with all available GPS data
     * @param [totalDistance] new total distance travelled
     */
    @UiThread fun onLocationUpdate(location: Location, totalDistance: Float)
}

private var listener: DataUpdateListener? = null

fun setListener(listener: DataUpdateListener) {
    this.listener = listener
}
```

Figure 4.12

The `DataProcessor` provides an `interface` that allows a UI class to listen for updates to the stroke rate or location and appropriately update the user interface. The `DataUpdateListener` interface is defined within the `DataProcessor` class, as shown in figure 4.12. The methods of the `DataUpdateListener` interface are marked with the `@UiThread` annotation so that an error message is logged if they are called from a non-UI thread.

The `listener` variable is marked as nullable (with Kotlin's typing system), since the `DataProcessor` may not always have a corresponding UI thread listening for stroke rate and location updates, or if the user interface goes out of view it will cease to listen for stroke rate updates. The `listener` variable is private to prevent other classes with access to the data processor instance from obtaining a reference to the listener. Only a setter is made available, so that a new listener can be installed (for example, for when a UI fragment is recreated).

Notable extracts of the implementation have been inserted as figures above; the full code for the `DataProcessor.kt` is displayed overleaf.

```
package net.zeevox.nearow.data

import ...

class DataProcessor(applicationContext: Context) {

    companion object {
        // rough estimate for sample rate
        private const val SAMPLE_RATE = 1000000 / DataCollectionService.ACCELEROMETER_SAMPLING_DELAY

        // how long of a buffer to store, roughly, in seconds
        // 10 second buffer -> 12spm min detection (Nyquist)
        private const val ACCEL_BUFFER_SECONDS: Int = 10

        // autocorrelation works best when the buffer size is a power of two
        private val ACCEL_BUFFER_SIZE = nextPowerOf2(SAMPLE_RATE * ACCEL_BUFFER_SECONDS)

        // smooth jumpy stroke rate -- take moving average of this period
        private const val STROKE_RATE_MOV_AVG_PERIOD = 3

        // milliseconds between stroke rate recalculations
        private const val STROKE_RATE_RECALCULATION_PERIOD: Long = 1000L

        // seconds to wait before starting stroke rate calculations
        private const val STROKE_RATE_INITIAL_DELAY = ACCEL_BUFFER_SECONDS * 1000L / 2

        // magic number determined empirically
        // https://stackoverflow.com/a/1736623
        private const val FILTERING_FACTOR = 0.1
        private const val CONJUGATE_FILTERING_FACTOR = 1.0 - FILTERING_FACTOR

        const val DATABASE_NAME = "nearow"

        /** Return smallest power of two greater than or equal to n */
        private fun nextPowerOf2(number: Int): Int {
            // base case already power of 2
            if (number > 0 && (number and number - 1 == 0)) return number

            // increment through powers of two until we find one larger than n
            var powerOfTwo = 1
            while (powerOfTwo < number) powerOfTwo = powerOfTwo shl 1
            return powerOfTwo
        }

        /** Calculate the magnitude of a three-dimensional vector */
        fun magnitude(@Size(3) triple: DoubleArray): Double =
            sqrt(triple[0] * triple[0] + triple[1] * triple[1] + triple[2] * triple[2])
    }

    private var listener: DataUpdateListener? = null

    fun setListener(listener: DataUpdateListener) {
        this.listener = listener
    }

    /** Perform CPU-intensive tasks on the `Default` coroutine scope */
    private val workerScope = CoroutineScope(Dispatchers.Default)

    /** Reference all other timestamps relative to the instantiation of this class */
    private val startTimestamp = System.currentTimeMillis()

    /** The total distance travelled over the course of this tracking session */
    private var totalDistance: Float = 0f
```

```
/**
 * Interface used to allow a class to update user-facing elements when new data are available.
 */
interface DataUpdateListener {
    /**
     * Called when stroke rate is recalculated
     * @param [strokeRate] estimated rate in strokes per minute
     */
    @UiThread fun onStrokeRateUpdate(strokeRate: Double)

    /**
     * Called when a new GPS fix is obtained
     * @param [location] [Location] with all available GPS data
     * @param [totalDistance] new total distance travelled
     */
    @UiThread fun onLocationUpdate(location: Location, totalDistance: Float)
}

/** The application database */
private val db: TrackDatabase =
    Room.databaseBuilder(applicationContext, TrackDatabase::class.java, DATABASE_NAME).build()

/** Interface with the table where individual rate-location records are stored */
private val track: TrackDao = db.trackDao()

/**
 * Increment sessionId each time a new rowing session is started Getting the session ID is
 * expected to be a very quick function call so we can afford to run it as a blocking function
 */
private var currentSessionId: Int = -1

/**
 * Check the database for existing sessions. The new session ID is auto-incremented from the
 * last recorded session. In case this is the first recorded session, the ID returned is 1.
 */
private suspend fun getNewSessionId(): Int = coroutineScope {
    val sessionId = async(Dispatchers.IO) { (track.getLastSessionId() ?: 0) + 1 }
    sessionId.await()
}

// somewhere to store acceleration readings
private val accelReadings = CircularDoubleBuffer(ACCEL_BUFFER_SIZE)

// and another one for their corresponding timestamps
// this is so that we can calculate the sampling frequency
private val timestamps = CircularDoubleBuffer(ACCEL_BUFFER_SIZE)

// store last few stroke rate values for smoothing
private val recentStrokeRates = CircularDoubleBuffer(STROKE_RATE_MOV_AVG_PERIOD)

private val smoothedStrokeRate: Double
    get() = recentStrokeRates.average()

/** The last known acceleration reading, used for ramping */
private val lastAccelReading = DoubleArray(3)

/** The last known location of the device */
private lateinit var mLocation: Location

init {
    // calculate the stroke rate in coroutine scope to not block UI
    strokeRateCalculator.start()
}
```

```
    /** A constantly-running job that periodically recalculates stroke rate */
    private val strokeRateCalculator: Job =
        workerScope.launch {
            // after a three-second stabilisation period
            delay(STROKE_RATE_INITIAL_DELAY)
            // recalculate stroke rate roughly once per second
            while (true) {
                // check that coroutine scope has not requested a shutdown
                ensureActive()

                getCurrentStrokeRate()

                // DataUpdateListener.onStrokeRateUpdate *must* be called on the UI thread, as it is
                // forbidden to alter UI elements from any other scope. As such, post the callback
                // onto the main thread. Ref. https://stackoverflow.com/a/56852228
                Handler(Looper.getMainLooper()).post {
                    listener?.onStrokeRateUpdate(smoothedStrokeRate)
                }

                delay(STROKE_RATE_RECALCULATION_PERIOD)
            }
        }

    /** Whether the [DataProcessor] is persisting values to the database */
    var isRecording: Boolean = false
        private set

    /**
     * Start recording a new rowing session. Once this method is called, processed values such as
     * stroke rate and GPS location are saved.
     *
     * @return whether recording was successfully started
     */
    fun startRecording(): Boolean {
        // if already recording, impossible
        if (isRecording) return false

        currentSessionId = runBlocking { getNewSessionId() }
        totalDistance = 0f
        isRecording = true
        return true
    }

    /**
     * Called when a new GPS measurement comes in. Informs any UI listener of this new measurement
     * and stores current location in memory
     */
    fun addGpsReading(location: Location) {
        workerScope.launch {
            // sum total distance travelled
            if (this@DataProcessor::mLocation.isInitialized && this@DataProcessor.isRecording)
                totalDistance += location.distanceTo(mLocation)

            // save this for calculating the distance travelled when next GPS measurement comes in
            mLocation = location

            // handle listener on main thread
            withContext(Dispatchers.Main) {
                // inform our listener of a new GPS location
                listener?.onLocationUpdate(location, totalDistance)
            }
        }
    }
```

```
/**
 * Stop recording the current rowing session.
 *
 * @return whether recording was successfully stopped
 */
fun stopRecording(): Boolean {
    // cannot stop if not yet started!
    if (!isRecording) return false

    isRecording = false
    return true
}

/** Calculate the sampling frequency of the accelerometer in Hertz */
private val accelerometerSamplingRate: Double
    get() = timestamps.size / (timestamps.head - timestamps.tail)

/** Convert an integer number of samples into a frequency in SPM */
private fun samplesCountToFrequency(samplesPerStroke: Int): Double =
    if (samplesPerStroke <= 0) 0.0 else 60.0 / samplesPerStroke * accelerometerSamplingRate

@WorkerThread
private suspend fun getCurrentStrokeRate(): Double {
    // rudimentary but efficient stillness detection
    if (accelReadings.average() < 0.1 && magnitude(lastAccelReading) < 0.1) {
        recentStrokeRates.addLast(0.0)
        return 0.0
    }

    val frequencyScores = Autocorrelator.getFrequencyScores(accelReadings)

    val currentStrokeRate =
        samplesCountToFrequency(
            Autocorrelator.getBestFrequency(
                frequencyScores, accelerometerSamplingRate.toInt() // no more than 60 spm
                ))

    recentStrokeRates.addLast(currentStrokeRate)

    // save into the database
    if (isRecording) {
        // if there is no known last location or the last known location is too old (20s) to be
        // useful, do not save location info into the database
        val trackPoint =
            if (!this@DataProcessor::mLocation.isInitialized ||
                System.currentTimeMillis() - mLocation.time > 20000L)
                TrackPoint(currentSessionId, System.currentTimeMillis(), smoothedStrokeRate)
            else
                TrackPoint(
                    currentSessionId,
                    System.currentTimeMillis(),
                    smoothedStrokeRate,
                    mLocation.latitude,
                    mLocation.longitude,
                    mLocation.speed)
        track.insert(trackPoint)
    }

    return currentStrokeRate
}
```

```
    /** Called when a new acceleration reading comes in, storing a damped value into the buffer */
    fun addAccelerometerReading(@Size(3) readings: FloatArray) {
        // saving of accel value is async so record timestamp now
        val timestamp = System.currentTimeMillis()

        // launch processing on worker coroutine scope
        workerScope.launch {
            // ramp-speed filtering https://stackoverflow.com/a/1736623
            val filtered =
                DoubleArray(3) {
                    readings[it] * FILTERING_FACTOR +
                        lastAccelReading[it] * CONJUGATE_FILTERING_FACTOR
                }

            // append the magnitude of the damped acceleration to the rear of the circular buffer
            accelReadings.addLast(magnitude(filtered))

            // store the corresponding timestamp as well
            timestamps.addLast(((timestamp - startTimestamp) / 1000L).toDouble())

            // save current readings into memory for when next readings come in
            System.arraycopy(filtered, 0, lastAccelReading, 0, 3)
        }
    }
}
```

4.4.2 `UnitConverter.kt`

A utility class that converts between different units of measurement, in particular rowing-specific ones such as split and watts. All of the functions it provides are moved inside the Kotlin `companion object` since they do not require instantiation of the class itself.

```kotlin
/**
 * Convert a speed, measured in metres per second, into the number of seconds that would be
 * required to cover 500m at this speed.
 */
private fun speedToSecondsPer500(speed: Float): Float = 500 / speed

/**
 * Convert a speed, measured in metres per seconds, into a formatted string with split in
 * m:ss.s/500m
 */
fun speedToSplitFormatted(speed: Float): String {
    // if speed is slower than 10 minute pace call it zero
    if (speed < 0.84) return "0:00.0"

    val totalSeconds = speedToSecondsPer500(speed)
    val minutes: Int = ((totalSeconds % 3600) / 60).toInt()
    val seconds: Double = (totalSeconds % 60).toDouble()
    return String.format("%d:%04.1f", minutes, seconds)
}
```

Figure 4.13: Functions for converting speed into a typically formatted rowing split

Figure 4.13 shows the code necessary for converting speed into a rowing split. A special behaviour is implemented for that case that the speed provided is exceptionally slow (i.e. less than 10:00/500m split $\simeq 0.84\,\mathrm{m\,s^{-1}}$), in which case the speed is considered to be $0.0\,\mathrm{m\,s^{-1}}$. An interaction unexpected to me was the Java string formatting placeholder for a float. In order to achieve a zero-padded number of minutes with one decimal place (i.e. 3 numerical digits), the `%04.1f` format specifier is used, since the decimal dot is counted as a numerical digit.

```kotlin
/**
 * Convert [speed], measured in metres per second, into a pace, where pace is time in
 * seconds over distance in meters.
 */
private fun speedToPace(speed: Float): Float = 1 / speed

/**
 * Convert [pace] to watts, where pace is time in seconds over distance in meters. For
 * example: a 2:05/500m split = 125 seconds/500 meters or a 0.25 pace. Watts are then
 * calculated as (2.80/0.25 ^ 3) or (2.80/0.015625), which equals 179.2.
 * https://www.concept2.com/indoor-rowers/training/calculators/watts-calculator
 */
private fun paceToWatts(pace: Float): Double = 2.80 / pace.pow( n: 3)

fun speedToWatts(speed: Float): Double = paceToWatts(speedToPace(speed))
```

Figure 4.14: Functions for converting speed into estimated power

A given speed can also be converted into an estimated power value, using a formula determined empirically by Concept2, the leading manufacturer of rowing machines.[17] The formula is:

$$\frac{2.80}{\left(\frac{\text{seconds}}{500}\right)^3} \tag{4.1}$$

4.5 Data storage

Each entity corresponds to a table in the associated Room database, and each instance of an entity represents a row of data in the corresponding table. The `TrackPoint` class is an entity. Each row in the table corresponds to a single track point, recording the timestamp of when it was taken, the stroke rate at that instance and the the geolocation. The speed of the boat, in metres per second is also recorded. This is because the Android location provider can produce a more accurate speed than simply measuring the distance between two successive track points, for example if the Doppler measurements from GNSS satellites are taken into account.

The database table is updated and queried through a *data access object*, defined in `TrackDao.kt`. The `TrackDao` class is a Kotlin `interface` that defines the methods that can be used to access the database. The SQL queries used for retrieving data from and inserting data into the database are defined within annotations for each method. No function body is provided, since this is autogenerated by the Room library. The SQL queries themselves are abstracted from application classes, which can only use the functions defined within the `TrackDao` interface to access data. Note that all of the functions in this class are marked `suspend`. This is because they deal with access to secondary storage, and as such could take a long time to complete. As such they must not be executed on the main thread but inside a coroutine, preferably on the I/O thread provided by the Android framework.

The `TrackDao` interface is used by the `TrackDatabase` class, which is defined in `TrackDatabase.kt`. The `TrackDatabase` class is a Kotlin `data class` that defines the database schema. A database version number is provided for version control and back-compatibility on potential future updates to the application.

The code for the four database-related classes is shown overleaf.

```
--- TrackDatabase.kt ---

/**
* [TrackDatabase] defines the database configuration and serves as the app's main access point to
* the persisted data.
*/
@Database(entities = [TrackPoint::class], version = 1)
abstract class TrackDatabase : RoomDatabase() {
   abstract fun trackDao(): TrackDao
}

--- TrackPoint.kt ---

/** Each instance of [TrackPoint] represents a row in a `records` table in the app's database. */
@Entity(tableName = "records", indices = [Index(value = ["trackId"])])
data class TrackPoint(
   val trackId: Int,
   @ColumnInfo(name = "timestamp") val timestamp: Long,
   @ColumnInfo(name = "stroke_rate") val strokeRate: Double,
   @ColumnInfo(name = "latitude") val latitude: Double? = null,
   @ColumnInfo(name = "longitude") val longitude: Double? = null,
   /**
    * The speed of the boat at the given timestamp, in metres per second
    */
   @ColumnInfo(name = "speed") val speed: Float? = null,

   /**
    * Automatically incremented point ID. For simple sequential ordering. It is placed last in the
    * constructor so that it is not necessary to use named arguments when creating a new
    * trackpoint.
    */
   @PrimaryKey(autoGenerate = true) val pointId: Int = 0,
)

--- TrackDao.kt ---

/**
* [TrackDao] provides the methods that the rest of the app uses to interact with data in the
* `tracks` table.
*/
@Dao
interface TrackDao {
   @Query(
         "SELECT trackId, MIN(timestamp) AS timestamp FROM records GROUP BY trackId ORDER BY timestamp
DESC")
   suspend fun getSessions(): List<Session>

   @Query("SELECT MAX(trackId) FROM RECORDS") suspend fun getLastSessionId(): Int?

   @Query("SELECT * FROM records WHERE trackId == :sessionId ORDER BY pointId ASC")
   suspend fun loadSession(sessionId: Int): List<TrackPoint>

   @Insert(onConflict = OnConflictStrategy.ABORT) suspend fun insert(vararg records: TrackPoint)
}

--- Session.kt ---

data class Session(
   val trackId: Int,
   @ColumnInfo(name = "timestamp") val timestamp: Long,
) {
   override fun toString(): String =
       SimpleDateFormat("EEE MMM d HH:mm ''yy", Locale.UK).format(Date(timestamp))
}
```

4.6 File export

The code for the `FitFileExporter` class is shown in the appendix on page 61. Given a list of `TrackPoint` objects, representing the rows of the database, a Garmin FIT activity file is produced. It was chosen to take a list of `TrackPoint` objects as input instead of fetching rows from the database within the `FitFileExporter` class in order to increase code cohesion. In general the code is rather straightforward: necessary file metadata and start / stop markers are inserted as well as a record message for every row in the database, storing its timestamp, speed, estimated power (calculated using that `UnitConverter` class), stroke rate and latitude and longitude if available. There were only two slight bumps in the road that are described in detail below.

Firstly, the Garmin SDK requires the latitude and longitude to be stored as integers. The documentation is lacking and so there was certainly an initial sense of confusion, as integer precision for normal coordinates would result in an accuracy to the nearest 100km! After a bit of a search around the internet, a StackOverFlow answer [18] was found that suggested that *obviously* one should multiply the decimal coordinates to be multiplied by 11930465. This is because Garmin stores the latitude and longitude in the funky unit of **semicircles**. The rationale behind this is that given an integer, which can take 2^{32} possible values, to maximise precision this is divided by the 360° of the Earth's circumference. The result is that the integer is multiplied by 11930465 (to the nearest integer).

To verify that the generated FIT files are actually valid, it was attempted to upload a recorded rowing session exported as a FIT file to Strava. Unfortunately on first attempt Strava rejected the FIT file, giving the descriptive error message of "File contains bad data". A FIT activity file generated by the NK SpeedCoach was turned into a CSV using the Garmin SDK FitCSVTool and compared with the file generated by the application. The timestamps, although recorded at a very similar time, were completely different. It turns out that Garmin stores timestamps using a separate epoch, defined as seconds since midnight on December 31, 1989, as opposed to the standard Unix timestamp epoch of second since midnight on January 1, 1970.[19] A constant offset of around 20 years, or 631065600 seconds, must be applied.

4.7 User interface

4.7.1 `MainActivity.kt`

`MainActivity.kt` is the entrypoint to the application. It is defined in `AndroidManifest.xml` as the launcher activity to start the application when its icon is pressed by the user in the app drawer. The class inherits from the `AppCompatActivity` class, which is the base class (provided by the Google first-party AndroidX library) for activities that wish to use some of the newer platform and design features on older Android devices.

The `onCreate` method is called when the application is started. If the application is in debug mode, `StrictMode` is enabled. `StrictMode` is a developer tool that is used to catch common programming errors and exceptions; it logs any long-running operations that are not explicitly marked as being allowed to run on the main thread, and is used to help determine any causes that are preventing the application from running as smoothly as possible. `StrictMode` is also configured to shut down the application if a memory leak is detected.

```kotlin
class MainActivity : AppCompatActivity() {

    /** for accessing UI elements, bind this activity to the XML */
    private lateinit var binding: ActivityMainBinding

    /** the UI fragment currently displayed in this activity */
    private lateinit var fragment: Fragment

    override fun onCreate(savedInstanceState: Bundle?) {
        if (BuildConfig.DEBUG)
                StrictMode.setThreadPolicy(
                    StrictMode.ThreadPolicy.Builder().detectAll().penaltyLog().build())
                StrictMode.setVmPolicy(
                    VmPolicy.Builder()
                        .detectLeakedSqlLiteObjects()
                        .detectLeakedClosableObjects()
                        .penaltyLog()
                        .penaltyDeath()
                        .build())

        setTheme(R.style.Theme_Nearow)

        super.onCreate(savedInstanceState)

        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        if (savedInstanceState == null) {
            fragment = PerformanceMonitorFragment()
            supportFragmentManager.commit {
                // Android Dev best practices say to always setReorderingAllowed to true
                // https://developer.android.com/guide/fragments/create
                setReorderingAllowed(true)
                add<PerformanceMonitorFragment>(R.id.fragment_container_view)
            }
        }
    }

    /** Called when the activity has detected the user's press of the back key. */
    override fun onBackPressed() {
        stopService(Intent(this@MainActivity, DataCollectionService::class.java))
        super.onBackPressed()
    }
}
```

Figure 4.15

MainActivity is nothing more than a shell for the user interface. The main UI components are detached from the activity and placed in a fragment. This is done so that the fragment remains independent of any activity-specific changes such as screen rotation. However, since fragments are not notified of these changes at all, handling of, for example, a press on the device's back button is performed by the `MainActivity`. When the back button is depressed, the activity shuts down the `DataCollectionService` before letting the activity be destroyed.

The fragment displayed in `MainActivity` is the `PerformanceMonitorFragment`. It is added to the layout of the `MainActivity` through the activity's `supportFragmentManager`.

4.7.2 `PerformanceMonitorFragment.kt`

This class handles the bulk of the user interface, as described in design section 3.5.1. The fragment's `onCreateView` method is automatically called when the fragment is added into an activity; in the context

of this application it is only ever added to `MainActivity`, but in theory this could be any activity.

If the GPS location permission has not been granted when the `startAndBindToDataCollectionService` subroutine is called, a permissions request is launched (the callback for which is initialised on class instantiation). If the user grants the permission, the service startup method is called again, otherwise a permission information dialog is shown to explain why, with the GPS location permission denied, it will not be possible to let the client make full use of the application. Once GPS location permission has been granted, the service is started. The function call for this is different depending on the Android OS version, as Android Oreo introduced new battery-optimisation features for long-running services. Finally, a bind request is issued so that the fragment can interact with the newly launched instance of the `DataCollectionService`.

Once (if) binding to the service is successful, the `onServiceConnected` callback function is launched. This is where the fragment is notified that the service has been bound to. The fragment obtains the `DataCollectionService` instance from the `LocalBinder` that is passed to the callback. The fragment then registers itself as the new `DataUpdateListener` for the service. A private boolean called `mBound` stores the value of whether there is an established connection with the service, and is set to `true` when the connection is established. It is appropriately set to `false` when the `onServiceDisconnected` callback is activated. This boolean is used in other parts of the class to ensure that a connection to the service still exists prior to making calls that interact with the bound service.

Android view-binding is enabled in the `build.gradle` buildscript of the project, which means that access to UI components is handled via the automatically generated `FragmentPerformanceTrackerBinding` class. This means that formerly abundant references to the `findViewById` function are no longer necessary.

The application toolbar is initialised with two menu elements: the "view sessions" button and a switch to enable or disable GPS. The button for session history is configured with a simple callback to launch the `SessionsActivity` when pressed. The GPS switch is configured with a function to call, if bound to the service, the `enableGps` and `disableGps` methods provided by it. In addition, the UI elements for split and distance are hidden or shown as appropriate depending on whether GPS is enabled.

A *floating action button* (FAB) in the bottom-right corner controls the recording state of the application. When it is pressed, the `startTracking` method is called, which performs the following tasks:

- resets the stopwatch to start counting up from zero,
- sets the button to be red with a stop icon,
- marks the toolbar actions as *disabled* to prevent accidental touches during a session
- calls the `startRecording` method of the `DataProcessor` so that stroke rate and location measurements are stored into the database

The `stopTracking` subroutine performs the inverse of the aforementioned tasks; however, it can only be triggered when the FAB is long-pressed by the user.

The code for the `PerformanceMonitorFragment` starts on the next page.

```
package net.zeevox.nearow.ui.fragment

import ...

class PerformanceMonitorFragment : Fragment(), DataProcessor.DataUpdateListener {

    private var _binding: FragmentPerformanceTrackerBinding? = null
    private val binding
        get() = _binding!!

    /** for communicating with the service */
    lateinit var mService: DataCollectionService

    /** whether there is an established link with the service */
    private var mBound: Boolean = false

    /** Defines callbacks for service binding, passed to bindService() */
    private val connection =
        object : ServiceConnection {

            override fun onServiceConnected(className: ComponentName, service: IBinder) {
                // We've bound to LocalService, cast the IBinder and get LocalService instance
                val binder = service as DataCollectionService.LocalBinder
                mService = binder.getService()
                mBound = true

                // Listen to callbacks from the service
                mService.setDataUpdateListener(this@PerformanceMonitorFragment)
            }

            override fun onServiceDisconnected(arg0: ComponentName) {
                mBound = false
            }
        }

    private lateinit var viewSessionsButton: MaterialButton
    private lateinit var toolbarSwitchGps: SwitchMaterial

    companion object {
        /** Logcat tag used for debugging */
        private val TAG = PerformanceMonitorFragment::class.java.simpleName
    }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?,
    ): View {
        _binding = FragmentPerformanceTrackerBinding.inflate(inflater, container, false)
        return binding.root
    }


    /**
     * Called when the view previously created by [onCreateView] has been detached from the
     * fragment.
     */
    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
```

```
    /** Called when the Fragment is visible to the user. */
    override fun onStart() {
        Log.i(TAG, "Fragment started")
        super.onStart()
        startAndBindToDataCollectionService()
    }

    /** Called when the Fragment is no longer started. */
    override fun onStop() {
        Log.i(TAG, "Fragment stopped")
        super.onStop()
        requireContext().unbindService(connection)
        mBound = false
    }

    /**
     * Register the permissions callback, which handles the user's response to the system
     * permissions dialog. https://developer.android.com/training/permissions/requesting
     */
    private val requestPermissionLauncher =
        registerForActivityResult(ActivityResultContracts.RequestPermission()) { isGranted: Boolean
            ->
            // if permission has been granted return to where we left off and start the service
            if (isGranted) {
                if (!mBound) startAndBindToDataCollectionService()
            } else {
                // create an alert (dialog) to explain functionality loss since permission has been
                // denied
                val permissionDeniedDialog =
                    this.let {
                        val builder = AlertDialog.Builder(requireContext())
                        builder.apply {
                            setTitle(getString(R.string.dialog_title_gps_permission_denied))
                            setMessage(getString(R.string.dialog_msg_gps_permission_denied))
                              setPositiveButton(android.R.string.ok) { dialog, _ -> dialog.dismiss()
}
                        }
                        // Create the AlertDialog
                        builder.create()
                    }

                // show the dialog itself
                permissionDeniedDialog.show()
            }
        }

    private fun openSessionsHistory() =
        startActivity(Intent(requireActivity(), SessionsActivity::class.java))
```

```
/**
 * Called immediately after [onCreateView] has returned, but before any saved state has been
 * restored in to the view.
 */
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    // do not let the screen dim or switch off
    binding.root.keepScreenOn = true

    // TODO create alternative landscape layout file to support rotation
    requireActivity().requestedOrientation = ActivityInfo.SCREEN_ORIENTATION_PORTRAIT

    toolbarSwitchGps = binding.pmToolbar.findViewById(R.id.pm_toolbar_switch_gps)
    viewSessionsButton = binding.pmToolbar.findViewById(R.id.pm_toolbar_action_session_history)

    toolbarSwitchGps.setOnCheckedChangeListener { _, isChecked ->
        if (isChecked) mService.enableGps() else mService.disableGps()
        val visibility = if (isChecked) View.VISIBLE else View.GONE
        binding.splitFrame.visibility = visibility
        binding.distanceFrame.visibility = visibility
    }

    viewSessionsButton.setOnClickListener { openSessionsHistory() }

    binding.startStopButton.setOnClickListener {
        if (!mService.dataProcessor.isRecording) startTracking()
        else Snackbar.make(
                    binding.root,
                    getString(R.string.info_use_long_press_to_stop),
                    Snackbar.LENGTH_LONG)
                .show()
    }

    // long click used to stop recording to prevent water splashes from stopping session
    binding.startStopButton.setOnLongClickListener {
        if (!mService.dataProcessor.isRecording) startTracking() else stopTracking()
        true
    }
}

/**
 * Called when stroke rate is recalculated [strokeRate]
 * - estimated rate in strokes per minute
 */
override fun onStrokeRateUpdate(strokeRate: Double) {
    binding.strokeRate.text = String.format("%.1f", strokeRate)
}

/** Called when a new GPS fix is obtained */
override fun onLocationUpdate(location: Location, totalDistance: Float) {
    binding.apply {
        splitFrame.visibility = View.VISIBLE
        distanceFrame.visibility = View.VISIBLE
        split.text = UnitConverter.speedToSplitFormatted(location.speed)
        distance.text = String.format("%.0f", totalDistance)
    }
}
```

```
    private fun startTracking() {
        // reset chronometer
        binding.timer.base = SystemClock.elapsedRealtime()
        // start counting up
        binding.timer.start()
        mService.dataProcessor.startRecording()
        binding.startStopButton.apply {
            text = getString(R.string.action_stop_tracking)
            backgroundTintList =
                ColorStateList.valueOf(ResourcesCompat.getColor(resources, R.color.end_red, null))
            // https://stackoverflow.com/a/29146895
            icon = ResourcesCompat.getDrawable(resources, R.drawable.ic_round_stop_24, null)
        }

        viewSessionsButton.isEnabled = false
        toolbarSwitchGps.isEnabled = false
    }

    private fun stopTracking() {
        binding.timer.stop()
        mService.dataProcessor.stopRecording()
        binding.startStopButton.apply {
            text = getString(R.string.action_start_tracking)
            backgroundTintList =
                ColorStateList.valueOf(
                    ResourcesCompat.getColor(resources, R.color.start_green, null))
            icon = ResourcesCompat.getDrawable(resources, R.drawable.ic_round_play_arrow_24, null)
        }

        viewSessionsButton.isEnabled = true
        toolbarSwitchGps.isEnabled = true
    }

    /** Bind to LocalService. If it is not running, automatically start it up */
    private fun startAndBindToDataCollectionService() {
        if (ActivityCompat.checkSelfPermission(
            requireContext(), Manifest.permission.ACCESS_FINE_LOCATION) !=
            PackageManager.PERMISSION_GRANTED) {
            // request GPS permission, callback will re-call this function to start the service
            requestPermissionLauncher.launch(Manifest.permission.ACCESS_FINE_LOCATION)
            return
        }

        val dataCollectionServiceIntent =
            Intent(requireContext(), DataCollectionService::class.java)

        // mark the service as started so that it is not killed
        // https://stackoverflow.com/a/43742797
        // the startService and startForegroundService methods can be called
        // as many times as necessary -- there will always only be one
        // instance of the service running
        // https://developer.android.com/guide/components/services#StartingAService
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O)
            requireContext().startForegroundService(dataCollectionServiceIntent)
        else requireContext().startService(dataCollectionServiceIntent)

        // bind and automatically create the service
        requireContext()
            .bindService(dataCollectionServiceIntent, connection, Context.BIND_AUTO_CREATE)
    }
}
```

4.7.3 Recorded sessions view

In somewhat similar fashion to `MainActivity` and `PerformanceMonitorFragment`, `SessionsActivity` (page 58) is a container for the `SessionsFragment`. The fragment is added to the layout of the `SessionsActivity` through the activity's `supportFragmentManager`. The key difference is the addition of a toolbar to the activity, which displays a back arrow and a title. The toolbar back button is configured to act identically to a press on the device back button, returning the user to the main screen.

The `SessionFragment` also inherits from the Android `Fragment` base class. A connection to the database is initialised and the list layout is inflated. The `onCreateView` method sets the adapter used by the `RecyclerView` to a custom `SessionsListAdapter`. The `SessionsListAdapter` class (page 60) is responsible for populating the list with an element for each recorded session in the database; its parent class is the base `RecyclerView.ListAdapter` class. This custom child adapter is initialised with a single function as an argument that is called when one of the sessions in the list has its share button pressed. When the `SessionFragment` is created, it launched a coroutine task to load a list of sessions from the database before calling the list adapter's `submitList` function to populate the list. The initialisation code for the `SessionFragment` is presented on page 59.

```
private fun onSessionShareButtonClicked(session: Session) {
    CoroutineScope(Dispatchers.Main).launch { this: CoroutineScope
        // display a snackbar while we export the activity
        // this is best practice as opposed to showing a dialog
        val snackbar =
            Snackbar.make(
                    requireView(),
                    "Preparing export…",
                    Snackbar.LENGTH_INDEFINITE)
                .apply { show() }

        // actually export the activity to a file
        val file: File? =
            withContext(Dispatchers.Default) { this: CoroutineScope
                val sessionData = track.loadSession(session.trackId)
                if (sessionData.size <= 2) null ^withContext
                else FitFileExporter(requireContext()).exportTrackPoints(sessionData) ^withContext
            }

        // cancel the snackbar once we have finished exporting
        snackbar.dismiss()

        if (file == null)
            Snackbar.make(
                    requireView(),
                    "Activity too short to export",
                    Snackbar.LENGTH_LONG)
                .show()
        else onTrackExported(file)
    }
}
```

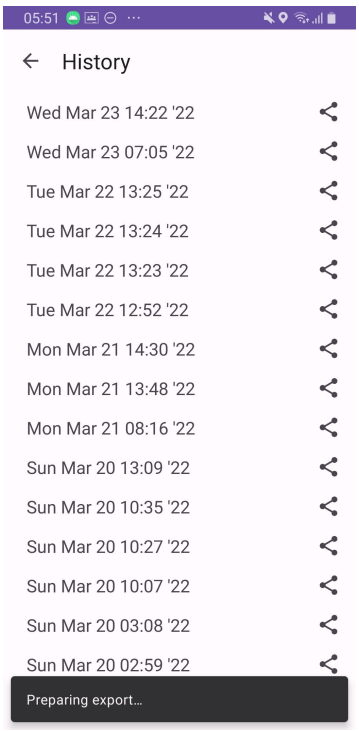

Figure 4.16: `onSessionShareButtonClicked` callback function code / snackbar appearance

The callback provided to the `SessionsListAdapter` is the `onSessionShareButtonClicked` function within the `SessionsFragment`. The `onSessionShareButtonClicked` method launches a coroutine task to export the file. While the file is being written, a *snackbar* is shown at the bottom of the screen. A snackbar is the preferred way of doing this, as opposed to a `progress dialog` which would show a spinny circle in the middle of the screen and block the user from doing anything else. The `onSessionShareButtonClicked` method is called when the user presses the share button for a session in the list. The `SessionFragment` loads the points of the recorded session from the database. If there are two or fewer datapoints (incredibly short session) then the export is cancelled. This is because the exporter relies on that fact that there must be a first and last data point for extracting start / stop

timestamps. The `exportTrackPoints` method of the `FitFileExporter` is called to export the session to a FIT file. If the export is successful, the generated `File` is passed to the `onTrackExported` method.

```
/**
 * Called once a session has been finished and successfully exported to a file.
 * https://developer.android.com/training/secure-file-sharing/share-file
 */
private fun onTrackExported(exportedFile: File) {
    val fileUri: Uri =
        try {
            FileProvider.getUriForFile(
                requireContext(), authority: "net.zeevox.nearow.fileprovider", exportedFile)
        } catch (e: IllegalArgumentException) {
            Log.e( tag: "File Selector", msg: "The selected file can't be shared: $exportedFile")
            null
        } ?: return

    startActivity(
        Intent.createChooser(
            Intent(Intent.ACTION_SEND).apply { this: Intent
                putExtra(Intent.EXTRA_STREAM, fileUri)
                putExtra(Intent.EXTRA_TITLE, value: "Exported fit file")
                putExtra(
                    Intent.EXTRA_TEXT, value: "Share the exported FIT file with another application")
                setDataAndType(fileUri, type: "application/vnd.ant.fit")
                flags = Intent.FLAG_GRANT_READ_URI_PERMISSION
            },
            title: null))
}
```

Figure 4.17: Code for the `onTrackExported` callback function

The `onTrackExported` method takes in a `File` and opens an Android *share sheet* that allows the user to send the file to another application installed on the device. Due to the restrictions on file access on Android in the name of privacy concerns, it is necessary to initialise a `FileProvider` that can then grant temporary read access to the receiving application. The file provider is registered in the `AndroidManifest.xml` and points to a `filepaths.xml` file shipped with the application that enumerates specific directories that can be shared from. The exported file is exported into the `exports/` directory of the internal application storage area. The `FileProvider` is then used to generate a URI for the exported file. The URI is then passed to the `Intent` that is used to launch the share sheet. Additional metadata is recorded into the intent, including the file mime type `application/vnd.ant.fit`[20].

5 | Testing

The MPAndroidChart library was used to display realtime charts of incoming data for debug purposes.

Figure 5.1: Iterations of the stroke rate debugger view

The first screenshot in figure 5.1 shows the acceleration pattern produced by tapping the phone periodically. A realtime smoothed line chart is updated on every new acceleration reading.

By the time the second screenshot in 5.1 was taken, a bar chart was added to the view to show the amplitude of each frequency bin in the Discrete Fourier transform. The acceleration pattern is that for shaking the phone wildly (small period), which is why the peak in the Fourier transform is nearer to the top of the chart.

The third screenshot of 5.1 is once again of tapping the phone, but this time to a metronome set to 105bpm. The app correctly detects 105bpm. Admittedly, this specific frequency was not chosen by accident. It is, of course, way more frequent than a rowing stroke could ever be. With a window size of 256 samples, it was empirically determined that stroke rate was accurate to roughly the nearest $\sim$ 6bpm. In this context, that meant that it would in essence round to one of these stroke rates: $\ldots, 99, 105, 111, \ldots$.

With the new autocorrelation algorithm in place, the application was tested on the water with a waterproof Samsung Galaxy S8 and Google Pixel 2. Displayed stroke rates seemed plausible, and were

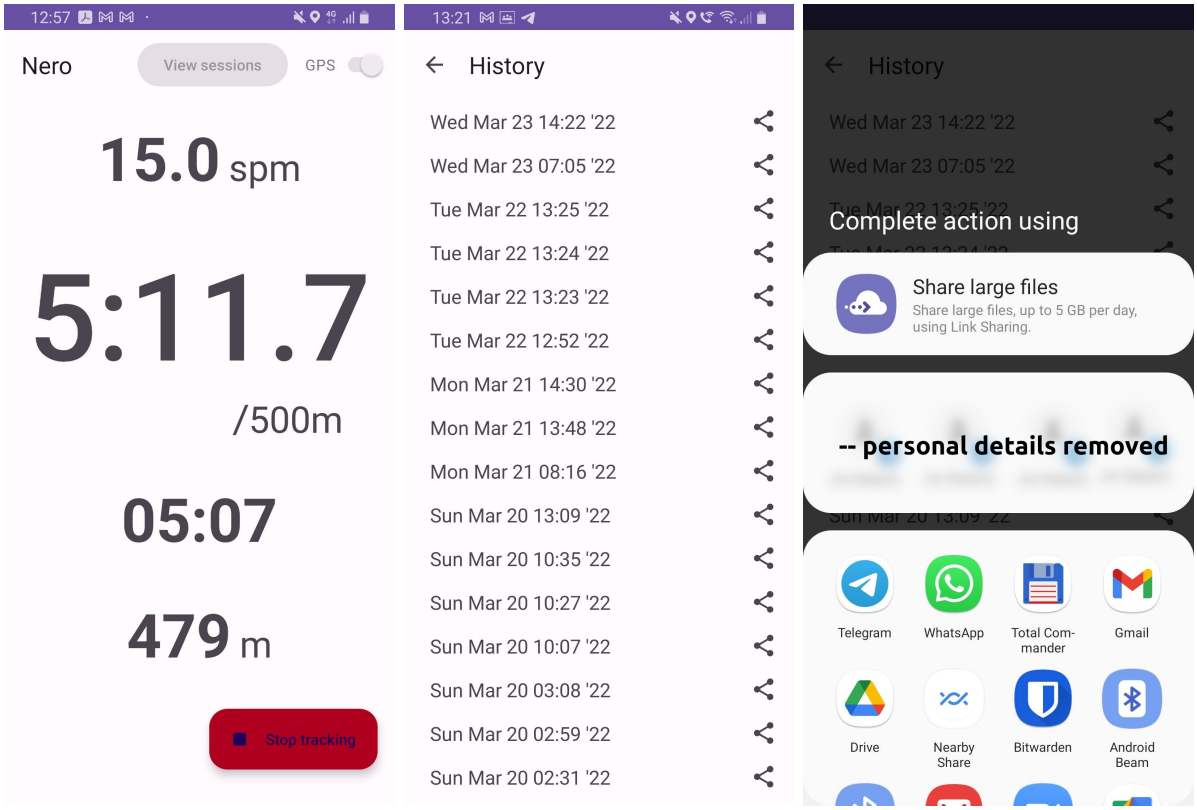

Figure 5.2: application tracking view / session history / session export share sheet

usually within a split of the SpeedCoach.

After the outing, the session was exported to the user's devices. The exported FIT file was imported into Strava for further analysis. Average stroke rate matched, the SpeedCoach reported 17.5 and Strava 18 (it rounds to the nearest integer). Average split was similar, albeit slightly lower than a SpeedCoach. This is presumed to be because the StrokeCoach does not record the time when the boat drifts with the stream, whereas the application does, increasing the total distance covered. The total distance covered was 6% greater than that reported by the SpeedCoach. This is within acceptable margins of error, but suggests that stillness detection can still be worked on.

Unlike many of the other parts of this application, the `UnitConverter` class can actually be tested with unit tests in the `UnitConversionTests` class. The JUnit testing library was selected for its simplicity and good integration with the Android Studio development environment. Speed to split and speed to power conversion is tested, as shown in figure 5.4.

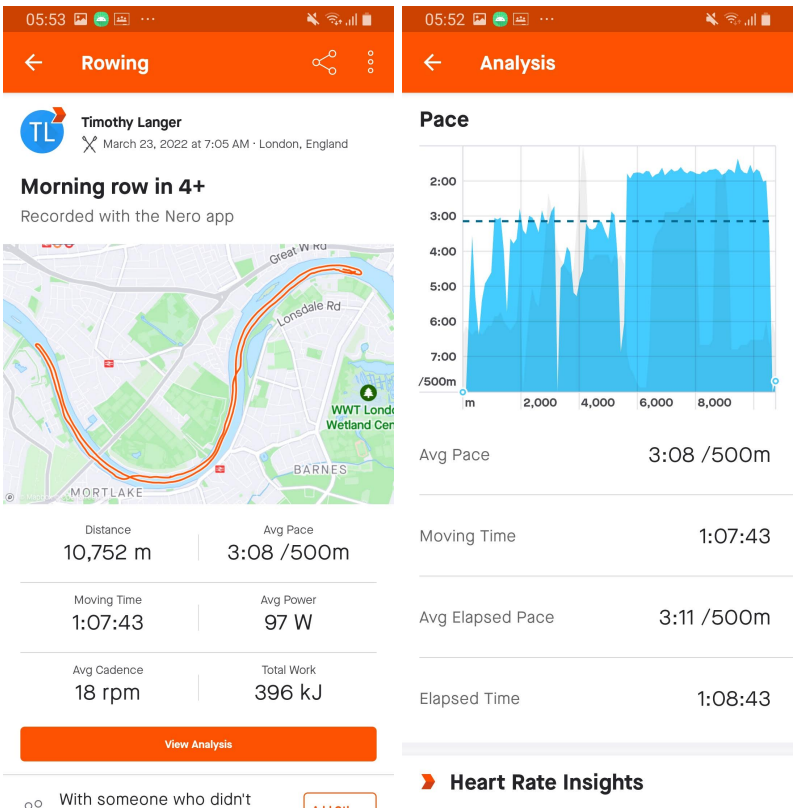

Figure 5.3: Strava session overview / pace analysis

```
package net.zeevox.nearow

import net.zeevox.nearow.utils.UnitConverter
import org.junit.Assert.assertEquals
import org.junit.Test

class UnitConversionTests {
    @Test
    fun speedToSplitFormatTest() {
        assertEquals( expected: "2:05.0", UnitConverter.speedToSplitFormatted( speed: 4.0f))
        assertEquals( expected: "1:40.0", UnitConverter.speedToSplitFormatted( speed: 5.0f))
        assertEquals( expected: "1:02.5", UnitConverter.speedToSplitFormatted( speed: 8.0f))
    }

    @Test
    // checking with values from
    // https://www.concept2.com/indoor-rowers/training/calculators/watts-calculator
    fun speedToWattsTest() {
        assertEquals( expected: 179.2, UnitConverter.speedToWatts( speed: 4.0f),  delta: 0.1)
        assertEquals( expected: 350.0, UnitConverter.speedToWatts( speed: 5.0f),  delta: 0.1)
        assertEquals( expected: 1433.6, UnitConverter.speedToWatts( speed: 8.0f),  delta: 0.1)
    }
}
```

Figure 5.4: Unit tests for the `UnitConverter` class

6 | Evaluation

6.1 Feedback from a 3rd party

The application was installed on a OnePlus 8T belonging to James Trotman (interviewed earlier) and his opinion was requested after a training session in the eight. He trialled the application on his device alongside an NK SpeedCoach; the results were discussed after the outing.

The user found their way around the app very quickly. The only note to make is that, as on installation there were no already recorded sessions, the purpose of the blank screen shown in the "view sessions" menu was not clear.

James said that the stroke rate was rather accurate, always within a single spm of the stroke rate given by the SpeedCoach. He did comment that the split, however, was rather jumpy, and was not as good as the SpeedCoach; according to James, it was oscillating around the same split as the SpeedCoach, but only accurate, as per his estimate, to the nearest 10 seconds. It is proposed that this is because the SpeedCoach samples the split once per stroke, whereas the application samples once per second, which, at low stroke rates, results in several samples being taken per stroke. This means that it reports a slower split on the recovery and a fast split on the drive. This could be corrected by implementing a moving average or changing the GPS client to perform sampling as a function of the rate.

Battery usage was also discussed. James' phone battery decreased by around 40% over the course of a two-hour row. Although not a major concern, we agreed that ideally battery usage should be decreased. The screen is the major factor for battery drain, and on a sunny day the high brightness further causes increased battery consumption. This could be targeted by, for example, giving the UI a black background to reduce battery usage on OLED screens.

6.2 Objectives analysis

The specification described in section 2.6.4 is copied below, augmented with a discussion of the achievements of the application.

1. be an application installable on any modern Android smartphone,

   The application was tested on several Android devices: a Pixel 2 (2017), a Pixel 4a 5G (2020), James' OnePlus 8T (2020) and a Samsung Galaxy S8 (2017). No device-specific issues were discovered.

2. require minimal configuration and interaction from the end user, and ideally none at all,

   The application is started up and starts measuring stroke rate and split immediately. The only initial configuration required is a request to access the user's GPS location. The application does not require any user interaction once recording is started.

3. be as easy to use and self-explanatory as possible for a new user,

James found his way around the application very quickly. He mentioned that he liked the long-press requirement for the stop button.

4. calculate the following metrics, as accurately as possible:

   - the boat's stroke rate,

     The application calculated stroke rate is accurate to within a single spm of the stroke rate given by the SpeedCoach. The SpeedCoach, however, does excel in picking up stroke rate changes almost instantaneously, whereas the application could take up to 10 seconds for particularly sudden changes in stroke rate.

   - the boat's split,

     The average boat split as found by the application is very close to that calculated by the SpeedCoach. However, realtime stroke rate is not as accurate as the SpeedCoach.

   - the total distance rowed over the course of the session,

     The SpeedCoach does not measure distance covered while drifting with the current. The application does, which means that the distance covered is higher. However, this does not make it inaccurate.

5. display the above metrics in a realtime and an easy-to-read manner,

   Metrics are updates once per second, which is more than enough. Potentially even too often, which is why split can seem to jump around.

6. collect data during a rowing session using nothing but the hardware on the smartphone,

   The application does not require any additional hardware. A waterproof case may be a requirement for some, but it is not necessary for the functionality.

7. provide start/stop functionality to record the rowing session for later review,

   A start/stop button is provided. Sessions are recorded to a database.

8. allow the user to export the recorded rowing session as a `FIT` activity file that can be imported into 3rd-party software, such as Strava, containing

   - the geolocation of the boat,
   - the speed of the boat,
   - the cadence, or stroke rate, of the rowers,
   - the estimated power generated by the rowers

   FIT activity export functionality is present. Files generated by the application can be imported into Strava and viewed on a map alongside a chart of split.

9. be as battery-efficient as possible

   Optimisations have been made and algorithms carefully considered to make them as battery-efficient as possible. Perhaps the UI could be adapted to reduce the battery drain caused by a bright white screen.

7 | Appendix

```
package net.zeevox.nearow.ui

import ...

class SessionsActivity : AppCompatActivity() {

    /** for accessing UI elements, bind this activity to the XML */
    private lateinit var binding: ActivitySessionsBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        setTheme(R.style.Theme_Nearow)

        super.onCreate(savedInstanceState)

        binding = ActivitySessionsBinding.inflate(layoutInflater)
        setContentView(binding.root)

        if (savedInstanceState == null) {
            supportFragmentManager.commit {
                // Android Dev best practices say to always setReorderingAllowed to true
                // https://developer.android.com/guide/fragments/create
                setReorderingAllowed(true)
                add<SessionsFragment>(R.id.fragment_container_view)
            }
        }

        val toolbar: MaterialToolbar = findViewById(R.id.sessions_toolbar)
        setSupportActionBar(toolbar)
        supportActionBar?.apply {
            setDisplayHomeAsUpEnabled(true)
            setDisplayShowHomeEnabled(true)
        }
    }

    override fun onSupportNavigateUp(): Boolean {
        onBackPressed()
        return super.onSupportNavigateUp()
    }
}
```

```kotlin
package net.zeevox.nearow.ui.fragment

import ...

/** A fragment representing a list of Items. */
class SessionsFragment : Fragment() {

    /** The application database */
    private lateinit var db: TrackDatabase

    /** Interface with the table where individual rate-location records are stored */
    private lateinit var track: TrackDao

    /** Coroutine scope for database queries and other long-running operations */
    private val scope = CoroutineScope(Dispatchers.Default)

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        db =
            Room.databaseBuilder(
                    requireContext(), TrackDatabase::class.java, DataProcessor.DATABASE_NAME)
                .build()

        track = db.trackDao()
    }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?,
    ): View? {
        val view = inflater.inflate(R.layout.fragment_sessions_list, container, false)

        // Set the adapter
        if (view is RecyclerView)
            with(view) {
                layoutManager = LinearLayoutManager(context)
                setHasFixedSize(true)
                val sessionsAdapter = SessionsListAdapter(::onSessionShareButtonClicked)
                adapter = sessionsAdapter
                scope.launch { sessionsAdapter.submitList(track.getSessions()) }
            }
        return view
    }

    private fun onSessionShareButtonClicked(session: Session) {
        // see other screenshots
    }

    private fun onTrackExported(exportedFile: File) {
        // see other screenshots
    }

}
```

```
package net.zeevox.nearow.ui.recyclerview

import ...

/** [ListAdapter] that can display a [Session]. */
class SessionsListAdapter(val clickListener: (Session) -> Unit) :
    ListAdapter<Session, SessionsListAdapter.ViewHolder>(SessionDiff()) {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder =
        ViewHolder(
            FragmentSessionsBinding.inflate(LayoutInflater.from(parent.context), parent, false))

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val item: Session = getItem(position)
        holder.contentView.text = item.toString()
        holder.shareButton.setOnClickListener { clickListener(item) }
    }

    inner class ViewHolder(binding: FragmentSessionsBinding) :
        RecyclerView.ViewHolder(binding.root) {
        val contentView: TextView = binding.content
        val shareButton: ImageButton = binding.shareButton

        override fun toString(): String {
            return super.toString() + " '" + contentView.text + "'"
        }
    }

    internal class SessionDiff : DiffUtil.ItemCallback<Session>() {
        override fun areItemsTheSame(oldItem: Session, newItem: Session): Boolean =
            oldItem === newItem

        override fun areContentsTheSame(oldItem: Session, newItem: Session): Boolean =
            oldItem.trackId == newItem.trackId
    }
}
```

```
package net.zeevox.nearow.output

import ...

class FitFileExporter(private val context: Context) {

    suspend fun exportTrackPoints(trackPoints: List<TrackPoint>): java.io.File {
        if (trackPoints.size <= 2)
            throw IllegalArgumentException("Too few trackPoints submitted to export to a file")

        val activity = createActivityFromTrackPoints(trackPoints)

        // create directory if not exists
        val directory = java.io.File(context.filesDir.path + "/exports")
        if (!directory.exists()) directory.mkdir()

        val file = java.io.File(directory, getFilenameForTimestamp(trackPoints.first().timestamp))
        writeMessagesToFile(activity, file)
        return file
    }

    private fun createActivityFromTrackPoints(trackPoints: List<TrackPoint>): List<Mesg> {

        val firstPoint = trackPoints.first()
        val lastPoint = trackPoints.last()

        val activityStartTime = DateTime(Date(firstPoint.timestamp))
        val activityEndTime = DateTime(Date(lastPoint.timestamp))

        val startLat = firstPoint.latitude?.let { decimalToGarmin(it) }
        val startLong = firstPoint.longitude?.let { decimalToGarmin(it) }
        val endLat = lastPoint.latitude?.let { decimalToGarmin(it) }
        val endLong = lastPoint.longitude?.let { decimalToGarmin(it) }

        val elapsedTime =
            ((activityEndTime.timestamp - activityStartTime.timestamp) / 1000).toFloat()

        return buildList {
            // Every FIT file MUST contain a File ID message
            add(getFileMetadata(activityStartTime))
            // A Device Info message is a BEST PRACTICE for FIT ACTIVITY files
            add(getDeviceInfo(activityStartTime))
            // Timer Events are a BEST PRACTICE for FIT ACTIVITY files
            add(createStartEvent(activityStartTime))
            // Create a RecordMesg for each TrackPoint and add it to the output queue
            trackPoints.mapTo(this, ::getRecordMesgForTrackPoint)
            // Every FIT file MUST contain at least one Lap message
            add(createLap(activityEndTime, elapsedTime, startLat, startLong, endLat, endLong))
            // Mark the activity as ended
            add(createEndEvent(activityEndTime))
            // Every FIT file MUST contain at least one Session message
            add(createSession(activityEndTime, elapsedTime, startLat, startLong))
            // Every FIT ACTIVITY file MUST contain EXACTLY one Activity message
            add(createActivityMesg(activityEndTime))
        }
    }
```

```
    private suspend fun writeMessagesToFile(messages: List<Mesg?>, file: java.io.File) {
        // Create the output stream
        val encoder: FileEncoder =
            try {
                FileEncoder(file, Fit.ProtocolVersion.V2_0)
            } catch (e: FitRuntimeException) {
                Log.e(javaClass.simpleName, "Error opening file ${file.name}")
                e.printStackTrace()
                return
            }

        withContext(Dispatchers.IO) { for (message in messages) encoder.write(message) }

        // Close the output stream
        try {
            encoder.close()
        } catch (e: FitRuntimeException) {
            Log.e(javaClass.simpleName, "Error closing encode.")
            e.printStackTrace()
            return
        }

        Log.d(javaClass.simpleName, "Encoded FIT Activity file ${file.name}")
    }

    companion object {

        // The combination of manufacturer id, product id, and serial number should be unique.
        // When available, a non-random serial number should be used.
        private const val TRACKING_PRODUCT_ID: Int = 1
        private const val MANUFACTURER_ID: Int = Manufacturer.DEVELOPMENT
        private const val SOFTWARE_VERSION = BuildConfig.VERSION_CODE
        private const val SERIAL_NUMBER: Long = 2469834L

        /**
         * Garmin stores lat/long as integers. Each decimal degree represents 2^32 / 360 = 11930465
         * https://gis.stackexchange.com/a/368905
         */
        fun decimalToGarmin(pos: Double): Int = (pos * 11930465).toInt()

        private fun getRecordMesgForTrackPoint(trackPoint: TrackPoint) =
            RecordMesg().apply {
                timestamp = DateTime(Date(trackPoint.timestamp))
                speed = trackPoint.speed
                power = UnitConverter.speedToWatts(speed).toInt()
                cadence = trackPoint.strokeRate.toInt().toShort()
                trackPoint.latitude?.let { positionLat = decimalToGarmin(it) }
                trackPoint.longitude?.let { positionLong = decimalToGarmin(it) }
            }

        private fun createStartEvent(start: DateTime): EventMesg =
            EventMesg().apply {
                timestamp = start
                event = Event.TIMER
                eventType = EventType.START
                timerTrigger = TimerTrigger.MANUAL
                eventGroup = 0
            }
```

```
    private fun createEndEvent(end: DateTime): EventMesg =
        EventMesg().apply {
            timestamp = end
            event = Event.TIMER
            eventType = EventType.STOP
            timerTrigger = TimerTrigger.MANUAL
            eventGroup = 0
        }

    private fun createLap(
        lapStartTime: DateTime,
        elapsedTime: Float,
        _startPositionLat: Int? = null,
        _startPositionLong: Int? = null,
        _endPositionLat: Int? = null,
        _endPositionLong: Int? = null,
    ): LapMesg =
        LapMesg().apply {
            messageIndex = 0
            startTime = lapStartTime
            timestamp = lapStartTime

            totalElapsedTime = elapsedTime
            totalTimerTime = elapsedTime

            _startPositionLat?.let { startPositionLat = it }
            _startPositionLong?.let { startPositionLong = it }
            _endPositionLat?.let { endPositionLat = it }
            _endPositionLong?.let { endPositionLong = it }

            event = Event.LAP
            eventType = EventType.STOP
            lapTrigger = LapTrigger.MANUAL
            sport = Sport.ROWING
            subSport = SubSport.GENERIC
        }

    private fun getDeviceInfo(mesgTimestamp: DateTime): DeviceInfoMesg =
        DeviceInfoMesg().apply {
            deviceIndex = DeviceIndex.CREATOR
            manufacturer = MANUFACTURER_ID
            product = TRACKING_PRODUCT_ID
            serialNumber = SERIAL_NUMBER
            softwareVersion = SOFTWARE_VERSION.toFloat()
            timestamp = mesgTimestamp
        }
```

```
        private fun createSession(
            activityStartTime: DateTime,
            elapsedTime: Float,
            _startPositionLat: Int? = null,
            _startPositionLong: Int? = null,
        ): SessionMesg =
            SessionMesg().apply {
                messageIndex = 0
                firstLapIndex = 0
                numLaps = 0

                startTime = activityStartTime
                timestamp = activityStartTime

                totalElapsedTime = elapsedTime
                totalTimerTime = elapsedTime

                _startPositionLat?.let { startPositionLat = it }
                _startPositionLong?.let { startPositionLong = it }

                sport = Sport.ROWING
                subSport = SubSport.GENERIC

                event = Event.SESSION
                eventType = EventType.STOP
            }

        private fun getFileMetadata(startTime: DateTime): FileIdMesg =
            FileIdMesg().apply {
                type = File.ACTIVITY
                manufacturer = MANUFACTURER_ID
                product = TRACKING_PRODUCT_ID
                timeCreated = startTime
                serialNumber = SERIAL_NUMBER
            }

        private fun createActivityMesg(activityStartTime: DateTime): ActivityMesg {
            val timeZone: TimeZone = TimeZone.getDefault()
            val timezoneOffset: Long = (timeZone.rawOffset + timeZone.dstSavings) / 1000L
            return ActivityMesg().apply {
                timestamp = activityStartTime
                numSessions = 1
                type = Activity.MANUAL
                event = Event.ACTIVITY
                eventType = EventType.STOP
                localTimestamp = activityStartTime.timestamp + timezoneOffset
                totalTimerTime =
                    (activityStartTime.timestamp - activityStartTime.timestamp).toFloat()
            }
        }

        private fun getFilenameForTimestamp(timestamp: Long): String {
            // Create a DateFormatter object for displaying date in specified format.
            val formatter = SimpleDateFormat("yyyy-MM-dd-HH-mm-ss", Locale.UK)

            // Create a calendar object that will convert the date and time value in milliseconds to
            // date.
            val calendar = Calendar.getInstance().apply { timeInMillis = timestamp }

            return "Nero-${formatter.format(calendar.time)}.fit"
        }
    }
}
```

Bibliography

[1] *The Boat Race : Arup & The Boat Race Company*. Oct. 2017. URL: https://www.theboatrace.org/wp-content/uploads/Arup-Report-The-Boat-Race.pdf (visited on 16th Sept. 2021).

[2] Dr Valery Kleshnev. 'Analysis of boat acceleration'. en. In: *Rowing Biomechanics Newsletter* (Nov. 2012), p. 2. URL: http://www.biorow.com/RBN_en_2012_files/2012RowBiomNews11.pdf (visited on 21st Sept. 2021).

[3] *Heuers on the Sea — 25 Years of Yacht Timers*. Aug. 2014. URL: http://www.onthedash.com/heuer-yacht-timers/ (visited on 20th Mar. 2022).

[4] *PaceCoach - Australian National Maritime Museum*. 1990. URL: https://collections.sea.museum/objects/12569/pacecoach-rowing-computer (visited on 20th Mar. 2022).

[5] *Motion sensors : Android Developers*. URL: https://developer.android.com/guide/topics/sensors/sensors_motion (visited on 9th Sept. 2021).

[6] *Android Benchmarks : Geekbench*. URL: https://browser.geekbench.com/android-benchmarks (visited on 9th Sept. 2021).

[7] Google. *Kotlin and Android | Android Developers*. 2022. URL: https://developer.android.com/kotlin.

[8] *Linear acceleration sensor : Android Developers*. URL: https://source.android.com/devices/sensors/sensor-types#linear_acceleration (visited on 9th Sept. 2021).

[9] Peter Fischer, Klaus Sembritzki and Andreas Maier. *Figure 2.11, Sine signal with additive noise after processing with a low-pass filter and a high-pass filter*. en. Text. Aug. 2018. URL: https://www.ncbi.nlm.nih.gov/books/NBK546153/figure/ch2.fig15/ (visited on 21st Mar. 2022).

[10] *Low-pass filter*. en. Page Version ID: 1039066300. Aug. 2021. URL: https://en.wikipedia.org/w/index.php?title=Low-pass_filter&oldid=1039066300 (visited on 21st Mar. 2022).

[11] Russell Bradford. 'Sliding is smoother than jumping'. In: (2005), p. 4. URL: http://www.music.mcgill.ca/~ich/research/misc/papers/cr1137.pdf (visited on 30th Dec. 2021).

[12] *Autocorrelation*. en. Page Version ID: 1059192357. Dec. 2021. URL: https://en.wikipedia.org/w/index.php?title=Autocorrelation&oldid=1059192357.

[13] *Save data in a local database using Room*. en. URL: https://developer.android.com/training/data-storage/room (visited on 8th Mar. 2022).

[14] *FIT SDK | Garmin Developers*. URL: https://developer.garmin.com/fit/overview/ (visited on 8th Oct. 2021).

[15] *fft - Is there a way to do a DFT with low latency and fine frequency resolution at low frequencies?* URL: https://dsp.stackexchange.com/questions/40296/is-there-a-way-to-do-a-dft-with-low-latency-and-fine-frequency-resolution-at-low (visited on 8th Jan. 2022).

[16] Google. *Coroutine context and dispatchers*. URL: https://kotlinlang.org/docs/coroutine-context-and-dispatchers.html.

[17] *Watts Calculator*. en. Text. May 2012. URL: https://www.concept2.com/indoor-rowers/training/calculators/watts-calculator (visited on 18th Oct. 2021).

[18] *wgs84 - Garmin FIT Coordinate System*. URL: https://gis.stackexchange.com/questions/371656/garmin-fit-coodinate-system (visited on 13th Mar. 2022).

[19] *Cookbook | FIT SDK | Garmin Developers*. URL: https://developer.garmin.com/fit/cookbook/datetime/ (visited on 13th Feb. 2022).

[20] *Flexible and Interoperable Data Transfer*. en. July 2020. URL: https://en.everybodywiki.com/Flexible_and_Interoperable_Data_Transfer.

[21] Olav Holten. *Imagick-FT*. Nov. 2020. URL: https://github.com/olavholten/imagick-FT.

[22] *Android getResources().getDrawable() deprecated API 22*. URL: https://stackoverflow.com/questions/29041027/android-getresources-getdrawable-deprecated-api-22.

[23] *android - Does bindService() with BIND_AUTO_CREATE always creates new service instance?* URL: https://stackoverflow.com/questions/43742758/does-bindservice-with-bind-auto-create-always-creates-new-service-instance.

[24] *Services overview*. en. URL: https://developer.android.com/guide/components/services.

[25] *Sharing a file*. en. URL: https://developer.android.com/training/secure-file-sharing/share-file.

[26] *Create a fragment*. en. URL: https://developer.android.com/guide/fragments/create.

[27] *Request location updates*. en. URL: https://developer.android.com/training/location/request-updates.

[28] *java - Running code in main thread from another thread*. URL: https://stackoverflow.com/questions/11123621/running-code-in-main-thread-from-another-thread.