Smartphone-based tracking of rowing training

AQA Computer Science A-Level

Non-Examined Assessment

Timothy Langer

St Paul's School

Dr C. A. Harrison

21st March 2022

Contents

1 | Introduction

Rowing in its modern form developed in England in the 1700s, 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.[5]

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. Regattas are preferable for spectating, because it is obvious as to which boat is physically in the lead. Typically, head races take place in in the autumn and winter, whereas regattas are frequent in late spring and summer. Head races in particular require accurate timing and tracking of competitors' boats, seeing as the results are based on this information.

1.0.1 The rowing stroke

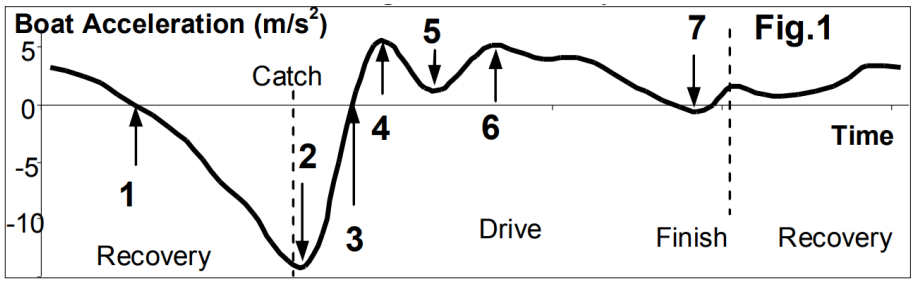

Figure 1.1: Typical pattern of boat acceleration during the stroke cycle [3]

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.

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 steering line changes, and does not allow for realtime feedback of the boat's speed.

2.2.2 Rowing stopwatches

The 1960s saw the emergence of specialised mechanical 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.

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 1990s saw the first PaceCoach, which coupled with inboard and outboard sensors such as an impeller to calcualte distance covered and real-time split.[1]

[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 [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 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. 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 many 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 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. display, in a realtime and an easy-to-read manner:
   - the boat's stroke rate,
   - the boat's split,
   - the total distance rowed over the course of the session,
   - the elapsed total session time
4. collect data during a rowing session using nothing but the hardware on the smartphone,
5. provide start/stop functionality to record the rowing session for later review,
6. 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
7. be as battery-efficient as possible

[5] modern version of an outrigger that spans across the middle from one saxboard to the other
[6] released within the last five years

3 | Design

3.1 Input

3.1.1 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.[10] 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.

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.[9]

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.

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. It is an efficient ($O(n)$) algorithm that can be used to calculate the frequency of a signal.

> The sliding DFT is a recursive algorithm to compute successive Fourier transforms of input data frames that are a single sample apart.[2]

Although the SDFT is an efficient algorithm, it is nonetheless rather calculation-heavy. Stroke rate calculation ought to be performed on a separate thread to not slow down the graphical user interface.

3.2.2 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.
- Kotlin makes handling nullability of variables and objects far easier, to help avoid Java's dreaded `NullPointerException`.

4 | Technical solution

4.1 Data collection

4.2 Data processing

4.2.1 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.

> 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]. ([2])

```kotlin
/**
* 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. 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 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. A DC[1] and Nyquist[2] bin are also included, so the number of available frequency bins is $\frac{N}{2} + 2$.

```
// 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

The sum of the real components of all the frequency bins is stored and recalculated on each addition of a new sample. This is also 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. A function is also provided to calculate the dominant frequency at the given moment in time – it returns the wavelength of the frequency bin with the greatest magnitude. The code for the two aforementioned methods is shown in figure 4.2.

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.

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

```
// 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

```
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

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.[7] 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.

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.

4.2.2 Autocorrelator.kt

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

`getFrequencyScores(data)`

The method shown in figure 4.5 slides 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.

$$\frac{\sum\left(X_i - \mu\right)\left(X_{i-\tau} - \mu\right)}{\sum\left(X_i - \mu\right)^2} \tag{4.1}$$

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

```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

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.

`getBestFrequency(correlations)`

```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 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.

4.2.3 `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.

```kotlin
/**
 * 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`.

```kotlin
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 list in Kotlin 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.

4.2.4 `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.[8]

```
/**
 * 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.

Figure 4.11 show the various configurable compile-time constants are configurable. In theory these could

```
/** 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

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 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.

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 a error message is logged if they are called from a non-UI thread.

4.3 Data storage

4.4 File export

4.5 User interface

```
// 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"
```

Figure 4.11

```
/**
 * 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

Bibliography

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

[2] 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).

[3] 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).

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

[5] *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).

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

[7] *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).

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

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

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