```kotlin
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
```

```kotlin
/**
 * 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)
            }
        }
    }
```

```kotlin
/**
 * 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)
        }
    }
}
```