UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FISICAS Y MATEMATICAS
DEPARTAMENTO DE INGENIERIA ELECTRICA

IMPLEMENTACIÓN DE MÉTODOS SUB-ÓPTIMOS DE ESTIMACIÓN Y PREDICCIÓN DEL ESTADO-DE-CARGA DE BATERÍAS DE ION-LITIO EN AMBIENTE ANDROID

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL ELECTRICISTA

ALEX JAVIER DÍAZ MILLÁN

PROFESOR GUÍA
MARCOS ORCHARD CONCHA

MIEMBROS DE LA COMISIÓN
HECTOR AUGUSTO ALEGRÍA
JAIME ALÉE GIL

SANTIAGO DE CHILE
2015

Resumen

La masiva demanda de combustibles derivados del petróleo abre la puerta a la búsqueda de nuevas formas de suplir las necesidades energéticas en el mundo. Debido a esto, los acumuladores de energía tienen un rol fundamental en el sector industrial en los últimos años. En particular, las baterías de Ion-Litio que por sus características las transforma en elementos claves dentro del diseño de sistemas autónomos. Los vehículos eléctricos nacen para poder suplir esta gran demanda y en particular, las bicicletas eléctricas no sólo permiten utilizar menos energía, sino que poder transportarse con mayor libertad y fluidez en las grandes ciudades.

Con el nacimiento de esta nueva alternativa para el trasporte se hace necesario poder contar con sistemas de supervisión de las baterías que puedan entregar información importante al usuario sobre su Estado de Carga (SOC, por sus siglas en inglés) y estado de salud (SOH, por sus siglas en inglés). Esta información le será útil no solo al usuario sino que también a la industria encargada de masificar el producto, ya que de esta forma puede generar planes preventivos y de mantención para optimizar recursos y mejorar la calidad de servicio. En la actualidad, hay múltiples productos que son capaces de estimar el SOC y predecir el tiempo de descarga (EOD, por sus siglas en inglés) de una batería de Ion-Litio; sin embargo, es mediante los métodos de Filtro de Partículas (FP) donde se obtienen los mejores resultados.

Con el objeto de unir tanto la teoría de los métodos de estimación y pronóstico del SOC de las baterías con el uso de la tecnología de acceso común, nace el objetivo de este Trabajo de Título, el cual busca implementar métodos sub-óptimos de estimación y pronóstico del Estado de Carga de baterías de Ion-Litio en un ambiente Android. Es decir, a través del presente trabajo se integraran estos métodos dentro de una aplicación (app) Android de uso común, aprovechando las capacidades y recursos de los Smartphone en la actualidad.

Se busca programar un algoritmo de estimación y pronóstico (AEP) en lenguaje Java y mostrar los resultados a través de la pantalla al usuario mediante una *app*. Se muestra la implementación realizada y su entorno gráfico basados en los requerimientos tales como uso de protocolo Bluetooh y conexión USB, almacenamiento de datos y servicios web. Finalmente, se entregan los resultados obtenidos del AEP tanto en la estimación y predicción del SOC como en la del EOD. Esto en conjunto con un análisis del uso de recursos de procesamiento y memoria que requiere el AEP para funcionar.

Como conclusión de los resultados expuestos en este trabajo, se puede afirmar que la implementación del algoritmo permite: (i) obtener información sobre el Estado de Carga de la batería de Ion-Litio en tiempo real; (ii) visualizar la predicción del tiempo de descarga basados en el perfil de uso de la bicicleta eléctrica. Todo esto a través de una *app* Android.

A Dios, mi esposa y mi familia.

Agradecimientos

Han sido largos años de esfuerzo y dedicación a una de las cosas que más me gusta en mi vida, el estudiar Ingeniería. Esta hermosa carrera permite vivir una experiencia hermosa que es crear cosas nuevas. Estos años han sido de constante aprendizaje intelectual, pero lo que más rescato es lo que he logrado aprender en lo personal. Aprendí a ganar amigos, aprendí que la responsabilidad y la perseverancia generalmente son más importantes que la misma inteligencia, y que la combinación de éstas logra el éxito en lo que uno se proponga. Aprendí que el compañerismo y el ayudar a los demás te hace mejor persona. Aprendí que el ser ingeniero no es más que una herramienta para vivir la vida tal como uno decida vivirla.

Si hoy estoy escribiendo estas líneas es gracias a mi esposa Romina que con su amor me ha entregado un apoyo incondicional en este camino. No solo estuvo conmigo en cada momento sino que me ayudó a hacer las cosas de la mejor forma posible. Me instó a mejorar y a superarme, a creer en mí mismo. Sin duda que no lo hubiese podido lograr sin ella y su fe en mí. Estoy muy agradecido de mis padres, Alex y Nitza, que fueron los que me impulsaron a tomar este camino, con su apoyo y su amor me guiaron para que lograra entrar a esta facultad. Me ayudaron desde pequeño a confiar en que lo podía lograr. Agradezco a mis abuelos que con su amor y servicio hicieron todo lo posible para que nunca me faltara nada. Agradezco a mis hermanos Dianne y Lucas y también a mi tío Javier, que con su amor me permitían compartir mis momentos de alegría. Agradezco a toda mi familia y a las personas que tengo en mi corazón y que me permitieron tener un lindo recuerdo de este camino.

Por último, agradezco a mi profesor Marcos Orchard quien impulsó este trabajo y me dio la oportunidad de realizar esta memoria. Agradezco toda la buena disposición, la confianza y las oportunidades dadas durante mi carrera.

Tabla de Contenido

Capítulo I 1

1.1 Introducción 1

1.2 Objetivo General del Trabajo de Título 2

1.3 Objetivos Específicos del Trabajo de Título 2

Capítulo II 3

2.1 Dispositivos almacenadores de energía y Estado-de-Carga 3

2.2 Estimación y Predicción del Estado de Carga con Filtro de Partículas 4

2.2.1 Inferencia Bayesiana 6

2.2.2 Modelación de Eventos Aleatorios mediante el uso de Cadenas de Markov ... 9

2.3 Algoritmo de Estimación y Predicción del Estado-de-Carga en Baterías de Ion-Litio 10

2.3.1 MATLAB® 10

2.3.2. Algoritmo de Estimación y Predicción del Estado de Carga 11

2.4 Lenguaje de programación Java 15

2.4.1 Etapas de desarrollo de un programa en Java 15

2.4.2 Estructura básica de un programa en Java 16

2.4.3 Elementos básicos de la orientación a objetos 17

2.4.4 Librerías 18

2.4.5 Análisis de rendimiento 19

2.5 Ambiente Android 20

2.6 Contextualización 23

Capítulo III 25

3.1. Requerimientos 25

3.1.1. Recibir datos vía Bluetooh / conexión USB 25

3.1.2. Almacenar datos recibidos y generados 25

3.1.3. Estimar y pronosticar el Estado de Carga de la batería 25

3.1.4. Subir información obtenida a la nube 26

3.2. Determinación de Componentes Principales a utilizar 26

3.2.1. Requerimiento 1: Recibir datos vía Bluetooh 4.0 o USB 26
3.2.2. Requerimiento 2: Datos generados almacenados 26
3.2.3. Requerimiento 3: Algoritmo de estimación y pronóstico off-line................. 27
3.2.4. Requerimiento 4: Subir información a un servidor 28
3.3. Diseño e implementación de cada Requerimiento... 28
3.3.1. Arquitectura de Hardware 28
3.3.2. Arquitectura de Software.. 29
3.3.3. Algoritmo de estimación y pronóstico off-line... 33
2.3.4 Algoritmo de estimación y pronóstico en una aplicación Android 38
Capítulo IV 42
3.1 Datos utilizados.. 42
3.2 Resultados del algoritmo de estimación y pronóstico del Estado de Carga en Java 43
3.3 Performance Java vs MATLAB® 47
3.4 Algoritmo de estimación y pronóstico del Estado de Carga dentro de una aplicación... 49
3.4.1 Cálculo de tiempos de importancia 50
Conclusiones.. 52
Trabajo Futuro 54
Bibliografía.. 55

Índice de Figuras

Figura 1. Cadena de Markov Primer Orden 2 estados utilizada para la predicción de la corriente ... 14
Figura 2. Módulo o Subprograma ... 18
Figura 3. Arquitectura Android ... 21
Figura 4. Ciclo de vida de una actividad en Android ... 22
Figura 5. Esquema del módulo Master que contiene el sistema de control de la bicicleta eléctrica del CIL ... 23
Figura 6. Sistema de comunicación ... 28
Figura 7. Arquitectura de Hardware ... 29
Figura 8. Ícono aplicación E-libatt ... 29
Figura 9. Interfaz de usuario ... 30
Figura 10. Interfaz de prueba ... 30
Figura 11. Algoritmo controlador ... 31
Figura 12. Paquetes utilizados en algoritmo de estimación y pronóstico del Estado de Carga en Java ... 33
Figura 13. Diagrama UML de clases contenidas en el paquete '*soc*'. En el sector intermedio se detallan los atributos de cada clase, mientras que en el sector inferior se indican los métodos que poseen. ... 34
Figura 14. Relación entre principales métodos ... 35
Figura 15. Algoritmo Estimación y Predicción del Estado de Carga ... 36
Figura 16. Diagrama de Flujos – Estimación ... 37
Figura 17. Diagrama de Flujos - Pronóstico ... 38
Figura 18. Arquitectura del banco de baterías utilizado en la E-bike ... 42
Figura 19. Conjuntos de datos de corriente y voltaje correspondientes al recorrido de la Ruta 5 de la E-bike ... 43
Figura 20. Predicción del voltaje en base a la PDF a posteriori de los estados. a) En azul se muestra el voltaje real, medido por los sensores de la E-bike, mientras que en rojo, se observa el voltaje predicho por los estados. b) Muestra un acercamiento del voltaje predicho, en rojo, y del voltaje observado, en azul. ... 43
Figura 21. Predicción del Voltaje a largo plazo. En azul se muestra el voltaje observado por los sensores, mientras que en verde se muestra la predicción del voltaje a partir del inicio de la predicción a largo plazo ... 44
Figura 22. Resultados del Estado de Carga. En azul el SOC “real” el cual se calcula utilizando los datos de voltaje y corriente observados por los sensores, en rojo el SOC estimado a partir de los estados y en verde el SOC predicho a partir del inicio de la predicción a largo plazo ... 44
Figura 23. Cadena de Markov. a) Determinación de los estados de corriente alta y baja. b) Discretización de la corriente en los estados determinados. ... 45

Figura 24. Ponderación EWMA de los estados de corriente alta y baja del perfil de uso de la batería de Ion-Litio. En color azul los nuevos estados y en color verde los estados antes de la ponderación...46
Figura 25. Realización del perfil de uso futuro de la corriente con transiciones entre sus dos estados. 46
Figura 26. Rendimiento del algoritmo de estimación y pronóstico del Estado de Carga en ambiente MATLAB®...47
Figura 27. Performance algoritmo off-line...49
Figura 28. Elibatt - Funcionamiento de la aplicación en su vista usuario donde se indica el Estado de Carga y el pronóstico del tiempo de descarga de la batería...............................50
Figura 29. Elibatt - Funcionamiento de la aplicación en su vista de prueba o debug donde se indican los parámetros de interés del algoritmo de estimación y pronóstico del Estado de Carga...50

Índice de Códigos

Código 1. Recibir archivo en el Servidor mediante PHP 32
Código 2. Función que sube el archivo de datos a servidor 32
Código 3. Declaración de variables de visualización 40
Código 4. Función que imprime datos en pantalla 40
Código 5. Click en botón Subir para almacenar datos dentro de un servidor 40
Código 6. Almacenar datos dentro de archivos de texto 41

Índice de Tablas

Tabla 1. Parámetros del modelo para la batería de Ion-Litio 12
Tabla 2. Comparativa tiempos de procesamiento.. 48

Lista de Acrónimos

AEP	Algoritmo de Estimación y Pronóstico
API	Interfaz de programación de aplicaciones (*Application Programming Interface*)
CIL	Centro de Innovación de Litio
CM	Cadena de Markov (*Markov Chain*)
EKF	Filtro Extendido de Kalman (*Extended Kalman Filter*)
EOD	Fin de la Descarga (*End of Discharge*)
ESD	Dispositivo Almacenador de Energía (*Energy Storage Device*)
EWMA	Medias Móviles con Ponderación Exponencial (*Exponentially Weighted Moving Average*)
FP	Filtro de Partículas (*Particle Filter*)
HTTP	Protocolo de transferencia de hipertexto (*Hypertext Transfer Protocol*)
PDF	Densidad de probabilidad (*Probability Density Function*)
PHP	*PHP Hypertext Pre-processor*
RMS	Error Cuadrático Medio (*Root Mean Square*)
SOC	Estado de Carga (*State Of Charge*)
SOH	Estado de Saludo (*State Of Health*)
SMC	Métodos Secuenciales de Monte Carlo (*Sequential Monte Carlo Method*)
UKF	Filtro Unscented de Kalman (*Unscented Kalman Filter*)
URL	Localizador de recursos uniforme (*Uniform Resource Locator*)

Capítulo I

1.1 Introducción

Hoy en día, el transporte constituye el principal sector consumidor de energía del mundo (25-35%) y además depende casi un cien por ciento del petróleo. La conversión a electricidad es un triángulo virtuoso de sustentabilidad energética, independencia del petróleo y reducción de CO_2. De esta forma, los dispositivos almacenadores de energía (ESDs, del inglés Energy Storage Devices), y en especial las baterías de Ion-Litio, juegan un rol significativo en el desarrollo de nuevos y más eficientes sistemas de comunicación y transporte. La problemática del consumo energético afecta a muchas personas y empresas que buscan de alguna manera reducir los costos de desplazarse de un lugar a otro.

En este sentido, el desarrollo de una bicicleta o vehículo eléctrico requiere, entre varias cosas, del uso de métodos de estimación y pronóstico para el análisis del Estado de Carga de la batería (SOC, por sus siglas en inglés) que permitan al usuario conocer la carga estimada actual del ESD y además, la predicción del tiempo de descarga (EOD, por sus siglas en inglés) de la batería. Los ESD's representan un medio para el manejo de energía y a la vez son una importante restricción práctica en términos del nivel de autonomía máxima que cualquiera de estos sistemas puede alcanzar. El Estado de Carga como información entregada es vital para el usuario ya que permite conocer con un cierto grado de precisión cuándo requiere recargar el dispositivo, así como también el tiempo restante de uso y otro tipo de información de interés.

Un aspecto útil que va de la mano al análisis del SOC es el Estado de Salud (SOH, por sus siglas en inglés) que mediante el uso de métodos de estimación y pronóstico permite conocer una predicción acerca de la vida útil remanente del ESD. Esto es, conocer con cierta precisión cuando es necesario reemplazar el dispositivo. Este tema es muy importante en el análisis completo de un ESD, pero queda fuera del alcance de este trabajo.

Por otro lado, con el acelerado proceso de crecimiento que vive la tecnología, hoy en día se cuenta con una herramienta poderosa como son los Smartphone que permiten introducir un entorno gráfico amigable e intuitivo para presentar toda la información de la batería y un nivel de procesamiento de información que permite manejar los datos utilizados en los métodos de estimación y predicción.

El objetivo de implementar un sistema inspirado en la teoría de estimación y pronóstico de un ESD de la bicicleta eléctrica a una aplicación Android de uso masivo es la idea central de este Trabajo de Título. En el Capítulo I se entregan los objetivos generales y específicos que se desean cumplir a lo largo de este trabajo. Pensando en el lector, el Capítulo II se encarga de entregar los conceptos teóricos que enmarcan la temática del proyecto y permite contextualizar

las ideas que se están desarrollando. El Capítulo III se encarga de mostrar los requerimientos del sistema y la determinación de los componentes principales a utilizar, además se muestra la implementación de Software y Hardware del AEP dentro de una aplicación Android. Una vez implementado el código, el Capítulo IV se encarga de mostrar los resultados obtenidos en la ejecución del AEP, se muestran resultados de estimación del SOC y pronóstico del EOD. Además, se entregan gráficos sobre el nivel de procesamiento de éste en términos de recursos utilizados tales como memoria, nivel de procesamiento y número de clases utilizados.

1.2 Objetivo General del Trabajo de Título

Es en este contexto que el presente Trabajo de Título tiene como objetivo general implementar un método sub-óptimo de estimación y predicción del Estado de Carga de dispositivos de almacenamiento de energía en un ambiente Android. Esto permitirá asegurar información acerca de la cantidad máxima de energía almacenada en el acumulador y de la cantidad de energía se tiene en un momento en particular, conocimiento vital en el manejo de ESD's. El objetivo es poder integrar un módulo de estimación y pronóstico que funcione en línea enfocado particularmente en la caracterización estadística del perfil de uso futuro del acumulador.

1.3 Objetivos Específicos del Trabajo de Título

En este sentido, se consideran como objetivos específicos del Trabajo de Título, los siguientes aspectos: (i) Implementar en tiempo real un algoritmo sub-óptimo de estimación y predicción del SOC de baterías de Ion-Litio en una aplicación Android; (ii) Cuantificar tiempo de procesamiento, consumo de recursos y energía del algoritmo en la plataforma; (iii) Establecer una frecuencia de estimación y predicción del SOC; (iv) Comparar desempeño con respecto a versión en MATLAB®.

.

Capítulo II

Descripción del Problema

El presente capítulo tiene por objetivo ubicar al lector en el entorno en el cual se desarrolla este Trabajo de Título, entregando los antecedentes previos y necesarios para su contextualización.

En primer lugar, en la Sección 2.1, se indica el concepto de estado-de-carga para dispositivos almacenadores de energía. En la Sección 2.2 se indican las características generales de la Estimación y Predicción del Estado-de-Carga. En la Sección 2.3 se presenta el algoritmo de estimación y pronóstico del SOC. En la Sección 2.4 se describen las características generales del lenguaje de programación Java y en el 2.5 se muestra el ambiente de desarrollo Android. Finalmente, la Sección 2.6 hace referencia a la contextualización del proyecto.

Es importante mencionar, que en la Sección 2.2 muestra una visión global del respaldo teórico del algoritmo invitando al lector, en el caso de querer obtener mayor cantidad de detalles, revisar el Trabajo de Título "Estimación y Pronóstico en línea del Estado de Carga de Baterías Ion-Litio basado en Filtro de Partículas e Implementado en Bicicletas Eléctricas" de la Universidad de Chile cuyo autor es el Sr. Cristóbal Inostroza [1].

2.1 Dispositivos almacenadores de energía y Estado-de-Carga

El almacenamiento de energía utiliza dispositivos donde la energía es conservada en cierta cantidad, para luego ser liberada y utilizada para diversos fines como, por ejemplo, energizar dispositivos móviles o una bicicleta eléctrica.

En particular, este estudio se enfocará en las baterías de iones de litio, también conocidas como baterías de Ion-Litio. Este es un dispositivo miembro de la familia de baterías recargables diseñado para almacenar energía eléctrica, el cual emplea como electrolito una sal de litio que procura los iones necesarios para la reacción electroquímica reversible que ocurre entre el cátodo y el ánodo.

Dentro de las propiedades de las baterías de Ion-Litio se encuentran la ligereza de sus componentes, su elevada capacidad energética y su resistencia a la descarga, es decir, tiene una lenta pérdida de carga cuando no está en uso. Junto con estas características se puede agregar el poco efecto memoria que sufren o su capacidad para funcionar con un elevado

número de ciclos de regeneración [4]. Esto ha permitido el diseño de acumuladores livianos, de pequeño tamaño y variadas formas, con un alto rendimiento, especialmente adaptados a las aplicaciones de la industria electrónica de gran consumo.

En el contexto de la escasez de combustibles derivados del petróleo, la industria del automóvil está comenzando, hace algún tiempo, a desarrollar y comercializar vehículos con motores eléctricos basados en la tecnología de las baterías de iones de litio, con los que se pueda disminuir la dependencia energética de estas fuentes, a la vez que se mantiene baja la emisión de gases contaminantes. En este sentido, se hace importante poder conocer el Estado de Carga de las baterías, ya que éste puede ser comparado con el depósito de combustible de un vehículo.

El Estado de Carga, representado en porcentaje, indica la cantidad de energía disponible en una batería. [5]. El SOC puede servir como indicador para recordar a los usuarios recargar sus baterías y para evitar que sea sobrecargada. Sin embargo, no hay manera fácil de visualizar con precisión el SOC de la batería y adquirir información sobre el estado actual de la batería [6]. De este punto de vista, la estimación y pronóstico del Estado de Carga permite entregar información al usuario sobre la cantidad de energía actual que posee y el tiempo de descarga de la batería.

2.2 Estimación y Predicción del Estado de Carga con Filtro de Partículas

Contar con información acerca del Estado de Carga y tiempo de descarga de la batería permite realizar estrategias de control en los vehículos eléctricos y/o híbridos, ya que permite tomar decisiones con respecto al modo de operación del vehículo de acuerdo a la energía disponible. Estimar el SOC presenta dificultades ya que éste no se puede medir en forma directa, y por lo tanto, se requiere estimar su valor utilizando métodos indirectos [2]. Hoy en día, la tendencia del estudio del SOC considera parámetros eléctricos y ambientales de la batería como corriente, voltaje en bornes y temperatura que permiten realizar estudios en línea [7].

Actualmente, se consideran varios enfoques para la estimación del Estado de Carga con métodos basados en modelos empíricos y/o en la información de los parámetros de la batería. Los métodos más utilizados en este enfoque son lógica difusa, redes neuronales, Filtro Extendido de Kalman (EKF) y Filtro de Partículas. La lógica difusa ha sido utilizada en algunas investigaciones para modelar la relación entre el Estado de Carga del ESD y sus parámetros medidos a través de la Impedancia Espectroscópica, en trabajos que cuentan con información escasa y contradictoria y para estimar directamente el SOC y el SOH de acumuladores. Las redes neuronales permiten modelar empíricamente el acumulador, es

decir, requiere de información de entrenamiento para generar modelos no-lineales y ajustar sus parámetros. Una vez que el modelo se ha generado es posible realizar estimación y predicción del SOC. Sus desventajas están relacionadas con el sobre-ajuste del modelo a la información de entrenamiento y otorgar resultados en óptimos locales. En relación a esto, una buena solución ha sido modelar los acumuladores mediante circuitos eléctricos equivalentes, con los que se busca representar los principios fisicoquímicos de los acumuladores. Sin embargo, estos modelos son no-lineales con lo que los métodos más utilizados en este enfoque se basan en una técnica de estimación sub-optima Bayesiana llamada Filtro Extendido de Kalman (EKF), el cual se basa en el modelo no-lineal de la batería e intenta aproximar la matriz de error de covarianza asociada a la estimación de estado, usando una versión linealizada de los sistemas dinámicos que representan la descarga del acumulador. Este procedimiento permite que el EKF adapte los valores de los parámetros y estados del modelo durante toda la etapa de estimación incorporando así toda la información disponible hasta el instante de predicción. Sin embargo, este método tiene su desventaja al momento de realizar la predicción o estimación a n-pasos, ya que los errores de aproximación debidos a la linealización son demasiado importantes para no ser considerados [2].

En este sentido, los métodos secuenciales de Montecarlo (SMC, por sus siglas en inglés) o Filtros de Partículas (FP) han demostrado tener un mejor desempeño al trabajar en procesos de estimación y predicción con modelos no lineales. Su principal característica es que es capaz de trabajar tanto en estimación como en predicción basándose en modelos no-lineales, y al igual que el EKF, es capaz de adaptar los valores de los parámetros y estados del modelo durante la etapa de estimación, lo que permite incorporar al modelo toda la información disponible hasta el instante inicial de predicción. Además, puede realizar una estimación del ruido del sistema y caracterizar una solución sub-optima del proceso de descarga en función de una densidad de probabilidad (PDF, por sus siglas en ingles). Las aplicaciones de FP son múltiples y están presentes en diversas disciplinas como en seguimiento de trayectorias, detección de fallas, pronóstico de eventos catastróficos, econometría, y en particular, en la determinación del estado-de-carga de las baterías [2].

Hoy en día se cuenta con un método capaz de estimar y, según el perfil de uso futuro, predecir en línea el Estado de Carga de un dispositivo almacenador de energía, y que entrega resultados con apropiada exactitud y precisión basados en un modelo de baterías de Ion-Litio que se analizará en la siguiente sección.

2.2.1 Inferencia Bayesiana

Los métodos de Montecarlo se basan en la inferencia bayesiana, es decir, estimar cantidades desconocidas a través de un conjunto de observaciones. En el caso particular de las baterías de Ion-Litio, las observaciones corresponden a la corriente y voltaje durante el proceso de descarga de un dispositivo almacenador de energía, y la variable que se pretende estimar es el Estado de Carga del acumulador.

Bajo un enfoque Bayesiano, el conocimiento disponible del sistema en estudio se puede utilizar para formular distribuciones a priori de la evolución de las cantidades desconocidas (corriente y voltaje), y funciones de verosimilitud para relacionar éstas con las observaciones. En este contexto, la inferencia sobre las variables latentes del sistema (variables que no se observan directamente sino que son inferidas a partir de otras variables que se observan) se basa en la distribución a posteriori obtenida mediante el Teorema de Bayes. Para realizar estimaciones en línea es necesario actualizar esta distribución a posteriori en forma secuencial cada vez que se reciben nuevas observaciones.

Filtro de Kalman (FK) permite derivar una expresión analítica óptima para la evolución de las distribuciones a posteriori, si es que las relaciones que definen los procesos de observación y transición de estado están dadas por un modelo de espacio-estado lineal y Gaussiano [9]. Sin embargo, en general, los procesos reales son complejos y no pueden ser descritos por este tipo de relaciones.

Para resolver este problema del filtrado Bayesiano, Filtro de Partículas posee gran facilidad de implementación, y tiene un amplio campo de aplicación. Este método es capaz de aproximar una secuencia de medidas de probabilidad de dimensión creciente mediante un conjunto de muestras ponderadas del espacio de estado, las cuales evolucionan en base al conocimiento previo del sistema y a las observaciones obtenidas del proceso. FP ha mostrado ser una alternativa superior al EKF y al filtro Unscented de Kalman (UKF, del inglés Unscented Kalman Filter) para sistemas no-lineales y no-Gaussianos, ya que con suficientes muestras, aproximan la estimación óptima del problema Bayesiano de forma más exacta [8]. Una ventaja es que sus propiedades de convergencia no dependen de la naturaleza del modelo del esquema de estimación, esto permite incorporar no-linealidades y procesos de innovación de diversas distribuciones.

2.2.1.1 Filtro de Partículas

Filtro de Partículas o métodos Secuenciales de Monte Carlo (SMC) son un conjunto de algoritmos de estimación en línea que estiman la densidad a posteriori del espacio de estado mediante la implementación directa de las ecuaciones de recursión Bayesiano.

Estos métodos de filtrado no tienen ningún supuesto restrictivo sobre la dinámica del espacio de estado o de la función de densidad. Además, proporcionan una metodología de generación de muestras de la distribución requerida, sin necesidad de suposiciones sobre el modelo de espacio de estado o las distribuciones de estado. El modelo de espacio de estados puede ser no lineal y las distribuciones iniciales del estado y de ruido pueden tomar cualquier forma requerida.

Los métodos Secuenciales de Monte Carlo utilizan ecuaciones de recursión Bayesiano donde las muestras de la distribución están representadas por un conjunto de partículas, cada partícula tiene un peso asignado a ella que representa la probabilidad de que la partícula sea muestreada a partir de la función de densidad de probabilidad. La disparidad de peso conduce al colapso, el cual es un problema común que se encuentra en estos algoritmos de filtrado, no obstante, puede ser mitigado mediante la inclusión de un nuevo paso de *remuestreo* antes de que los pesos se vuelvan demasiado desiguales. En la etapa de *remuestreo*, las partículas con pesos insignificantes son reemplazadas por nuevas partículas en la proximidad de las partículas con pesos más altos. Sin embargo, estos métodos no funcionan bien cuando se aplica a los sistemas de alta dimensión, ya que cada vez que se desee estimar la secuencia de densidades $\pi_k(x_{0:k}|y_{1:k})$ se necesita de todo el conjunto de valores $y_{1:k}$, haciendo el procedimiento computacionalmente ineficiente debido a la alta dimensionalidad de la densidad objetivo.

Suponiendo que la secuencia de densidades que se desea aproximar es $\{\pi_k(x_{0:k})\}_{k\geq 1}$, donde $\forall k\ \pi_k(x_{0:k})$ está definida en un espacio medible $(\chi^{k+1}, \Sigma_{k+1})$ y puede ser evaluada punto a punto salvo una constante de normalización, la implementación del Filtro de Partículas consiste en generar una colección de $N >> 1$ muestras aleatorias ponderadas $\left\{w_k^{(i)}, w_{0:k}^{(i)}\right\}_{i=1,\ldots,N}$, $w_k^{(i)} \geq 0, \forall i, k$; que permite aproximar $\pi_k(x_{0:k})$, $\forall\ k$ por la distribución empírica [2]:

$$\pi_k^N(x_{0:k}) = \sum_{i=1}^{N} w_k^{(i)} \delta_{x_{0:k}^{(i)}}(x_{0:k}) \tag{1}$$

donde $\delta_\alpha(\cdot)$ es el delta de Dirac centrada α.

Para resolver el problema de aproximación de esperanzas en el contexto de inferencia Bayesiana, se debe considerar que la distribución objetivo $\pi_k(x_{0:k}) = p(x_{0:k}|y_{1:k})$ es la PDF a posteriori de $x_{0:k}$ y reemplazar la distribución empírica $\pi_k^N(\cdot)$ en la esperanza de predicción. Con esta sustitución, las esperanzas pueden ser aproximadas mediante [2]:

$$\int_{\chi^{k+1}} \phi_k(x_{0:k})\, \pi_k(x_{0:k}) dx_{0:k} \simeq \int_{\chi^{k+1}} \phi_k(x_{0:k})\, \pi_k^N(x_{0:k}) dx_{0:k}$$
$$= \int_{\chi^{k+1}} \phi_k(x_{0:k}) \sum_{i=1}^{N} w_k^{(i)} \delta_{x_{0:k}^{(i)}}(x_{0:k}) dx_{0:k} = \sum_{i=1}^{N} w_k^{(i)} \phi_k\left(x_{0:k}^{(i)}\right) \quad (2)$$

Con estas relaciones, el problema de inferencia Bayesiana se reduce a la selección secuencial de las muestras y sus respectivos pesos. El Remuestreo Secuencial de Importancia (SIR, del inglés Sequential Importance Resampling) es el algoritmo más básico basado en FP que resuelve este problema [2].

2.2.1.2 Muestreo y Remuestreo Secuencial

Para abordar el problema de degeneración, esto es, después de unas pocas iteraciones, todas menos una de las partículas tienen un peso insignificante, se utiliza el remuestreo secuencial (resampling, en inglés).

El algoritmo de remuestreo se lleva a cabo para eliminar las partículas con pequeños pesos, concentrando los esfuerzos computacionales en los que tienen pesos grandes. Teniendo en cuenta esto último, el algoritmo SIR para Filtro de Partículas se presenta a continuación:

1. Muestreo de importancia
   - Para $i = 1, \dots, N$, muestrear $\tilde{x}_k^{(i)} \rightarrow \pi\left(x_k \middle| \tilde{x}_{0:k-1}^{(i)}, y_{0:k}\right)$ y establecer $\tilde{x}_{0:k}^{(i)} \triangleq \left(\tilde{x}_{0:k-1}^{(i)}, \tilde{x}_k^{(i)}\right)$
   - Establecer pesos importantes

$$w\left(\tilde{x}_{0:k}^{(i)}\right) = \tilde{x}_{0:k-1}^{(i)} \cdot \frac{p\left(y_k \middle| \tilde{x}_k^{(i)}\right) p\left(\tilde{x}_k^{(i)} \middle| \tilde{x}_{0:k-1}^{(i)}\right)}{q\left(\tilde{x}_k^{(i)} \middle| \tilde{x}_{0:k-1}^{(i)}\right)} \quad (3)$$

$$w_{0:k}^{(i)} = w\left(\tilde{x}_{0:k}^{(i)}\right) \cdot \left(\sum_{k=1}^{N} w\left(\tilde{x}_{0:k}^{(i)}\right)\right)^{-1} \quad (4)$$

2. Algoritmo de remuestreo
   Si $\widehat{N}_{eff} \geq N_{thres}$, ($\widehat{N}_{eff}$ una estimación del tamaño efectivo de la muestra N_{eff} y N_{thres} un umbral fijo)
   - $\breve{x}_{0:k}^{(i)} = \tilde{x}_{0:k}^{(i)}$ para k=1,…,N. De otra manera

- Para $k = 1, \dots, N$, muestrear un índice $j(k)$ distribuido mediante una distribución discreta satisfaciendo $P(j(k) = l) = w_k^{(i)}$ para $l = 1, \dots, N$.
- Para $k = 1, \dots, N$, $\breve{x}_{0:k}^{(i)} = \tilde{x}_{0:k}^{(i)}$ y $\breve{w}_k^{(i)} = N^{-1}$

Después de este proceso de remuestreo, la nueva población de partículas $\left\{\breve{x}_{0:k}^{(i)}\right\}_{i=1,..,N}$ es una muestra i.d.d. de la siguiente distribución empírica

$$\breve{\pi}_k^N(x_{0:k}) = \frac{1}{N}\sum_{i=1}^{N} N^{(i)}\delta\left(x_{0:k} - \tilde{x}_{0:k}^{(i)}\right) = \frac{1}{N}\sum_{i=1}^{N}\delta\left(x_{0:k} - \breve{x}_{0:k}^{(i)}\right) \tag{5}$$

Y por lo tanto, los pesos son reestablecidos a $\breve{w}_k^{(i)} = N^{-1}$.

2.2.2 Modelación de Eventos Aleatorios mediante el uso de Cadenas de Markov

Para efectos de este trabajo las Cadenas de Markov en tiempo discreto son ocupadas para la caracterización de los diversos perfiles de uso que se pueden presentar durante el proceso de descarga del banco de baterías.

Las Cadenas de Markov, son modelos matemáticos simples que permiten representar fenómenos aleatorios que evolucionan en el tiempo. Poseen una estructura básica, pero completa que hace posible describir el comportamiento de procesos en muchas aplicaciones.

La principal característica de las Cadenas de Markov es que no poseen memoria de donde se ha estado en el pasado. Esto significa que solo el estado actual del proceso estocástico $(X_k)_{k\geq 0}$ determina el estado siguiente (primer orden):

$$\Pr\{X_{k+1} = s_{k+1} | X_0 = s_0, X_1 = s_1, \dots, X_k = s_k\} = \Pr\{X_{k+1} = s_{k+1} | X_k = s_k\} \tag{6}$$

con $k \in \mathbb{Z}^+ = \{0,1,2, \dots\}$ y $s_k \in S$ estados de un conjunto finito y numerable.

En este sentido, las probabilidades de transición $P_{(i,j)}$ entre los estados de una Cadena de Markov pueden ser representadas por una matriz estocástica $P = \left(P_{(i,j)}: i, j \,\epsilon\, S\right)$ donde la suma de cada fila $\left(P_{(i,j)}: j \;\in S\right)$ resulta 1.

Por otro lado, la propiedad de homogeneidad se centra en el caso en que la probabilidad de evolucionar del estado i en el instante k al estado j en el instante $k+m$ es independiente de k. Esta condición equivale a imponer que la matriz de probabilidad de estado es constante e independiente de k.

$$P_{k+1}^{k} = P \quad \forall k \tag{7}$$

Con esto entonces la evolución del sistema queda definida solamente por $\Pr\{X_0 = s_0\}$, probabilidad inicial del sistema, y por la matriz de probabilidades de transición P.

La siguiente sección introducirá la implementación de un algoritmo de estimación y pronóstico del Estado de Carga mediante el uso de Cadenas de Markov para la caracterización de los perfiles de uso de equipos energizados con baterías.

2.3 Algoritmo de Estimación y Predicción del Estado-de-Carga en Baterías de Ion-Litio

El algoritmo de estimación y predicción del Estado de Carga ha sido desarrollado e implementado en ambiente MATLAB® por el Sr. Cristóbal Inostroza en su Trabajo de Título llamado "Estimación y Pronóstico en línea del Estado de Carga de Baterías Ion-Litio basado en Filtro de Partículas e Implementado en Bicicletas Eléctricas" de la Universidad de Chile [1].

2.3.1 MATLAB®

MATLAB® (abreviatura de MATrix LABoratory, "laboratorio de matrices") es una herramienta de software matemático que ofrece un entorno de desarrollo integrado (IDE) con un lenguaje de programación propio (lenguaje M) y está disponible para las plataformas Unix, Windows, Mac OS X y GNU/Linux.

Entre sus prestaciones básicas se hallan: la manipulación de matrices, la representación de datos y funciones, la implementación de algoritmos, la creación de interfaces de usuario (GUI) y la comunicación con programas en otros lenguajes y con otros dispositivos hardware. Precisamente estas funcionalidades son las que permiten obtener ventajas en la utilización de este lenguaje, ya que, facilitan la programación al tener instrucciones ya preparadas para trabajar con números y matrices. Además, sus herramientas gráficas permiten visualizar tendencias y realizar análisis para verificar el funcionamiento del algoritmo.

MATLAB® posee herramientas de compilación a lenguajes Java y .Net mediante MATLAB Builder ™ el cual permite crear clases que pueden integrarse en los programas

creados en los lenguajes mencionados utilizando el MATLAB Compiler Runtime (MCR). Esta última característica facilita el proceso de compilación y creación del algoritmo dentro del ambiente Java, sin embargo, el objetivo de este Trabajo de Título no es sólo la implementación del algoritmo, sino también, la optimización de éste. Por ello, se programa el algoritmo en lenguaje nativo Java, ya que, los programas compilados en código nativo son más rápidos en tiempos de ejecución que los programas traducidos, debido a la sobrecarga del proceso de traducción.

2.3.2. Algoritmo de Estimación y Predicción del Estado de Carga

El desarrollo e implementación del módulo de pronóstico del tiempo de descarga del banco de baterías, requiere una adecuada caracterización del modelo de dispositivo almacenador de energía y además, información sobre las futuras condiciones de operación.

Las próximas dos secciones se enfocarán en mostrar el modelo a utilizar, que caracteriza el impacto de diferentes corrientes de descarga sobre el voltaje de la batería.

1.2.2.1 Estimación de Estados

La corriente y el voltaje del banco de baterías son utilizados como entradas al sistema. Considerando esta información disponible se utiliza el siguiente modelo de descarga, con el que se explica la caída del voltaje en los bornes.

Modelo de transición de estados:

$$\begin{aligned} x_1(k+1) &= x_1(k) + \omega_1(k) \\ x_2(k+1) &= x_2(k) - v(k) \times i(k) \times \Delta t \times E_{crit}^{-1} + \omega_2(k) \end{aligned} \tag{8}$$

Ecuación de observación:

$$\begin{aligned} v(k) = v_L + (v_0 - v_L)e^{\gamma(x_2(k)-1)} + \alpha v_L(x_2(k) - 1) \\ + (1-\alpha)v_L\left(e^{-\beta} - e^{-\beta\sqrt{x_2(k)}}\right) - i(k)x_1(k) + \eta(k) \end{aligned} \tag{9}$$

Donde la corriente de descarga $i(k)\,[A]$ y el tiempo de muestreo Δt son variables de entrada, mientras que el voltaje de la batería $v(k)$ es la salida y de observación del sistema. Las cantidades v_0, v_L, α, β y γ son parámetros del modelo que deben ser estimados en forma *off-line* mediante una prueba de descarga controlada cuyo objetivo es disminuir el error cuadrático medio entre la curva de voltaje obtenida al descargar la batería

(completamente cargada) a su corriente nominal y la ecuación de observación presentada en (9). La Tabla 1 presenta los parámetros utilizados para la batería.

Tabla 1. Parámetros del modelo para la batería de Ion-Litio

α	β	γ	V_0	V_L	E_{crit}
0.14	9.29	6.69	41.49	39.2	1065600

Los estados son definidos como $x_1(k)$ (asociado a la resistencia interna del ESD) y $x_2(k)$ (SOC, energía remanente de la batería normalizada por el parámetro E_{crit}), E_{crit} es la energía total esperada entregada por el ESD en condiciones nominales o al no presentar signos de degradación.

Se observa que el modelo presentado es no-lineal, por tanto el método que más se ajusta para trabajar con estimación y predicción es el método de Filtro de Partículas. Por último, es en esta etapa de estimación donde se realiza el proceso de remuestreo para resolver el problema de degeneración de partículas.

1.2.2.2 Predicción de Estados

Una vez realizada la estimación de estados, lo que sigue es utilizar esta información para efectos de pronóstico del proceso de descarga del banco de baterías de Ion-Litio. Para lograr esto, se utiliza un marco de pronóstico de eventos basado en Filtro de Partículas, en el cuál se considera la distribución de probabilidad del estado, dados los datos medidos, como condición inicial y una caracterización estadística de perfiles de uso del banco de baterías, con el objetivo de estimar la distribución de probabilidad del tiempo de operación remanente de la batería.

Una de las principales ventajas de los métodos de predicción en base a Filtro de Partículas, es que la etapa de predicción puede realizarse en tiempo real; mediante la caracterización de una serie de posibles trayectorias (dependientes de realizaciones de perfiles de operación futura) y un conjunto de pesos $w_k^{(i)}$ que representan la probabilidad de seguir efectivamente dicha trayectoria. Más aún, una vez definida una "zona de peligro" asociada a una condición de riesgo en la batería, es factible calcular la distribución de probabilidad del tiempo remanente hasta la descarga usando la ley de probabilidades totales, lo que a su vez no sólo permite entregar información respecto al valor esperado de la predicción, sino que además entrega estadísticos que describen de manera apropiada la distribución de probabilidad del riesgo asociado al momento en que el evento de interés ocurre (descarga de la batería). Ejemplos de estos estadísticos se tienen: intervalos de confianza, varianza del EOD, Just-in-Time Point (JITP γ%), entre otros [2].

Con respecto a la caracterización del perfil de uso futuro de la corriente, existen dos enfoques abordados en el algoritmo: Corriente Promedio; Cadenas de Markov.
El primer enfoque establece un único valor calculado como el promedio de todas las corrientes.

$$I_{mean} = \frac{1}{k}\sum_{i=1}^{k} I_i \tag{10}$$

Por otro lado, un segundo enfoque para la caracterización de la corriente futura es el de Cadenas de Markov, tal como se vio en la Sección 1. En este caso el algoritmo de estimación y pronóstico del Estado de Carga utiliza una cadena de primer orden de dos estados, donde los estados y probabilidades de transición están asociados directamente al uso que se le ha dado al banco de baterías hasta el instante inicial de predicción. Dado esto, se utiliza el siguiente procedimiento para la determinación de los parámetros de la Cadena de Markov:

1. Segmentar los datos de corriente en intervalos iguales
2. Determinar los valores de los estados de cada intervalo mediante el uso del algoritmo de k-Means
3. Discretizar los datos de corriente de cada intervalo en sus dos estados
4. Calcular las probabilidades de transición entre los estados para cada intervalo, según:

$$p_{ij}^{(m)} = \frac{\sum_{k=2}^{N^{(m)}} \left[i(k) = \frac{j}{i(k-1)} = i \right]}{N^{(m)} - 1} \quad \forall i, j \, ; i, j \, \in \left(i_{baja}^{(m)}, i_{alta}^{(m)} \right) \tag{11}$$

   donde $p_{ij}^{(m)}$ es la probabilidad de pasar del estado i al estado j en el intervalo m; $N^{(m)}$ es el número de datos del intervalo m; $i_{baja}^{(m)}, i_{alta}^{(m)}$ corresponden al estado bajo y alto de la corriente en el intervalo m.
5. Finalmente, con el objetivo de incorporar la información de los intervalos previos, y otorgarle mayor importancia a los datos correspondientes a los últimos intervalos y así, obtener los valores definitivos tanto de los estados como de las probabilidades de transición de la CM con la que se caracterizara el perfil de uso futuro, se realiza una ponderación exponencial (EWMA) de los valores de estados y probabilidades de transición en cada intervalo, obteniendo así nuevos valores de los estados y probabilidades de transición para cada intervalo, los cuales tienen incorporada la información del perfil de uso de los intervalos anteriores según:

$$\bar{\iota}_{baja}^{(m)} = (1-\alpha) \times i_{baja}^{(m)} + \alpha \times \bar{\iota}_{baja}^{(m-1)} \;\; \forall m \tag{12}$$

$$\bar{\iota}_{alta}^{(m)} = (1-\alpha) \times i_{alta}^{(m)} + \alpha \times \bar{\iota}_{alta}^{(m-1)} \;\; \forall m \tag{13}$$

$$\bar{p}_{ij}^{(m)} = (1-\alpha) \times p_{ij}^{(m)} + \alpha \times \bar{p}_{ij}^{(m-1)} \;\; \forall i,j,m \tag{14}$$

Donde $\bar{\iota}_{baja}^{(m)}$, $\bar{\iota}_{alta}^{(m)}$ corresponden respectivamente a los valores ponderados del estado bajo y alto de la corriente en el intervalo m; $\bar{p}_{ij}^{(m)}$ es la probabilidad ponderada de pasar del estado i al estado j en el intervalo m; y α corresponde al factor de olvido, el que es considerado como α = 0.35.

Una vez determinados todos los parámetros de la Cadena de Markov, la caracterización estadística del perfil de uso futuro queda ilustrada en:

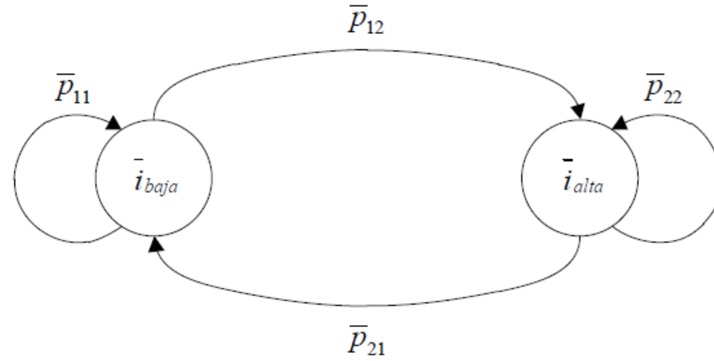

Figura 1. Cadena de Markov Primer Orden 2 estados utilizada para la predicción de la corriente

Es así como el modelo no-lineal propuesto para la descarga de los ESD permite la implementación de Filtro de Partículas dentro de módulos de pronósticos en tiempo real y a la vez, permite la caracterización estadística del perfil de uso futuro del acumulador. En lo que respecta al algoritmo *off-line*, se utiliza una iteración del Filtro de Partículas, veinte realizaciones de Cadenas de Markov y se utilizan treinta partículas para lograr un nivel de desempeño adecuado en la aplicación de pronóstico del tiempo de descarga de la batería.

El número de realizaciones de CM e incluso el número de partículas son valores que afectan directamente el desempeño computacional del algoritmo. Este es un tema importante, recordando que el objetivo de este trabajo es implementar el algoritmo en un ambiente Android. Se sabe que los dispositivos móviles poseen recursos limitados, por lo que la optimización de procesamiento y memoria al momento de programar en Java es de alta importancia. La siguiente Sección 2.4 introduce al lector en este lenguaje de programación que es el que permite programar una aplicación en Android.

2.4 Lenguaje de programación Java

Java es un lenguaje de programación orientado a objetos, desarrollado por Sun Microsystems en la década de 1990. Moldeado en base a C++, el lenguaje Java se diseñó para ser pequeño, sencillo y portátil a través de plataformas y sistemas operativos.

Existen varias versiones de Java como Java Standard Edition (SE 7), Java Enterprise Edition (Java EE) y Java Micro Edition (ME). Java EE está orientada hacia el desarrollo de aplicaciones de red distribuidas, de gran escala, y aplicaciones basadas en Web. Mientras que Java ME está orientada hacia el desarrollo de aplicaciones para pequeños dispositivos con memoria restringida, como los teléfonos inteligentes BlackBerry. El sistema operativo Android de Google — que se utiliza en muchos teléfonos inteligentes, tablet's, lectores electrónicos y otros dispositivos— utiliza una versión personalizada de Java que no se basa en Java ME.

Java posee etapas para el desarrollo de un código que pueda ser ejecutado dentro de un dispositivo móvil. Este código debe seguir cierta estructura básica e integrar fundamentos básicos de la programación orientada a objetos, tales como, modularidad, abstracción y encapsulamiento. Las siguientes secciones muestran las bases sobre las cuales el código fue diseñado.

2.4.1 Etapas de desarrollo de un programa en Java

Para desarrollar un programa en Java, es necesario pasar por las siguientes etapas: edición, compilación y ejecución.

De esta forma en la etapa de edición, se escribe las instrucciones del programa usando el lenguaje Java y luego se guarda el archivo bajo la extensión *.java*. Por ejemplo, un archivo puede quedar con el nombre de "HolaMundo.java". A este programa escrito en Java se le denomina código fuente.

En la etapa de compilación, se traduce el código fuente usando un compilador Java, que es un programa llamado *javac.exe*, con lo que se obtiene un nuevo código conocido como código de bytes, que posee el mismo nombre que el archivo de código fuente, pero con la extensión *.class*. Asimismo, el código de bytes quedará almacenado en un archivo llamado "HolaMundo.class".

Por último, en la etapa de ejecución, el código de bytes es ejecutado por la Máquina Virtual de Java (JVM, por sus siglas en inglés), donde el código de bytes es el lenguaje de la JVM. Existe una JVM para la mayor parte de los sistemas operativos; pero, todas las JVM pueden

ejecutar el mismo código de bytes. Así, el código de bytes es independiente de la plataforma. Esto hace que los programas Java puedan ser ejecutados en cualquier máquina que disponga de una JVM.

2.4.2 Estructura básica de un programa en Java

La estructura básica de un programa en Java consiste en una declaración de clase, por ejemplo, "public class HolaMundo". La palabra clave *class* introduce una declaración de clase, la cual va seguida del nombre de la clase. Cada clase comienza con la palabra clave *public*. Al guardar el código, el nombre del archivo debe coincidir con el nombre de la clase, por ejemplo "HolaMundo.java".

Las declaraciones de clases comienzan con una *llave izquierda* " { ". Su correspondiente *llave derecha "* } ", debe terminar con la declaración de una clase. A las llaves izquierda y derecha se les conoce como *bloque*. Un bloque puede contener o anidar otros bloques.

Todo programa en Java debe de contener un método principal al que se le denomina *main*. Se dice entonces que una aplicación en Java es un programa que contiene el método main.

El método main es el punto de inicio de toda aplicación en Java. Los paréntesis después del identificador main indican que este es un *método*. La palabra clave *void* indica que este método realizara una tarea, pero no devolverá ningún tipo de información cuando complete su tarea. Las palabras *String[] args* entre paréntesis son una parte requerida de la declaración del método main.

Ya que se tiene la estructura básica para crear aplicaciones en Java lo único que falta es agregar instrucciones para que el programa realice una o más acciones. Todas las instrucciones en Java termina con un *punto y coma* (;).

Los *comentarios* ayudan a documentar un programa y mejorar su legibilidad. El compilador de Java ignora estos comentarios. Un comentario que comienza con // se llama *comentario de fin de línea* (o *de una sóla línea*). Los *comentarios tradicionales* (o de múltiples líneas) se distribuyen en varias líneas y comienza con el delimitador /* y termina con */.

La descripción de los componentes de una clase en Java, y su correspondiente uso en la programación son sólo una introducción al lector. Las características fundamentales que identifican a cualquier lenguaje orientado a objetos y la forma como se implementan en Java se muestran en la Sección **¡Error! No se encuentra el origen de la referencia.**. unque la descripción en detalle de estos conceptos es demasiado amplia, no se pretende

profundizar demasiado, sino únicamente dar una aproximación de los mismos a nuestro lenguaje de interés.

2.4.3 Elementos básicos de la orientación a objetos

A pesar de que no hay convergencia sobre los cuales deben ser los elementos fundamentales que caractericen la programación orientada a objetos, en la mayoría de teorías sobre el tema se logran identificar los siguientes:

- Abstracción
- Encapsulamiento
- Modularidad
- Herencia
- Polimorfismo

En el campo de la programación se entiende por abstracción como una visión externa de un elemento, como una clase, en el cual no interesa aquello que va por dentro. Esta abstracción determina el comportamiento y funcionalidad de la clase. Por ejemplo, la etapa de pronóstico es una clase que realiza la predicción del Estado de Carga de la batería. El proceso de realizar la predicción involucra una serie de complejas acciones que incluyen otros algoritmos y elementos matemáticos. Pero lo interesante es que todo eso queda aislado del interés del programador, y sólo se pone a disposición ciertos métodos y propiedades que permiten hacer uso de esta etapa del algoritmo.

El encapsulamiento hace referencia a la protección que debe tener una clase con los elementos que la conforman, especialmente de aquellos que contienen los datos, por ejemplo, del SOC estimado a través del tiempo. En general, una clase protege a sus elementos del mundo exterior mediante el uso de una interfaz y una implementación. La interfaz permite acceder a los elementos desde el mundo exterior, mientras que la implementación permite establecer el valor o comportamiento en cada uno de los objetos definidos en aquella clase.

La modularidad consiste en dividir un programa en módulos o subprogramas con el fin de hacerlo más legible y manejable. Al aplicar la programación modular, un problema complejo debe ser divido en varios sub-problemas más simples. Esto debe hacerse hasta obtener sub-problemas lo suficientemente simples como para poder ser resueltos fácilmente con algún lenguaje de programación. Ésta técnica se llama refinamiento sucesivo, divide y vencerás o análisis descendente.

Un módulo es cada una de las partes de un programa que resuelve uno de los sub-problemas en que se divide el problema complejo original. La Figura 2 muestra la idea de módulo o sub-problema. Cada uno de estos módulos tiene una tarea bien definida y algunos necesitan de otros para poder operar. En caso de que un módulo necesite de otro, puede comunicarse con éste mediante una interfaz de comunicación que también debe estar bien definida.

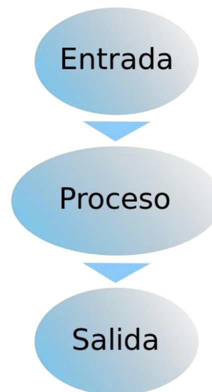

Figura 2. Módulo o Subprograma

Si bien un módulo puede entenderse como *una parte* de un programa en cualquiera de sus formas y variados contextos, en la práctica se los suele tomar como sinónimos de procedimientos y funciones. Pero no necesaria, ni estrictamente, un módulo es una función o un procedimiento ya que el mismo puede contener muchos de ellos. No debe confundirse el término "modulo" (en el sentido de programación modular) con términos como "función" o "procedimiento", propios del lenguaje que lo soporte.

Los últimos dos fundamentos son la herencia y el polimorfismo. El primero permite crear clases que se derivan de otras clases y este es uno de los aspectos vitales en la reutilización de componentes. El segundo es la posibilidad de que una entidad tome muchas formas, es decir, permite hacer referencia a objetos de diferentes clases por medio de una misma operación, la cual se ejecuta y aplica de acuerdo al objeto que la invoca. No se profundiza con mayor detalle en la herencia ni en el polimorfismo, ya que, los tres primeros son los más utilizados a lo largo del trabajo.

2.4.4 Librerías

Es posible crear cada clase y método que se necesite para formar un programa Java. Sin embargo, es bastante útil aprovechar las amplias colecciones de clases existentes en las librerías o también conocidas como bibliotecas de clases de Java, que también se conocen como API (Interfaces de programación de aplicaciones) de Java.

El utilizar estas librerías en vez de programar desde cero puede mejorar el rendimiento de los programas ya que estas clases y métodos están escritos de manera cuidadosa para funcionar con eficacia, lo que permite reducir el tiempo de programación.

Una vez escrito el código en Java se procede a someter al programa a un análisis de rendimiento y robustez. La siguiente sección muestra algunas herramientas utilizadas para realizar este análisis.

2.4.5 Análisis de rendimiento

El rendimiento del equipo (performance) se caracteriza por la cantidad de trabajo útil realizado por un programa en comparación con el tiempo y los recursos utilizados. Las componentes que afectan la performance de un programa son la CPU, dispositivos de entrada y salida, llamadas a métodos nativos y los sistemas de memoria.

Para poder analizar el rendimiento de un programa existen diversos aspectos dentro de los cuales se encuentran el tiempo de inicio de un programa, el tiempo de respuesta, la cantidad de datos (throughput) que fluye a través del sistema, requerimientos de hardware y escalabilidad. En relación al tiempo de inicio, esto se refiere al tiempo de carga que requiere la aplicación para comenzar. El tiempo de respuesta, se refiere al tiempo que requiere el sistema para reaccionar ante una solicitud del usuario. La cantidad de datos que pueden ser procesados por el programa dentro de un tiempo dado. Los requerimientos de hardware se refieren a cuanto procesador y cuanta memoria RAM se requiere para ejecutar el programa. Por último, la escalabilidad es la propiedad deseable del programa, que indica su habilidad para reaccionar y adaptarse sin perder calidad, o bien manejar el crecimiento continuo de trabajo de manera fluida, o bien para estar preparado para hacerse más grande sin perder calidad en los servicios ofrecidos.

Una buena forma simple de medir el rendimiento del programa, es la de calcular u obtener su tiempo de ejecución el cual dicho en otras palabras, es el tiempo que tarda un método u aplicación en realizar su tarea. Lo primero que hay que hacer es capturar un tiempo inicial, luego una vez terminada la acción a realizar tomar el tiempo final, todo a través del comando *System.currentTimeMillis()*.

Por otro lado, se tiene Java Monitoring & Management Console (JConsole) que es una herramienta que proporciona una interfaz visual para ver la información detallada sobre las aplicaciones Java mientras se están ejecutando en una máquina virtual de Java (JVM) [10]. JConsole utiliza la extensa instrumentación de la JVM para proporcionar información sobre el rendimiento y el consumo de recursos de las aplicaciones que se ejecutan en la plataforma Java. JConsole también permite capturar datos sobre el software JVM y guardarlos para luego poder ver los datos e incluso poder compartirlos.

2.5 Ambiente Android

Para diseñar una aplicación en Android, es necesario tener en claro los elementos que la componen y la funcionalidad de cada uno de ellos. Android es un sistema operativo Open Source basado en Linux diseñado principalmente para dispositivos móviles con pantalla táctil, como Smartphones y/o Tablets.

Aunque la mayoría de las aplicaciones están escritas en Java, no hay una máquina virtual Java en la plataforma. El código Java no es ejecutado, sino que primero se compila en el ejecutable Dalvik y corre en la Máquina Virtual Dalvik. Dalvik es una máquina virtual especializada diseñada específicamente para Android y optimizada para dispositivos móviles.

Para la comprensión completa del documento, es necesario explicar algunos conceptos claves utilizados por Android tal como ilustra la Figura 3.

- **Aplicaciones**: las aplicaciones base incluyen un cliente de correo electrónico, programa de SMS, calendario, mapas, navegador, contactos y otros. Todas las aplicaciones están escritas en lenguaje de programación Java.
- **Marco de trabajo de aplicaciones**: los desarrolladores tienen acceso completo a los mismos APIs del framework usados por las aplicaciones base. La arquitectura está diseñada para simplificar la reutilización de componentes; cualquier aplicación puede publicar sus capacidades y cualquier otra aplicación puede luego hacer uso de esas capacidades (sujeto a reglas de seguridad del framework). Este mismo mecanismo permite que los componentes sean reemplazados por el usuario.
- **Librerías**: Android incluye un conjunto de bibliotecas de C/C++ usadas por varios componentes del sistema. Estas características se exponen a los desarrolladores a través del marco de trabajo de aplicaciones de Android; algunas son: System C library (implementación biblioteca C estándar), bibliotecas de medios, bibliotecas de gráficos, 3D y SQLite, entre otras.
- **Máquina virtual Dalvik**: Android incluye un set de librerías base que proporcionan la mayor parte de las funciones disponibles en las librerías base del lenguaje Java. Cada aplicación Android corre su propio proceso, con su propia instancia de la máquina virtual Dalvik. Dalvik ha sido escrito de forma que un dispositivo puede correr múltiples máquinas virtuales de forma eficiente. Dalvik ejecuta archivos en el formato Dalvik Executable (.dex), el cual está optimizado para memoria mínima. La Máquina Virtual está basada en registros y corre clases compiladas por el compilador de Java que han sido transformadas al formato.dex por la herramienta incluida "dx".

- **Núcleo Linux**: Android depende de Linux para los servicios base del sistema como seguridad, gestión de memoria, gestión de procesos, pila de red y modelo de controladores. El núcleo también actúa como una capa de abstracción entre el hardware y el resto de la pila de software.

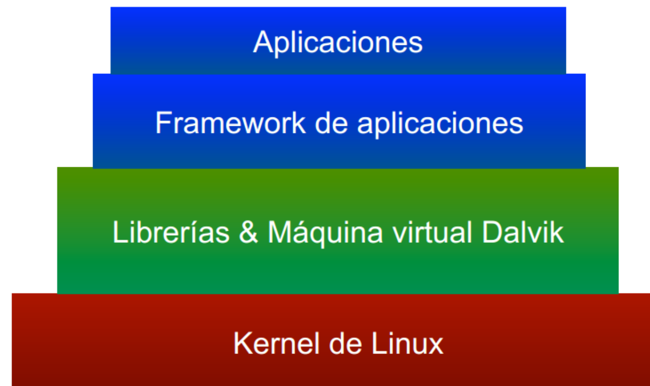

Figura 3. Arquitectura Android

Para cada proyecto Android se utiliza una estructura de directorios que contiene una serie de elementos. En el caso particular de Eclipse, que es un programa informático con herramientas de programación de código abierto multiplataforma, va a tener los siguientes elementos:

AndroidManifest.xml

Es un archivo XML, donde se configura el comportamiento de la aplicación, se habilitan o deshabilitan permisos. En este archivo se ubican las actividades y las diferencias actividades que tiene la aplicación.

/src

Carpeta src que significa source, donde están ubicados todos los archivos Java.

/gen

Carpeta gen, donde están ubicados los archivos generados por Eclipse.

/assets

Carpeta donde van ubicados archivos importantes como JSON, de video, ect.

/libs

Carpeta que contiene las librerías en JAR.

/res

/res/layout

Carpeta que contiene los archivos XML o todos los recursos para la visualización de la aplicación.

/res/values

Carpeta que contiene los valores asociados a la aplicación. Acá está contenido un archivo string.xml que facilita la internacionalización.

En cuanto a los componentes, existen varios de éstos que permitirán realizar la construcción de la aplicación. Una aplicación desarrollada en Android puede contener uno o más elementos `activity`. Cada uno de esos elementos generalmente es una ventana con la que el usuario puede interactuar. En términos de implementación corresponde a una clase Java, que hereda de la clase `activity`. Al momento de crear esta ventana se muestra una interfaz de usuario, la cual se define utilizando una instancia de la clase `View`.

Al momento de iniciar una aplicación se mostrará al usuario una actividad desde la cual se puede llamar a otra actividad y esa otra a su vez puede llamar a otra. Al momento de finalizar la actividad se vuelve a la actividad que la llamó. Debido a que se pueden llamar muchas actividades durante la ejecución de una aplicación, Android utiliza una pila para gestionarlas.

En la Figura 4 [11] se observa que cada actividad posee un ciclo de vida, pasando por diversos estados:

- ejecución (*`resumed`*): la actividad es visible (está en *`foreground`*) y el usuario puede interactuar con ella.
- pausa (*`paused`*): la actividad es visible pero otra ventana (por ejemplo una ventana transparente o que no ocupe toda la pantalla) tiene el foco (está en *`foreground`*).
- detenida (*`stopped`*): la actividad no es visible (está en *`background`*).

Al momento de estar en pausa o detenida, el objeto *activity* es mantenido en memoria, manteniendo sus estados e información.

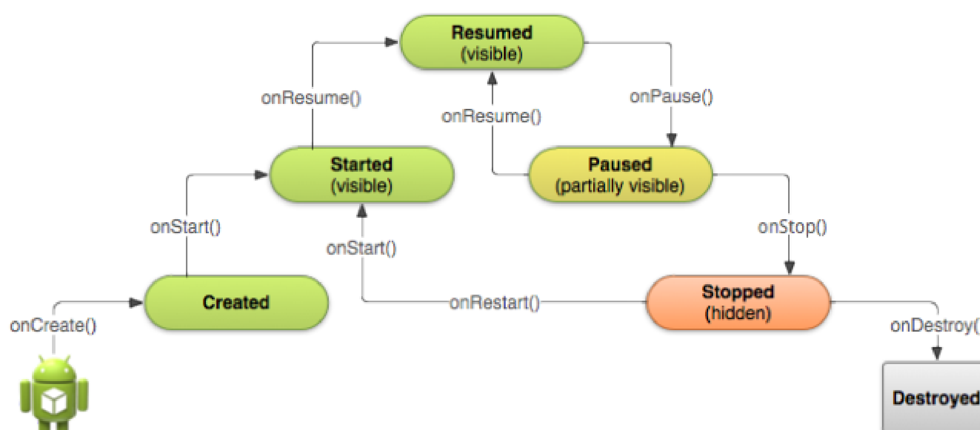

Figura 4. Ciclo de vida de una actividad en Android

Una vez definido el ciclo de vida, hay que tener en cuenta qué métodos son importantes en cada uno de ellos. Se presentan a continuación los métodos más importantes de una actividad:

- *onCreate()*: se invoca al crear una actividad. Este evento sólo se invoca la primera vez que se llama a una actividad.
- *onRestart()*: se invoca después que la actividad ha sido detenida y justo antes de ser iniciada de nuevo. Después de ella se invoca a *onStart()*.
- *onStart()*: es invocado cada vez que la actividad es visible para el usuario. Es decir, la primera vez que se muestra y las veces que en las que vuelve a aparecer tras haber estado oculta. Si la actividad vuelve a aparecer (pasa a *foreground*) se llama a *onResume()*. Si la actividad se oculta se llama a *onStop()*.
- *onResume()*: Se invoca antes que la actividad empiece a interactuar con el usuario. En este punto la actividad está en la parte superior de la pila.
- *onPause()*: Se invoca cuando otra actividad va a entrar a ejecución. Por tanto, la actividad pasa al fondo de la pila.

Los fundamentos presentados en esta sección buscan introducir al lector en el ambiente Android y darle una breve noción de la estructura del código desarrollado a lo largo de estos capítulos. A continuación la Sección 2.6 explica el contexto en el cual este Trabajo de Título ha sido desarrollado.

2.6 Contextualización

El Centro de Innovación de Litio (CIL) ha desarrollado una bicicleta eléctrica con sistemas de control que permiten enviar datos como la corriente y el voltaje a través del protocolo Bluetooh 4.0 y/o una conexión USB hacia un dispositivo móvil. La Figura 5 (Imagen obtenida de la presentación de E-libatt octubre 2014) muestra el esquema del módulo Master del sistema de control de la bicicleta eléctrica del CIL.

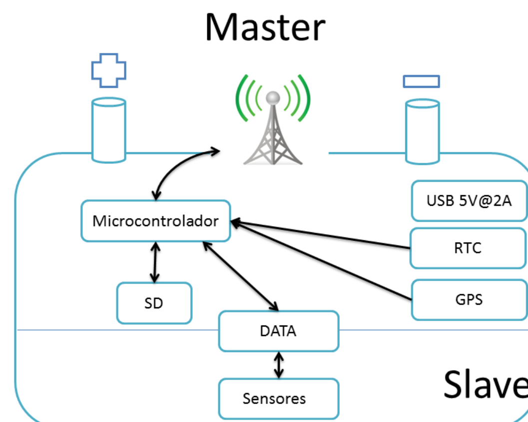

Figura 5. Esquema del módulo Master que contiene el sistema de control de la bicicleta eléctrica del CIL

Por otro lado, se encuentran desarrollando una aplicación para Android y IOS llamada E-libatt que posee las siguientes funcionalidades:

- Medir el estado-de-carga de la batería
- Informar acerca de:
    - Velocidad actual
    - Distancia recorrida
    - Altitud
    - Potencia del motor
    - Temperatura de las baterías
- Mostrar trayectoria en un mapa
- Mostrar gráficos con el historial de cómo han cambiado estas variables
- Enviar datos al servidor

La *app* permite conocer, almacenar y visualizar información importante acerca de la bicicleta eléctrica, en particular el SOC de la batería. Sin embargo, en estos momentos se cuenta con una estimación del SOC muy simplificada, donde se mide el voltaje cuando la carga es máxima y cuando es mínima y se procede a calcular linealmente el SOC. El resultado obtenido es mostrado en la pantalla del dispositivo mediante la aplicación.

Este Trabajo de Título busca aportar mediante la integración del algoritmo, explicado en los apartados anteriores, a un ambiente Android con el objetivo ser utilizado por bicicletas eléctricas. Este algoritmo reemplazará el sistema de estimación actual y entregará información más precisa y exacta que beneficiará al usuario final de la bicicleta eléctrica.

Capítulo III

En el Capítulo II fue presentado el algoritmo de estimación y pronóstico del Estado de Carga para bicicletas eléctricas. En este apartado se implementará el algoritmo *off-line* en lenguaje Java para posteriormente ser introducido dentro de una aplicación Android. Esto para que el CIL, en una etapa posterior, pueda integrarlo a la aplicación que ellos desarrollen.

El ciclo de vida de un proyecto con lleva numerosas tareas agrupadas en etapas. Una etapa común para casi todos los desarrollos es la de obtención de requisitos. Acá se identifican los recursos que se tienen y los requerimientos establecidos.

3.1. Requerimientos

3.1.1. Recibir datos vía Bluetooh / conexión USB

La batería de la bicicleta generará datos de corriente y voltaje los cuales serán obtenidos a través de sensores y enviados vía Bluetooh o a través de una conexión USB. El módulo de control de cada bicicleta se encargará de realizar estas tareas mencionadas. En lo que respecta al presente trabajo, el primer requerimiento es poder recibir los datos bajo el protocolo de comunicación Bluetooh para ser procesados posteriormente.

3.1.2. Almacenar datos recibidos y generados

Los datos recibidos de corriente y voltaje son almacenados dentro del módulo de control de la propia bicicleta. Sin embargo, para efectos prácticos de simulación y pruebas es importante poder contar con los datos recibidos dentro del dispositivo móvil que contiene la aplicación, esto permite analizar el comportamiento y performance del algoritmo de estimación y pronóstico del Estado de Carga. En el caso de estimación se almacena el voltaje y SOC estimado, mientras que para la predicción se guarda la predicción del voltaje y SOC, en conjunto con el EOD o tiempo de descarga.

3.1.3. Estimar y pronosticar el Estado de Carga de la batería

Una vez recibido los datos de entrada, la aplicación debe proporcionar, en pantalla, información sobre el Estado de Carga y Tiempo de Descarga de la batería. Cada usuario que ingrese a la *app*, debe visualizar un porcentaje de carga de la batería y un tiempo de uso pronosticado antes que la batería se descargue. Estos datos proporcionados al usuario deben ser actualizados automáticamente cada cierto intervalo de tiempo.

3.1.4. Subir información obtenida a la nube

La información obtenida durante la recolección de datos debe ser subida a un servidor con el objetivo de acceder a ella en cualquier momento que se requiera. Un ejemplo de uso de esta característica es en el caso de que un banco de baterías esté defectuoso. Esto permite tener un historial del uso de la batería y así generar procedimientos preventivos relativos al estado de salud de las baterías.

Estos cuatro requerimientos principales permitirán otorgar un prototipo que permita al usuario obtener información relevante sobre su bicicleta eléctrica. En lo que sigue, se detallan los componentes principales a utilizar para cumplir con los requerimientos mencionados.

3.2. Determinación de Componentes Principales a utilizar

Lo primero a mencionar en las componentes a utilizar es la API de Android utilizada la cual corresponde a la API 20 Android 4.4W.

3.2.1. Requerimiento 1: Recibir datos vía Bluetooh 4.0 o USB

La bicicleta E-bike posee un módulo Bluetooh cuyo estándar de comunicación con aplicaciones móviles es de bajo consumo Bluetooth 4.0 (BLE) que permite 24Mbit/s.

Para realizar la comunicación entre ambos dispositivos se utiliza la librería *RBLService.java* que implementa la comunicación con el dispositivo BLE en segundo plano.

3.2.2. Requerimiento 2: Datos generados almacenados

Los datos recibidos y generados son almacenados dentro de archivos de texto (.txt), separados por comas. Para ello se utilizan varias librerías que proporciona Java para el manejo de archivos las cuales se mencionan a continuación:

- `java.io.BufferedReader`
- `java.io.BufferedWriter`
- `java.io.File`
- `java.io.FileReader`
- `java.io.FileWriter`
- `java.io.IOException`

Estos archivos se encuentran ubicados en la carpeta */sdcard/* de la memoria del dispositivo móvil.

3.2.3. Requerimiento 3: Algoritmo de estimación y pronóstico off-line

El algoritmo de estimación y pronóstico del Estado de Carga ha sido programado en Java siguiendo buenas prácticas de programación para optimizar el código para Android. Esto debido a dos motivos principalmente. El primero es que si se optimiza la manera en que se escribe el código se generará menos código intermedio con lo cual la máquina virtual tendrá que transformar menos código a código binario y por ende la aplicación irá más rápido. El segundo motivo, es que como efecto de optimizar el código, la aplicación consumirá menos batería, el cuál es un objetivo deseable para cada usuario.

Dentro de las buenas prácticas utilizadas se pueden nombrar las siguientes:

1. Evitar concatenaciones de String
2. Crear métodos con el menor número de entradas posibles
3. Pre-calcular operaciones complejas y utilizarlos como constantes
4. Sacar fuera de los bucles las constantes y creación de nuevos objetos
5. Evitar el acceso repetido al índice de un arreglo

Por otro lado, Java posee varias herramientas de análisis de código para realizar búsqueda de errores. Una de estas herramientas es FindBugs, que analiza el bytecode (instrucciones que la máquina virtual de Java ejecuta) buscando algunos patrones conocidos. Lo bueno de este programa es que no se limita a una búsqueda de expresiones regulares, sino que intenta comprender lo que el programa quiere hacer. Mediante el uso de FindBugs se corrigen principalmente los errores de comparación de tipo *double*, debido a la inexactitud que existe en este tipo de datos.

El algoritmo de estimación y pronóstico del SOC, requiere de diversos métodos matemáticos para ser implementado. Por ello, fue necesario desarrollar una librería llamada "*MathLib*" que contiene todas las funciones matemáticas ocupadas en el código y que MATLAB® provee por defecto. Por ejemplo, la función *zeros(int n, int m)* que entrega una matriz de ceros con dimensión $n \times m$, o también funciones más complejas como *kmens(double[][] X, int k)* que recibe un vector fila de datos y el número de cluster para ejecutar el algoritmo de clustering. Si bien se tienen todas las funcionalidades necesarias para operar en forma correcta, esta librería no posee la plenitud de los métodos que MATLAB® ofrece.

3.2.4. Requerimiento 4: Subir información a un servidor

Para subir la información generada a la nube es necesario poder contar con un servidor de manera de realizar una conexión HTTP a través una URL. Se utiliza el método POST para hacer envío del archivo mediante el uso de varias librerías Java que se muestran a continuación.

- `java.io.DataOutputStream`
- `java.io.FileInputStream`
- `java.io.IOException`
- `java.io.InputStream`
- `java.net.HttpURLConnection`
- `java.net.MalformedURLException`
- `java.net.URL`

3.3. Diseño e implementación de cada Requerimiento

3.3.1. Arquitectura de Hardware

Para contextualizar un poco la elección de la arquitectura de hardware, se presenta en la Figura 6 un diagrama conceptual del sistema de comunicación de la E-bike con un dispositivo móvil.

Figura 6. Sistema de comunicación

Se analizaron tres arquitecturas como soluciones posibles para implementar el algoritmo de estimación y pronóstico. La primera considera ejecutar el código dentro del sistema de control de la batería, la segunda en una *app* dentro del dispositivo móvil y la tercera en algún servidor. Finalmente, se decide por utilizar la segunda opción debido a lo siguiente:

- La primera opción fue descartada debido a que el procesador que utiliza el sistema de control de la batería es una placa Arduino UNO y sus capacidades de procesamiento y memoria son bastante limitadas para el nivel de robustez del algoritmo.

- Se descarta la tercera opción debido a la posibilidad de utilizar la aplicación sin necesidad de poseer Internet Móvil. Esta independencia de utilizar la aplicación en sectores donde no exista cobertura de redes de celulares (3G, 4G o EDGE) genera en el usuario fidelidad sobre la aplicación. De todas formas esta alternativa es igualmente válida si el usuario final al que se apunta, tenga como requisito para el uso de la aplicación el poseer algún servicio de Internet.

Por tanto, la arquitectura de hardware seleccionada es presentada en la Figura 7.

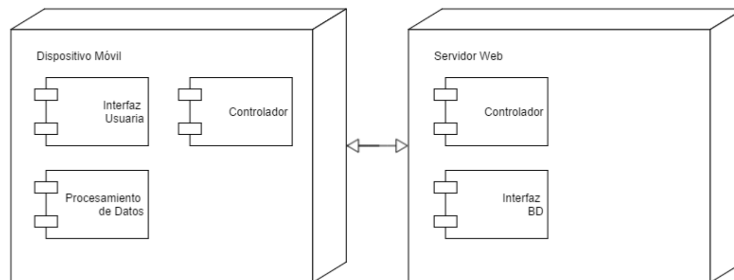

Figura 7. Arquitectura de Hardware

3.3.2. Arquitectura de Software

La Figura 8 muestra el icono de la aplicación desarrollada. Este diseño es una extracción del logo del proyecto desarrollado por el CIL.

Figura 8. Ícono aplicación E-libatt

La aplicación cuenta con los siguientes cuatro módulos principales: Interfaz usuaria, Interfaz de prueba, controlador, interfaz de base de datos. A continuación se describe cada uno.

3.3.2.1.Interfaz Usuaria

Pensando desde el punto de vista del usuario se entrega en pantalla la información necesaria que se puede obtener a través del AEP, es decir, el Estado de Carga porcentual estimado y la predicción del fin de la descarga. El SOC porcentual se muestra con valores dentro del rango entre 0 y 100, mientras que el EOD se visualiza mediante horas, minutos y segundos.

La Figura 9 muestra la interfaz de usuario implementada.

Figura 9. Interfaz de usuario

La información sobre el diseño de esta interfaz está contenida dentro de un archivo XML llamado *user_layout.xml* ubicado en la carpeta *res/layout* del proyecto de la aplicación. Ahí se encuentra toda la información sobre colores, tamaños y letras utilizadas. Además, para el caso de la barra de progreso que indica el porcentaje de batería restante, su diseño se encuentra en un archivo llamado *myprogressbar.xml* dentro de la carpeta *res/drawable* del proyecto.

3.3.2.2.Interfaz de Prueba

La interfaz de prueba se diseña en función de los parámetros de interés en el AEP. Se muestra el voltaje y corriente de entrada; voltaje, SOC y resistencia interna estimados; voltaje, SOC y EOD predichos. La Figura 10 muestra esta interfaz implementada.

Figura 10. Interfaz de prueba

Al igual que en el caso anterior la información sobre el diseño de esta interfaz está contenida dentro de un archivo XML llamado *debug_layout.xml* ubicado en la carpeta *res/layout* del proyecto de la aplicación.

3.3.2.3.Controlador

El algoritmo encargado de controlar el comportamiento de la aplicación se encarga de recibir los datos de corriente y voltaje, permite Estimar y Pronosticar en los momentos adecuados, visualizar datos y almacenar datos. Un diagrama simplificado de este algoritmo se muestra en la Figura 11.

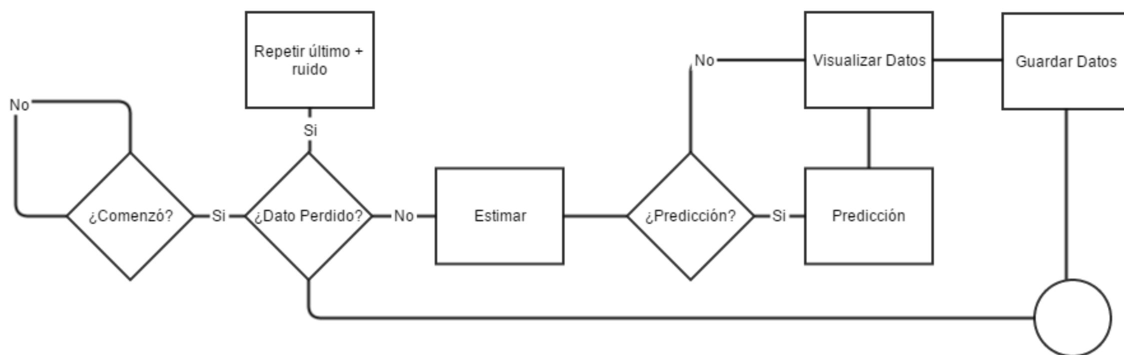

Figura 11. Algoritmo controlador

La implementación de las etapas de análisis de dato de entrada, visualización y almacenamiento de datos (véase la Sección 2.3.4.1), estimación y predicción (véase las secciones 2.3.3.3 y 2.3.3.4) serán mostradas en las secciones siguientes.

3.3.2.4. Interfaz a servidor

Para realizar el envío de datos dentro de un archivo *.txt* a un servidor se utilizan las librerías Java mencionadas en Requerimiento 4: Subir información a un servidor. Desde el lado del servidor se utiliza un archivo *.php* que se encarga de recibir el archivo del cliente Android y guardarlo en la carpeta indicada. Una vez teniendo el archivo en el servidor es posible leer y administrar la información de tal forma de almacenarla en cualquier base de datos, como por ejemplo, MySQL.

```php
<?php

      $target_path = basename( $_FILES['uploadedfile']['name']);

      if(move_uploaded_file($_FILES['uploadedfile']['tmp_name'],
      "uploaded/" . $target_path)) {
            echo "Archivo ". $target_path . "subido correctamente";
      } else {
            echo "Error al subir el archivo";
      }

?>
```

Código 1. Recibir archivo en el Servidor mediante PHP

En el código anterior *basename* se encarga de devolver el último componente de nombre de una ruta el cual es utilizado en la siguiente línea para copiar el archivo dentro de la ruta elegida. Así mediante *move_uploaded_file* se mueve el archivo subido a la nueva ubicación. Por último, *echo* se encarga de mostrar las cadenas de texto en caso de éxito o error.

Por el lado de la aplicación se utiliza la clase *HttpFileUploader* la cual en su constructor recibe la URL del servicio PHP anteriormente comentado y el nombre del archivo a subir. Mediante el método *doStart()* recibirá el archivo y realizará la subida al servidor. El uso de este método se realiza desde una clase, que implementa la interfaz *Runnable*, diferente al de la interfaz de usuario, esto para evitar dejar bloqueada la interfaz mientras se realiza la subida.

El código encargado de realizar la subida se muestra a continuación.

```java
public void uploadFile(String filename){
      String fpath                  = "/sdcard/"+ filename + ".txt";
      try {
            FileInputStream fis     = new FileInputStream(fpath);
            HttpFileUploader htfu   = new
            HttpFileUploader("http://www.tuservidor.cl/upload.php","nop
            aramshere", fpath);
            htfu.doStart(fis);
            Toast.makeText(getApplicationContext(), filename+".txt
            upload", Toast.LENGTH_SHORT).show();
      } catch (FileNotFoundException e) {
            Toast.makeText(getApplicationContext(), "I/O error",
            Toast.LENGTH_SHORT).show();
            e.printStackTrace();
      }
}
```

Código 2. Función que sube el archivo de datos a servidor

También es importante mencionar el uso de las notificaciones en Android o *Toast*, las cuales indican al usuario, a través de mensajes que se muestra en pantalla, si el archivo fue subido correctamente o un error ocurrió.

3.3.3. Algoritmo de estimación y pronóstico off-line

Para la implementación del algoritmo de estimación y pronóstico dentro de la aplicación Android, se debe traspasar todo el código de MATLAB® en lenguaje Java. Para ello, tal como muestra la Figura 12, se crean tres paquetes: *data*; *mathlib*; *soc.*

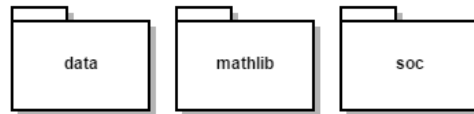

Figura 12. Paquetes utilizados en algoritmo de estimación y pronóstico del Estado de Carga en Java

Un paquete o *package* es una agrupación de clases afines. Este concepto es equivalente al de librería. El primer paquete *data* es el encargado del manejo de los datos. Lee y guarda datos en archivos Excel y también permite imprimirlos en la consola. El segundo paquete *mathlib* contiene una clase con todos los métodos necesarios para la ejecución del algoritmo. Acá se puede encontrar cálculos de medias, el algoritmo k-means utilizado para el cálculo de las corrientes futuras para las Cadenas de Markov, métodos que retornan un número aleatorio con distribución normal utilizados en la estimación del voltaje, entre otros. El tercer paquete *soc* es el paquete principal y tal como muestra la Figura 13 contiene cuatro clases: *Algorithm*; *Model*; *ParticleFilter*; *MarkovChain*.

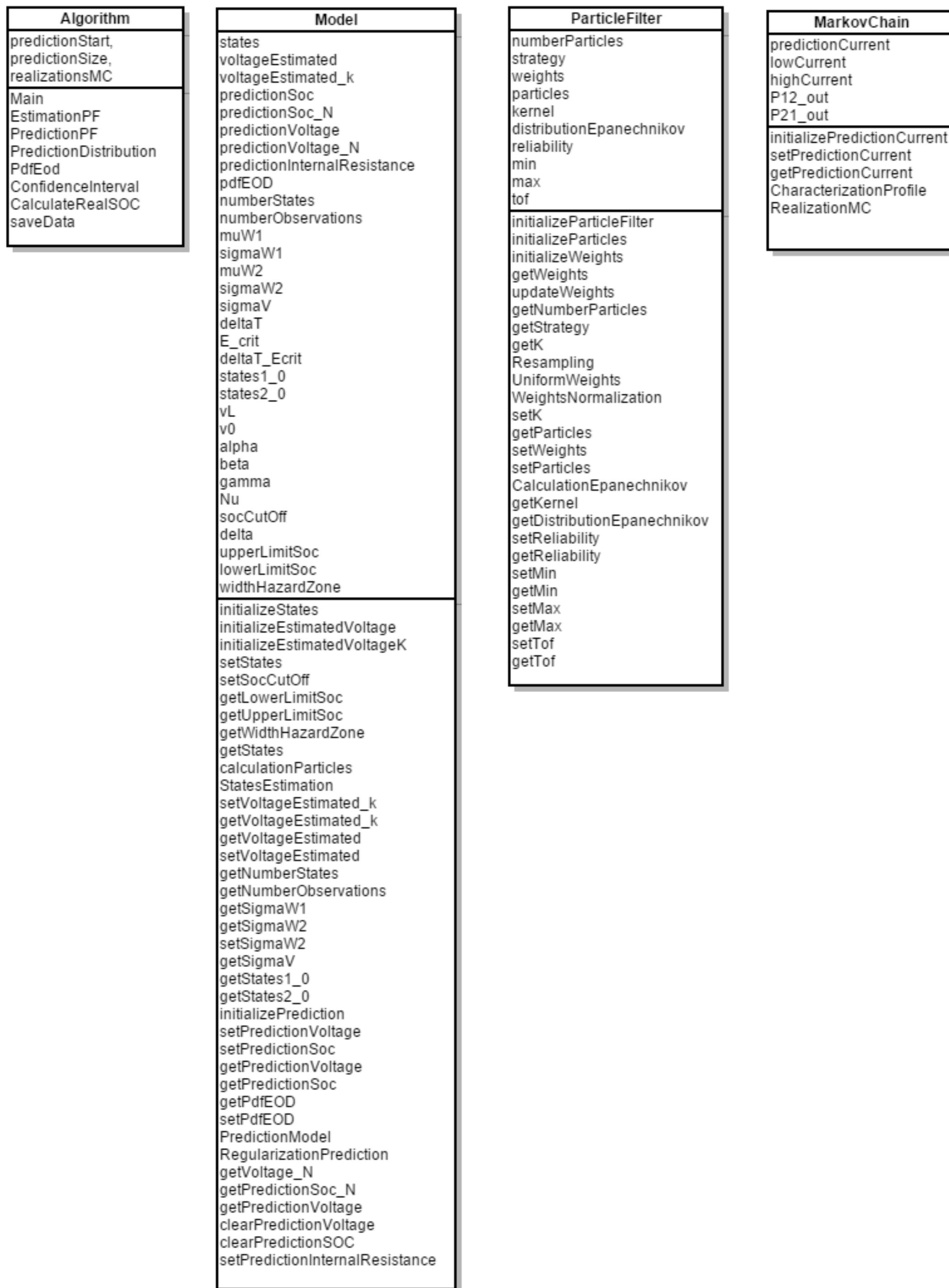

Figura 13. Diagrama UML de clases contenidas en el paquete '*soc*'. En el sector intermedio se detallan los atributos de cada clase, mientras que en el sector inferior se indican los métodos que poseen.

La clase principal de todo el proyecto es *Algorithm.java* y está encargada de ejecutar el algoritmo de estimación y pronóstico de Estado de Carga. Esta clase contiene varios

métodos y entre ellos se encuentran *EstimationPF* y *PredictionPF* que realizan la estimación y pronóstico respectivamente. Para ejecutar este algoritmo se requiere de un modelo, tal como se describe en la Sección 1.2.2.1. Por esta razón, la segunda clase del paquete *soc* es la llamada *Model.java* que contiene todos los parámetros y funciones necesarias para realizar la estimación de los estados y del voltaje en función del modelo presentado. En esta clase se aplican los conceptos de modularidad y encapsulamiento, así los parámetros se encuentran protegidos del mundo exterior y basta con cambiar su valor en una sola variable para que el algoritmo funcione en ese nuevo estado.

Existen dos clases más, *ParticleFilter.java* y *MakovChain.java*. La primera contiene parámetros como el número de partículas a utilizar, un arreglo de partículas y otro con sus respectivos pesos. Además, contiene métodos de inicialización de las mismas partículas y pesos, métodos de resampling, normalización de pesos y cálculos de pesos uniformes. La última clase del paquete *soc* es *MarkovChain.java* la cual permite encontrar los estados de baja y alta corriente según el perfil de uso que se le ha dado a la batería antes del instante de predicción. También contiene las probabilidades de transición entre estados y dos métodos importantes: *CharacterizationProfile* y *RealizationMC*. El primer método permite caracterizar el perfil de uso de la batería mediante el algoritmo de *k-means*. El segundo realiza la Cadena de Markov y establece la predicción de la corriente dado las probabilidades de transición y los estados calculados en el método anterior.

La Figura 14 muestra una simplificación del funcionamiento del algoritmo *off-line*. Esta figura intenta mostrar la relación existente entre los principales métodos de las cuatro clases existentes.

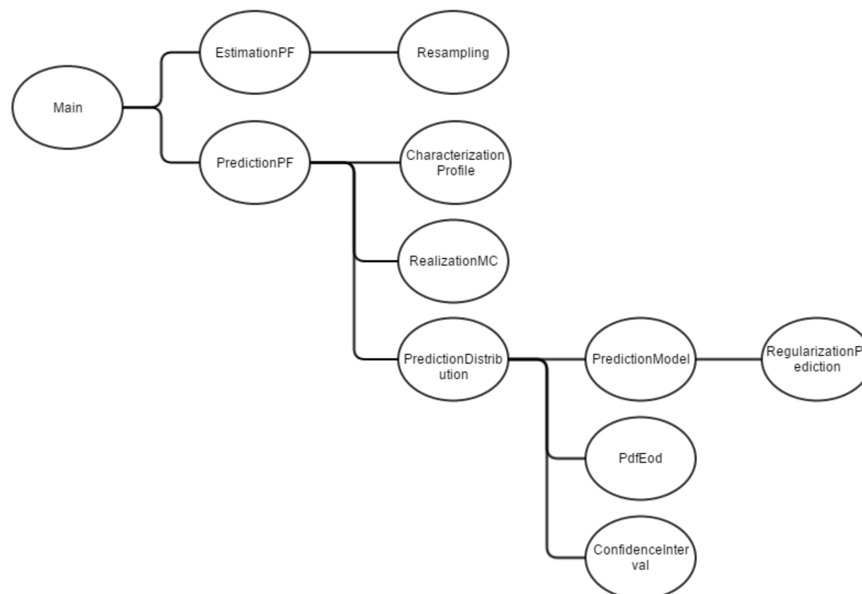

Figura 14. Relación entre principales métodos

El método EstimationPF realiza la estimación del voltaje y del Estado de Carga, mientras que PredictionPF realiza la predicción del voltaje, la predicción del SOC y entrega el tiempo de descarga, ambos métodos serán detallados a continuación. La Figura 15 muestra un diagrama de flujos simplificado del "Main" o script principal.

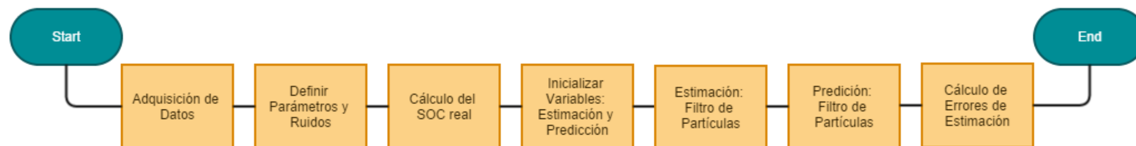

Figura 15. Algoritmo Estimación y Predicción del Estado de Carga

2.3.3.1 Adquisición de datos

El código del algoritmo *off-line* comienza con la adquisición de datos, carga los valores de la parametrización de la batería y carga las mediciones de los sensores de voltaje y corriente a través de las clases *ParameterizationBattery* y *Route* respectivamente.

En esta etapa se leen los datos desde un archivo Excel gracias a la librería POI 3.9 [12] que permite manejar archivos de la clase estándar Office Open XML. También se colola como alternativa el uso de las librerías Java para la lectura de archivos de texto.

2.3.3.2 Inicialización de parámetros y objetos

Se inicializan dos objetos claves de las clases *Model* y *ParticleFilter*. El primero permite establecer valores asociados al modelo como $V_l, V_0, \alpha, \beta, \gamma$, ruidos de observación y proceso y matrices de estados. Mientras que el segundo permite inicializar las partículas y sus pesos respectivos.

2.3.3.3 Etapa de Estimación

En esta etapa se realiza la estimación de cada partícula y la actualización de sus pesos respectivos para cada estado. La estimación se realiza siguiendo el modelo descrito en las ecuaciones (8) y (9) dentro de la clase *Model*, mientras que la actualización de los pesos se realiza al multiplicar el valor de los pesos del instante anterior con una distribución de probabilidad normal de media 0 $[V]$ y desviación estándar dada por 0,3 $[V]$ dentro de la clase *ParticleFilter*.

Se realiza la normalización del vector de pesos y luego se calcula el índice de eficiencia, en función de los pesos de cada partícula, que genera una condición para aplicar el método de

remuestreo (véase Sección 2.2.1.2) que es aplicado cuando hay degeneración de las partículas. En este método, ubicado dentro de la clase *ParticleFilter*, se implementan dos estrategias que pueden ser utilizadas: *Multinomial* y *Sistemático*. La estrategia *Multinomial* obtiene una muestra ponderada de los índices de las partículas usando los pesos como probabilidades para la ponderación. Luego de esto el método retorna un vector de pesos uniformes y normalizados junto con los estados remuestreados. La estrategia *Sistemático* crea un vector de límites como una suma acumulada de los pesos de las partículas. Luego genera los índices que permiten crear un nuevo vector de partículas las cuales son reemplazadas por degeneración.

A continuación, se muestra un diagrama de flujos simplificado sobre el funcionamiento de la etapa de estimación usando Filtro de Partículas.

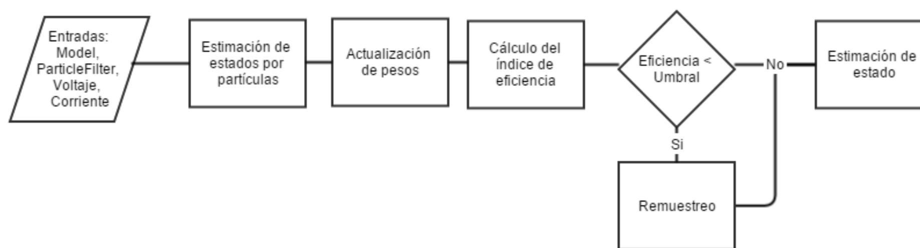

Figura 16. Diagrama de Flujos – Estimación

2.3.3.4 Etapa de Pronóstico

La etapa de pronóstico llega después de varias iteraciones donde ya se han recibido un buen número de datos. Esto es relevante para poder obtener buenas predicciones, el no tener una cantidad de datos adecuada permite que los resultados no sean demasiado acertados.

Esta etapa requiere establecer un perfil de uso de la bicicleta a través del tiempo, es decir, debe intentar adivinar la forma de uso que se le dará a ésta, dados los datos que ha recibido hasta el momento. Un ejemplo sencillo de entender es en el caso de que la bicicleta esté subiendo una pendiente inclinada (como un cerro) durante algún tiempo, por ende la corriente usada será alta. Al avanzar un poco en el tiempo, lo más probable es que la bicicleta siga subiendo y por ende consumiendo corrientes altas. Para lograr esto, se pueden utilizar dos enfoques distintos. El primero utiliza la corriente promedio de las mediciones hasta el momento actual. Mientras que el segundo intenta caracterizar el perfil generando una matriz de probabilidades de transición, dados los datos recibidos, para ser usados en las realizaciones de la Cadena de Markov. Además, este segundo enfoque utiliza una ponderación EWMA o de pesos ponderados la cual le asigna mayor peso a la historia reciente.

Una vez generado el perfil de uso de la batería se procede a ejecutar varias realizaciones de predicción con el objetivo de promediar los resultados obtenidos en cada iteración y así obtener resultados con mejor exactitud y precisión [2]. En esta parte, si el enfoque seleccionado es el de Cadenas de Markov, se ejecuta el método *RealizationCM* que entrega un perfil de uso futuro de la corriente de la batería.

En este método de pronóstico se obtienen las predicciones, la función densidad de probabilidad del tiempo de descarga, el tiempo de descarga y los límites del intervalo de confianza a través del método *PredictionDistribution*. Dentro de este método se realizan las predicciones basadas en el modelo descrito en las ecuaciones (8) y (9), también se genera la distribución acumulada del SOC y la función densidad de probabilidad del EOD.

A continuación, la Figura 17 muestra un diagrama de flujos simplificado del funcionamiento del método de pronóstico.

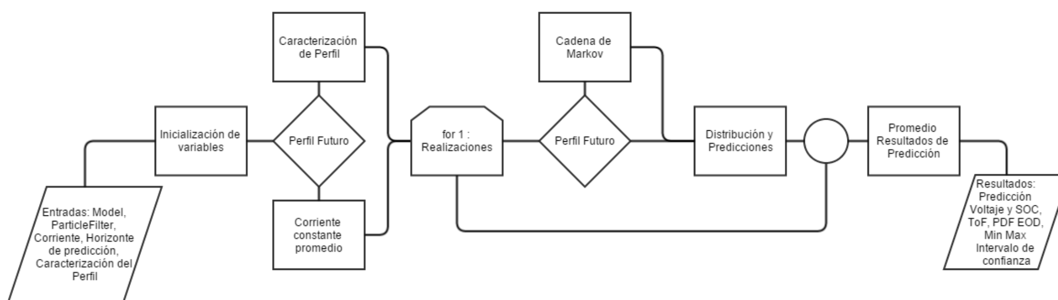

Figura 17. Diagrama de Flujos - Pronóstico

2.3.4 Algoritmo de estimación y pronóstico en una aplicación Android

2.3.4.1 Actividad de Usuario

El algoritmo de estimación y pronóstico es implementado dentro de una clase interna llamada *AEP* ubicada en la actividad *UserActivity* de la aplicación. Esta clase contiene el algoritmo de estimación y pronóstico en línea. Todas las variables de interés son inicializadas en el método *onCreate* que es donde se crea la actividad.

La función encargada de recibir los datos es *onReceive* que funciona como puente entre el servicio y la actividad. Básicamente, el objeto *mGattUpdateReceiver* es el nexo entre el servicio y la actividad. El servicio, cuando tiene nuevos datos, ejecuta el método *onReceive* del *listener*, que a su vez ejecuta un método de la actividad (para el procesamiento de los datos) pasando como parámetro un *Intent*, que es el objeto que trae los bytes desde el servicio a l actividad. En el último *else if* de la función se observa el método *displayData* que es la llamada que ve la actividad con los datos que envía el sistema controlador de la bicicleta eléctrica, en este caso una placa Arduino.

Cada vez que el controlador de la bicicleta envía datos se realiza el proceso de estimación y se repite el proceso en forma cíclica. Una vez que se ha llegado a un número adecuado de datos para realizar pronóstico del tiempo de descarga se desencadena el proceso de estimación y luego de pronóstico. Finalizado el proceso se muestra en pantalla el Estado de Carga y el tiempo de descarga de la batería mediante la función *addData*.

Ojo que el servicio llama a onReceive con algunos datos, pero no necesariamente es una línea de texto, yo ví que eran unos pocos bytes, a veces 0 bytes. Puedes ver el método displayData y processCharacter para ver como acumulo datos de a poco y como se filtran los bytes que no son texto (vi también, que el arduino a veces enviaba basura).

Cada vez que llega un dato es necesario analizar si es un dato correcto o no. El método *onReceive* recibe algunos datos que no siempre son correctos, a veces 0 bytes, para ello se utiliza el método *processCharacter*, que permite acumular datos caracter por caracter a través de un buffer y que filtra los datos (bytes) que no son texto. Una vez que llega un caracter "\n" se envía el buffer como un string al método *onNewLine*. Este método separa los datos de corriente, voltaje y temperatura enviados desde el controlador de la bicicleta. Acá se realiza un pre procesamiento de datos, para revisar que los sensores estén enviando datos dentro de los márgenes esperados, por ejemplo el voltaje no debe bajar de los 30 volts recordando que el banco de baterías tiene un voltaje de corte asignado por el controlador, que se encuentra aproximadamente en ese valor.. En estos casos la acción a tomar es simple (15), se toma el último dato y se le agrega algo de ruido con distribución normal de media cero y desviación estándar igual al ruido de observación para generar un nuevo dato y con este realizar el proceso de estimación.

$$V_k = 0\ ?\ V_k = V_{k-1} + normrnd(0, \sigma_v) : V_k = V_k \tag{15}$$

De todas formas, esta es una medida preventiva si se considera que la pérdida de datos se ha mejorado mediante el uso de mejores sensores de voltaje y corriente.

Se describe a continuación la visualización de datos en la pantalla del dispositivo móvil y posteriormente el almacenamiento de estos dentro de un archivo de texto.

En Android se utiliza el método *onCreate* para realizar la declaración de las variables de texto (*TextView*), botones (*Button*) o barras de progreso (*ProgressBar*) que son las utilizadas para crear una vista de pantalla. Estas son referenciadas mediante un identificador declarado en el archivo XML comentado en las secciones 3.3.2.1 y 3.3.2.2. El Código 3 muestra la declaración de variables utilizadas en la vista de prueba.

```
mXTextView          = (TextView) findViewById(R.id.tvX);
mYTextView          = (TextView) findViewById(R.id.tvY);
salidaTextView      = (TextView) findViewById(R.id.salida_text);
socTextView         = (TextView) findViewById(R.id.soc_text);
resistTextView      = (TextView) findViewById(R.id.resist_text);
jTextView           = (TextView) findViewById(R.id.j_text);
TextView04          = (TextView) findViewById(R.id.TextView04);
TextView07          = (TextView) findViewById(R.id.TextView07);
TextView09          = (TextView) findViewById(R.id.TextView09);
subir               = (Button) findViewById(R.id.button1);
```

Código 3. Declaración de variables de visualización

Visualización de datos mediante la función *setText*, *setText2* y *setText3*. El Código 4 muestra la primera función encargada de visualizar los datos de entrada (corriente y voltaje).

```
private void setText(final String xText, final String yText) {
      runOnUiThread(new Runnable() {
            @Override
            public void run() {
                  mXTextView.setText(xText);
                  mYTextView.setText(yText);
            }
      });
}
```

Código 4. Función que imprime datos en pantalla

El botón *subir* es el encargado de subir la información almacenada en el archivo de texto cada vez que es presionado en la aplicación. Para poder conocer cuando el botón es presionado se utiliza el método *setOnClickListener* tal como muestra en el Código 5.

```
subir.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View arg0) {
            String filename = "datosOut";
            uploadFile(filename);   // Subir archivo
      }
});
```

Código 5. Click en botón Subir para almacenar datos dentro de un servidor

Para almacenar los datos se utilizan archivos de texto donde los datos están separados por comas. Esto facilita la visualización a través de hojas de cálculos como Excel.

```
String readfilename      = "datosOut";
String filecontent       = Float.toString((float) (aux1));
String filecontent2      = Float.toString((float) (aux2));
String filecontent3      = Float.toString((float) (aux3));
FileOperations fop       = new FileOperations();
String text              = fop.read(readfilename);
if(text != null){
      text += filecontent + "," + filecontent2 + "," + filecontent3;
      fop.write(readfilename, text);
} else {
      text = filecontent + "," + filecontent2 + "," + filecontent3;
      fop.write(readfilename, text);
}
```

Código 6. Almacenar datos dentro de archivos de texto

Capítulo IV

Discusión de Resultados

En este capítulo se analiza el comportamiento *off-line* del algoritmo en lenguaje Java y se muestran los resultados de la implementación dentro de una aplicación. Desde el punto de vista de eficacia del código en aspectos como exactitud y tiempo de ejecución se realiza una comparación de performance entre Java y MATLAB®. Además, gracias a la herramienta JConsole se observan monitoreos gráficos en tiempo real de Memoria, CPU, Threads y Classes.

3.1 Datos utilizados

Los datos de voltaje y corriente del acumulador son utilizados como entradas al modelo. Ambas mediciones se pueden realizar durante el proceso de descarga de la batería, lo que permite trabajar en línea. En particular, los datos utilizados para desarrollar el trabajo consisten en un set de datos tomados a la E-bike, el cual está compuesto por mediciones simultáneas de voltaje, corriente y tiempo. Los conjuntos de datos fueron tomados de un banco de baterías (véase Figura 18) de Ion-Litio de $4000\ [mA-hr]$, 3.7 $[V]$ nominales y $< 19\ [m\Omega]$ cada celda. Estos datos obtenidos corresponden al recorrido de una ruta de la E-bike, la cual fue sometida a una operación con carga y velocidad aleatoria.

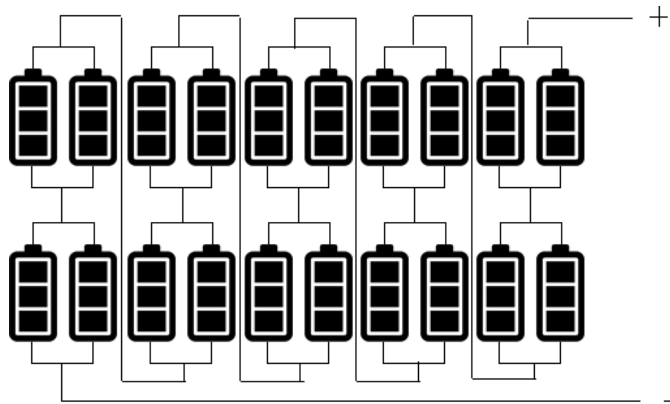

Figura 18. Arquitectura del banco de baterías utilizado en la E-bike

Para las pruebas realizadas se utiliza una ruta llamada *Ruta 5* la cual entrega los siguientes datos que se muestran en la Figura 19. Esta ruta se encuentra en el archivo *Ruta5.m* ubicados en la carpeta */Matlab/* de los archivos adjuntos a este Trabajo de Título.

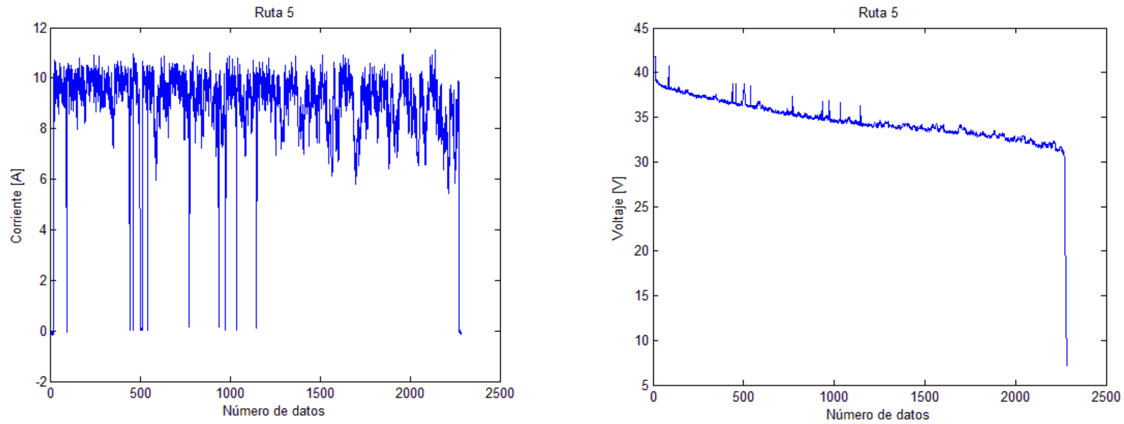

Figura 19. Conjuntos de datos de corriente y voltaje correspondientes al recorrido de la Ruta 5 de la E-bike

3.2 Resultados del algoritmo de estimación y pronóstico del Estado de Carga en Java

A continuación, se presentan los resultados obtenidos de estimación y pronóstico del algoritmo en lenguaje Java. La Figura 20 muestra que el voltaje predicho por el modelo sigue muy de cerca al voltaje de real entrada. El valor cuadrático medio o RMS promedio es de 0.4613, valor calculado mediante diez realizaciones del algoritmo *off-line*.

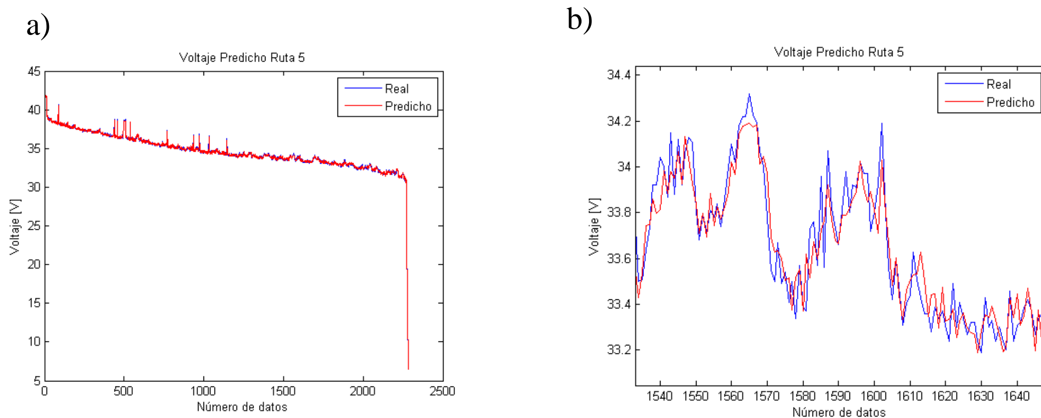

Figura 20. Predicción del voltaje en base a la PDF a posteriori de los estados. a) En azul se muestra el voltaje real, medido por los sensores de la E-bike, mientras que en rojo, se observa el voltaje predicho por los estados. b) Muestra un acercamiento del voltaje predicho, en rojo, y del voltaje observado, en azul.

El inicio de la predicción a largo plazo comienza en el dato número 900 obtenido y es aquí donde se realiza el pronóstico del voltaje, SOC y EOD. Los resultados de la predicción del voltaje a partir de este punto son mostrados en la Figura 21.

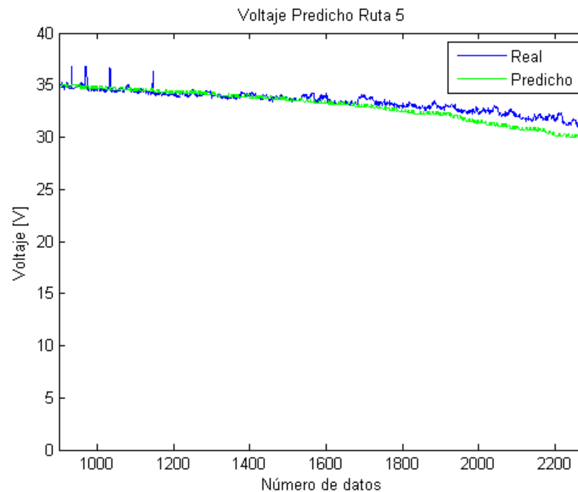

Figura 21. Predicción del Voltaje a largo plazo. En azul se muestra el voltaje observado por los sensores, mientras que en verde se muestra la predicción del voltaje a partir del inicio de la predicción a largo plazo

Hasta el momento se han presentado los resultados de pronóstico de voltaje del banco de baterías. En lo que respecta al Estado de Carga, la implementación del algoritmo *off-line* en Java entrega los siguientes resultados para la estimación y predicción, los cuales son mostrados en la Figura 22. Acá el error cuadrático medio es mayor al de voltaje y promedia el 5% en diez realizaciones.

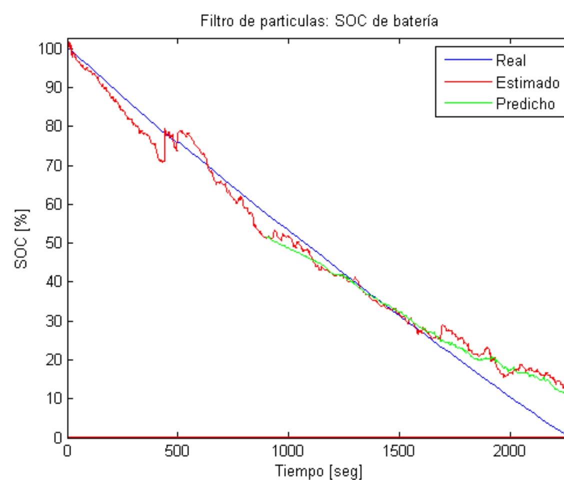

Figura 22. Resultados del Estado de Carga. En azul el SOC "real" el cual se calcula utilizando los datos de voltaje y corriente observados por los sensores, en rojo el SOC estimado a partir de los estados y en verde el SOC predicho a partir del inicio de la predicción a largo plazo

La Figura 22 muestra el SOC "real" el cual es calculado utilizando los datos de voltaje y corriente observados por los sensores de la E-bike y la energía crítica o energía nominal de la batería, valor teórico obtenido en forma experimental [1] y que tiene un valor de $1.065.600\ [V * A * seg]$.

Por otro parte, como se explicó en el Capítulo II para la realización del perfil de uso futuro de la bicicleta eléctrica en la etapa de pronóstico se utilizan Cadenas de Markov (véase Figura 23). Los estados de cada intervalo son determinados utilizando Clustering K-means (Figura 23 a), para posteriormente discretizar la corriente en dos estados: corriente alta y corriente baja (Figura 23 b).

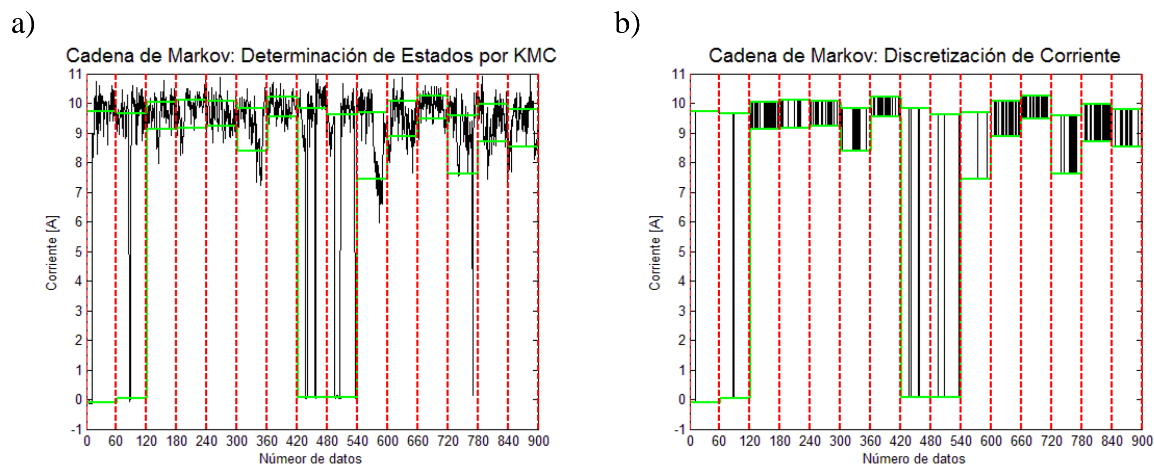

Figura 23. Cadena de Markov. a) Determinación de los estados de corriente alta y baja. b) Discretización de la corriente en los estados determinados.

El largo de cada intervalo es un parámetro de diseño de valor 60 que se ajustó y optimizó en forma *off-line* [2] y cuyo criterio de elección fue que el SOC disminuyese un $2 - 3\%$ de la estimación a priori del comportamiento del SOC en una prueba de descarga [13]. Este valor es muy relevante para la pendiente de la curva de predicción del SOC, ya que si el ancho es muy grande esto conlleva una pendiente baja y por ende se sobreestima el tiempo de descarga de la batería.

Para incorporar la información del perfil de uso de los intervalos anteriores se realiza una ponderación exponencial mencionada en la Sección 1.2.2.2.

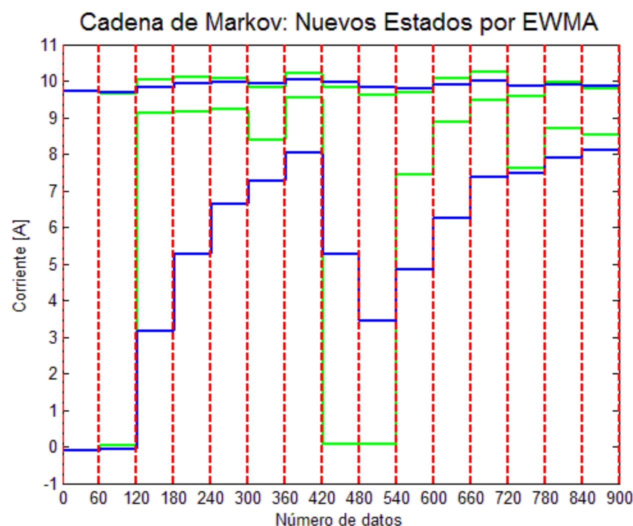

Figura 24. Ponderación EWMA de los estados de corriente alta y baja del perfil de uso de la batería de Ion-Litio. En color azul los nuevos estados y en color verde los estados antes de la ponderación.

La Figura 25 muestra una de las realizaciones del perfil de uso para uno de los intervalos de los datos de corriente antes del inicio de predicción. Se observan las transiciones entre los dos estados de corriente baja y alta en un intervalo de 60 datos. Por último, una vez realizada la caracterización del perfil de uso, se procede a ejecutar varias realizaciones de CM.

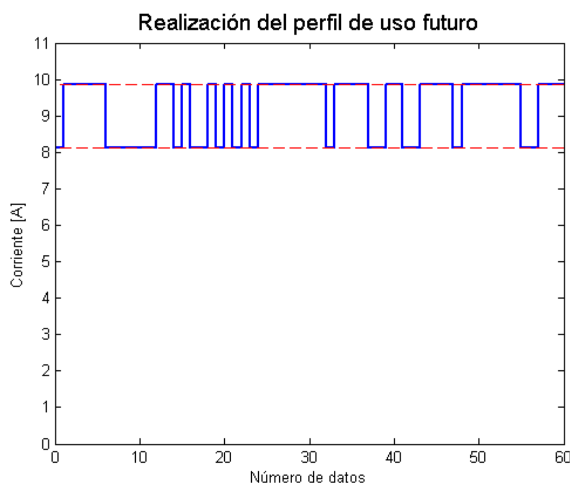

Figura 25. Realización del perfil de uso futuro de la corriente con transiciones entre sus dos estados.

Las figuras mostradas en esta sección muestran un correcto funcionamiento de los métodos y etapas del algoritmo de estimación y pronóstico del Estado de Carga. Cada uno de estos gráficos han sido creados utilizando datos generados por el algoritmo en Java, los cuales han sido extraídos mediante la librería *POI* a una hoja de cálculo.

A continuación, se muestra la comparación de exactitud, tiempo de ejecución y uso de recursos del algoritmo en los ambientes de Java y MATLAB®.

3.3 Performance Java vs MATLAB®

MATLAB® posee una herramienta llamada *Matlab Profiler* para analizar los tiempos de ejecución de cada línea de cada función. Para hacer un llamado a esta herramienta basta con ejecutar "profile viewer" en la línea de comandos de MATLAB®. El uso es bastante intuitivo, se ingresa el nombre de la función a analizar donde dice "Run this code:".

El método principal en este ambiente es *Solución_General* y es similar a la clase *Algorithm* mencionado en la implementación de la Sección 3.3.3. Al utilizar *Matlab Profiler* se obtiene el siguiente informe que se muestra en la Figura 26.

Function Name	Calls	**Total Time**	Self Time*	Total Time Plot (dark band = self time)
Solucion_General	1	8.163 s	0.559 s	
Pronostico_FP	1	5.218 s	0.040 s	
Distribucion_Pred	1	5.176 s	0.840 s	
Prediccion_Modelo	1	4.317 s	0.567 s	
Prediccion_Reg	2099	3.749 s	3.221 s	
Estimacion_FP	2284	1.925 s	0.854 s	

Figura 26. Rendimiento del algoritmo de estimación y pronóstico del Estado de Carga en ambiente MATLAB®

Con esto es posible observar que la etapa de pronóstico (*Pronostico_PF*) es la que más tiempo tarda en ejecutarse, centrándose la mayor parte del tiempo ocupado en el método de regularización *Prediccion_Reg* que utiliza la densidad de Epanechnikov.

Este dato es importante si se piensa desde el punto de vista de la aplicación ejecutándose en el ambiente Android en tiempo real, ya que muestra el comportamiento del algoritmo al momento de pronosticar el tiempo de descarga del banco de baterías y su posterior visualización en pantalla.

Por el lado de Java, este posee una interfaz gráfica de usuario llamada Java Monitoring & Management Console (JConsole) que permite monitorear el rendimiento y consumo de recursos del algoritmo de estimación y pronóstico del SOC que se ejecuta en la plataforma de Java.

En función de esto se realiza una primera comparativa con respecto a los tiempos de procesamiento. Para ello es importante mencionar que las pruebas han sido realizadas en condiciones idénticas en términos de capacidades de procesamiento y memoria. Esto es en un computador con Procesador Intel Core i3 CPU 2.20GHz dual core y memoria RAM 8 Gb.

A continuación, se presenta el promedio de diez realizaciones del algoritmo para cada lenguaje. Para ambos casos el tiempo calculado se mide luego de cargar los datos de parametrización y ruta. El final de la medición corresponde al instante posterior al cálculo de los errores de estimación y no considera, en el caso de Java, el almacenamiento de datos en archivos *.xlm*, ni, en el caso de MATLAB®, la visualización de gráficos.

Tabla 2. Comparativa tiempos de procesamiento

	Tiempo [milisegundos]
Java	$2319 \pm 2\%$
MATLAB®	$5522 \pm 2\%$

Mediante el uso de los elementos básicos de la orientación a objetos como modularidad, encapsulamiento, entre otros (véase Sección 2.4.3) el algoritmo de estimación y pronóstico reduce aproximadamente un 60% el tiempo de procesamiento de un ambiente a otro, esto es, cerca de 3 $[s]$. El tiempo de procesamiento de algoritmo en Java es un buen resultado pensando en la implementación dentro de la aplicación móvil.

Utilizando la herramienta JConsole se realizan varias ejecuciones del algoritmo *off-line*, donde se observa que los recursos utilizados en el momento de mayor demanda (etapa de pronóstico) no superan los 25 MB de memoria ni el 40% de CPU usada en promedio. La Figura 27 muestra una realización del algoritmo.

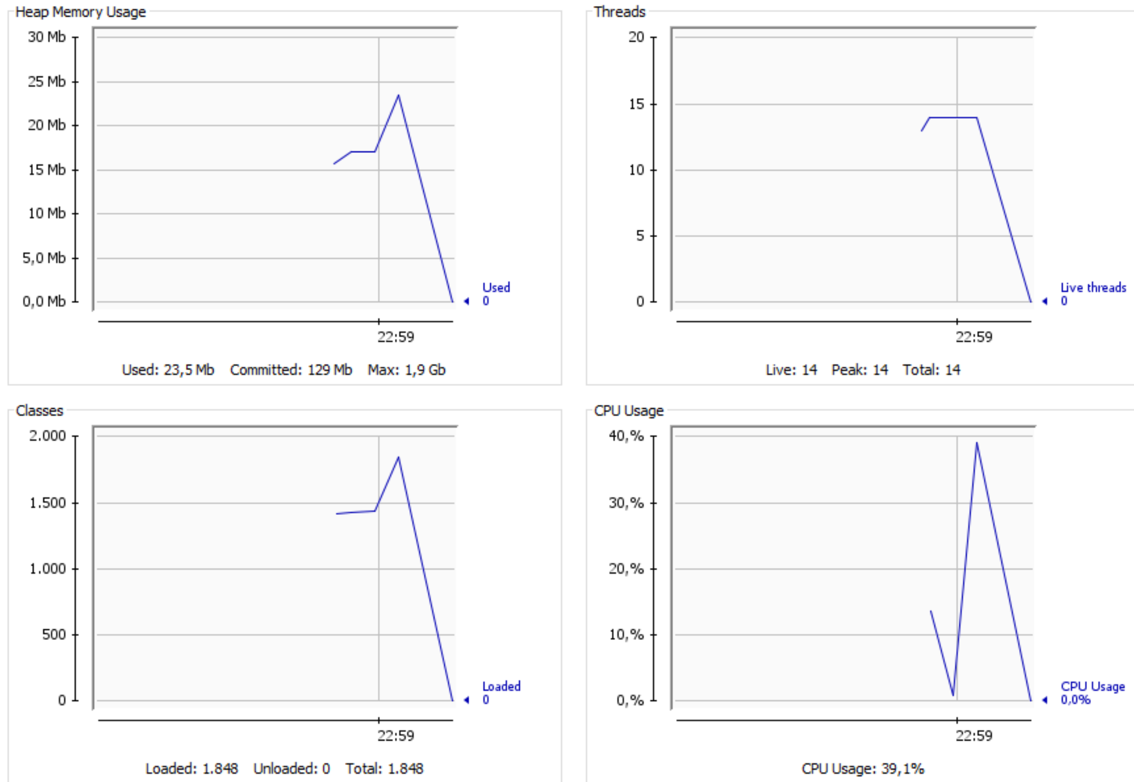

Figura 27. Performance algoritmo off-line

El gráfico del costado izquierdo superior muestra el uso de memoria durante el transcurso del algoritmo, se observa un pick asociado a la etapa de pronóstico. En el gráfico del uso de la CPU ubicado en el costado inferior derecho también se puede observar el pick mencionado asociado a la predicción. Los otros dos gráficos muestran el uso de clases e hilos de ejecución dentro del algoritmo.

3.4 Algoritmo de estimación y pronóstico del Estado de Carga dentro de una aplicación

E-libatt es el nombre de la aplicación desarrollada. Esta ha sido instalada en un Smartphone Motorola Moto X considerado de gama media. Todas las pruebas han sido realizadas en este dispositivo. Las siguientes figuras muestran el funcionamiento de la aplicación en tiempo real tanto para la vista usuario como para la vista de prueba.

Figura 28. Elibatt - Funcionamiento de la aplicación en su vista usuario donde se indica el Estado de Carga y el pronóstico del tiempo de descarga de la batería

Figura 29. Elibatt - Funcionamiento de la aplicación en su vista de prueba o debug donde se indican los parámetros de interés del algoritmo de estimación y pronóstico del Estado de Carga

En las figuras anteriores se puede observar los datos que entrega el algoritmo en cada ciclo. La predicción comienza a partir del dato número 900, el tiempo de descarga es entregado en milisegundos tal como se muestra en el costado derecho de la Figura 29. Este tiempo es procesado para ser mostrado en horas, minutos y segundos tal como se muestra en el costado derecho de la Figura 28.

3.4.1 Cálculo de tiempos de importancia

En esta sección se proponen tiempos de importancia para la ejecución del algoritmo. Para tomar estas decisiones hay que considerar algunos aspectos, como los recursos utilizados y

la preferencia del usuario. La cantidad de recursos utilizados es un limitador sobre la frecuencia de aplicación del algoritmo. Se ha observado en la sección anterior que la parte de pronóstico requiere de un gran número de recursos computacionales. Por otro lado, la entrega de la información en los tiempos adecuados permite al usuario tomar medidas preventivas y decisiones de uso con respecto a la bicicleta eléctrica. Es así que se proponen los siguientes tiempos y/o frecuencias.

- Frecuencia de muestreo

  La frecuencia de adquisición de datos es cada $1\,[s]$. Este es un parámetro establecido en el diseño del Firmware del controlador de la bicicleta.

- Tiempo de pronóstico

  El inicio del pronóstico tiene relación con el nivel de precisión en la etapa de pronóstico. Este tiempo recomendado es de $900\ [s]$ indicado en [1].

- Horizonte de pronóstico

  El horizonte de pronóstico propuesto es de 3600 [s], es decir, 1 hora hacia adelante. Este parámetro se propone en función de la observación de varios set de datos donde la batería muestra una duración promedio de sesenta minutos dentro de un uso continuo.

- Frecuencia de refresco en pantalla

  La frecuencia de actualización de los datos de pantalla se ha fijado en 1 segundo, es decir, en cada ciclo se refresca la información entregada al usuario. Los recursos utilizados para actualizar la pantalla son mínimos comparados a los del AEP por los que no afectan en demasía el rendimiento de la aplicación.

Un tiempo importante de observar es el de predicción en tiempo real. El ejecutar esta etapa tarda en promedio $16[s]$ hasta entregar el tiempo de descarga. Este comportamiento se explica tal como se muestra en la Figura 26 y Figura 27 donde el mayor tiempo de ejecución se centra en esta etapa de predicción. No sólo el tiempo que tarda en ejecutarse es extenso sino que también se pierden los datos de entrada durante esa ventana de tiempo. La solución que se propone a esto es implementar un servicio en Android que se ejecute en segundo plano, es decir, mientras se realizan los cálculos de pronóstico se continúan recibiendo datos y realizando la estimación de estados.

Conclusiones

Este Trabajo de Título, presenta la implementación de un algoritmo de estimación y pronóstico del Estado de Carga de un banco de baterías de Ion-Litio utilizado en bicicletas eléctricas. La implementación incluye toda la teoría base que posee este algoritmo en el uso de Filtro de Partículas para la predicción del EOD. El aterrizar estos conceptos dentro de algo práctico como lo es una aplicación Android para dispositivo móvil ha generado un incentivo adicional para poder realizar este trabajo.

La conversión del lenguaje a Java entrega los mismos resultados que el algoritmo offline original en ambiente MATLAB® e incluso opera $3[s]$ en promedio más rápido por cada ejecución. Una vez definido el código en Java este ha sido validado usando datos experimentales de descarga. En una primera etapa, se observa el gran trabajo que realiza FP para ajustar los estados del sistema en la medida que incorpora información característica del banco de baterías y ciclo de descarga con un error de precisión bastante bajo. Por otro lado, en la segunda etapa de pronóstico es posible observar el trabajo realizado en la caracterización del perfil de uso futuro y las realizaciones de la Cadena de Markov de primer orden. Acá se definen los valores de los estados y de las probabilidades de transición.

Una vez cumplido el desafío de programar el código en Java se integra dentro de una aplicación para dispositivos móviles Android y además, se realizan pruebas en tiempo real de su funcionamiento. Es acá donde se observa el poder de estimación que tiene al entregar en cada segundo un valor estimado del SOC con exactitud. Esta característica es importante desde el punto de vista del usuario ya que permite tomar decisiones con muy buena precisión sobre el uso de la bicicleta eléctrica. Además, no solo se entrega un porcentaje del SOC de la batería sino también, se muestra el tiempo de descarga del banco de baterías según el perfil de uso que el usuario le ha dado en los instantes previos. Un ejemplo claro, es que si un usuario está subiendo un cerro la aplicación mostrará un EDO mucho más bajo que si el usuario baja una colina. Se desarrolló otra aplicación que permite al desarrollador poder conocer en detalle los valores de interés que el código genera en cada ciclo.

No solo la creación de la aplicación es un objetivo a cumplir, sino que se intenta crear un diseño atractivo que generara en el usuario aceptación y comodidad a la hora de revisar cuanta energía disponible tiene. Se establecieron letras de gran tamaño, una imagen visual que simula una batería llena y un fondo que contraste las letras blancas para mejorar la visualización.

Estas implementaciones fueron integradas a un servicio web, donde se da la posibilidad de crear y diseñar una base de datos a partir de los datos que se le envía al servidor, este

servicio implementa el estándar HTTP, por lo que puede ser reutilizado por cualquier aplicación o usuario.

Con el trabajo realizado se dan por cumplido cada uno de los objetivos específicos planteados en este Trabajo de Título. Se logró implementar en tiempo real el algoritmo sub-óptimo de estimación y predicción del SOC de baterías de Ion-Litio en una aplicación Android y mediante las buenas prácticas de programación se optimizó el tiempo de procesamiento del algoritmo. Con la aplicación en marcha se definieron los tiempos y frecuencias de estimación y predicción así como la frecuencia de recepción de datos.

Trabajo Futuro

La idea de optimizar recursos para el uso de la aplicación conlleva un menor uso de batería del dispositivo utilizado. A través del análisis realizado en el Capítulo IV se observa que la etapa de pronóstico consumía la mayor parte de los recursos. Es ahí donde se podría estudiar dos opciones: disminuir cantidad de realizaciones CM; utilizar enfoque de corriente promedio. Esto disminuiría el nivel de procesamiento y mejoraría la performance del algoritmo y la aplicación. Sin embargo, puede que disminuya la precisión de la predicción, por lo cual se requiere un estudio de diversos casos. Además, también se observa que el tiempo de procesamiento en esta etapa es alto y evita la recepción de datos por un cierto tiempo, por lo que se plantea la opción de la creación de un servicio que se ejecute en segundo plano para poder seguir recibiendo datos y continuar estimando.

Por otro lado, el algoritmo utiliza un modelo de descarga del banco de baterías de Ion-Litio. El proyecto del CIL tiene pensado comercializar bancos de baterías que pueden ser montados uno sobre otro para poder entregar mayor autonomía a la bicicleta eléctrica. Por lo tanto, se propone establecer un protocolo de comunicación entre el controlador de la bicicleta y la aplicación que le indique el número de bancos conectados. Con esta información es posible tomar la decisión de usar distintos parámetros para modelar el nuevo banco de baterías conectado y realizar el proceso de estimación y pronóstico con buenos resultados.

No solo se le debe indicar la cantidad de bancos de batería. Otro desafío es modificar el algoritmo para responder frente a cargas intermedias. Es decir, cuando el usuario carga su batería mientras el algoritmo está corriendo y no se conoce si se realizó una carga parcial o completa. Una opción es almacenar los valores estimados en el momento de la carga y utilizarlos como condición inicial a comenzar la descarga del banco de baterías nuevamente.

Bibliografía

[1] Cristóbal Inostroza, “Estimación y Pronóstico en línea del Estado de Carga de Baterías Ion-Litio basado en Filtro de Partículas e Implementado en Bicicletas Eléctricas”, Trabajo de Título, Universidad de Chile 2014.

[2] Matías Cerda “Estimación en línea del tiempo te descarga de Baterías de Ion-Litio utilizando Caracterización del Perfil de Utilización y Métodos Secuenciales de Monte Carlo”, tesis de magister, Universidad de Chile 2012.

[3] Android (2014, 19 de diciembre). Android Developers en developer.android.com.

[4] Panasonic.com. "Rechargeable Batteries: Lithium Ion". En http://na.industrial.panasonic.com/products/batteries/rechargeable-batteries/lithium-ion.

[5] Bingjun Xiao, Yiyu Shi, Lei He, "A Universal State-of-Charge algorithm for Batteries", *47th IEEE Design Automation Conference (DAC'10)*, Anaheim, CA, June 13-18, 2010.

[6] Ranjbar, A.H.; Banaei, A.; Khoobroo, A.; Fahimi, B. Online estimation of state of charge in Ion-Litio batteries using impulse response concept. IEEE Trans. Smart Grid 2012, 3, 360–367.

[7] Pattipati, B., Sankavaram, C., Pattipati, K., "System Identification and Estimation Framework for Pivotal Automotive Battery Management System Characteristics," Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on, vol.41, no.6, pp.869-884, Nov. 2011.

[8] Orchard, M., Vachtsevanos, G., "A Particle Filtering Approach for On-Line Fault Diagnosis and Failure Prognosis," Transactions of the Institute of Measurement and Control, vol. 31, no. 3-4, pp. 221-246, June 2009.

[9] Kalman, R. E. "A new approach to linear filtering and prediction problems," Journal of Basic Engineering 82(1): 35-45 1960.

[10] Oracle, “Using JConsole”. En Java Documentation, Chapter 3 http://docs.oracle.com/javase/8/docs/technotes/guides/management/jconsole.html

[11] [El organigrama de ciclo de una actividad]. Recuperado de http://yevbes.es/ciclo-de-vida-de-aplicacion-android/

[12] Librería POI

[13] Hugo F. Navarrete Echeverría, “Caracterización estadística del perfil de uso de baterías para el pronóstico del Estado-de-Carga”, Trabajo de Título, Universidad de Chile 2014.

[14] MIT Electric Vehicle Team, “A Guide to Understanding Battery Specifications”, December 2008.