ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Desarrollo de un software de telemetría para el control de una moto de carreras

Autor: Pedro José Conesa Sánchez

Director: José María Molina García-Pardo

Septiembre 2014

Autor	Pedro José Conesa Sánchez
E-Mail Autor	pjcs90@gmail.com
Director	José María Molina García-Pardo
E-Mail Director	josemaria.molina@upct.es
Título del PFC	Desarrollo de un software de telemetría para el control de una moto de carreras.

Resumen:

MotoStudent es una competición promovida por la fundación *Moto Engineering Fundation* (MEF) entre universidades españolas y europeas. Consiste en diseñar y desarrollar un prototipo de moto de competición de 125 centímetros cúbicos y dos tiempos, y superar unas pruebas de evaluación que se llevaran a cabo en las instalaciones de la Ciudad del Motor de Aragón.

Para el mejor control de la moto, y también su diseño, se hace necesario medir los diferentes parámetros de la moto, ya sea la temperatura del motor, su posición, velocidad, el eje de inclinación en una curva como la aceleración, etc. Por ello se ha diseñado este software que recoge todos los parámetros que nos envían los sensores para poder mostrarlos a tiempo real y así optimizar la configuración y las prestaciones de la moto de competición.

Este proyecto se enmarca dentro de un trabajo multidisciplinar donde se recogen los datos de la telemetría en una placa en la moto, se transmiten vía radio, y se muestran con el trabajo desarrollado en este proyecto.

Por otro lado, todos estos datos se almacenarán en una base de datos para su posterior consulta y para ver los cambios introducidos por el equipo en la moto para comprobar que el rendimiento ha mejorado realmente.

Otro de los parámetros a medir será la posición GPS, la cual se ilustrará en tiempo real mediante la creación de mapas *KML*, los cuales mostrarán la posición real de la moto en la plataforma *GoogleEarth*.

Titulación	Ingeniero de Telecomunicación
Departamento	Tecnologías de la Información y las Comunicaciones
Fecha de Presentación	Septiembre - 2014

AGRADECIMIENTOS

Gracias a mis padres, Pedro y Faustina, a mis hermanas Carmen María y Susana y a mi pareja y amiga Oriali, por creer todos ellos en mí y brindarme todo su apoyo en todo momento a lo largo de mi carrera y en general, en mi vida.

GRACIAS.

ÍNDICE

FIGURAS

CAPÍTULO 1. "INTRODUCCIÓN"

1.1. MOTIVACIÓN

MotoStudent [1] es una competición promovida por la fundación *Moto Engineering Fundation* (MEF) entre universidades españolas y europeas. Consiste en diseñar y desarrollar un prototipo de moto de competición de 125 centímetros cúbicos y dos tiempos, y superar unas pruebas de evaluación que se llevaran a cabo en las instalaciones de la Ciudad del Motor de Aragón.

Para el mejor control de la moto, y también su diseño, se hace necesario medir los diferentes parámetros de la moto, ya sea la temperatura del motor, su posición, velocidad, el eje de inclinación en una curva como la aceleración, etc. Por ello se ha diseñado este software que recoge todos los parámetros que nos envían los sensores para poder mostrarlos a tiempo real y así optimizar la configuración y las prestaciones de la moto de competición.

Este proyecto se enmarca dentro de un trabajo multidisciplinar donde se recogen los datos de la telemetría en una placa en la moto, se transmiten vía radio, y se muestran con el trabajo desarrollado en este proyecto.

Por otro lado, todos estos datos se almacenarán en una base de datos para su posterior consulta y para ver los cambios introducidos por el equipo en la moto para comprobar que el rendimiento ha mejorado realmente.

Otro de los parámetros a medir será la posición GPS, la cual se ilustrará en tiempo real mediante la creación de mapas *KML* [2], los cuales mostrarán la posición real de la moto en la plataforma *GoogleEarth* [3][4].

1.2. OBJETIVOS

1.2.1. OBJETIVO GENERAL

El objetivo general consiste en obtener el mayor rendimiento de la moto, así como mejorar sus prestaciones prueba tras prueba, y para conseguirlo es necesario recopilar la mayor cantidad posible de información en tiempo real.

Para ello se diseña un sistema software que permita el almacenamiento y visualización de los datos obtenidos, de una manera rápida, eficiente y que sea sencilla de manipular e interpretar.

El sistema se basa en un servidor independiente, que podrá conectarse a la red de datos o a una estación base, y que el mismo permitirá almacenar y visualizar los datos recibidos en el servidor.

1.2.2. OBJETIVOS ESPECÍFICOS

Los objetivos específicos son:

- Crear un servidor que permita que la comunicación entre el equipo transmisor de la moto y el servidor sea de manera rápida y efectiva.
- Almacenar los datos recibidos de manera correcta y eficiente, permitiendo el manejo de los mismos a posteriori.
- Representar los datos obtenidos en tiempo real, de la manera más clara posible para su comprensión.

1.3. SOFTWARE DE TELEMETRÍA

El software de telemetría se trata de una aplicación que actúa como servidor y permite recibir toda la información que se desee enviar desde los sensores de la moto.

Para mejorar la funcionalidad de la aplicación, se utiliza un sistema de almacenamiento de la información en forma de base de datos, el cual permite que se almacene toda la información ordenada y de manera rápida, y que la misma esté disponible para cualquier otra aplicación o usuario que lo desee.

Además, se interactúa con programas para visualizar la posición geográfica de la moto y que de este modo sea fácil saber en cualquier instante donde se encuentra.

Los datos recibidos por la aplicación se muestran inmediatamente de manera gráfica, de manera que se pueden observar el estado actual de cada sensor y la evolución que ha llevado el mismo.

Entre otras funcionalidades más, permite la exportación de los datos obtenidos a formatos típicos como pueden ser las Hojas de Cálculo o un fichero de texto plano.

1.4. ESTRUCTURA DE LA MEMORIA

La memoria está distribuida de la siguiente manera:

- Una introducción sobre el software de telemetría
- Un segundo capítulo sobre el estado del arte de la telemetría que consta de 4 partes:
  - En la primera parte se describe la telemetría de forma general.
  - En la segunda parte se relata el origen y la evolución de la telemetría.
  - En la tercera parte describimos algunas de las aplicaciones más populares de la telemetría en la actualidad.
  - En la cuarta parte hablamos de las líneas futuras de la telemetría
- En el tercer capítulo trata los diferentes entornos de trabajo que se han utilizado para desarrollar el software de telemetría.
- En el siguiente capítulo, se explica con más detalle las distintas características de la aplicación y sus funcionalidades.
- En los últimos capitulos, se hace una pequena evaluacion del proyecto y de los resultados obtenidos, así como unas líneas futuras del mismo.
- Por último se incluye en el anexo un manual de uso de la aplicación.

CAPÍTULO 2. "ESTADO DEL ARTE DE TELEMETRÍA"

2.1. INTRODUCCIÓN A LA TELEMETRÍA

La telemetría incluye un conjunto de procedimientos para medir magnitudes físicas y químicas desde una posición distante al lugar donde se producen los fenómenos que queremos analizar y además, abarca el posterior envío de la información hacia el operador del sistema [5]. El término telemetría procede de los términos griegos "*tele*" que significa remoto, y "*metron*", que significa medida. Aunque el término telemetría se suele aplicar para sistemas remotos sin cables, en algunas bibliografías también se puede encontrar para definir sistemas de transmisión cableados.

Un sistema de telemetría normalmente está constituido por un transductor como dispositivo de entrada, un medio de transmisión en forma de líneas de cable u ondas de radio, dispositivos de procesamiento de señales, y dispositivos de grabación o visualización de datos.

El dispositivo de entrada se puede distinguir como conjunto de dos partes fundamentales: el sensor, que es el elemento sensible primario que responde a las variaciones de estado de las magnitudes físicas de estudio, y el transductor, que es el que se encarga de convertir el valor de temperatura, presión o vibraciones en la señal eléctrica correspondiente, una vez detectada la variable a seguir, como puede ser la velocidad, las revoluciones del motor, la posición de la moto o el estado de las suspensiones.

El medio de transmisión puede establecerse de forma guiada por medios como redes de telefonía clásica, redes de ordenadores o enlaces de fibra óptica, o de forma no guiada, por ondas de radio, comunicación por *bluetooth* o *wifi* o incluso por redes de telefonía móvil, que será la más adecuada para nuestra aplicación.

El dispositivo de procesamiento de la señal está compuesto por un servidor remoto encargado de analizar y transformar los datos, según sea conveniente, para almacenar toda la información en una base de datos interna del propio ordenador.

Para la visualización de los datos utilizaremos una herramienta software capaz de mostrar automáticamente los valores recogidos en las gráficas pertinentes. Y además, será capaz de representar la posición GPS de la moto en tiempo real.

2.2. ORÍGENES Y EVOLUCIÓN DE LA TELEMETRÍA

Aunque el origen de la telemetría no está claro, el primer circuito conocido de transmisión de datos se creó en 1915, durante la I Guerra Mundial, desarrollada por el alemán Khris Osterhein y el italiano Franchesco Di Buonanno para medir la distancia hasta los objetivos de artillería. Otras fuentes aseguran que el origen de la telemetría es algo anterior, situando el primer sistema de telemetría en la revolución industrial [6].

Los sistemas impulsados a vapor representan las raíces de la telemetría industrial, pues se consideran los primeros procesos industriales controlados de forma fiable y determinista. En concreto, los avances introducidos por James Watt en el motor de vapor de Thomas Newcomen's incluyen varios aparatos de monitorización y control, como el indicador de presión de mercurio o su famoso "flyball governor" (Figura 2.1.), conocido actualmente como el regulador de Watt. Estos dispositivos monitorizaban y controlaban el proceso interno de un motor de vapor a distancia. La distancia no resultaba ser muy grande pero el lugar donde se realizaba la medición era bastante inaccesible.

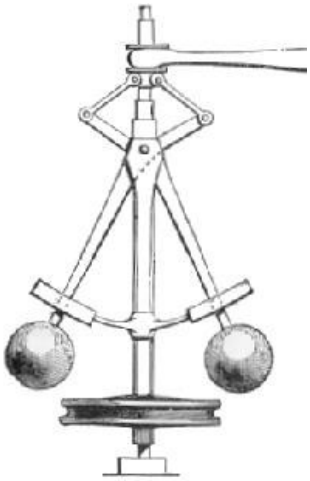


Figura 2.1. Regulador Watt [A]

Otra de las primeras aplicaciones de monitorización y control remoto fue el desarrollo de un sistema de seguridad ferroviaria (Figura 2.2.), patentado en 1872 por William Robinson [7]. Consistía en un circuito eléctrico formado por las vías férreas del tren. Éstas creaban una diferencia de potencial, y al entrar un tren en el circuito, las ruedas metálicas cortocircuitan los carriles y así se detectaba que esa sección estaba ocupada. Este aviso era "telemetrado" a varias millas de distancia por una señal de luz que indicaba al ingeniero del tren que se aproximaba que debía esperar hasta que la zona quedase libre.

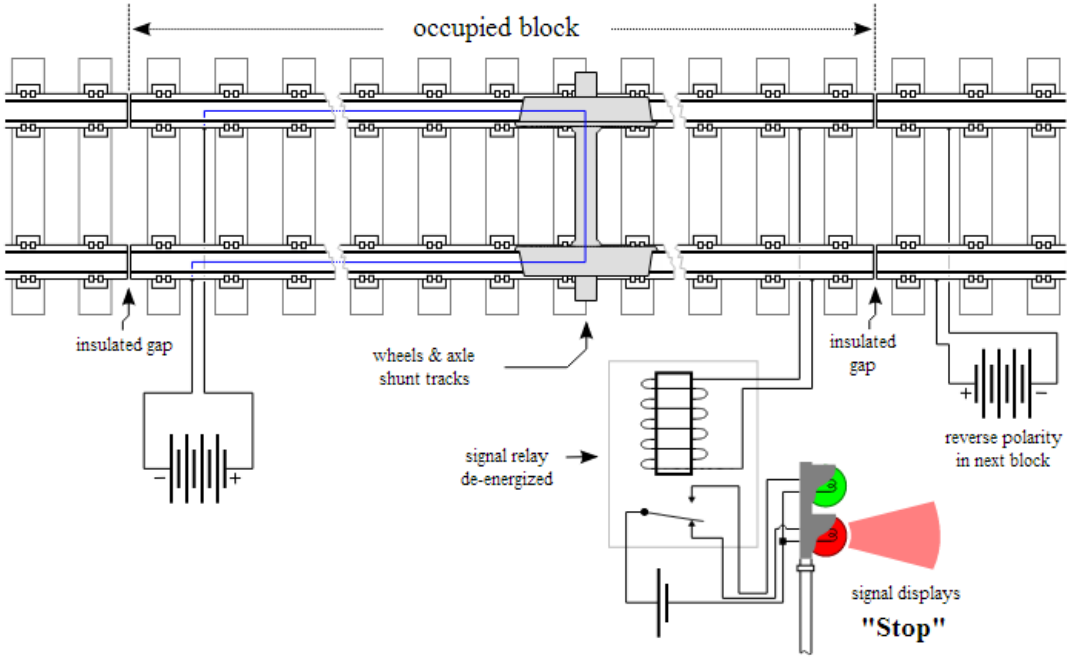


Figura 2.2. Circuito de vía [B]

El control y monitorización de los trenes continuó su desarrollo a un ritmo constante en el siglo XX. Los circuitos de las vías sigue siendo la técnica de indicación de ocupación más predominante, pero en esta ocasión el estado llega a ser comunicado a cientos e incluso miles de millas de distancia para controlar el proceso de movimiento de los trenes a tiempo real.

Con la invención de la radiosonda en 1930 [8], se comenzó a observar el tiempo de forma mucho más efectiva, llegando a monitorizar la temperatura, la humedad y la presión de la atmósfera superior. Este instrumento de medida podía alcanzar altitudes de 50000 pies por lo que rápidamente sustituyó a todas las sondas presentes utilizadas.

A pesar de todos estos avances, esta tecnología de telecontrol no sufrió un cambio significativo hasta los años 40. Durante la Segunda Guerra Mundial se idearon nuevas soluciones para establecer una comunicación ante la necesidad de dirigir señales de aviso.

La historia de la telemetría industrial abarca cerca de 200 años, eventos como la Revolución Industrial, las dos Guerras Mundiales y el desarrollo de los ordenadores y la informática. En la actualidad, la telemetría abarca multitud de ámbitos industriales como el aeroespacial, control de plantas químicas, exploración científica de naves no tripuladas (submarinos, aviones de reconocimiento y satélites), ámbitos médicos, y el mundo de la competición entre los que destacan categorías como Moto GP y Fórmula1.

En la Fórmula 1, los primeros sistemas de telemetría llegaron en los años noventa de la mano de las escuderías Williams y McLaren. Supuso un gran avance tecnológico y todos los equipos que no la usaron se quedaron atrás en la competición.

Gracias a esta tecnología, los ingenieros recibían en tiempo real información sobre los monoplazas que rodaban por la pista (tiempo por vuelta, revoluciones del motor, presión del aceite, velocidad del viento, constantes vitales del piloto, etc.) e incluso podían modificar parámetros del coche desde el propio muro de boxes. La mejora de los sistemas fue tan grande que a partir del año 2003 la Federación Internacional de Automovilismo (FIA) prohibió que los parámetros del monoplaza fuesen manipulados desde los garajes, y ahora sólo el piloto es el que puede hacerlo desde su volante.

2.3. APLICACIONES ACTUALES DE LA TELEMETRÍA

A continuación se presentan algunos de los numerosos campos en los que se aplica esta tecnología en la actualidad:

COMPETICIÓN. FÓRMULA1 Y MOTOGP

En el caso de Fórmula1 [9] los sistemas de telemetría son los sistemas auxiliares más importantes de los que se dispone. Los sistemas que utilizan se basan en ondas microondas en la banda UHF (300MHz-300GHz) y en conexiones punto a punto coche-portátil (PC). En las transmisiones inalámbricas la propagación ha de ser por línea de vista, es decir, que no haya ningún obstáculo sólido entre las antenas, porque las ondas utilizadas no son capaces de superarlos. Por ello se trabaja con envío de información a corta distancia mediante el uso de distintas antenas, aunque cuando el coche pasa lejos de los boxes puede haber pérdida de información. Podrían usarse también ondas de radio, que serían más rápidas, pero también menos fiables y con un menor ancho de banda (y por lo tanto, no podría transmitirse tanta información). Para poder enviar información a corta distancia, a lo largo de todos y cada uno de los circuitos del Mundial existen una serie de antenas repetidoras a las que llegan los datos desde los monoplazas.

Cada monoplaza lleva incorporada una pequeña (y aerodinámica) antena situada en el morro y a más de 10cm de altura, para evitar que la curvatura de la tierra sea un obstáculo más. Es omnidireccional, trabaja a una frecuencia de entre 1,45 y 1,65 GHz, tiene una ganancia de aproximadamente +3 dBi y una potencia de 160W. En la parte trasera del coche también se incorpora una segunda antena unidireccional.

Figura 2.3. Antena de telemetría de un monoplaza [C]

Esta antena base va conectada a una unidad emisora/receptora CBR-610 que actúa como modem y des/encripta la señal con los datos codificados. Cuenta con una tasa de transferencia con picos de hasta 100Mbps. Esta unidad prepara la información registrada por los sensores de coche de tal forma que pueda gestionarse mediante el potente software 'Atlas', que permite la lectura de los datos mediante complejas gráficas.

Figura 2.4. Centralita de datos [C]

Desde la propia "centralita de datos" también se envía la información directamente a la fábrica de la escudería vía satélite, usando antenas parabólicas trabajando en la banda SHF.

Por otro lado, el elemento clave sin el cual no sería posible la telemetría en la Fórmula1 es la ECU (Electronic Control Unit). Podríamos decir que es la CPU del monoplaza, que se encarga de recoger todos los datos de los sensores. Es estándar y obligatoria para los 24 coches de la parrilla y está fabricada por la escudería McLaren en colaboración con Microsoft.

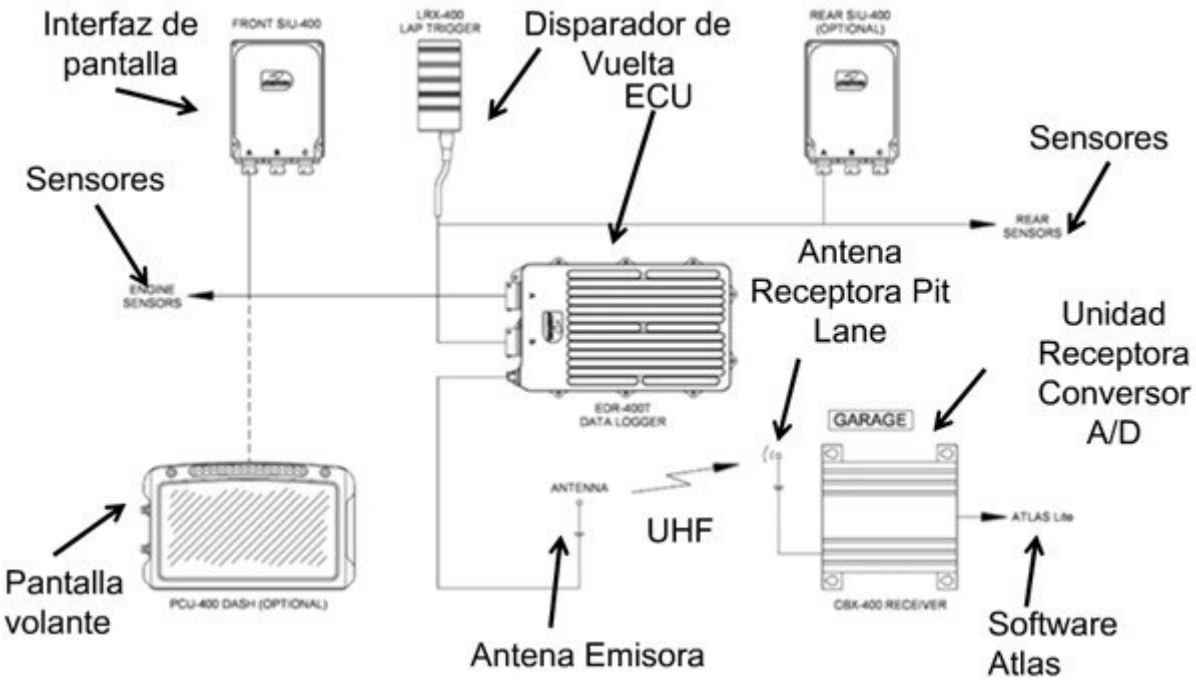


Figura 2.5. Esquema monoplaza [C]

La ECU está basada en la arquitectura Power-PC, cuenta con dos procesadores de 40MHz, 1GB de memoria estática, 1MB de memoria flash ROM y 1MB de memoria SRAM. Su tasa máxima de transmisión de datos es de 230Kbps. Los ingenieros usan un cable Ethernet o RS-232 para conectarla con un ordenador portátil y configurarla adecuadamente (aunque está bastante limitada por la normativa de la FIA).

En la competición MotoGP [10] se lleva a cabo un sistema de telemetría totalmente distinto. Por reglamento, en MotoGP la telemetría es offline, es decir, se recaba la información mientras el piloto está en pista, y luego, mediante el software propio de cada centralita, se descarga en el ordenador desde el que se trabajará .A diferencia de la Fórmula1 que se trabaja en tiempo real, en MotoGP este sistema está prohibido.

En esta competición cada moto descarga toda la información después de cada expedición en pista. La información se transmite a través de canales, por ejemplo la Ducati GP11 puede llegar a tener más de doscientos. Una vez registrada la información, los gráficos resultantes de la misma ayudarán al piloto, junto al analista de datos y al ingeniero de pista, a conocer la situación de su moto y de su propio pilotaje, en cada curva del circuito. Asimismo, podrá averiguar la marcha con la que negocia cada ángulo o la presión que ejerce sobre los frenos en todo momento, para conocer sus límites, y la mejor estrategia para sobrepasarlos sin exponerse a un accidente.

A continuación, podemos ver un gráfico que contempla variables como las revoluciones por minuto, las marchas, las suspensiones. Es un gráfico real de la telemetría del piloto MAPFRE Aspar, Héctor Barberá, en el circuito australiano de Phillip Island.

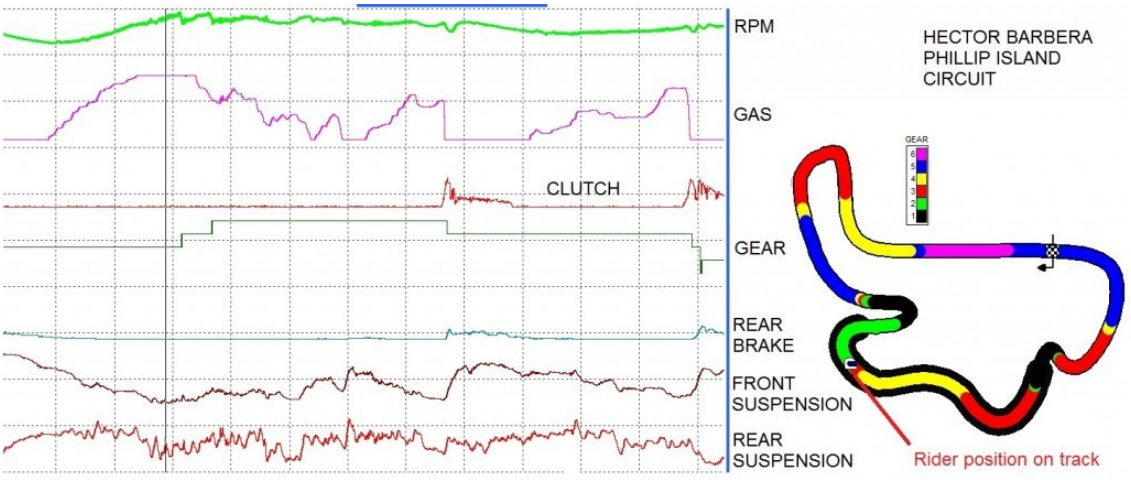


Figura 2.6. Ejemplo telemetría MotoGP [D]

DOMÓTICA

A mediados de los años 90, los automatismos destinados a edificios de oficinas, junto con otros específicos, se comenzaron a aplicar a las viviendas particulares y otro tipo de edificios, dando origen a la vivienda domótica [11].

La vivienda domótica es un hogar capaz de, ante diferentes eventos, tomar las decisiones óptimas. Para ello integra una serie de mecanismos en materia de electricidad, electrónica, robótica, informática y telecomunicaciones, con el objetivo de simplificar algunas labores, dotar de mayor confort y seguridad, y aumentar el ahorro energético.

Con la integración de la telemetría en el campo de la domótica, no sólo es posible manipular un equipo de la casa in situ, sino monitorizar y controlar su estado de forma remota en tiempo real. Esta técnica se puede implementar mediante pasarelas residenciales, a través de las cuáles se puede acceder a la red interna de la vivienda, tan sólo disponiendo de un servidor web con conexión a Internet, entre otras herramientas. Esto permite a propietarios controlar y gestionar los componentes de su vivienda desde cualquier lugar.

SENSORES

Instrumentos utilizados para conocer el estado de las variables a controlar en la vivienda como puede ser la iluminación, la temperatura, la presencia de intrusos, la detección de lluvia o escapes de gas, etc. Con ello se proporciona al sistema de control la información correspondiente sobre su estado.

Mostramos algunos ejemplos a continuación:

- Sondas de temperatura para gestión de calefacción, necesarias para controlar de forma correcta distintos tipos de calefacción eléctrica (por ejemplo, sondas limitadoras para suelo radiante).

- Sonda de humedad, destinada a detectar posibles escapes de agua en cocinas, aseos, etc.
- Detector de fugas de gas, para la detección de posibles fugas de gas en cocina, etc.
- Detector de humo y/o fuego, para la detección de conatos de incendio.
- Detector de radiofrecuencia (RF) para detectar avisos de alerta médica emitidos por un emisor portátil de radiofrecuencia (de idéntico parecido a los mandos para apertura de puertas de garaje).
- Sensor de presencia, para detección de intrusiones no deseadas en la vivienda.

ACTUADORES

Dispositivos que contienen una serie de salidas a través de las cuales pueden controlar y manejar el estado de luces, alarmas, persianas, electroválvulas, etc, y una serie de entradas a través de las que recibe las órdenes a ejecutar. Son obtenidas mediante sensores y transmisores como por ejemplo, pulsadores.

PASARELAS RESIDENCIALES

Enlace que establece la comunicación entre las redes interiores de la vivienda domótica y la red exterior por la que se accede a la misma (Figura 2.7.). Para ello debe ser capaz de adaptar los protocolos y características de la red interior a la exterior. Las redes exteriores a las que se puede conectar pueden ser: RTC, RDSI, Xdsl, cable, PLC, satélite, fibra, LMDS, etc.

Figura 2.7. Ejemplo de pasarela residencial para acceso remoto [E]

ROBÓTICA

La robótica es una rama de la tecnología que estudia el diseño y construcción de máquinas capaces de desempeñar tareas realizadas por el ser humano, o que requieren del uso de inteligencia [12]. La informática, la electrónica, la mecánica y la ingeniería son sólo algunas de las disciplinas que se combinan para elaborar un sistema robótico.

Estos sistemas pueden realizar tareas como manipular materiales radioactivos, limpiar residuos tóxicos, minería, búsqueda y rescate de personas y localización de minas terrestres. Para ser posible, es necesario recibir información del entorno e interpretarla, para lo que existen técnicas como la telemetría láser. Ésta consiste en lanzar un rayo láser que determina la distancia a la que se encuentran los objetos que lo rodean, lo que le permite desplazarse e interactuar por el mismo.

MEDICINA

En medicina, la telemetría es comúnmente usada para registrar eventos electrocardiográficos a distancia. Los radiotransmisores están conectados al paciente mediante 5 electrodos adheridos a la piel; esto permite a los pacientes libertad para deambular y moverse. El ordenador central refleja los E.C.G. de los pacientes conectados a él y guarda los eventos importantes ocurridos durante las últimas 24 horas. En un estudio realizado por Pérez Titos CB y Oliver Ramos MA, del Hospital Universitario Médico-Quirúrgico «Virgen de las Nieves» de Granada [13] se obtuvo que el 80% de los pacientes estudiados registraron eventos en la telemetría y el 23%, de éstos, fueron eventos graves.

Figura 2.8. Electrocardiograma obtenido por telemetría [F]

2.4. LÍNEAS DE INVESTIGACIÓN EN LA TELEMETRÍA

A continuación se presentan diversos estudios actuales que muestras las distintas posibilidades que ofrece la telemetría de cara al futuro:

LÁSER DE ESTADO SÓLIDO DE PULSOS CORTOS PARA TELEMETRÍA

La determinación de distancias por telemetría láser es una de las aplicaciones de mayor interés y de las primeras en que se utilizara esta fuente de radiación. [14] Uno de los procedimientos más frecuentemente utilizado, requiere del diseño de una cavidad que funcione en régimen pulsado, generando pulsos cortos adecuados para su empleo en técnicas de medidas remotas, en que la medición a realizar es el tiempo transcurrido entre el envío de la señal y el registro del eco producido por un objetivo elegido como "blanco". Por la rapidez en la determinación, el grado de colimación propio del láser y la condición de incerteza fija en todo el rango de medición, es el método más difundido, cuando es tolerable la incerteza característica del mismo. Basa su funcionamiento en la detección del eco de una señal láser de muy corta duración, a partir de la determinación del tiempo transcurrido entre la emisión y la recepción de la misma (tiempo de vuelo).

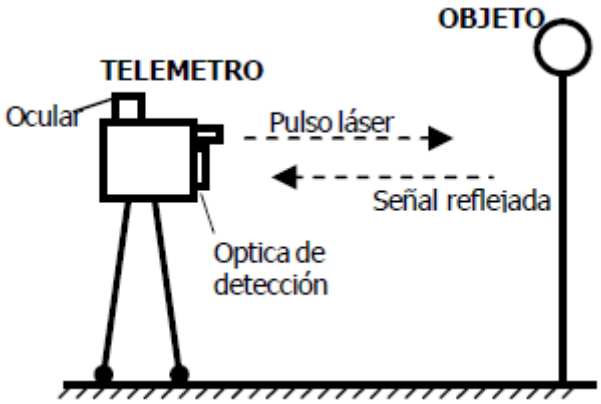


Figura 2.9. Telemetría láser [G]

El dispositivo se compone de una fuente de radiación láser de Nd:YAG bombeada por un diodo láser y un detector de alta ganancia utilizado en la etapa de recepción de la señal de eco. Se utilizaron pulsos de 0,3mJ de energía y 50ns de duración a una frecuencia de repetición de 60Hz. El sistema completo se ensayó exitosamente para establecer la distancia hacia diferentes objetos lejanos, habiendo registrando un alcance máximo de 5280m con precisión de unos pocos metros.

CONTROL DE VEHÍCULOS AÉREOS NO TRIPULADOS

En el entorno de los aviones aéreos no tripulados (UAV), una de las partes más importantes es la telemetría del avión. Gracias a ella se consigue localizar, monitorizar y extraer medidas en tiempo real relevantes para el estudio de la aerodinámica del aparato. Para que la transferencia de datos sea adecuada, hace falta tener en cuenta muchos factores como por ejemplo la calidad del envío de la señal, la banda de frecuencia en la que se trabaja, las interferencias con otros dispositivos, etc. Dependiendo del tipo de antena en la estación de control de tierra utilizada para la comunicación con el avión, puede ser importante que la orientación de dicha antena apunte al UAV con el fin de conseguir la mayor potencia de transmisión posible.

INTEGRACIÓN DE LA MONITORIZACIÓN Y CONTROL EFICIENTE EN TIEMPO REAL DE REDES DE AGUA POTABLE

La gestión de la redes de agua potable comprende dos niveles. El primer nivel consiste en la monitorización de la red, que implica la observación del sistema mediante la utilización de los sistemas de telemetría, registrando las variables más significativas del funcionamiento de la red así como la correcta operación de los sistemas de control/actuación instalados en la red. El segundo nivel consiste en el control operacional de la red que, a partir de las medidas proporcionadas por la telemetría y la previsión de la demanda, se encarga de planificar y ejecutar mediante el telecontrol las

políticas de gestión óptima que persiguen optimizar el consumo de agua y energía de la red. El proyecto Effinet persigue desarrollar una plataforma software que permita integrar la monitorización y el control tanto de la red como del comportamiento de la demanda.

Desde la perspectiva de la gestión de la red, el nivel de monitorización implica todas las tecnologías y metodologías para la observación del estado de la misma, así como la detección y aislamiento de cualquier funcionamiento anómalo de cualquiera de sus elementos (fallos en sensores/actuadores, fugas, etc.). La monitorización de redes de agua de distribución se basa frecuentemente en una sectorización de la red, es decir, en la subdivisión de la red en sectores independientes, o DMA (District Metered Areas), en los cuales se controla el caudal y la presión de entrada. En cada uno de los sectores se puede hacer un mejor seguimiento y control de la presión y de las posibles pérdidas. Las metodologías de monitorización desarrolladas en Effinet se basan en la utilización de modelos matemáticos hidráulicos, de series temporales y de detección y diagnóstico de fallos con el objetivo de detectar y localizar fugas y posibles eventos de empeoramiento de la calidad del agua.

A nivel de red, el nivel de control operacional consiste en determinar las estrategias de control admisibles para los actuadores del sistema (bombas y válvulas) de forma que optimicen el funcionamiento de la red minimizando los costes de operación. Los costes asociados a los bombeos son básicamente los de la energía eléctrica y se pueden optimizar aprovechando la variación de tarifas eléctricas entre día/noche y teniendo en cuenta cómo varían los precios en el mercado eléctrico a varios días vista. Para resolver este problema de control óptimo, en el proyecto Effinet se están desarrollando estrategias de control predictivo económico que tengan en cuenta la incertidumbre asociada a la variación de la demanda y al coste de la energía eléctrica. El control en el ámbito de la demanda se conoce como gestión de la demanda y agrupa un conjunto de medidas que pueden utilizar los gestores del agua para modificar el comportamiento de la demanda. Entre ellas, se incluyen técnicas de regulación de presiones, sistemas tarifarios y, finalmente, sistemas de información personalizada a los

usuarios para que ellos puedan optimizar sus patrones de consumo y el coste asociado. Por otra parte, el proyecto Effinet estudia la caracterización de los patrones de consumo de los clientes mediante telelectura y el envío de información personalizada a los clientes sobre sus patrones de consumo, de modo que estos puedan adaptarlo en función de posibles tarificaciones dinámicas del agua y la energía eléctrica.

METODOLOGÍAS PARA LA COLECTA DE MUESTRAS EN FAUNA SILVESTRE IN SITU

El estudio de la fauna silvestre implica el manejo de las poblaciones y su hábitat, ya sea para el aprovechamiento de las especies cinegéticas y de importancia comercial, el control de las poblaciones que causan daño a los intereses humanos, o para la conservación de especies amenazadas. Determinar los diferentes métodos para la colecta de muestras en la investigación del médico de la conservación se hace una actividad transversal a otras ciencias de forma transdisciplinaria cuando se trabaja en condiciones in situ. Gran parte de la diversidad biológica se pierde como consecuencia de las enfermedades o de problemas de tipo antrópico que afectan la fauna silvestre, la incidencia de enfermedades y de contaminantes provoca cambios en las poblaciones afectando procesos evolutivos y ecológicos que regulan la biodiversidad; debido a esto se han establecido y aplicado nuevas metodologías y técnicas de recolección de muestras con el fin de tomar datos en campo, estas actividades se están mostrando eficaces para establecer planes de conservación y así avanzar en la comprensión del conocimiento en las ciencias veterinarias según los resultados ofrecidos en la investigación científica, en este campo, la telemetría de las variables de interés juega un papel principal.

ALERTA DE DESBORDE DE RÍOS Y CONSULTA DE PARÁMETROS DE HUMEDAD Y TEMPERATURA VÍA SMS

Se trata del desarrollo de un módulo de telemetría basado en la red de telefonía móvil GSM (Global System Mobile) con el fin de alertar a una persona o grupo de personas ante el posible desborde de un río, y permitir la consulta de parámetros de temperatura y humedad relativa vía mensaje de texto ó SMS (Short Message Service). Para la medición del nivel de agua se utilizó tecnología de ultrasonido y para la medición de humedad y temperatura el sensor digital SHT-11

MEDICIÓN DE PROFUNDIDAD DE RÍOS Y LAGOS

Para tratar los problemas de sequías e inundaciones es necesario conocer bien los caudales de los ríos o la profundidad de los lagos, ya que esta labor ha sido siempre realizada de manera artesanal. A través de un sistema de telemetría por radio capaz de medir la profundidad a la que se encuentra el suelo del nivel del agua y estaciones hidrométricas para obtener los valores de caudal del rio, podemos obtener los valores reales y actualizados para el correcto funcionamiento de las centrales hidroeléctricas y controlar mejor las posibles sequías e inundaciones que pudiera haber.

Como podemos ver, existen infinidad de campos en los que la telemetría juega un papel fundamental, por lo que poco a poco, y con la evolución de la tecnología, su uso se hace más común y necesario en todos los campos de trabajo y estudios.

CAPÍTULO 3. "ENTORNO DE TRABAJO"

3.1. ESCENARIO DE APLICACIÓN

Este proyecto se enmarca en el conjunto de proyectos para el equipo de *MotoStudent*, en especial en el área de telemetría. Todos estos proyectos juntos forman un sistema completo adaptado a las necesidades del equipo de competición.

El primero de los proyectos, "*Desarrollo de un dispositivo de telemetría basado en la plataforma Arduino y Shield 3G + GPS*" [15], se encarga de la adquisición de los datos de los sensores con un microcontrolador y su envío mediante una tarjeta de telefonía, comercial o privada.

El otro proyecto "*Implementación de un sistema de radiocomunicaciones para la transmisión de la telemetría de una moto de carreras*" [16], es el encargado de establecer la comunicación entre el microcontrolador y el servidor, creando una comunicación inalámbrica punto-a-punto entre la fuente de datos y el receptor, el cual a su vez reenviará por cable los datos al servidor.

Este proyecto es la parte final del conjunto de los tres proyectos. Está encargado de la recepción de los datos, el tratamiento de los mismos, su almacenamiento y la representación de forma legible, de manera que permita obtener unos resultados para poder mejorar el rendimiento de la moto de competición.

3.1.1. DESARROLLO DEL PROYECTO

Para el desarrollo de este proyecto se han utilizado diversas herramientas de trabajo. Estas herramientas van desde la parte del diseño del software, como el almacenamiento de los datos obtenidos, así como la representación y comprobación de los mismos.

Para el diseño de la aplicación, en especial para facilitar la creación de la interfaz gráfica, se ha utilizado el entorno de desarrollo *NetBeans* [17].

En cuanto a los datos, serán almacenados sistemáticamente en una misma base de datos. Para ello utilizamos la herramienta de código abierto *MySQL* [18], en concreto la versión servidor, que nos permitirá guardar los datos con mayor facilidad.

La moto estará representada en el mapa, y es por ello que nos ayudaremos de la herramienta *GoogleEarth* para ver en tiempo real donde se encuentra localizada la moto de competición.

Por último, utilizaremos la herramienta *Navicat* [19] para conectarnos a la base de datos y así poder observar las distintas tablas que dicha base de datos posea y los datos que cada una de las tablas contenga.

3.2. NETBEANS

NetBeans es un entorno de desarrollo de software libre, en especial dedicado al lenguaje de programación Java, aunque también es posible utilizarlo para el desarrollo de otros lenguajes de programación como son C++, HTML o PHP.

Es conocido que existen otros entornos de trabajo también populares para el desarrollo de aplicaciones en Java, que es nuestro caso, pero para facilitar la creación de los distintos entornos gráficos y menús, se decide optar por la aplicación *NetBeans*, pues ésta facilita la creación de dichos menús, quitando así carga al programador para así centrarse en el funcionamiento de la aplicación.

El entorno de trabajo de *NetBeans IDE 7.4* presenta el siguiente aspecto:

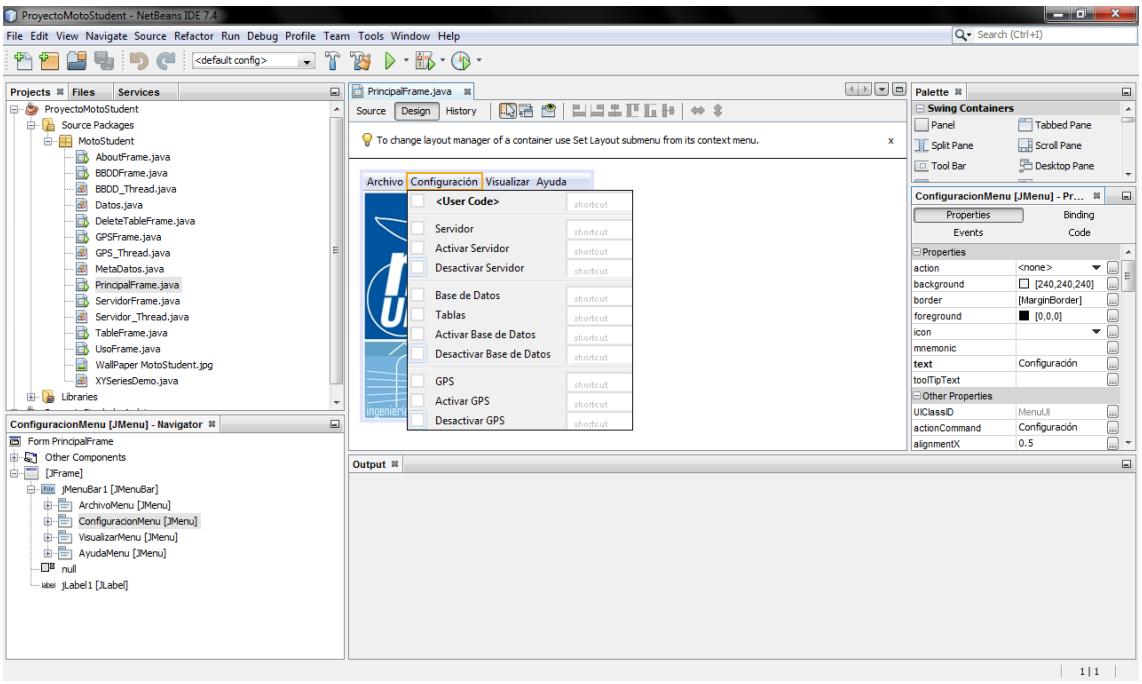


Figura 3.1. Entorno de diseño gráfico de NetBeans

Como podemos ver en la Figura 3.1., es posible crear los menús de forma gráfica, además, como se muestra en la imagen a continuación, también es posible diseñar la misma directamente en código de programación:

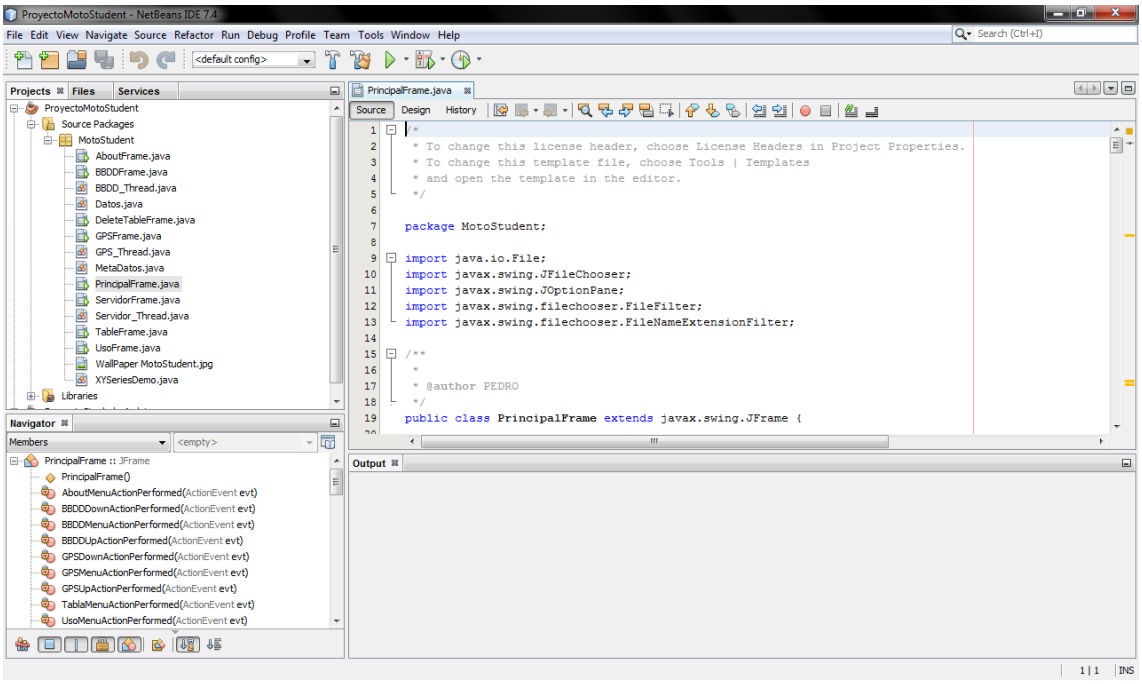


Figura 3.2. Entorno de trabajo de NetBeans

Como se puede observar en la Figura 3.2., también disponemos de una vista de todos los ficheros que forma el proyecto, permitiendo así navegar a través de él y facilitar trabajar con el mismo.

3.3. MYSQL SERVER

MySQL Server es un sistema de gestión de bases de datos relacional, multihilo y multiusuario. MySQL desarrolla el lenguaje que lleva su nombre, que a su vez es una adaptación del lenguaje para el acceso a base de datos, conocido como *SQL* (*Structured Query Language*). Las diferencias entre ambos lenguajes directivos son escasas y poseen muchas similitudes, tanto en el vocabulario como en la sintaxis.

Con este sistema, conseguimos tener una base de datos instalada en el equipo, indicando el nombre de la base de datos y a su vez un usuario y contraseña.

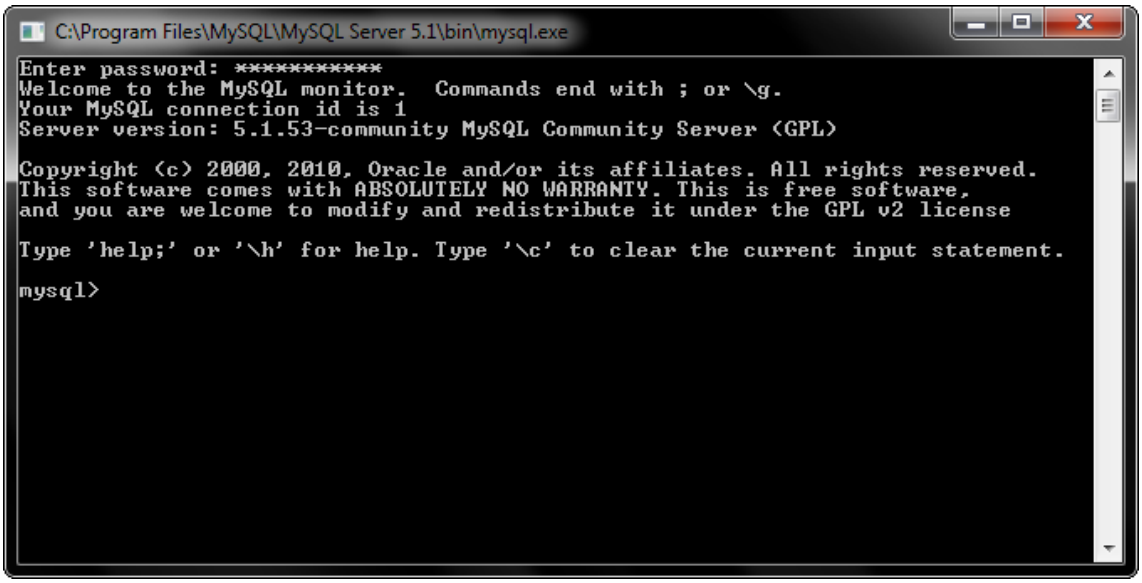


Figura 3.3. Configuración MySQL Server

Como vemos en la figura anterior, podemos gestionar la base de datos desde el ejecutable que nos abrirá la consola de comandos. Desde ella podremos crear tablas, entradas de datos en las mismas y gestionar todo lo relacionado con el servidor de base de datos mediante la ejecución de instrucciones *SQL*.

Par facilitar este trabajo, una vez ejecutada la aplicación por primera vez y creado una base de datos y un nombre de usuario y contraseña, la base de datos será gestionada por otra aplicación con un entorno gráfico más amigable como el programa *Navicat*.

Para trabajar con esta base de datos desde la aplicación final se utilizará el lenguaje estructurado *SQL*, estableciendo una conexión, realizando consultas e inserciones y borrando datos de las tablas que componen la base de datos, así como de realizar diferentes relaciones entre tablas y finalizar la conexión establecida con la base de datos.

3.4. GOOGLEEARTH

GoogleEarth es un programa informático similar a un Sistema de Información Geográfica (SIG), creado por la empresa *Keyhole Inc*., que permite visualizar imágenes en 3D del planeta, combinando imágenes de satélite, mapas y el motor de búsqueda de Google.

La interfaz de *GoogleEarth* es la de la Figura 3.4..:

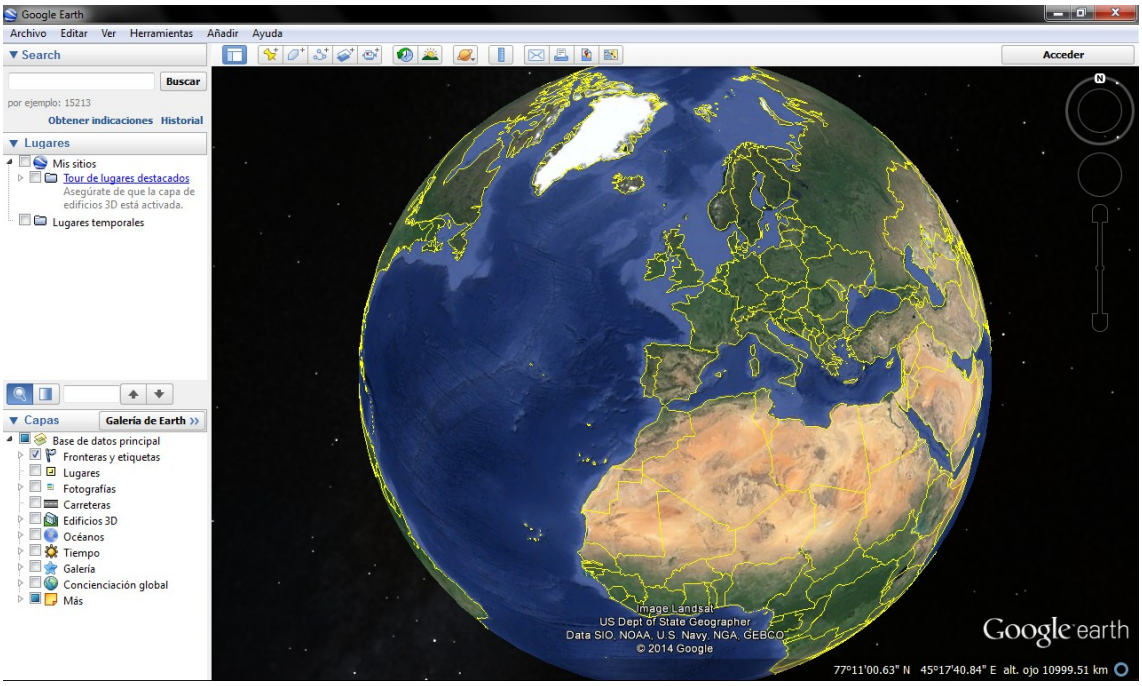


Figura 3.4. Interfaz de GoogleEarth

Muchos usuarios utilizan la aplicación para añadir sus propios datos, haciéndolos disponibles mediante varias fuentes como blogs. *GoogleEarth* es capaz de mostrar diferentes capas de imagen encima de la base y es también un cliente válido para un *Web Map Service*. *GoogleEarth* soporta datos geoespaciales tridimensionales mediante los archivos *.kml* (*Keyhole Markup Language*).

KML es un lenguaje de marcado basado en *XML* para representar datos geográficos en tres dimensiones. Un fichero *KML* especifica una característica (un lugar, una imagen o un polígono) para *GoogleEarth*. Contiene título, una descripción básica del lugar, sus coordenadas (latitud y longitud) y alguna otra información.

En nuestro caso, el documento solo cambiará la posición geográfica de la moto, en lo demás tendrá la misma estructura, como la siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
<Document>
   <Style id="randomColorIcon">
      <IconStyle>
         <scale>1.1</scale>
         <Icon>
            <href>moto_icon.png</href>
         </Icon>
      </IconStyle>
   </Style>
  <Placemark>
    <name>UPCT Moto</name>
    <styleUrl>#randomColorIcon</styleUrl>
    <description>
      Aqui esta la posicion de la moto UPCT
    </description>
    <Point>
       <!-- <coordinates>Longitud,Latitud,Altura</coordinates> -->
      <coordinates>-1.0339898,37.6432535,0</coordinates>
    </Point>
  </Placemark>
</Document>
</kml>
```

Ejecutando el código el fichero *.kml* que forma el código anterior veremos cómo se sitúa el icono de la moto en las coordenadas indicadas por el documento.

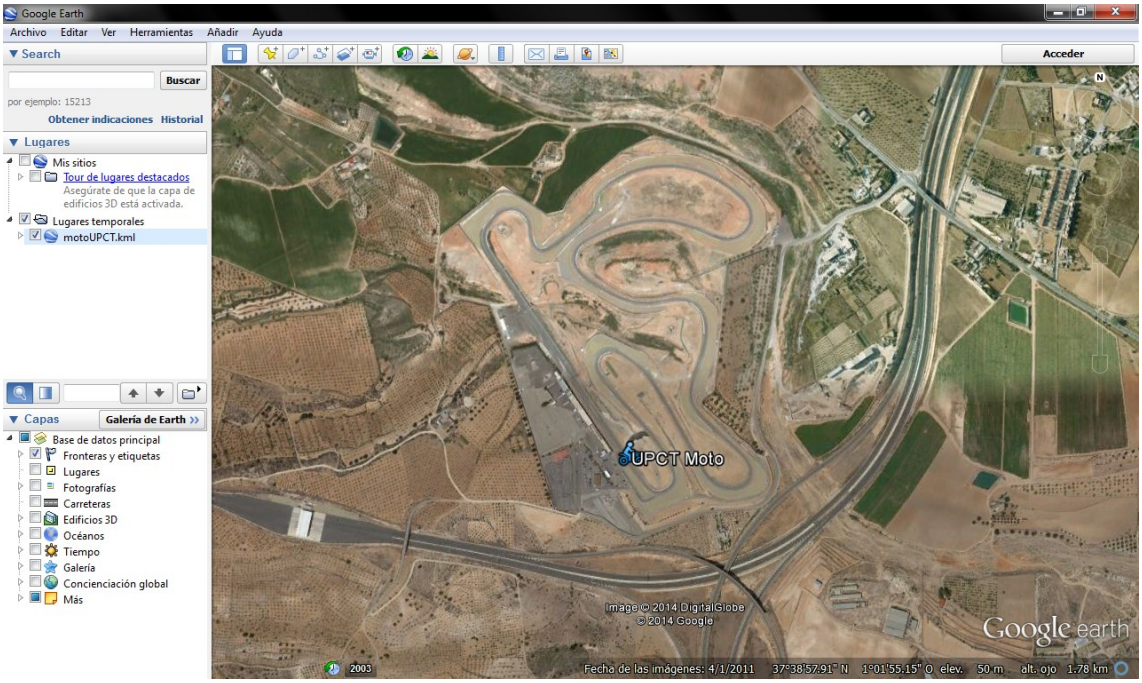


Figura 3.5. Posición de ejemplo KML

3.5. NAVICAT

Navicat es un administrador gráfico de base de datos y un software de desarrollo producido por *PremiumSoft CyberTech Ltd*. para *MySQL*, *MariaDB*, *Oracle*, *SQLite*, *PostgreSQL* y *Microsoft SQL Server*.

Navicat cuenta con un explorador como interfaz gráfica de usuario soportando múltiples conexiones para bases de datos locales y remotas. Su diseño está pensado para satisfacer las diferentes necesidades de un amplio sector del público, desde administradores y programadores de bases de datos a diferentes empresas que dan soporte y o comparten información con clientes o socios.

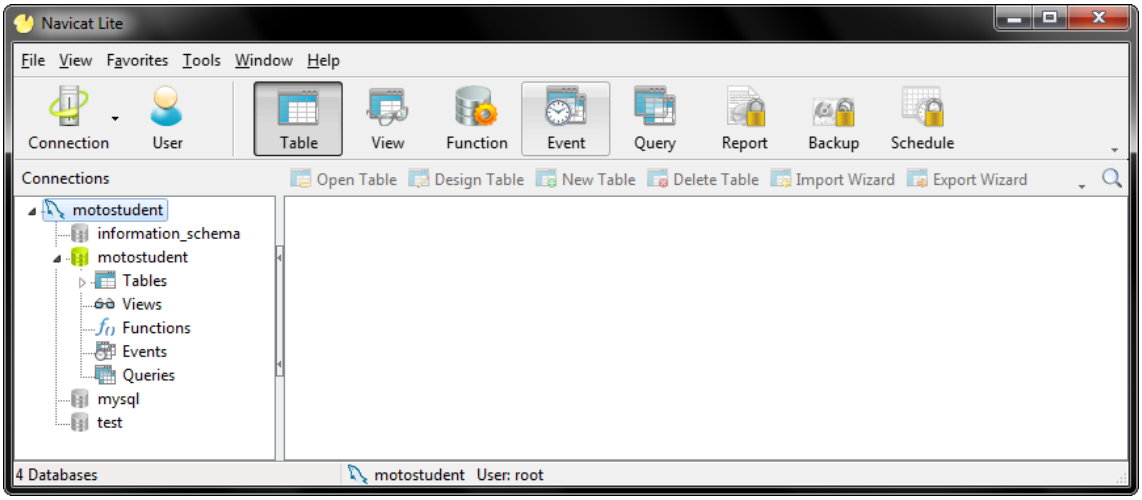
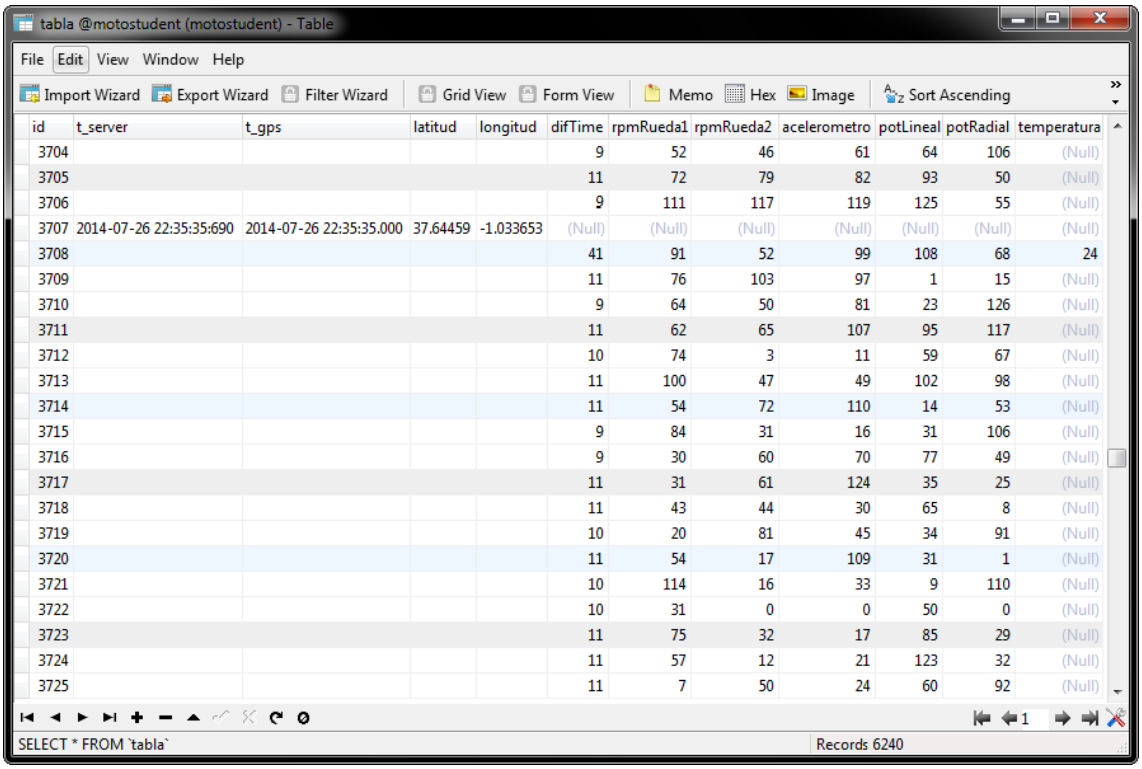


Figura 3.6. Interfaz de Navicat

Dentro de *Navicat* es fácil crear nuevas base de datos, nuevas tablas y los parámetros que incluyen, además de visualizar sus parámetros y establecer relaciones entre tablas y claves primarias. La visualización de una tabla ya rellena, a la cual podremos añadir aún más datos, posee una estructura similar a la de la Figura 3.7.



id	t_server	t_gps	latitud	longitud	difTime	rpmRueda1	rpmRueda2	acelerometro	potLineal	potRadial	temperatura
3704					9	52	46	61	64	106	(Null)
3705					11	72	79	82	93	50	(Null)
3706					9	111	117	119	125	55	(Null)
3707	2014-07-26 22:35:35:690	2014-07-26 22:35:35.000	37.64459	-1.033653	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)
3708					41	91	52	99	108	68	24
3709					11	76	103	97	1	15	(Null)
3710					9	64	50	81	23	126	(Null)
3711					11	62	65	107	95	117	(Null)
3712					10	74	3	11	59	67	(Null)
3713					11	100	47	49	102	98	(Null)
3714					11	54	72	110	14	53	(Null)
3715					9	84	31	16	31	106	(Null)
3716					9	30	60	70	77	49	(Null)
3717					11	31	61	124	35	25	(Null)
3718					11	43	44	30	65	8	(Null)
3719					10	20	81	45	34	91	(Null)
3720					11	54	17	109	31	1	(Null)
3721					10	114	16	33	9	110	(Null)
3722					10	31	0	0	50	0	(Null)
3723					11	75	32	17	85	29	(Null)
3724					11	57	12	21	123	32	(Null)
3725					11	7	50	24	60	92	(Null)

Figura 3.7. Tabla en Navicat

La clave primaria de esta tabla sería la variable "*id*". Después podemos observar como algunas casillas están incompletas debido a la falta de valores. La tabla mostrada se obtuvo mediante una simulación. En posteriores capítulos veremos que significa cada parámetro de la tabla mostrada.

CAPÍTULO 4. "DESARROLLO DE LA APLICACIÓN"

La aplicación se compone de diversas partes que se irán explicando en los distintos apartados de este capítulo.

4.1. PROCEDIMIENTO GENERAL DE LA APLICACIÓN

En el procedimiento general de la aplicación, una vez que estén todos los parámetros configurados, por cada nueva recepción de un paquete UDP[1] el servidor lanzará un nuevo hilo de ejecución[2], donde se ejecutarán las operaciones necesarias para tratar los datos adecuadamente.

[1] **UDP** (*User Datagram Protocol*) es un protocolo del nivel de transporte basado en el intercambio de datagramas. Permite el envío de datagramas a través de la red sin que se haya establecido previamente una conexión, ya que el propio datagrama incorpora suficiente información de direccionamiento en su cabecera.

[2] **Hilo de ejecución**, hebra o subproceso es la unidad de procesamiento más pequeña que puede ser planificada por un sistema operativo. La creación de un nuevo hilo es una característica que permite a una aplicación realizar varias tareas a la vez (concurrentemente). Los distintos hilos de ejecución comparten una serie de recursos tales como el espacio de memoria, los archivos abiertos, situación de autenticación, etc. Esta técnica permite simplificar el diseño de una aplicación que debe llevar a cabo distintas funciones simultáneamente.

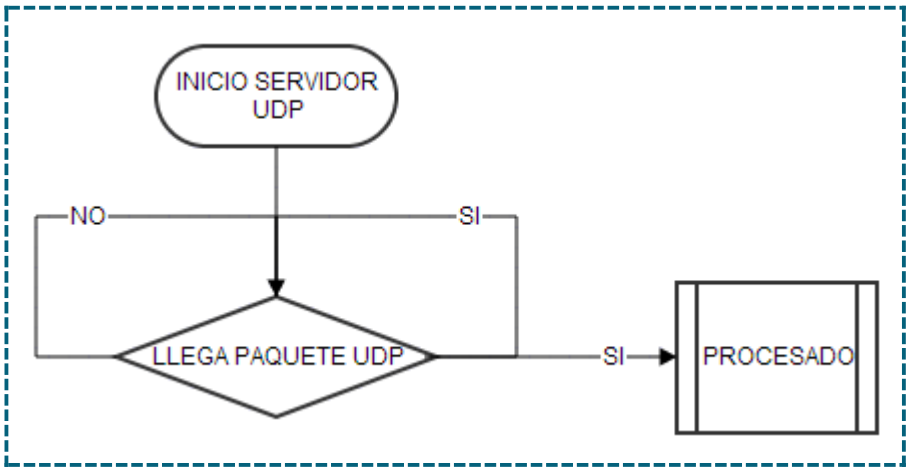


Figura 4.1. Flujograma general de la aplicación

El proceso que trata los datos que llegan al servidor cumple el siguiente esquema:

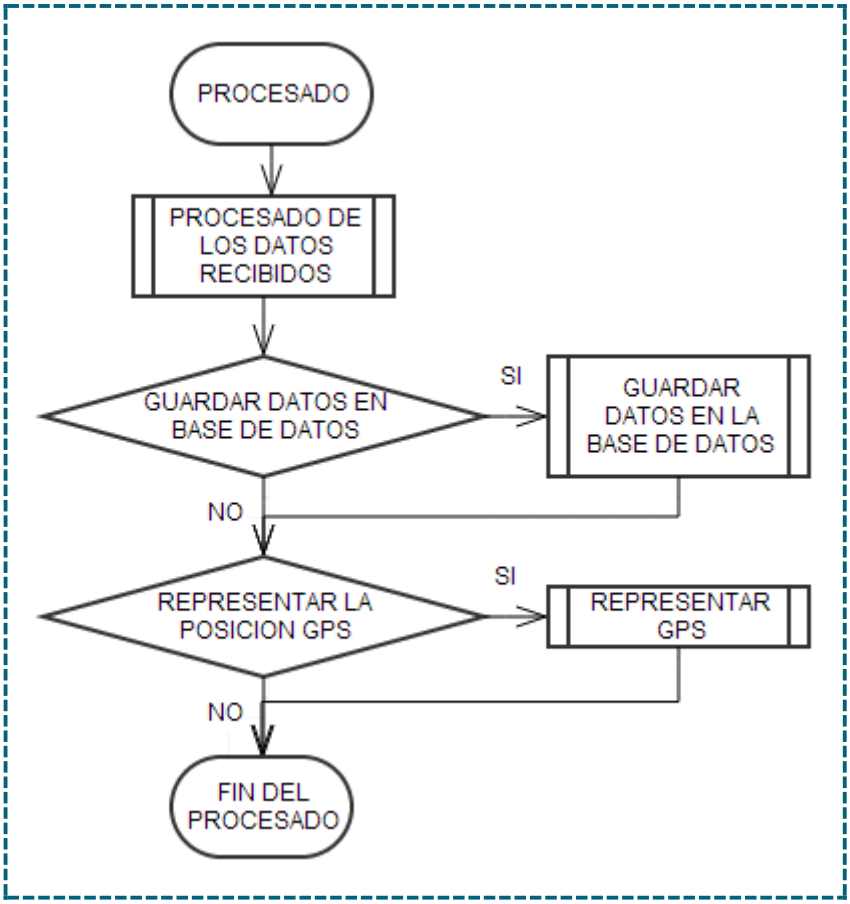


Figura 4.2. Flujograma del proceso general

Como podemos observar en el diagrama de flujo, en el hilo de ejecución que lanzamos para procesar los datos se realizan muchas operaciones:

1. Descodificamos el datagrama UDP, encapsulándola en una nueva clase llamada `Datos`, la cual encapsula todos los datos de los sensores una vez descodificados y ordenados de manera que su utilización sea sencilla y efectiva.
2. Guardar los datos en la tabla de la base de datos que se haya seleccionado, siempre y cuando hayamos indicado en la aplicación que queremos almacenar los datos.
3. Representar gráficamente los datos de los sensores que deseemos. Este paso es independiente de si los datos están almacenados en la base de datos o no.
4. Mostar la posición de la moto de competición en el mapa de *GoogleEarth*, siempre y cuando hayamos decidido que deseamos esta opción.

Con todo ello, el procesado de un solo paquete de datos UDP termina, independientemente de si mientras se realizaba este proceso llegaba otro paquete de datos o no ha llegado ninguno, ya que como se ha mencionado antes, por cada paquete de datos UDP que llega, lanza un nuevo hilo de procesado.

El código del procesado es el siguiente:

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

package MotoStudent;

/**
 *
 * @author PEDRO
 */
public class Proceso_Thread extends Thread{

    public void run(String str){
        // Desfragmentamos los datos que llegan al servidor de forma
        // que los tengamos divididos en los diferentes valores

        // Variables para debug de rendimiento
        long start_time = 0;
        long end_time = 0;
        double difference = 0;
```

```
        if(MetaDatos.debugTime)
            start_time = System.nanoTime();

        Graph_Thread gT = new Graph_Thread(); // Thread para mostrar las gráficas
        Datos d = new Datos(str);

        // Añadimos los datos a la tabla de la BBDD (?)
        if(MetaDatos.bbdd_status){
            BBDD_Thread bbdd = new BBDD_Thread(); // Guardamos los datos en la BBDD
            bbdd.run(d);
        }

        if(MetaDatos.begin){
            for(int i = 0; i < MetaDatos.gNum; i++){
                MetaDatos.graficas[i] =
MetaDatos.MySQLtoGraph(i+5,MetaDatos.gnombres[i], MetaDatos.titleAxisY[i]);
                MetaDatos.graficas[i].setFocusableWindowState(false);
                // Creamos las gráficas y la hacemos que no tenga el foco
            }
            MetaDatos.begin = false;
        }

        gT.run(d); // Actualizamos las gráficas


        // Mostramos la moto en el mapa (?)
        if(MetaDatos.gps_status){
            if(!d.getError()[1] && !d.getError()[2]){
                GPS_Thread gps = new GPS_Thread(); // Posicionamos en GoogleEarth
                gps.run(d);
            }
        }

        for(int i = 0; i < MetaDatos.gNum; i++){
            MetaDatos.graficas[i].setVisible( MetaDatos.gstatus[i] );
        }

        if(MetaDatos.debugTime){
            end_time = System.nanoTime();
            difference = (end_time - start_time)/1e6;
            System.out.println("\n==============\nT_PROCESO:\t" + difference +
"ms.\n==============");
        }
    }
}
```

4.2. EL SERVIDOR UDP

Utilizamos un servidor UDP ya que nos ofrece un servicio de envío de paquetes más rápido que el que proporciona el protocolo TCP[3], además de permitirnos mayor cantidad de conexiones que dicho protocolo, que es justo lo que buscamos en nuestra aplicación, el envío rápido y de mucha cantidad de información. No nos preocuparemos de los datagramas que se pierdan, la aplicación intentará obtener la mayor cantidad de datos que sean posibles, sin bloquear la comunicación, por ello se opta por usar el protocolo de comunicaciones UDP frente a TCP.

Al tratarse de un servidor, solo podremos configurar su número de puerto, esto se hará a través del menú `Configuración > Servidor`.

[3] **TCP** (*Transmission Control Protocol*) es un protocolo de comunicación orientado a conexión fiable del nivel de transporte. El protocolo garantiza que los datos serán entregados en su destino sin errores y en el mismo orden en que se transmitieron, proporciona un transporte fiable de flujo de bits entre aplicaciones. Está pensado para poder enviar grandes cantidades de información de forma fiable, liberando al programador de la dificultad de gestionar la fiabilidad de la conexión (retransmisiones, pérdida de paquetes, orden en el que llegan los paquetes, duplicados de paquetes...) que gestiona el propio protocolo.

Figura 4.3. Localización del menú Servidor

Nos aparecerá el siguiente menú para administrar el puerto de comunicaciones:

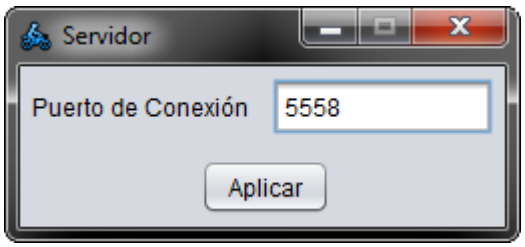


Figura 4.4. Menú Servidor

Al activar la aplicación, el servidor lanzará las posibles gráficas que ya estén activas, y si estas tienen valores guardados en la base de datos, se mostrarán. Por cada recepción se lanzará un nuevo hilo de procesado de los datos que han llegado, dejando listo el servidor para el siguiente datagrama de datos UDP.

El código del servidor es el siguiente:

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

package MotoStudent;

/**
 *
 * @author PEDRO
 */

import java.net.*;

public class Servidor_Thread extends Thread{

    public void run(){

        Proceso_Thread pT; // Thread de procesado de datos
        try{ // Intentamos abrir las gráficas si ya existen..
            for(int i = 0; i < MetaDatos.gNum; i++){
                MetaDatos.graficas[i] =
MetaDatos.MySQLtoGraph(i+5,MetaDatos.gnombres[i],MetaDatos.titleAxisY[i]);
                MetaDatos.graficas[i].setFocusableWindowState(false);
                // Creamos las gráficas y la hacemos que no tenga el foco
                MetaDatos.connect();
                MetaDatos.ejeX =
(MetaDatos.getAxisX("difTime"))[MetaDatos.getAxisX("difTime").length-1];
                MetaDatos.shutdown();
            }
            for(int i = 0; i < MetaDatos.gNum; i++){
                MetaDatos.graficas[i].setVisible( MetaDatos.gstatus[i] );
            }
            MetaDatos.begin = false;
        }catch(Exception e){
            MetaDatos.begin = false;
            if(MetaDatos.debug)
                e.printStackTrace();
        }

        byte[] receiveData = new byte[1024]; // Paquete de datos de 1024 bytes
        String str = null; // String para convertir los datos
        DatagramSocket serverSocket = null; // Socket del servidor

        try{
            serverSocket = new DatagramSocket(MetaDatos.npuerto);

        }
        catch(Exception e){
            if(MetaDatos.debug)
                e.printStackTrace();
        }

        // Variables para debug de rendimiento
        long start_time = 0;
        long end_time = 0;
        double difference = 0;


        while(MetaDatos.status){

            if(MetaDatos.begin)
                for(int i = 0; i < MetaDatos.gNum; i++){
                    MetaDatos.graficas[i].setVisible( MetaDatos.gstatus[i] );
                }
```

```
            while(MetaDatos.servidor_status & MetaDatos.status){
                str = null;
               DatagramPacket receivePacket = new DatagramPacket(receiveData,
receiveData.length);

                if(MetaDatos.debugTime)
                    start_time = System.nanoTime();

                try{
                    serverSocket.receive(receivePacket);
                    str = new String( receiveData, 0, receivePacket.getLength() );
                    if(MetaDatos.debug){
                        System.out.println("Esperamos al paquete de datos...");
                    }
                }catch(Exception e){
                    if(MetaDatos.debug)
                        e.printStackTrace();
                }

                if(MetaDatos.debug){
                    System.out.println("Llega: \n\t" + str);
                }

                // Lanzamos a correr la aplicacion de tratamiento de datos
                pT = new Proceso_Thread();
                pT.run(str);

                if(MetaDatos.debugTime){
                    end_time = System.nanoTime();
                    difference = (end_time - start_time)/1e6;
                    System.out.println("\n==============\nT_SERVIDOR:\t" + difference +
"ms.\n==============");
                }
            }
        }
        serverSocket.close();
    }
}
```

4.3. EL FORMATO DE LOS DATOS

Al recibir un datagrama UDP presenta los datos de los sensores con una codificación dada. Cada datagrama de datos consta de 500 bytes exactos, ya sea todo con datos o con relleno hasta completar dicha cantidad.

El datagrama codificado presenta la siguiente estructura:

```
g[<latitude>],[<N/S>],[<longitude>],[<E/W>],[<date>],[<UTC_time>][f(dF)]{a(dA)[b(dB)][c(dC)][d(dD)][e(dE)][h(dH)]}{...}## (Hasta 500 caracteres)
```

A continuación se presenta un ejemplo de un datagrama codificada:

```
g3758.713483,N,00107.827293,W,270514,134141.0f4a*bdcdd e«h*a
bdcdd eãh/a
bdcdd eÿh/a
bdcdd eêh/a
bdcdd eþh/a
bdcdd eîh/a
bdcdd eûh/a
bdcdd eóh*a
bdcdd e÷h*a
bdcdd e÷h*a
bdcdd eóh*a
bdcdd eüh*a
bdcdd eïh*a
bdcdd eÿh*a
bdcdd eëh*a
bdcdd eÿh*a
bdcdd eèh/a
bdcdd eÿh/a
bdcdd eæh/a
bdcdd eÿh/a
bdcdd eæh/a
bdcdd eÿh/a
bdcdd eèh*a
bdcdd eÿh*a
bdcdd eìh*a
bdcdd eþh*a
bdcdd eïh*a
bdcdd eúh*a
bdcdd eóh/a
bdcdd eõh/a
bdcdd eøh/a
bdcdd eðh*a
bdcdd eüh*abdcddeìh*a
bdcdd eÿh/a
bdcdd eéh/a
bdcdd eÿh/############
```

A continuación se explica cómo leer la información del datagrama de datos UDP:

Diferencia de tiempo	**IDENTIFICADOR:** a	**FRECUENCIA:** Varias veces por datagrama	**TAMAÑO:** 1byte
	Describe la separación en tiempo entre adquisiciones de datos, desde un conjunto de sensores, en milisegundos (respecto al anterior).		
Sensor RPM Rueda 1	**IDENTIFICADOR:** b	**FRECUENCIA:** Varias veces por datagrama	**TAMAÑO:** 1byte
	Representa el número de vueltas de la rueda 1 en revoluciones por minuto.		
Sensor RPM Rueda 2	**IDENTIFICADOR:** c	**FRECUENCIA:** Varias veces por datagrama	**TAMAÑO:** 1byte
	Representa el número de vueltas de la rueda 2 en revoluciones por minuto.		
Acelerómetro	**IDENTIFICADOR:** d	**FRECUENCIA:** Varias veces por datagrama	**TAMAÑO:** 1byte
	Representa el ángulo de inclinación (en grados) de la moto.		
Potenciómetro Lineal	**IDENTIFICADOR:** e	**FRECUENCIA:** Varias veces por datagrama	**TAMAÑO:** 1byte
	Mide distancia, aunque se expresa en milivoltios, y representa el recorrido de la suspensión de la moto.		
Potenciómetro Radial	**IDENTIFICADOR:** h	**FRECUENCIA:** Varias veces por datagrama	**TAMAÑO:** 1byte
	Mide distancia, aunque se expresa en milivoltios, y representa el recorrido del acelerador de la moto.		
Sensor de Temperatura	**IDENTIFICADOR:** f	**FRECUENCIA:** 1 vez por datagrama o menos	**TAMAÑO:** 1byte
	Representa la temperatura en grados Celsius. Como la temperatura no varía bruscamente, adquirimos la medida del sensor con menos frecuencia que de otros sensores.		
GPS	**IDENTIFICADOR:** g	**FRECUENCIA:** 1 vez por datagrama o menos	**TAMAÑO:** 44byte
	Representa la posición de la moto en latitud [N/S] y longitud [E/W] y la fecha en que se realizó la adquisición de la misma. Como es un proceso considerablemente lento, su adquisición se limita a momentos puntuales.		

Conociendo como está compuesto un datagrama UDP, podemos decodificar la información que contiene y encapsularla dentro de una nueva clase de datos creada, que presenta la siguiente estructura:

`private int nP = 11;`	Número de parámetros.
`private int nT;`	Número de tramas dentro del datagrama UDP.
`private String time;`	Instante de llegada del paquete UDP al servidor.
`private String gpsTime;`	Instante de adquisición de la posición GPS.
`private String longitud;`	Longitud de posición.
`private String latitud;`	Latitud de posición.
`private int[] difTime;`	*Array*[4] con las diferencias de tiempo.
`private int[] rpmR1;`	*Array* con las RPM de la rueda 1.
`private int[] rpmR2;`	*Array* con las RPM de la rueda 2.
`private int[] acelerometro;`	*Array* con los datos del acelerómetro
`private int[] potLineal;`	*Array* con los datos del potenciómetro lineal.
`private int temperature;`	Dato de la temperatura.
`private int [] potRadial;`	*Array* con los datos del potenciómetro radial.
`private boolean[] er = new boolean[nP];`	*Array* booleano de errores producidos en la decodificación de los datos.

Cuando un valor que recibimos no es correcto o no aparece, se marca con un valor característico. Aquí hay un ejemplo con el constructor de la clase vacio:

```
public Datos(){
    time = MetaDatos.gSimpleTime("yyyy-MM-dd HH:mm:ss:SSS");
    latitud = "99.999999";
    longitud = "-999.999999";
    gpsTime = "yyyy-MM-dd HH:mm:ss:SSS";
    difTime[0] = -9999;
    rpmR1[0] = -9999;
    rpmR1[0] = -9999;
    acelerometro[0] = -9999;
    potLineal[0] = -9999;
    temperatura = -9999;
    potRadial[0] = -9999;
    for(int i=0; i<nT; i++)
        er[i] = true;
}
```

[4] **Array** es una zona de almacenamiento continuo, que contiene una serie de elementos del mismo tipo, los elementos de la matriz. Desde el punto de vista lógico una matriz se puede ver como un conjunto de elementos ordenados en fila (o filas y columnas si tuviera dos dimensiones).

A continuación se muestra el código completo de la nueva clase creada para encapsular los datos recibidos:

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

package MotoStudent;

import java.nio.charset.Charset;
import java.sql.Timestamp;

/**
 *
 * @author PEDRO
 */


public class Datos {

    // <editor-fold defaultstate="collapsed" desc="VALORES RECIBIDOS EN EL SERVIDOR">
    private int nP = 11; // N° de Parámetros
    /** PARAMETROS
     * p0  -> General
     * p1  -> Latitud
     * p2  -> Longitud
     * p3  -> Tiempo GPS
     * p4  -> Diferencia de Tiempo
     * p5  -> RPM Rueda 1
     * p6  -> RPM Rueda 2
     * p7  -> Acelerometro
     * p8  -> Potenciometro lineal
     * p9  -> Potenciometro radial
     * p10 -> Temperatura
     */
    private int nT; // N° de Tramas
    private String time;
    private String gpsTime;
    private String longitud;
    private String latitud;
    private int[] difTime;
    private int[] rpmR1;
    private int[] rpmR2;
    private int[] acelerometro;
    private int[] potLineal;
    private int temperatura;
    private int [] potRadial;

    private boolean[] er = new boolean[nP]; // Definimos si hay error
    // </editor-fold>

    // <editor-fold defaultstate="collapsed" desc="GET & SET">
    public boolean[] getError(){
        return er;
    }

    public int getNItems(){ // Numero de Items
        return nP;
    }

    public int getNTramas(){ // Numero de sub-Tramas
        return nT;
    }
```

```
    public int getSDifTime(int pos){
        return difTime[pos];
    }

    public int getSRPMR1(int pos){
        return rpmR1[pos];
    }

    public int getSRPMR2(int pos){
        return rpmR2[pos];
    }

    public int getSAcelerometro(int pos){
        return acelerometro[pos];
    }

    public int getSPotLineal(int pos){
        return potLineal[pos];
    }

    public int getSTemperatura(){
        return temperatura;
    }

    public int getSPotRadial(int pos){
        return potRadial[pos];
    }

    public String getLatitud(){
        return latitud;
    }

    public String getLongitud(){
        return longitud;
    }

    public String getTime(){ // Tiempo del Servidor
        return time;
    }

    public String getGPSTime(){ // Tiempo del GPS
        return gpsTime;
    }
    // </editor-fold>

    // <editor-fold defaultstate="collapsed" desc="CONTRUCTORES">
    public Datos(){
        time = MetaDatos.gSimpleTime("yyyy-MM-dd HH:mm:ss:SSS");
        latitud = "99.999999";
        longitud = "-999.999999";
        gpsTime = "yyyy-MM-dd HH:mm:ss:SSS";
        difTime[0] = -9999;
        rpmR1[0] = -9999;
        rpmR1[0] = -9999;
        acelerometro[0] = -9999;
        potLineal[0] = -9999;
        temperatura = -9999;
        potRadial[0] = -9999;
        for(int i=0; i<nT; i++)
            er[i] = true;
    }

    public Datos(String trama){

        /** Formato de la trama:
         *
g[<latitude>],[<N/S>],[<longitude>],[<E/W>],[<date>],[<UTC time>]{[a(dA)][b(dB)][c(dC)][
d(dD)][e(dE)]}{...}## (500)
         */
```

```
        // Variables para debug de rendimiento
        long start_time = 0;
        long end_time = 0;
        double difference = 0;

        if(MetaDatos.debugTime)
            start_time = System.nanoTime();

        time = MetaDatos.gSimpleTime("yyyy-MM-dd HH:mm:ss:SSS");

        for(int i=1; i<nP; i++) // Inicializamos variables de errores a true
            er[i] = true;

        int nSubTramas = 0; // Numero de SubTramas dentro de la trama

         if(trama.charAt(0)== 'g'){
            // Latitud y longitud GPS
            //  [<latitude>]ddmm.mmmmmm,[<N/S>],[<longitude>]dddmm.mmmmmm,[<E/W>]
            latitud = getCoordinates(trama.substring(1,12), trama.substring(13,14), 1);
            longitud = getCoordinates(trama.substring(15, 27), trama.substring(28,29),
2);

            //Tomamos la hora proveniente del sensor
            // ddmmyy,hhmmss.s -> yyyy-MM-dd HH:mm:ss:SSS
            gpsTime = "20" + trama.substring(34,36) + "-" + trama.substring(32,34) + "-"
+ trama.substring(30,32) + " " + trama.substring(37,39) + ":" + trama.substring(39,41) +
":" + trama.substring(41,45) + "00";
            try{
                Timestamp ts = Timestamp.valueOf(gpsTime);
                er[3] = false;
            }catch(Exception e){
                if(MetaDatos.debug)
                    if(MetaDatos.debug)
                        e.printStackTrace();
                gpsTime = "yyyy-MM-dd HH:mm:ss:SSS";
                er[3] = true;
            }

            //Ocupa 42 caracteres.
            trama=trama.substring(41);
        }

        int len = 100;
        int[] TramasA = new int [len];
        int[] TramasB = new int [len];
        int[] TramasC = new int [len];
        int[] TramasD = new int [len];
        int[] TramasE = new int [len];
        int[] TramasH = new int [len];
        for(int i = 0; i<len; i++){
            TramasA[i] = -9999;
            TramasB[i] = -9999;
            TramasC[i] = -9999;
            TramasD[i] = -9999;
            TramasE[i] = -9999;
            TramasH[i] = -9999;
        }
        int TramasF = -9999;

        byte[] b = trama.getBytes(Charset.forName("UTF-8")); // Transformamos String to
Bytes[]
        boolean condition = true; // Condición del bucle
        int posB = 0; // Posición
        // Establecemos las posiciones de cada sensor
        while(condition){
            switch(b[posB]){
                case 'a':
                    try{
```

```
                        TramasA[nSubTramas] = MetaDatos.byte2int(b, posB+1,1);
                        nSubTramas++; // Contamos las subtramas, ya que siempre empiezan
por el sensor 'a'
                        er[4]=false;
                    }catch(Exception e){
                        if(MetaDatos.debug)
                            e.printStackTrace();
                        er[4]=true;
                    }
                    break;

                case 'b':
                    try{
                        TramasB[nSubTramas-1] = MetaDatos.byte2int(b, posB+1,1);
                        er[5]=false;
                    }catch(Exception e){
                        if(MetaDatos.debug)
                            e.printStackTrace();
                        er[5]=true;
                    }
                    break;
                case 'c':
                    try{
                        TramasC[nSubTramas-1] = MetaDatos.byte2int(b, posB+1,1);
                        er[6]=false;
                    }catch(Exception e){
                        if(MetaDatos.debug)
                            e.printStackTrace();
                        er[6]=true;
                    }
                    break;
                case 'd':
                    try{
                        TramasD[nSubTramas-1] = MetaDatos.byte2int(b, posB+1,1);
                        er[7]=false;
                    }catch(Exception e){
                        if(MetaDatos.debug)
                            e.printStackTrace();
                        er[7]=true;
                    }
                    break;
                case 'e':
                    try{
                        TramasE[nSubTramas-1] = MetaDatos.byte2int(b, posB+1,1);
                        er[8]=false;
                    }catch(Exception e){
                        if(MetaDatos.debug)
                            e.printStackTrace();
                        er[8]=true;
                    }
                    break;
                case 'f':
                    try{
                        TramasF = MetaDatos.byte2int(b, posB+1,1);
                        er[9]=false;
                    }catch(Exception e){
                        if(MetaDatos.debug)
                            e.printStackTrace();
                        er[10]=true;
                    }
                    break;
                case 'h':
                    try{
                        TramasH[nSubTramas-1] = MetaDatos.byte2int(b, posB+1,1);
                        er[9]=false;
                    }catch(Exception e){
                        if(MetaDatos.debug)
                            e.printStackTrace();
                        er[9]=true;
```

```
                    }
                    break;
                case '#':
                    condition = false;
                    break;
                default:
                    break;
            }
            posB = posB+2;
            if(posB > 499)
                condition = false;
        }

        // N° total de subtramas
        nT = nSubTramas;

        difTime = new int[nSubTramas];
        rpmR1 = new int[nSubTramas];
        rpmR2 = new int[nSubTramas];
        acelerometro = new int[nSubTramas];
        potLineal = new int[nSubTramas];
        potRadial = new int[nSubTramas];

        System.arraycopy(TramasA, 0, difTime, 0, nSubTramas);
        System.arraycopy(TramasB, 0, rpmR1, 0, nSubTramas);
        System.arraycopy(TramasC, 0, rpmR2, 0, nSubTramas);
        System.arraycopy(TramasD, 0, acelerometro, 0, nSubTramas);
        System.arraycopy(TramasE, 0, potLineal, 0, nSubTramas);
        System.arraycopy(TramasH, 0, potRadial, 0, nSubTramas);
        temperatura = TramasF;

        for(int i=1; i<nP; i++) // Indicamos que hay error
            if(er[i]){
                er[0]=true;
                break;
            }

        if(MetaDatos.debugTime){
            end_time = System.nanoTime();
            difference = (end_time - start_time)/1e6;
            System.out.println("\n==============\nT_DATOS:\t" + difference +
"ms.\n==============");
        }
    }
    // </editor-fold>

    // <editor-fold defaultstate="collapsed" desc="FUNCIONES">
    public String getCoordinates(String str, String cor, int op){
        // Modifica Latitud & Longitud para adaptarlas del Arduino al GoogleEarth
        er[op]=true;
        if(false){
            System.out.println("Valor:\t" + str +"\nCoordenada:\t" + cor);
        }

        String result;
        try{
            int pos=str.indexOf('.', 0);
            float unidad = Float.parseFloat(str.substring(0, pos-2));
            float decimal = Float.parseFloat(str.substring(pos-2, str.length())) / 60;
            result = Float.toString((float)(unidad + decimal));

            if(cor.equals("N") | cor.equals("E") ){
                er[op]=false;
            }
            if(cor.equals("S") | cor.equals("W") ){
                er[op] =false;
                result = "-" + result;
            }
            // Latitud =>   (N -> Positivo ; S -> Negativo)
```

```
            // Longitud =>  (E -> Positivo ; W -> Negativo)

        }catch(Exception e){
            er[op]=true;
            if(op == 1)
                result = "99.999999";
            else
                result = "-999.999999";
        }
        return result;

    }
    // </editor-fold>
}
```

Para la conversión de los datos en bytes se han diseñado dos funciones que convierten un *array* de bytes en un *array* de números enteros (byte a byte) o un *array* de bytes, de máximo 4 bytes, en un solo número entero.

```
// <editor-fold defaultstate="collapsed" desc="CODIFICACION">
    /**  byteA2intA
     *      Convierte un array de bytes en su equivalente en Integer
     *      barray  -> Array de bytes de entrada
     *      pos     -> Posición inicial a convertir en el array
     *      len     -> Longitud en Bytes de conversión
     */
    public static int[] byteA2intA(byte[] barray, int ini, int len){
        byte[] bint = new byte[len];
        System.arraycopy(barray, ini, bint, 0, len); //Copiamos el array
        int[] iarray = new int[len];
        int i = 0;
        for (byte b : bint)
            iarray[i++] = b & 0xff;
        return iarray;
    }

    /**  byte2int
     *      Convierte un array de bytes en el Integer correspondiente
     *      b       -> Array de bytes de entrada
     *      pos     -> Posición inicial a convertir en el array
     *      len     -> Longitud en Bytes de conversión
     */
    public static int byte2int(byte[] b, int pos, int len){
        byte[] bint = new byte[] {00, 00, 00, 00}; // Integer = 4 x Byte
        System.arraycopy(b, pos, bint, 4-len, len); //Copiamos el array
        int result = ((0xFF & bint[0]) << 24) | ((0xFF & bint[1]) << 16) | ((0xFF &
bint[2]) << 8) | (0xFF & bint[3]);
        return result;
    }
// </editor-fold>
```

4.4. BASE DE DATOS, CONEXIONES Y ALMACENAMIENTO

Tras haber descodificado el datagrama recibido, deseamos guardar estos resultados en un sistema permanente, independientemente de que la aplicación se esté ejecutando o no, y a su vez que nos dé la oportunidad de que estos valores puedan ser utilizados por otra aplicación, para ello guardaremos los resultados en una base de datos.

Primeramente, habrá que indicar a qué base de datos nos queremos conectar. Esta base de datos ha de estar dentro del propio equipo que ejecute la aplicación. Debemos indicar el nombre de la base de datos y además el usuario y contraseña autorizados. Para indicar estos valores lo haremos en el siguiente menú, que se encuentra en `Configuración > Base de Datos.`

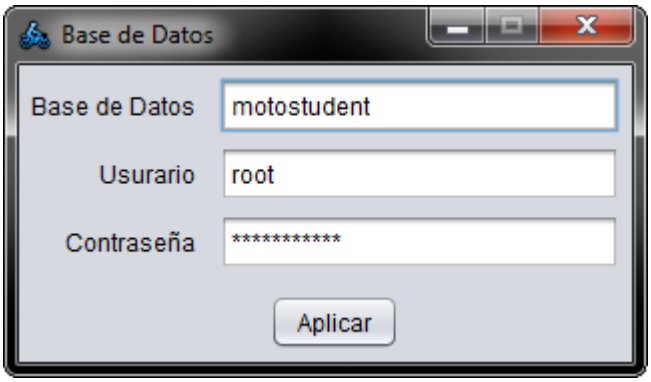


Figura 4.5. Menú de selección de la base de datos

Seguidamente, hemos de indicar la tabla de datos, dentro de la base de datos dada, se almacenarán u obtendrán los valores. Por defecto, cada vez que seleccionemos esta opción, si no hay ya una tabla seleccionada, la aplicación propondrá una nueva tabla, con nombre `tabla_yymmdd_hhmm`, en el caso de que ya haya una tabla seleccionada, nos indicará la misma. Tendremos la opción de elegir entre una nueva tabla y si hay tablas dentro de la base de datos, nos mostrará todas aquellas que aparecen. Este menú se encuentra en `Configuración > Tablas`.

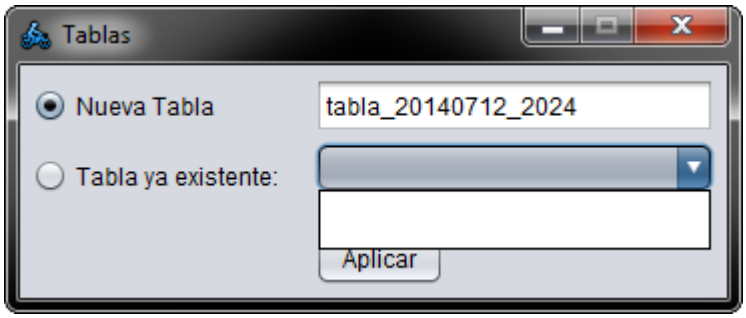


Figura 4.6. Menú para la selección de la tabla activa

Para agilizar los procesos con la base de datos, dentro de la clase auxiliar `MetaDatos`, se establecen unos parámetros.

```
// <editor-fold defaultstate="collapsed" desc="BASE DE DATOS">
    public static Connection connection;
    public static Statement statement;
    public static String params[] =
{"t_sever","t_gps","latitud","longitud","difTime","rpmRueda1","rpmRueda2","acelerometro"
,"potLineal", "potRadial", "temperatura"};
    private static String url = "jdbc:mysql://localhost/";
    static String tableCreate = " (id int(25) unsigned NOT NULL AUTO_INCREMENT PRIMARY
KEY, t_server VARCHAR(25) DEFAULT NULL, t_gps VARCHAR(25) DEFAULT NULL, latitud
VARCHAR(12) DEFAULT NULL, longitud VARCHAR(12) DEFAULT NULL, difTime int(32) unsigned
DEFAULT NULL, rpmRueda1 int(32) DEFAULT NULL, rpmRueda2 int(32) DEFAULT NULL,
acelerometro int(32) DEFAULT NULL, potLineal int(32) DEFAULT NULL, potRadial int(32)
DEFAULT NULL, temperatura int(32) DEFAULT NULL)";
    static String tableAdd = " (id, t_server, t_gps, latitud, longitud, difTime,
rpmRueda1, rpmRueda2, acelerometro, potLineal, potRadial, temperatura)";
// </editor-fold>
```

Una vez configurada la base de datos y la tabla deseada, podemos realizar una conexión a la base de datos, de la siguiente manera:

```
// <editor-fold defaultstate="collapsed" desc="CONNECT">
    public static void connect() throws Exception{
        if(debug){
            System.out.println("DB (" + bbdd + "): Connected");
        }
        try {
            connection = DriverManager.getConnection(url + bbdd, usuario, passwd);
            statement = (Statement) connection.createStatement();
        }
        catch(Exception e){
            if(MetaDatos.debug)
                e.printStackTrace();
            throw e;
        }
    }
// </editor-fold>
```

También es necesaria la desconexión de la base de datos, que se realiza con la siguiente función:

```
// <editor-fold defaultstate="collapsed" desc="SHUTDOWN">
    public static void shutdown(){
        try {
            if(debug){
                System.out.println("DB (" + bbdd + "): Shutdown");
            }
            connection.close();
        }
        catch(Exception e){
            if(MetaDatos.debug)
                e.printStackTrace();
        }
    }
// </editor-fold>
```

Podemos eliminar tablas de la base de datos, siempre y cuando no sea aquella que esté activa. Esta opción se encuentra en el menú `Archivo > Borrar tabla de datos`….

Figura 4.7. Menú para borrar una tabla no activa

El código que ejecuta la acción es el siguiente:

```
// <editor-fold defaultstate="collapsed" desc="DELETE SQL_TABLE">
    public static void deleteTableSQL(String tablename){
        try{
            connect();
        }
        catch (Exception ex) {
            if(MetaDatos.debug)
                ex.printStackTrace();
        }

        try {
            if(debug){
                System.out.println("Borramos la tabla de la BBDD");
            }
            statement.executeUpdate("DROP TABLE IF EXISTS " + tablename);
        }
        catch (SQLException ex) {
            if(MetaDatos.debug)
                ex.printStackTrace();
        }

        shutdown();
    }
// </editor-fold>
```

Otra función indispensable es la de crear una tabla, que encapsulamos en el siguiente código:

```
// <editor-fold defaultstate="collapsed" desc="CREATE SQL_TABLE">
    public static void createTableSQL(String tablename){
        try{
            connect();
        }
        catch (Exception ex) {
            if(MetaDatos.debug)
                ex.printStackTrace();
        }

        try {
            if(debug){
                System.out.println("Creamos tabla \"" + tablename +"\" si no existe en
la BBDD..");
            }
            statement.executeUpdate("CREATE TABLE IF NOT EXISTS " + tablename +
tableCreate);
        }
        catch (SQLException ex) {
            if(MetaDatos.debug)
                ex.printStackTrace();
        }
        shutdown();
    }
// </editor-fold>
```

Ya tratando con la inserción de los datos en la base de datos, para optimizar el tiempo de inserción, se realizará una sola inserción con todos los campos, indicando cada fila los campos que son nulos, para así poder asignar a cada columna su valor correspondiente. De esta forma, el hilo de procesado de inserción de datos queda de la siguiente forma:

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

package MotoStudent;

import java.sql.SQLException;


/**
 *
 * @author PEDRO
 */
public class BBDD_Thread extends Thread{

    public void run(Datos d){

        // Variables para debug de rendimiento
        long start_time = 0;
        long end_time = 0;
        double difference = 0;

        if(MetaDatos.debugTime)
            start_time = System.nanoTime();
```

```
        try{
            MetaDatos.connect(); // Nos conectamos a la base de datos
        }
        catch (Exception e){
            if(MetaDatos.debug)
                e.printStackTrace();
        }

        try {
            if(MetaDatos.debug){
                System.out.println("Insertamos los nuevos datos en la tabla de la
BBDD..");
            }
            String sqlSentenceComplete;

            // Realizamos la sentencia SQL
            sqlSentenceComplete= insertInto(d, d.getNTramas());

            MetaDatos.statement.executeUpdate(sqlSentenceComplete);

        }
        catch (SQLException ex) {
            if(MetaDatos.debug)
                ex.printStackTrace();
        }

        MetaDatos.shutdown(); // Cerramos la base de datos


        if(MetaDatos.debugTime){
            end_time = System.nanoTime();
            difference = (end_time - start_time)/1e6;
            System.out.println("\n==============\nT_BBDD:\t" + difference +
"ms.\n==============");
        }
    }

    // <editor-fold defaultstate="collapsed" desc="FUNCIONES ">
    public static String insertInto(Datos d, int len){
        // " (id, t_server, t_gps, latitud, longitud, difTime, rpmRueda1, rpmRueda2,
acelerometro, potLineal, potRadial, temperatura)"
        int pos = 0;

        String str = "INSERT INTO " + MetaDatos.tablaActual;
        str += "(t_server, t_gps, latitud, longitud, difTime, rpmRueda1, rpmRueda2,
acelerometro, potLineal, potRadial, temperatura)VALUES(\"" + d.getTime() +"\",\"" +
d.getGPSTime() + "\",\"" + d.getLatitud() + "\",\"" + d.getLongitud() +
"\",null,null,null,null,null,null,null)";
        str += ",(\"\",\"\", \"\", \"\"," + Integer.toString(d.getSDifTime(pos))+ "," +
Double.toString(d.getSRPMR1(pos))+ "," + Double.toString(d.getSRPMR2(pos))+ "," +
Double.toString(d.getSAcelerometro(pos))+ "," + Double.toString(d.getSPotLineal(pos)) +
"," + Double.toString(d.getSPotRadial(pos)) + ",";
        if(d.getSTemperatura()!=-9999)
            str += Double.toString(d.getSTemperatura()) + ")";
        else
            str+= "null)";

        for(pos = 1; pos <len; pos++)
            str += ",(\"\",\"\", \"\", \"\"," + Integer.toString(d.getSDifTime(pos))+
"," + Double.toString(d.getSRPMR1(pos))+ "," + Double.toString(d.getSRPMR2(pos))+ "," +
Double.toString(d.getSAcelerometro(pos))+ "," + Double.toString(d.getSPotLineal(pos)) +
"," + Double.toString(d.getSPotRadial(pos)) + ",null)";
        str += ";";

        return str;
    }
    // </editor-fold>
}
```

En el menú para seleccionar la tabla, aparece un elemento llamado `combobox`, el cual muestra las distintas tablas que hay en la base de datos. Para rellenar este elemento se realiza una consulta a la base de datos para obtener las distintas tablas que almacena:

```
// <editor-fold defaultstate="collapsed" desc="COMBOBOX">
    public static DefaultComboBoxModel fillCombo(String tablaException){
        try{
            Vector comboBoxItems=new Vector();
            connect();
            DatabaseMetaData md = connection.getMetaData();
            ResultSet rs = md.getTables(null, null, "%", null);
            while (rs.next()) {
                if( !(tablaException.equals(rs.getString(3))) ){
                    // Comprobamos que no sea igual a la expceción para añadir al
ComboBox
                    comboBoxItems.add(rs.getString(3));
                }
            }
            shutdown();
            DefaultComboBoxModel model = new DefaultComboBoxModel(comboBoxItems);
            return model;
        }
        catch(Exception e){
            if(MetaDatos.debug)
                e.printStackTrace();
        }

        return null;
    }
    // </editor-fold>
```

4.5. REPRESENTACIÓN DE GRÁFICAS

Cuando se desee representar los valores obtenidos por un sensor durante el transcurso de un tiempo, hemos de indicar, dentro del menú `Visualizar`, qué gráficas se desean mostrar.

Figura 4.8. Menú del visualizador de las gráficas

Para la creación de las gráficas nos ayudaremos de la librería `org.jfree.chart`, para crear gráficos por puntos (X, Y). Además nos da la posibilidad de importar las gráficas directamente a formato *PNG* o imprimirlas según se desee.

La visualización final de las gráficas presenta el siguiente estilo:

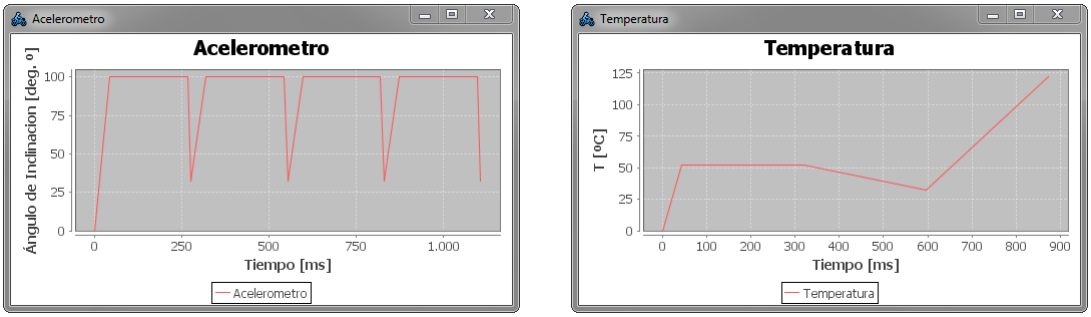


Figura 4.9. Gráficas de ejemplo

Entrando ya a nivel de programación, para facilitar aún más la creación de las gráficas, se crea una nueva clase que gestionará la interfaz gráfica de la misma.

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

package MotoStudent;

/**
 *
 * @author PEDRO
 */

import java.awt.Image;
import java.awt.Toolkit;
import java.awt.event.WindowEvent;
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.data.xy.XYSeries;
import org.jfree.data.xy.XYSeriesCollection;
import org.jfree.ui.ApplicationFrame;

/**
 * A simple demo showing a dataset created using the {@link XYSeriesCollection} class.
 *
 */
public class XYSeriesDemo extends ApplicationFrame{

    /**
     * A demonstration application showing an XY series containing a null value.
     *
     * @param title  the frame title.
     */
    private XYSeries series;

    public XYSeriesDemo(final String title, XYSeries series, String header,String aX,
String aY) {

        super(title);
```

```
        // <editor-fold defaultstate="collapsed" desc="ICON">
        try{
            Toolkit kit = Toolkit.getDefaultToolkit();
            Image imgIcon = kit.createImage("img/icon.png");
            setIconImage(imgIcon);
        }catch(Exception e){
            if(MetaDatos.debug)
                e.printStackTrace();
        }
        // </editor-fold>

        this.series = series;

        final XYSeriesCollection data = new XYSeriesCollection(series);
        final JFreeChart chart = ChartFactory.createXYLineChart(
            header,
            aX,
            aY,
            data,
            PlotOrientation.VERTICAL,
            true,
            true,
            false
        );

        final ChartPanel chartPanel = new ChartPanel(chart);
        chartPanel.setPreferredSize(new java.awt.Dimension(500, 270));
        setContentPane(chartPanel);
    }


    public void windowClosing(WindowEvent e){
        String title = this.getTitle();
        if(MetaDatos.debug){
            System.out.println("Titulo de la ventana: \"" + title + "\"");
        }
        for(int i = 0; i<MetaDatos.gnombres.length; i++){
            if(MetaDatos.gnombres[i].equals(title)){
                MetaDatos.gstatus[i] = false;
                PrincipalFrame.closeGraph(i);
            }
        }
        this.dispose();
    }

    public XYSeries getSeries() {
        return series;
    }

}
```

Por otro lado, disponemos de una serie de variables que ayudan a facilitar la programación.

```
// <editor-fold defaultstate="collapsed" desc="GRAFICAS">
    public static boolean begin;        // Indica si es la primera vez con la gráfica
    public static int gNum = 6;         // Número de gráficas
    public static int ejeX;             // Valor del Eje X
    public static XYSeriesDemo[] graficas = new XYSeriesDemo [gNum]; // Array de
gráficas
    public static boolean[] gstatus = new boolean[gNum];    // Estado visible de las
gráficas
    public static String[] gnombres = {"RPM Rueda 1", "RPM Rueda 2", "Acelerometro",
"Potenciometro Lineal (Suspension)", "Potenciometro Radial (Acelerador)",
"Temperatura"};
```

```
    public static String titleAxisY[] = {"RPM [r/min]","RPM [r/min]","Ángulo de
Inclinacion [deg. °]","Voltaje [mV]", "Voltaje [mV]", "T [°C]",};
// </editor-fold>
```

También se ha creado una función para facilitar la adquisición de los valores almacenados en la base de datos para mostrarlos en la gráfica pertinente.

```
// <editor-fold defaultstate="collapsed" desc="MySQL 2 GRAPH">
    public static XYSeriesDemo MySQLtoGraph(int col,String titleGraph, String titleX){
        try {
            //createTableSQL(tablaActual);
            XYSeriesDemo demoGraph01;
            connect();
            int[] axisX = getAxisX(params[col]);
            demoGraph01 = generateGraph(params[col], titleX,axisX,titleGraph);
//Columna = col + 1
            shutdown();
            //demoGraph01.dispose();
            return demoGraph01;
        }
        catch (Exception e) {
            if(MetaDatos.debug)
                e.printStackTrace();
            return null; // Error!!
        }
        //return null; // Error!!
    }

    public static XYSeriesDemo generateGraph(String sensor, String titleAxisY,int[]
axisX, String graphname)throws SQLException, FileNotFoundException, IOException {
        // Create new Graph
        final XYSeries series = new XYSeries(graphname);
        final XYSeriesDemo demo = new XYSeriesDemo(graphname, series, graphname,"Tiempo
[ms]", titleAxisY);
        demo.pack();
        //setDefaultCloseOperation(javax.swing.WindowConstants.DISPOSE ON CLOSE);
        RefineryUtilities.centerFrameOnScreen(demo);
        demo.setVisible(true);

        // Execute SQL query
        try{
            PreparedStatement stmt = connection.prepareStatement("SELECT " + sensor + "
FROM " + tablaActual + " WHERE " + sensor + " IS NOT NULL");
            ResultSet rs = stmt.executeQuery();


            int i = 1;
            series.add(axisX[0], 0); // Añadimos en la gráfica
            while (rs.next()) {
                try{
                    if(rs.getInt(1) != -9999)
                        series.add(axisX[i], rs.getInt(1)); // Añadimos en la gráfica
                    else
                        series.add(axisX[i], null);
                }catch(Exception e){
                    series.add(axisX[i], null);
                    if(debug){
                        if(MetaDatos.debug)
                            e.printStackTrace();
                    }
                }
                i++;
            }
        }
        catch(Exception e){
```

```
            if(MetaDatos.debug)
                e.printStackTrace();
            MetaDatos.status = false;
            //changeStatus.setText("Activar");
            JOptionPane.showMessageDialog(new JOptionPane(),"NO HAY TABLA
CREADA","ERROR",JOptionPane.WARNING_MESSAGE);
    '       demo.dispose();
            return null; // Error
        }

        return demo;
    }
// </editor-fold>
```

Sabiendo que los datos no presentan la misma frecuencia, se ha de generar el eje de tiempos según indica la acumulación de marcas temporales asociadas al sensor que queremos representar, para ello se ha creado la siguiente función:

```
// <editor-fold defaultstate="collapsed" desc="AXYS X">
    public static int[] getAxisX(String sensor){

        // Execute SQL query
        PreparedStatement s, sX;
        ResultSet r, rX;
        try{
            s = MetaDatos.connection.prepareStatement("SELECT id, difTime FROM " +
tablaActual + " WHERE difTime IS NOT NULL", Statement.RETURN_GENERATED_KEYS);
            r = s.executeQuery();

            sX = MetaDatos.connection.prepareStatement("SELECT id, difTime FROM " +
tablaActual + " WHERE " + sensor + " IS NOT NULL", Statement.RETURN_GENERATED_KEYS);
            rX = sX.executeQuery();


            rX.last();
            int [] axisX = new int[rX.getRow()+ 1];

            r.last();
            int [] valores = new int[r.getRow()+ 1];

            rX.beforeFirst();
            r.beforeFirst();

            int ind =1;
            int indX = 1;
            int pv = 0;
            valores[0] = 0;
            axisX[0] = 0;
            valores[1] = 0;
            axisX[1] = 0;

            rX.next();

            while (r.next()) {
                valores[ind] = r.getInt("difTime") + pv;
                pv = valores[ind];
                try{
                    if( (r.getInt("id")) == (rX.getInt("id")) ){
                        axisX[indX] = valores[ind];
                        indX++;
                        rX.next();
                    }
                }catch(Exception e){
                    break;
                }
```

```
                ind++;

            }

            return axisX;

        }catch(Exception e){
            if(MetaDatos.debug)
               e.printStackTrace();
        }
        int [] er = {-1, -1};
        return er;

    }
// </editor-fold>
```

A la hora de representar las gráficas, como ya hemos visto antes, en el servidor intentamos visualizar aquellos datos que estuvieran previamente guardados en la tabla seleccionada.

Cuando llegan nuevos datos y son almacenados, éstos se muestran en las gráficas que estén activas mediante un nuevo hilo de procesado por cada recepción.

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

package MotoStudent;

import org.jfree.data.xy.XYSeries;

/**
 *
 * @author PEDRO
 */
public class Graph_Thread extends Thread{
     public void run(Datos d){
        // Añadimos en la gráfica

        // Variables para debug de rendimiento
        long start_time = 0;
        long end_time = 0;
        double difference = 0;

        if(MetaDatos.debugTime)
            start_time = System.nanoTime();


        XYSeries series; // Gráfico de series

        // Temperatura
        if(d.getSTemperatura() != -9999){
            series = MetaDatos.graficas[5].getSeries();
            series.add(MetaDatos.ejeX + d.getSDifTime(0), new Integer (
d.getSTemperatura() ));
        }

        for(int i = 0; i<d.getNTramas(); i++){
            MetaDatos.ejeX = MetaDatos.ejeX + d.getSDifTime(i);

            // RPM Rueda 1
```

```
            series = MetaDatos.graficas[0].getSeries();
            if(d.getSRPMR1(i) != -9999)
                series.add(MetaDatos.ejeX , new Integer( d.getSRPMR1(i) ));
            else
                series.add(MetaDatos.ejeX , null);

            // RPM Rueda 2
            series = MetaDatos.graficas[1].getSeries();
            if(d.getSRPMR2(i) != -9999)
                series.add(MetaDatos.ejeX , new Integer( d.getSRPMR2(i) ));
            else
                series.add(MetaDatos.ejeX , null);

            // Acelerometro
            series = MetaDatos.graficas[2].getSeries();
            if(d.getSAcelerometro(i) != -9999)
                series.add(MetaDatos.ejeX , new Integer( d.getSAcelerometro(i) ));
            else
                series.add(MetaDatos.ejeX , null);

            // Potenciometro Lineal
            series = MetaDatos.graficas[3].getSeries();
            if(d.getSPotLineal(i) != -9999)
                series.add(MetaDatos.ejeX , new Integer( d.getSPotLineal(i) ));
            else
                series.add(MetaDatos.ejeX , null);

            // Potenciometro Radial
            series = MetaDatos.graficas[4].getSeries();
            if(d.getSPotRadial(i) != -9999)
                series.add(MetaDatos.ejeX , new Integer( d.getSPotRadial(i) ));
            else
                series.add(MetaDatos.ejeX , null);
        }


        for(int i = 0; i < MetaDatos.gNum; i++){
            MetaDatos.graficas[i].setVisible( MetaDatos.gstatus[i] );
        }

        if(MetaDatos.debugTime){
            end_time = System.nanoTime();
            difference = (end_time - start_time)/1e6;
            System.out.println("\n==============\nT_GRAFICAS:\t" + difference +
"ms.\n==============");
        }

    }

}
```

4.6. POSICIONAMIENTO EN GOOGLEMAPS

Dentro del menú `Configuración > GPS` se puede editar el nombre del fichero *KML*, el cual por defecto es `MotoUPCT.kml`.

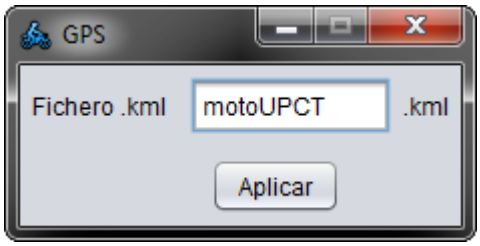


Figura 4.10. Menú para elegir el nombre del fichero KML

En cuanto a la programación, se ha generado un conjunto de variables que facilitan la gestión de la aplicación.

```
// <editor-fold defaultstate="collapsed" desc="KML">
    private static String inicioKLM = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n" +
"<kml xmlns=\"http://www.opengis.net/kml/2.2\">\n" +
"<Document>\n" +
"   <Style id=\"randomColorIcon\">\n" +
"      <IconStyle>\n" +
"         <scale>1.1</scale>\n" +
"         <Icon>\n" +
"            <href>img/icon.png</href>\n" +
"         </Icon>\n" +
"      </IconStyle>\n" +
"   </Style>\n" +
"  <Placemark>\n" +
"    <name>UPCT Moto</name>\n" +
"    <styleUrl>#randomColorIcon</styleUrl>\n" +
"    <description>\n" +
"      Aqui esta la posicion de la moto UPCT\n" +
"    </description>\n" +
"    <Point>\n" +
"      <coordinates>";

    private static String finalKLM = ",0</coordinates>\n" +
"    </Point>\n" +
"  </Placemark>\n" +
"</Document>\n" +
"</kml>";
// </editor-fold>
```

Para generar el archivo *.kml*, se ha creado la siguiente función:

```
// <editor-fold defaultstate="collapsed" desc="FICHERO KML">
    public static void generateKML(Datos d){
        FileWriter fichero = null;
    '
        try{
            fichero = new FileWriter(kml + ".kml",false);
            PrintWriter pw = new PrintWriter(fichero);
            pw.println(inicioKLM + d.getLongitud() + "," + d.getLatitud() + finalKLM);
        }
        catch (Exception e) {
            if(MetaDatos.debug)
                e.printStackTrace();
        }

        try {
            if (null != fichero){
                    fichero.close();
            }
       }
       catch (Exception e2) {
            if(MetaDatos.debug)
                e2.printStackTrace();
        }
    }
// </editor-fold>
```

En el servidor, una vez recibido un dato correcto de la posición GPS, y siempre y cuando hayamos indicado que queremos ver donde se encuentra la moto, el servidor lanzará un nuevo hilo de procesado encargado de realizar este trabajo.

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

package MotoStudent;

import java.awt.Desktop;
import java.io.File;

/**
 *
 * @author PEDRO
 */
public class GPS_Thread extends Thread{
    public void run(Datos d){
        MetaDatos.generateKML(d);
        try{
            if (Desktop.isDesktopSupported()){
                if(MetaDatos.debug){
                    System.out.println("Ejecutamos el nuevo fichero '.kml'");
                }
                Desktop.getDesktop().open(new File(MetaDatos.kml + ".kml"));
            }
        }
        catch(Exception e){
            if(MetaDatos.debug)
                e.printStackTrace();
        }
    }
}
```

Como podemos observar en el código, se genera un nuevo fichero que contiene toda información sobre la posición actual de la moto. Este fichero se ejecuta, situando en el programa *GoogleEarth* la posición de la moto de competición.

4.7. EXPORTACIÓN DE RESULTADOS

Para mejorar la funcionalidad de la aplicación, se ha optado habilitar la exportación de los datos recogidos por los distintos sensores de la moto en formatos comunes para cualquier equipo. La tabla a exportar es siempre la tabla que está activa, por lo que si queremos exportar una tabla ya creada, solo tendremos que seleccionarla en el menú `Configuración > Tabla`. Los formatos disponibles para exportar son los siguientes:

- Hoja de Cálculo Excel (`.xls`): Se establecerá en la primera fila el nombre del valor que almacenará cada columna. Para cada celda, se rellenará el valor que corresponde, tal y como aparece en la tabla activa de la base de datos.
- Texto Plano (`.txt`): Para no depender de un sistema operativo, o queramos crear una aplicación para realizar cálculos sobre estos valores, exportamos con un formato dado los resultados obtenidos, donde cada línea será el conjunto fila de la base de datos, y las distintas columnas están se crean por tabulaciones.

Para exportar los resultados almacenados, utilizaremos el explorador de Windows, que accedemos a través del menú `Archivo > Exportar datos a…`

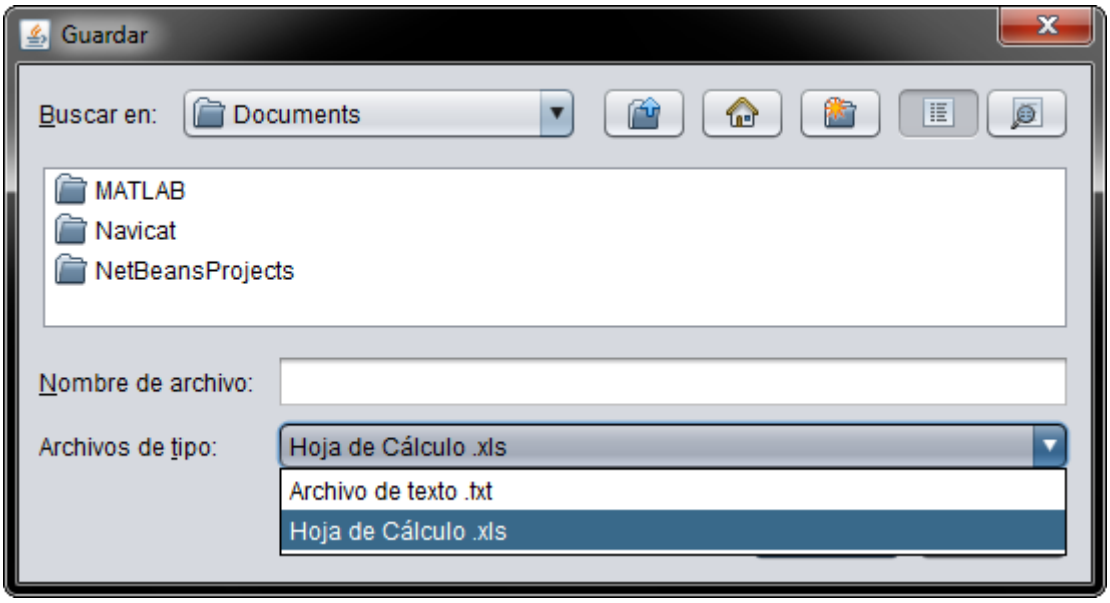


Figura 4.11. Menú para exportar datos de la tabla activa

Para realizar esta exportación, se maneja el objeto `FileChooser`:

```
    private void exportMenuActionPerformed(java.awt.event.ActionEvent evt) {
        JFileChooser chooser = new JFileChooser(); // Creamos el FileChooser
        chooser.removeChoosableFileFilter(chooser.getFileFilter()); // Borramos la
opción de "Todos los Archivos"

        FileFilter filter01 = new FileNameExtensionFilter("Archivo de texto .txt",
"txt");
        FileFilter filter02 = new FileNameExtensionFilter("Hoja de Cálculo .xls",
"xls");

        chooser.setFileFilter(filter01); // Añadimos la opción de .txt
        chooser.setFileFilter(filter02);  // Añadimos la opción de .xls

        if(MetaDatos.tabla_status){
            chooser.setSelectedFile(new File(MetaDatos.tablaActual));
        }

        int response = chooser.showSaveDialog(null); // Abrimos el FileChooser en modo
Guardar, en la carpeta actual del programa
        if(response == JFileChooser.APPROVE_OPTION){ // Opción GUARDAR
            File f = chooser.getSelectedFile();
            String filename = f.getAbsolutePath();
            if(MetaDatos.debug){
                System.out.print("Guardando archivo: " + filename);
            }

            String extension = chooser.getFileFilter().getDescription();
            if(extension.equals("Archivo de texto .txt")){
                if(MetaDatos.debug){
                    System.out.println(".txt");
                }
                try{
                    MetaDatos.MySQLtoTXT(filename);
                }
```

```
                catch(Exception e){
                    if(MetaDatos.debug)
                        e.printStackTrace();
                    JOptionPane.showMessageDialog(this,"La tabla seleccionada no está
creada.","Inane warning",JOptionPane.WARNING_MESSAGE);
                }
            }
            if(extension.equals("Hoja de Cálculo .xls")){
                if(MetaDatos.debug){
                    System.out.println(".xls");
                }
                try{
                    MetaDatos.MySQLtoXls(filename);
                }
                catch(Exception e){
                    if(MetaDatos.debug)
                        e.printStackTrace();
                    JOptionPane.showMessageDialog(this,"La tabla seleccionada no está
creada.","Inane warning",JOptionPane.WARNING_MESSAGE);
                }
            }
        }
        else{ // Opción CANCELAR
            if(MetaDatos.debug){
                System.out.println("Operación cancelada...");
            }
        }
    }
```

Como se observa en el código, una vez elegida la ruta, el nombre del fichero y el tipo de exportación, se llama a la función correspondiente para que realice la exportación de los datos.

Para exportar un fichero en texto plano el código utilizado es el siguiente:

```
// <editor-fold defaultstate="collapsed" desc="MySQL 2 TXT">
    public static void MySQLtoTXT(String nombreTXT)throws Exception{
        try{
            connect();
        }
        catch(Exception e){
            throw e;
        }

        try{
            generateTxt(tablaActual, nombreTXT + ".txt");
            shutdown();
        }
        catch (Exception e) {
            throw e;
        }
    }

    public static void generateTxt(String tablename, String filename)throws
SQLException, FileNotFoundException, IOException, Exception {
        // Create new .txt
        File file = new File(filename);
        FileWriter fw = new FileWriter(file.getAbsoluteFile(),false);
       BufferedWriter bw = new BufferedWriter(fw);

        // Execute SQL query
```

```
        PreparedStatement stmt = connection.prepareStatement("SELECT * FROM " +
tablename);
        ResultSet rs = stmt.executeQuery();

        // Get the list of column names and store them as the first row of the
spreadsheet.
        ResultSetMetaData colInfo = rs.getMetaData();
        List<String> colNames = new ArrayList<String>();
        for (int i = 1; i <= colInfo.getColumnCount(); i++) {
            colNames.add(colInfo.getColumnName(i));
            bw.write(colInfo.getColumnName(i));
            bw.write("\t");
        }
        bw.write("\r\n");

        // Save all the data from the database table rows
        while (rs.next()) {
            for (String colName : colNames) {
                if(rs.getString(colName) == null)
                    bw.write("null");
                else
                    bw.write(rs.getString(colName));
                bw.write("\t");
            }
            bw.write("\r\n");
        }
        bw.close();
    }
    // </editor-fold>
```

Para exportar a una Hoja de Cálculo se hace uso de la librería `org.apache.poi.hssf` que facilita el trabajo con las Hojas de Cálculo. El código generado es el siguiente:

```
// <editor-fold defaultstate="collapsed" desc="MySQL 2 EXCEL">
    public static void MySQLtoXls (String nombreXLS)throws Exception{
        try{
            connect();
        }
        catch(Exception e){
            throw e;
        }

        try{
            generateXls(tablaActual, nombreXLS + ".xls");
            shutdown();
        }
        catch (Exception e) {
            throw e;
        }

    }

    public static void generateXls(String tablename, String filename)throws
SQLException, FileNotFoundException, IOException, Exception {
        // Create new Excel workbook and sheet
        HSSFWorkbook xlsWorkbook = new HSSFWorkbook();
        HSSFSheet xlsSheet = xlsWorkbook.createSheet();
        short rowIndex = 0;

        // Execute SQL query
        PreparedStatement stmt =
```

```
        connection.prepareStatement("SELECT * FROM " + tablename);
        ResultSet rs = stmt.executeQuery();

        // Get the list of column names and store them as the first
        // row of the spreadsheet.
        ResultSetMetaData colInfo = rs.getMetaData();
        List<String> colNames = new ArrayList<String>();
        HSSFRow titleRow = xlsSheet.createRow(rowIndex++);

        for (int i = 1; i <= colInfo.getColumnCount(); i++) {
            colNames.add(colInfo.getColumnName(i));
            titleRow.createCell((short) (i-1)).setCellValue(
            new HSSFRichTextString(colInfo.getColumnName(i)));
            xlsSheet.setColumnWidth((short) (i-1), (short) 4000);
        }

        // Save all the data from the database table rows
        while (rs.next()) {
            HSSFRow dataRow = xlsSheet.createRow(rowIndex++);
            short colIndex = 0;
            for (String colName : colNames) {
                dataRow.createCell(colIndex++).setCellValue(
                new HSSFRichTextString(rs.getString(colName)));
            }
        }

        // Write to disk
        xlsWorkbook.write(new FileOutputStream(filename));
        xlsWorkbook = null; // Cerramos el fichero (?)
    }
// </editor-fold>
```

4.8. EL ENTORNO GRÁFICO

Dentro del menú `Ayuda > Uso` podemos ver un resumen de cómo utilizar la aplicación con las utilidades básicas.

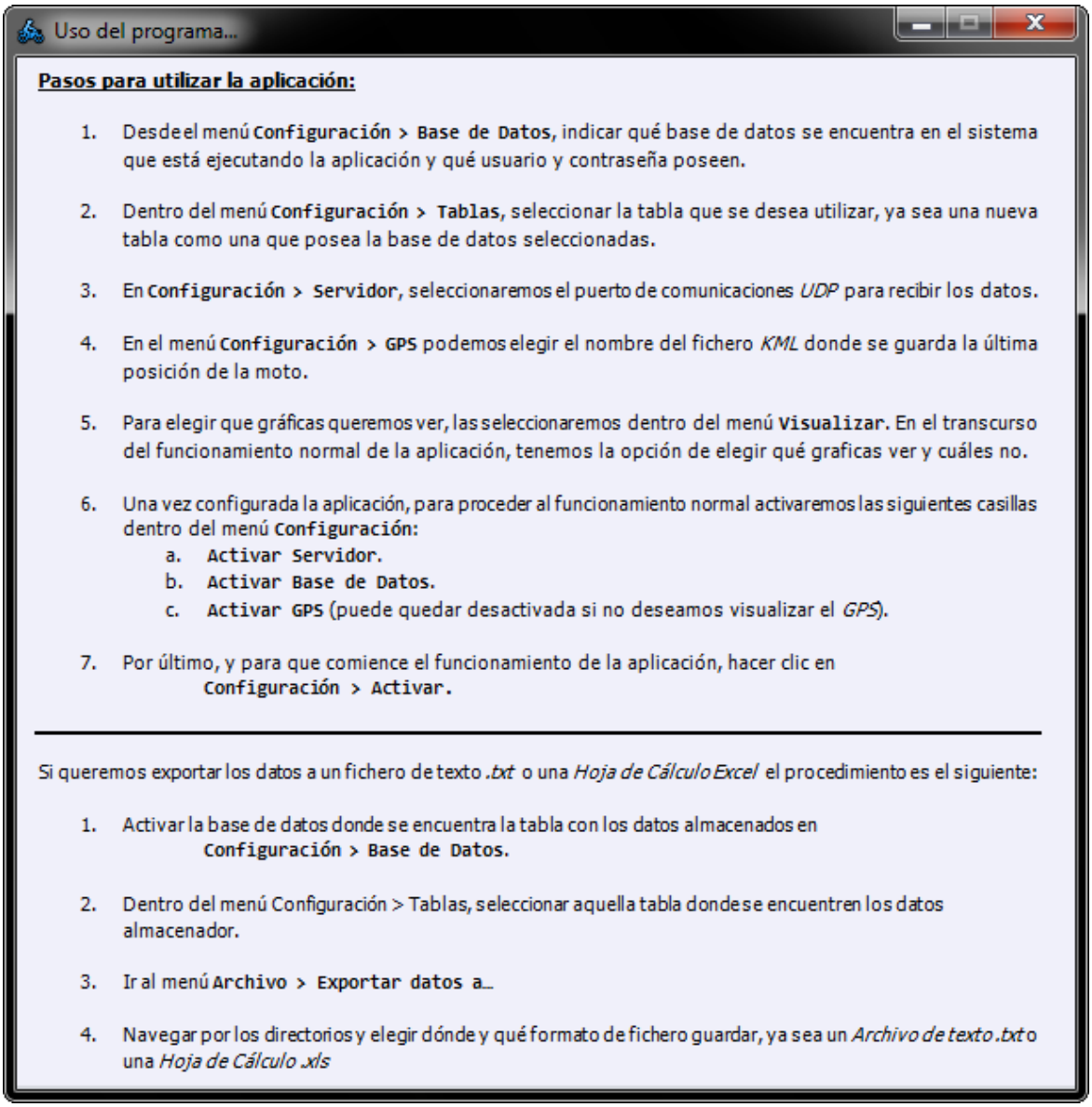


Figura 4.12. Menú de ayuda del funcionamiento del programa

Por otro lado, en el menú `Ayuda > Acerca de…` define aquellas personas y departamentos implicados en el desarrollo de esta aplicación.

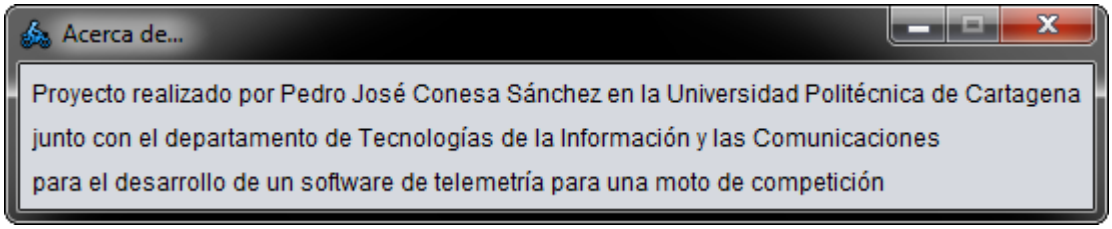


Figura 4.13. Menú explicativo de la aplicación

Una de las partes importantes de la aplicación es el control que tenemos sobre su funcionamiento solamente a nivel gráfico, es decir, las funcionalidades agregadas a la interfaz gráfica que nos permiten controlar las funciones del programa.

Primeramente, se establecen unos valores por defecto unas variables del control del programa, como son:

- `status`: Establece si la aplicación se encuentra activa o no, por defecto la aplicación no está activa.
- `servidor_status`: Establece si el servidor se encuentra activo o no, por defecto el servidor no está activo.
- `dbdd_status`: Establece si desea almacenar datos en la base de datos o no, por defecto no se permiten almacenar datos en la base de datos.
- `tabla_status`: Indica si hay tabla seleccionada dentro de la base de datos.
- `begin`: Variable para determinar si hemos mostrado por primera vez las gráficas, con el fin de recorrer todos los registros anteriores que haya en la base de datos.
- `ejeX`: Guarda en memoria el último momento de recogida de datos, para su uso en la inclusión de las gráficas.
- `npuerto`: Establece el número de puerto, por defecto el puerto `5558`.
- `bbdd`: Guarda el nombre de la base de datos, por defecto "`motostudent`".
- `usuario`: Guarda el nombre de usuario de la base de datos, por defecto "`root`".

- `passwd`: Guarda la contraseña de usuario de la base de datos, por defecto "`motostudent`".
- `tablaActual`: Guarda la tabla de la base de datos que se está utilizando, por defecto la fecha de creación, con el formato "`tabla_yyyyMMdd_HHmm`".
- `kml`: Guarda el nombre del fichero `.kml`, por defecto "`motoUPCT`".

```
// <editor-fold defaultstate="collapsed" desc="DEFAULT OPTIONS">
        for(int i = 0; i<MetaDatos.gNum; i++){
            MetaDatos.gstatus[i] = false;
        }

        MetaDatos.status = false;
        MetaDatos.servidor_status = false;
        MetaDatos.bbdd_status = false;
        MetaDatos.gps_status = false;
        MetaDatos.tabla_status = false;
        MetaDatos.begin = true;
        MetaDatos.ejeX = 0;

        MetaDatos.npuerto = 5558;
        MetaDatos.bbdd = "motostudent";
        MetaDatos.usuario = "root";
        MetaDatos.passwd = "motostudent";
        MetaDatos.tablaActual = "tabla_" + MetaDatos.gNewNameTable(); // La tabla por
defecto
        MetaDatos.kml = "motoUPCT";
// </editor-fold>
```

Cuando la aplicación está en funcionamiento y queremos realizar algún cambio importante para la misma, como puede ser cambiar el puerto de comunicaciones o la base de datos a la que se conecta, la aplicación se desactivará y lanzará un mensaje informándonos de que la aplicación está desactivada.

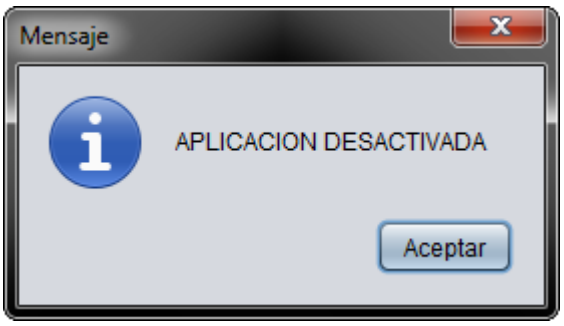


Figura 4.14. Mensaje de aviso

El código es el siguiente:

```
    public static void setDown(boolean adv){
        // Desactivamos el servidor en un nuevo Thread
        if(MetaDatos.status){
            MetaDatos.servidor.interrupt();
            for(int i = 0; i < MetaDatos.gNum; i++){
                MetaDatos.graficas[i].dispose();
                //gCheck[i].setSelected(false);
            }
            MetaDatos.status = false;
        }
        MetaDatos.begin = true;
        MetaDatos.ejeX = 0;
        MetaDatos.status = false;
        changeStatus.setText("Activar");
        if(adv)
            JOptionPane.showMessageDialog(new JOptionPane(),"APLICACION DESACTIVADA");
    }
```

Como podemos ver en el código, con cada cambio en el funcionamiento del programa modificamos también estas variables que definen el estado del programa.

4.9. OTRAS FUNCIONES DEL PROGRAMA

Para cada datagrama UDP recibido, capturamos el instante temporal en el que se produce, de modo que podemos comparar la diferencia de tiempo que hay en la transmisión de los datos utilizando los valores recibidos por el GPS y enviados al servidor, de este modo se hace necesaria una función que obtenga este valor de tiempo actual.

```
// <editor-fold defaultstate="collapsed" desc="TIMESTAMP">
    public static String gSimpleTime(String formato){
        // Formato completo --> yyyy-MM-dd HH:mm:ss:SSS
        DateFormat dateFormat = new SimpleDateFormat(formato);
       Calendar cal = Calendar.getInstance();
        String time = dateFormat.format(cal.getTime());
        return time;
    }
// </editor-fold>
```

De este modo, obtenemos una cadena de texto con el instante actual, lista para almacenar en la base de datos.

Aprovechando esta funcionalidad, se crea la función para generar el nombre por defecto a la nueva tabla en la base de datos, según el instante actual.

```
// <editor-fold defaultstate="collapsed" desc="TABLENAME">
    public static String gNewNameTable(){
        String n = gSimpleTime("yyyy/MM/dd HH:mm");
        String n1 = n.replace("/", "");
        String n2 = n1.replace(":","");
        return n2.replace(" ", "_");
    }
// </editor-fold>
```

4.10. DEPURACIÓN DEL PROGRAMA

Para esta aplicación se ha desarrollado dos sistemas independientes para depurar y mejorar el funcionamiento del programa, permitiendo localizar fallos o zonas de mejora en modo de depuración.

Se establecen dos variables booleanas, que activarán dichos modos cuando se desee comprobar el correcto funcionamiento de la aplicación, estas variables solo se pueden modificar en modo de programación, nunca en modo de ejecución de la aplicación.

```
// <editor-fold defaultstate="collapsed" desc="DEBUG">
    public static boolean debug = false;    // Variable para debuging
    public static boolean debugTime = false; // Variable de debug rendimiento
// </editor-fold>
```

La primera de las variables, debug, permitirá que la aplicación muestre por consola mensajes del estado en que se encuentra la aplicación o los errores que la misma produzca. A continuación se muestran algunos ejemplos:

Para mostrar un error en la aplicación:

```
try{
    ... // Código
}
catch(Exception e){
    if(MetaDatos.debug)
        e.printStackTrace();
}
```

En el proceso del servidor podemos encontrar el siguiente código que ejemplifica como vemos el estado de la aplicación:

```
try{
    serverSocket.receive(receivePacket);
    str = new String( receiveData, 0, receivePacket.getLength() );
    if(MetaDatos.debug){
        System.out.println("Esperamos al paquete de datos...");
    }
}catch(Exception e){
    if(MetaDatos.debug)
        e.printStackTrace();
}

if(MetaDatos.debug){
    System.out.println("Llega: \n\t" + str);
}
```

La segunda variable, `debugTime`, como bien indica su nombre, se utiliza para comprobar el rendimiento de la aplicación. Es muy necesario, ya que una aplicación lenta no podría recibir tantos paquetes de datos e incluso podría bloquearse, convirtiéndose así en una aplicación poco eficiente y casi improductiva teniendo encuentra el objetivo de la misma, que es la recogida del máximo número de datos proveniente de los sensores, por lo que necesitamos una aplicación ágil, rápida y eficiente.

Cuando activamos esta variable, mostramos por consola la duración de los procesos que nos interesan en milisegundos, para así tener una estima del rendimiento medio de la aplicación.

Para comenzar, es necesario precargar unas variables:

```
// Variables para debug de rendimiento
long start_time = 0;
long end_time = 0;
double difference = 0;
```

Cuando nos interese, comenzaremos a medir el tiempo que tarda en ejecutarse cierto código, para ello, guardamos el instante de tiempo. Una vez ejecutado dicho código, volvemos a guardar el instante de tiempo en el que se produce. Realizando una simple resta y convirtiendo el tiempo de nanosegundos a milisegundos obtenemos el tiempo de duración del fragmento del código, y por tanto del proceso que nos interesa:

```
if(MetaDatos.debugTime)
    start_time = System.nanoTime();

    ... // Código

if(MetaDatos.debugTime){
    end_time = System.nanoTime();
    difference = (end_time - start_time)/1e6;
    System.out.println("\n==============\nT_CODIGO:\t" + difference +
"ms.\n==============");
}
```

Podemos ver en una tabla los tiempos medios de duración de los diferentes puntos que se han medido en la aplicación:

Tiempo del procesado de datos	~ 0,3ms
Tiempo de almacenamiento en la base de datos	~ 60ms
Tiempo para mostrar las gráficas	~ 230ms
Tiempo del proceso completo	~ 260ms
Tiempo del proceso del paquete en el servidor	~ 300ms

Cabe recordar que se tratan de hilos de procesado independientes, también que el hilo del servidor solo depende del tiempo que tarda en llegarle un datagrama, por lo que podemos decir que el tiempo promedio son unos 260microsegundos de procesado de un datagrama de datos.

CAPÍTULO 5. "RESULTADOS Y EJEMPLOS"

Para el desarrollo y comprobación del correcto funcionamiento de la aplicación, se ha desarrollado otra aplicación que simula el comportamiento del microcontrolador *Arduino*[5], que es el encargado de adquirir y enviar los datos de los sensores para su procesado.

5.1. SIMULADOR DEL MICROPROCESADOR Y ENVIÓ DE DATOS

La idea de crear este simulador surge tras ver los inconvenientes de realizar las pruebas pertinentes de desarrollo del software de telemetría, ya sea por la necesidad de crear pruebas básicas, pruebas específicas o de rendimiento de la aplicación.

[5] **Arduino** es una plataforma de hardware libre, basada en una placa con un microcontrolador y un entorno de desarrollo, diseñada para facilitar el uso de la electrónica en proyectos multidisciplinares.

El simulador tiene la siguiente interfaz gráfica que lo hace muy versátil:

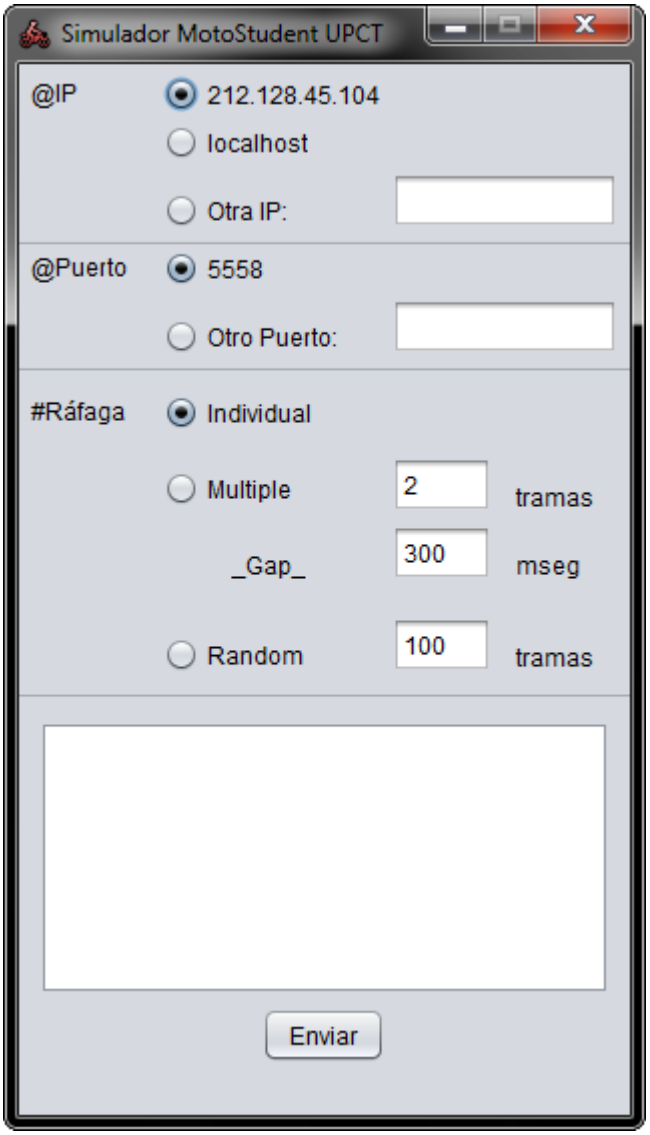


Figura 5.1. Menú simulador microcontrolador Arduino

En el primer bloque de opciones se puede elegir a que dirección IP enviar los datos, por defecto aparece la dirección IP en la que se encuentra el software de

telemetría. La segunda opción, *localhost*[6], es para realizar conexiones dentro de la misma máquina. En caso de querer realiza una conexión a otra máquina, se puede realizar seleccionando la tercera opción.

Al poder cambiar el puerto del servicio en el software de telemetría, se hace necesario para la comunicación que el simulador también pueda escoger a qué puerto dirigirse.

Por último, podemos elegir como se comporta la aplicación a la hora de enviar los datagramas de información simulados. Hay tres opciones:

- `Individual`: Tomará el texto que hay en el cuadro inferior y enviará dicha información a la dirección indicadas
- `Múltiple`: Permite enviar varios datagramas, tantos como aparezca en el recuadro que se indica y con un espaciado temporal constante en milisegundos.
- `Random`: Envía tantos datagramas de datos como se le indique, y genera de forma aleatoria los nuevos datagramas con la información, calculando exactamente su espaciado temporal que se aplicará en consecuencia.

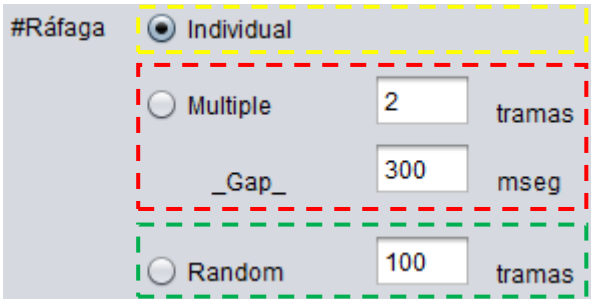


Figura 5.2. Detalle del tipo de modos de envío del simulador

[6]***Localhost***, en el contexto de redes *TCP/IP*, es un nombre reservado que tienen todas las computadoras, ratón o dispositivo independientemente de que disponga o no de una tarjeta de red Ethernet. El nombre *localhost* es traducido como la dirección IP de *loopback* 127.0.0.1 en IPv4, o como la dirección ::1 en IPv6.

El funcionamiento de la aplicación se muestra en el siguiente código.

```
// <editor-fold defaultstate="collapsed" desc="EVENTO">
    private void jButton1MouseClicked(java.awt.event.MouseEvent evt) {//GEN-
FIRST:event jButton1MouseClicked
        try{
            DatagramSocket clientSocket = new DatagramSocket();
            InetAddress IPAddress;
            String ip_send;
            if(defaultIP.isSelected())
                IPAddress = InetAddress.getByName("212.128.45.104");
            else{
                if(localIP.isSelected())
                    IPAddress = InetAddress.getByName("127.0.0.1");
                else{
                    ip_send = IPField.getText();
                    IPAddress = InetAddress.getByName(ip_send);
                }
            }

            byte[] sendData = new byte[512];

            int port_send;
            if(defaultPort.isSelected())
                port_send = 5558;
            else{
                port_send = Integer.parseInt(portField.getText());
            }

            if (!rafagaRandom.isSelected()){
                int iteraciones = 0;
                int duration = 10; // Duracion del sleep de 10ms para una sola trama
                if(rafagaIndividual.isSelected())
                    iteraciones = 1;
                else{
                    iteraciones = Integer.parseInt(nTramaMultiple.getText());
                    duration = Integer.parseInt(t_gapMultiple.getText());
                }
                String texto = textSend.getText();

                sendData = texto.getBytes();
                DatagramPacket sendPacket;
                for(int i = 0; i<iteraciones; i++){
                    sendPacket= new DatagramPacket(sendData, sendData.length,
IPAddress, port_send);
                    clientSocket.send(sendPacket);
                    try {
                        Thread.sleep(duration); // Dormimos (duration)ms
                    }catch(InterruptedException ex) {
                        Thread.currentThread().interrupt();
                    }
                }
                clientSocket.close();
            }
            else{
                String texto;
                int time_ms;
                DatagramPacket sendPacket;
                int iteraciones = Integer.parseInt(nTramaRandom.getText());
                for(int i=0; i<iteraciones; i++)
                {
                    texto = getInformation((i%10)==9,(i%10)==9);
                    time_ms = contTime(texto, (i%10)==9, (i%10)==9);

                    sendData = texto.getBytes();
```

```
                    try {
                        Thread.sleep(time_ms); // Dormimos (time_ms)ms
                    }catch(InterruptedException ex) {
                        Thread.currentThread().interrupt();
                    }

                    sendPacket= new DatagramPacket(sendData, sendData.length,
IPAddress, port_send);
                    clientSocket.send(sendPacket);
                }
            }
            textSend.setText("");
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }//GEN-LAST:event_jButton1MouseClicked
// </editor-fold>
```

En general, para que los datagramas que no son aleatorios, la única complejidad está en la espera del espaciado temporal entre datagramas. Este se realiza cambiando el estado del hilo de programación a modo `sleep` el tiempo que deseemos:

```
try {
    Thread.sleep(time_ms); // Dormimos (time_ms)ms
}catch(InterruptedException ex) {
    Thread.currentThread().interrupt();
}
```

Cuando se desea realizar el envió de datagramas aleatorios éstos se generan conforme a la estructura de una trama, dejando solo a la aleatoriedad los resultados de los sensores. También se ha simulado la periodicidad de los datagramas que incluyen información del GPS y del sensor de temperatura, ya que estos sensores tienen una velocidad de adquisición distinta a los demás, por lo que se transmiten con menos frecuencia.

Por otro lado, las subtramas que llevan información acerca del GPS tardan en capturarse en torno a los 40milisegundos, mientras que un datagrama normal, con el resto de sensores, está en torno a los 11milisegundos. Estas condiciones también se han incluido dentro de las opciones del datagrama aleatorio.

```
// <editor-fold defaultstate="collapsed" desc="SENSORES">
    public static String getInformation(boolean GPS, boolean Temp){
        String stream = "";
        int pos = 0;
        if(GPS)
        {
            stream += getGPS();
            pos += 45;
        }

        if(Temp)
        {
            stream += "f" + randomValue(0,127);
            pos += 2;
```

```
        }

        if(GPS){
            stream += "a" + randomValue(40,45);
        }

        while(true){
            if(pos + 12 < 500){
                if(!GPS)
                    stream += "a" + randomValue(9,12);
                GPS = false;
                stream += "b" + randomValue(0,127);
                stream += "c" + randomValue(0,127);
                stream += "d" + randomValue(0,127);
                stream += "e" + randomValue(0,127);
                stream += "h" + randomValue(0,127);
                pos += 12;
            }
            else
            {
                for(int i = pos; i<500; i++)
                    stream += "#";
                break;
            }
        }

        return stream;
    }
// </editor-fold>
```

Cuando creamos la trama GPS, localizaremos la posición de la moto en un entorno cercano, pero variando las coordenadas de la misma, de modo que simule el comportamiento de la moto como si estuviera en un circuito de competición.

```
// <editor-fold defaultstate="collapsed" desc="GPS">
    public static String getGPS(){
        //"g3758.7nnnnn,N,00107.8nnnnn,W,ddMMyy,HHmmss.0
        //<coordinates>-1.0339898,37.6432535,0</coordinates>
        DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        Calendar cal = Calendar.getInstance();
        String t = dateFormat.format(cal.getTime());

        String time = t.substring(8,10) + t.substring(5,7) + t.substring(2,4)
                + "," + t.substring(11,13) + t.substring(14,16) + t.substring(17,19);

        String str = "g3738.6" + randomValueGPS() + ",N,00102.0"
                + randomValueGPS()+ ",W," + time + ".0";

        return str;
    }
// </editor-fold>
```

Por otro lado, una vez generada la trama completa, es necesario saber cuánto tiempo habría tenido el microcontrolador en generarla, por ello calculamos el valor que ha tardado cada subtrama en generarse, según el carácter después del carácter '`a`', como se indicaba en apartado "4.3. El formato de los datos".

```
// <editor-fold defaultstate="collapsed" desc="COUNTER TIME">
    public static int contTime(String stream, boolean GPS, boolean temp){
        int pos = 0;
        int time = 0;
        if(GPS)
            pos += 45;
        if(temp)
            pos += 2;
        for(int i = pos; i<500; i += 12)
            if(stream.charAt(i)== 'a')
               time += stream.charAt(i+1);

        return time;
    }
// </editor-fold>
```

Por último, se han creado dos funciones para generar los valores aleatorios, la primera de ella genera un carácter aleatorio dado un intervalo de números enteros. La segunda de ellas genera un `string` de 5 caracteres numéricos para terminar de establecer la posición GPS de la moto de competición.

```
// <editor-fold defaultstate="collapsed" desc="RAND_CHAR">
    public static char randomValue(int min, int max){
        Random r = new Random();
        char c = (char)(r.nextInt(max-min) + min);
        return c;
    }
// </editor-fold>

// <editor-fold defaultstate="collapsed" desc="RAND_STRING">
    public static String randomValueGPS(){
        Random r = new Random();
        String alphabet = "0123456789";
        String str = "";
        for (int i = 0; i < 5; i++)
            str += alphabet.charAt(r.nextInt(alphabet.length()));
        return str;
    }
// </editor-fold>
```

5.2. SIMULACIÓN Y RESULTADOS

Para comenzar con la simulación, se han mandado 20 datagramas aleatorios para poder visualizar las gráficas con un poco más de detalle, ya que hay que recordar de que en esta simulación los datos que mandamos son aleatorios y no tienen sentido acorde con los sensores, pero aún así son resultados correctos.

Las gráficas resultantes han sido:

RPM RUEDA 1

Representa las vueltas por minuto de la primera de las ruedas.

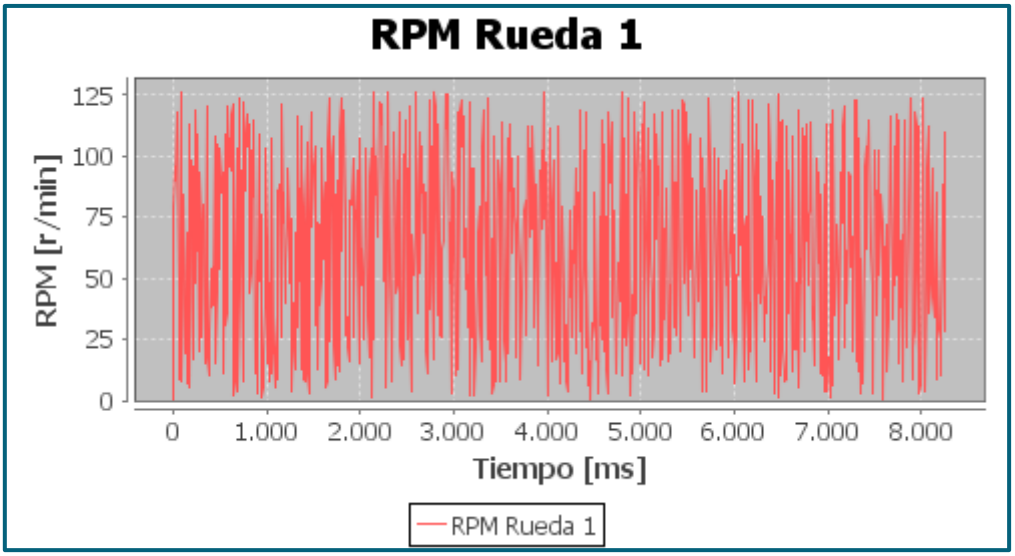


Figura 5.3. Gráfica "RPM Rueda 1" tras 20 datagramas de datos

RPM RUEDA 2

Representa las vueltas por minuto de la segunda de las ruedas.

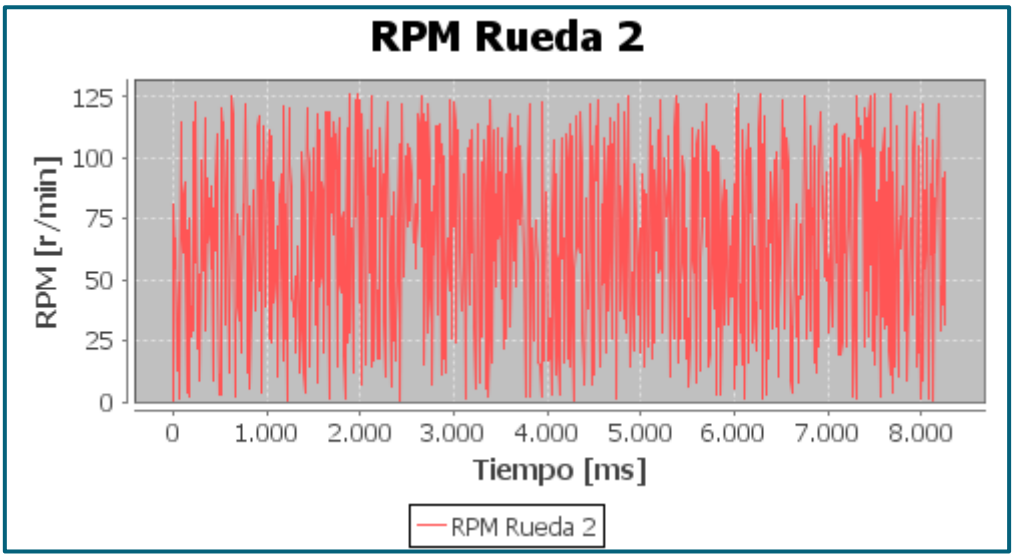


Figura 5.4. Gráfica "RPM Rueda 2" tras 20 datagramas de datos

ACELERÓMETRO

Representa el ángulo de inclinación de la moto.

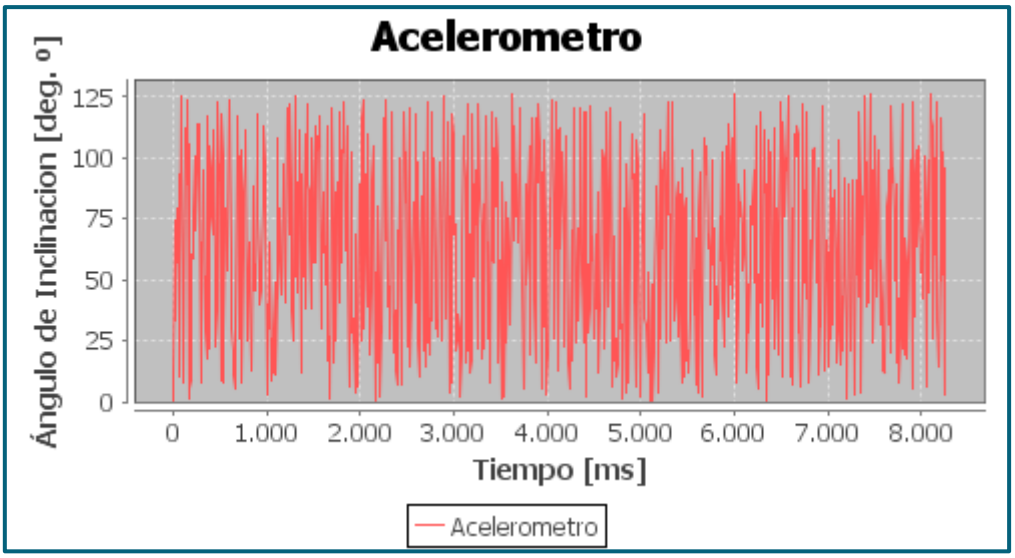


Figura 5.5. Gráfica "Acelerómetro" tras 20 datagramas de datos

POTENCIÓMETRO LINEAL

Representa el recorrido de la suspensión. Habrá que transformar el valor obtenido en milivoltios en distancia con la función de conversión adecuada.

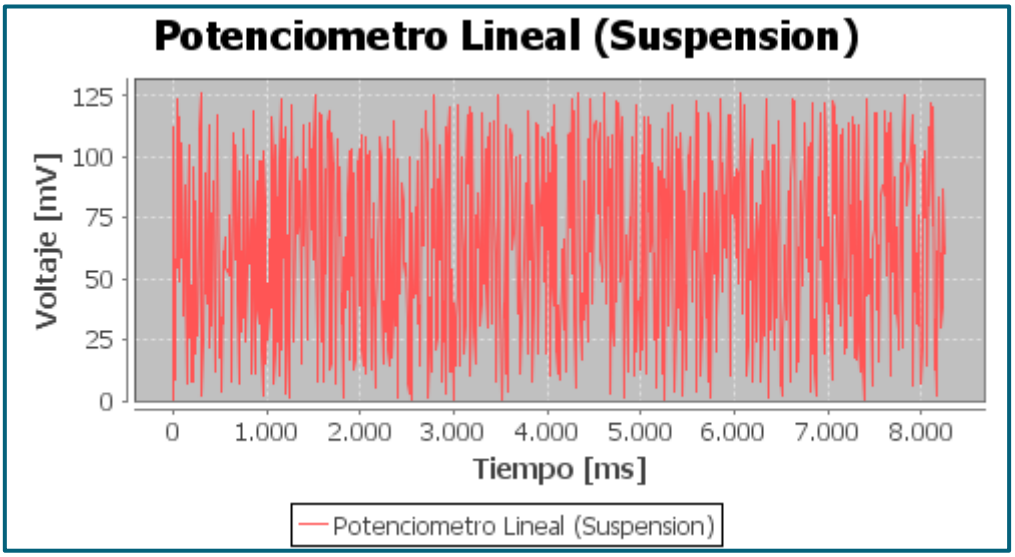


Figura 5.6. Gráfica "Potenciómetro Lineal" tras 20 datagramas de datos

POTENCIÓMETRO RADIAL

Representa el recorrido de giro del puño de acelerador de la moto. Habrá que transformar el valor obtenido en milivoltios en distancia con la función de conversión adecuada.

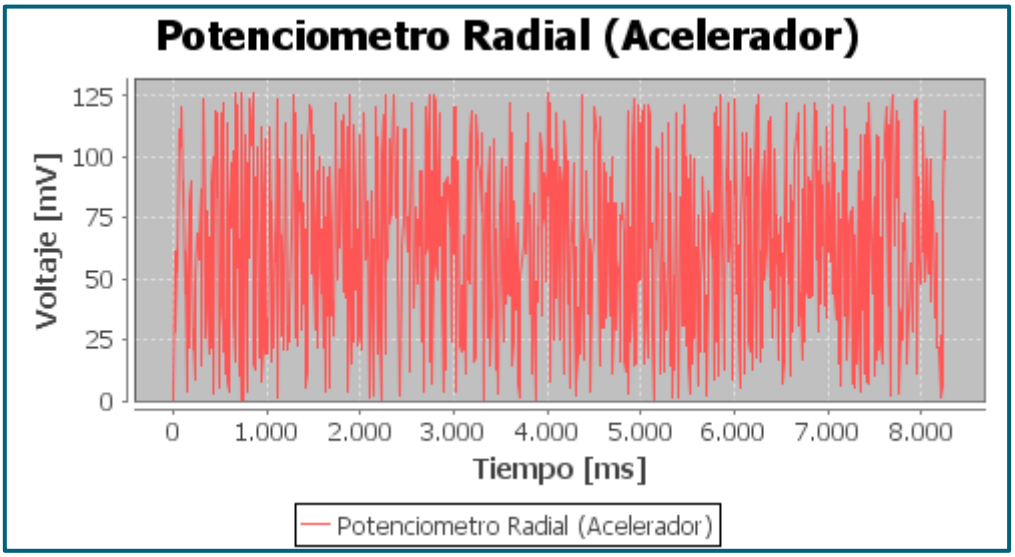


Figura 5.7. Gráfica "Potenciómetro Radial" tras 20 datagramas de datos

TEMPERATURA

Representa la temperatura del motor u otro elemento. Como este parámetro varía con menor frecuencia, la cantidad de valores obtenidos también es menor.

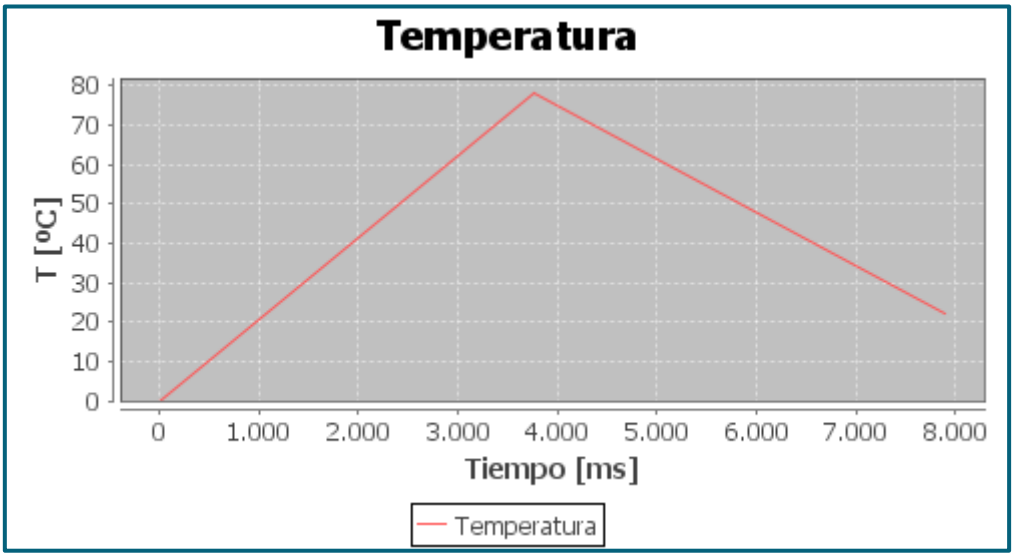


Figura 5.8. Gráfica "Temperatura" tras 20 datagramas de datos

Como se observa en las gráficas anteriores, vemos que para 20 tramas de datos, cada una de las tramas ha tardado de media un poco menos de 400milisegundos en enviarse.

GPS

También se ha obtenido la posición de la moto, que ha variado conforme transcurría el programa, en total, con 20 tramas ha variado dos veces la posición.

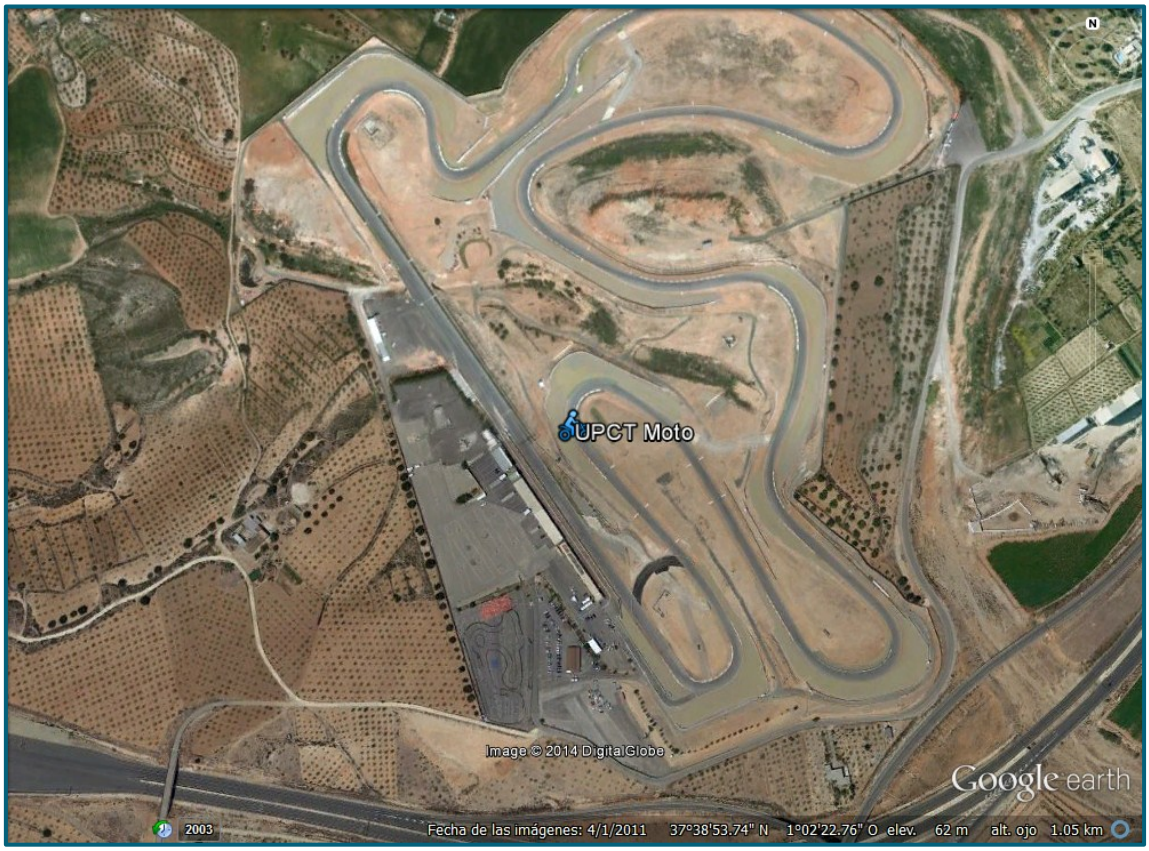


Figura 5.9. Posición en GoogleEarth tras 20 datagramas de datos

Continuando con el ejemplo anterior, se realizan un total de 200 datagramas de datos, 180 tramas más, visualizándose así las gráficas.

RPM RUEDA 1

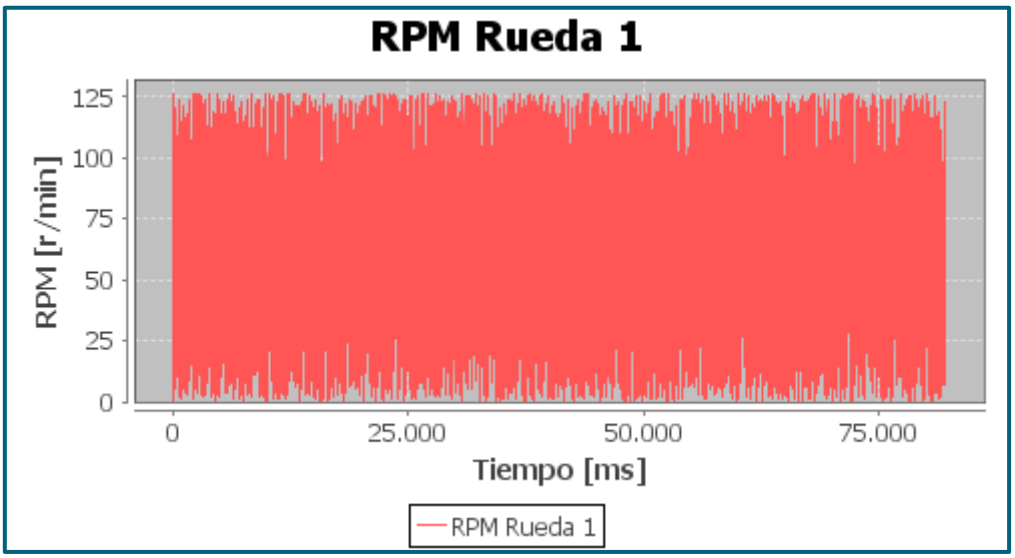


Figura 5.10. Gráfica "RPM Rueda 1" tras 200 datagramas de datos

RPM RUEDA 2

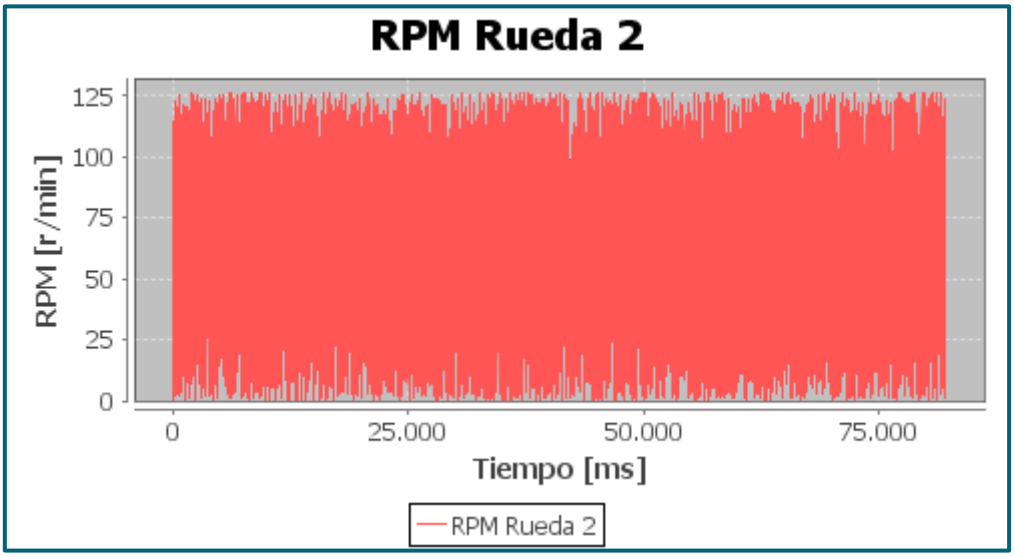


Figura 5.11. Gráfica "RPM Rueda 2" tras 200 datagramas de datos

ACELERÓMETRO

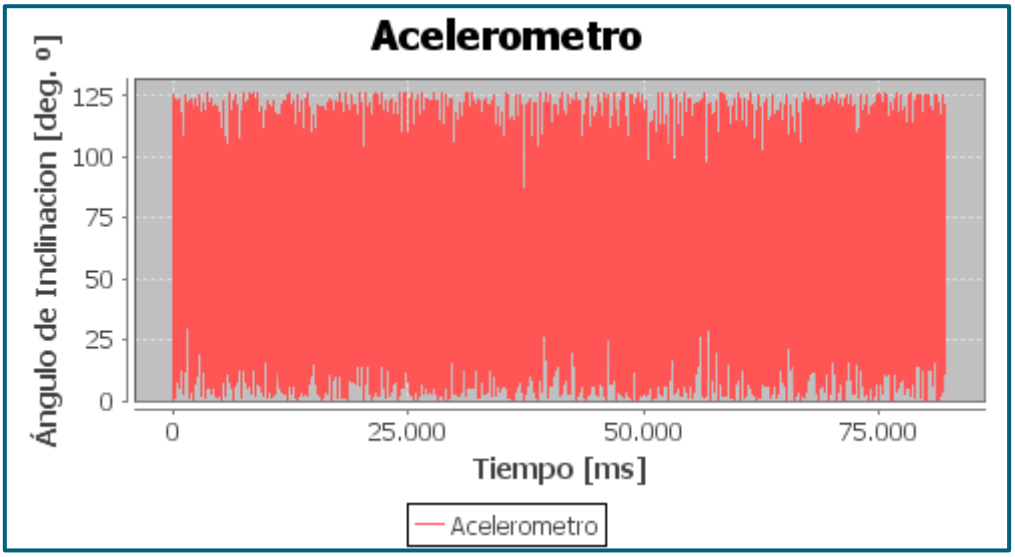


Figura 5.12. Gráfica "Acelerómetro" tras 200 datagramas de datos

POTENCIÓMETRO LINEAL

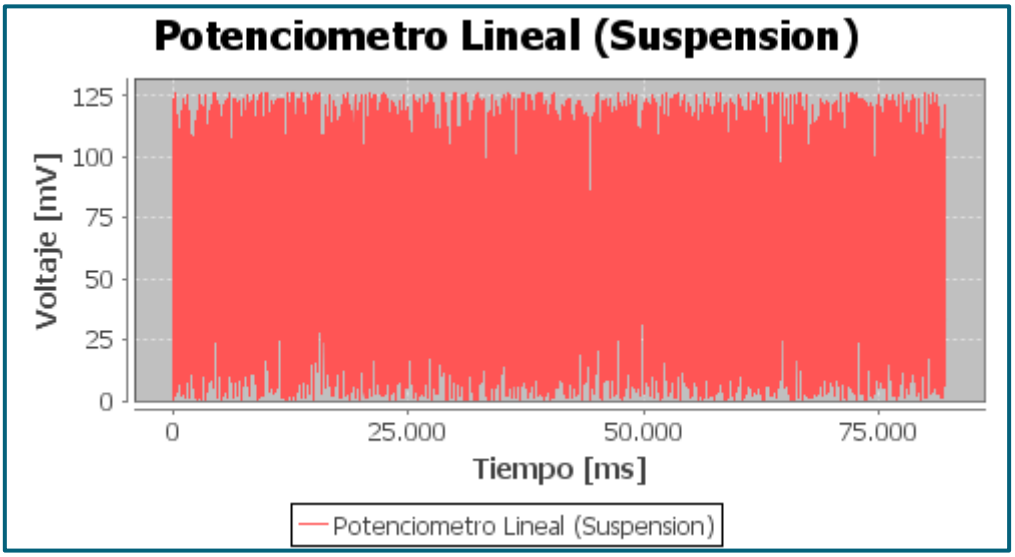


Figura 5.13. Gráfica "Potenciómetro Lineal" tras 200 datagramas de datos

POTENCIÓMETRO RADIAL

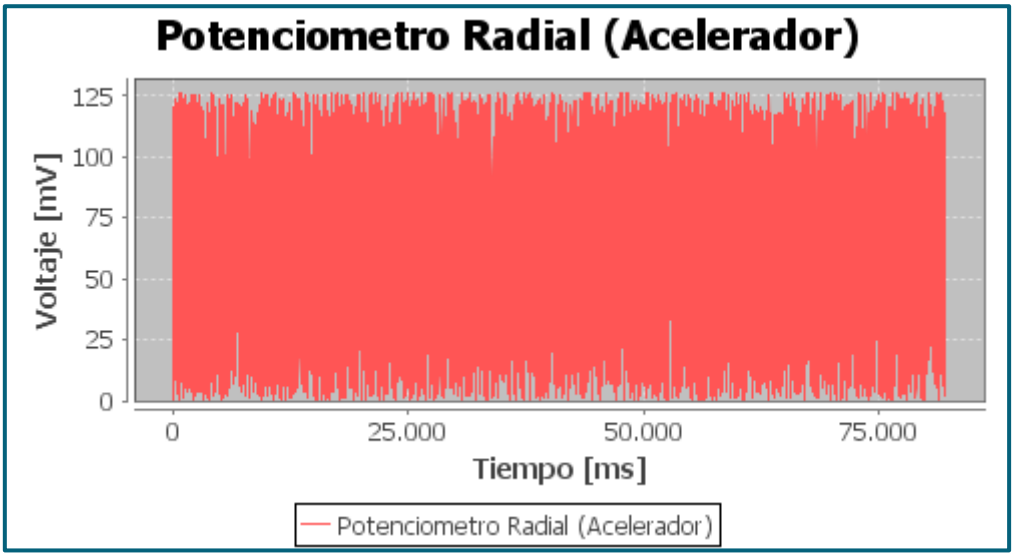


Figura 5.14. Gráfica "Potenciómetro Radial" tras 200 datagramas de datos

TEMPERATURA

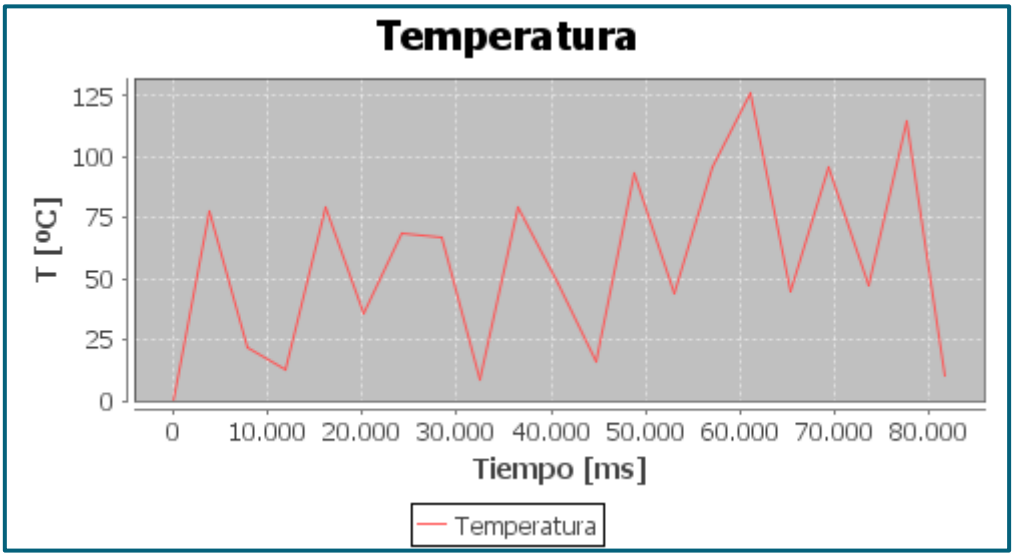


Figura 5.15. Gráfica "Temepratura" tras 200 datagramas de datos

En esta extensión de la simulación, con un total de 200 tramas de datos, cada una tarda en enviarse otros 400milisegundos de media.

GPS

Al igual que en la anterior simulación, también se ha obtenido la posición de la moto, que ha variado conforme transcurría el programa, en total, con 200 tramas ha variado veinte veces la posición.

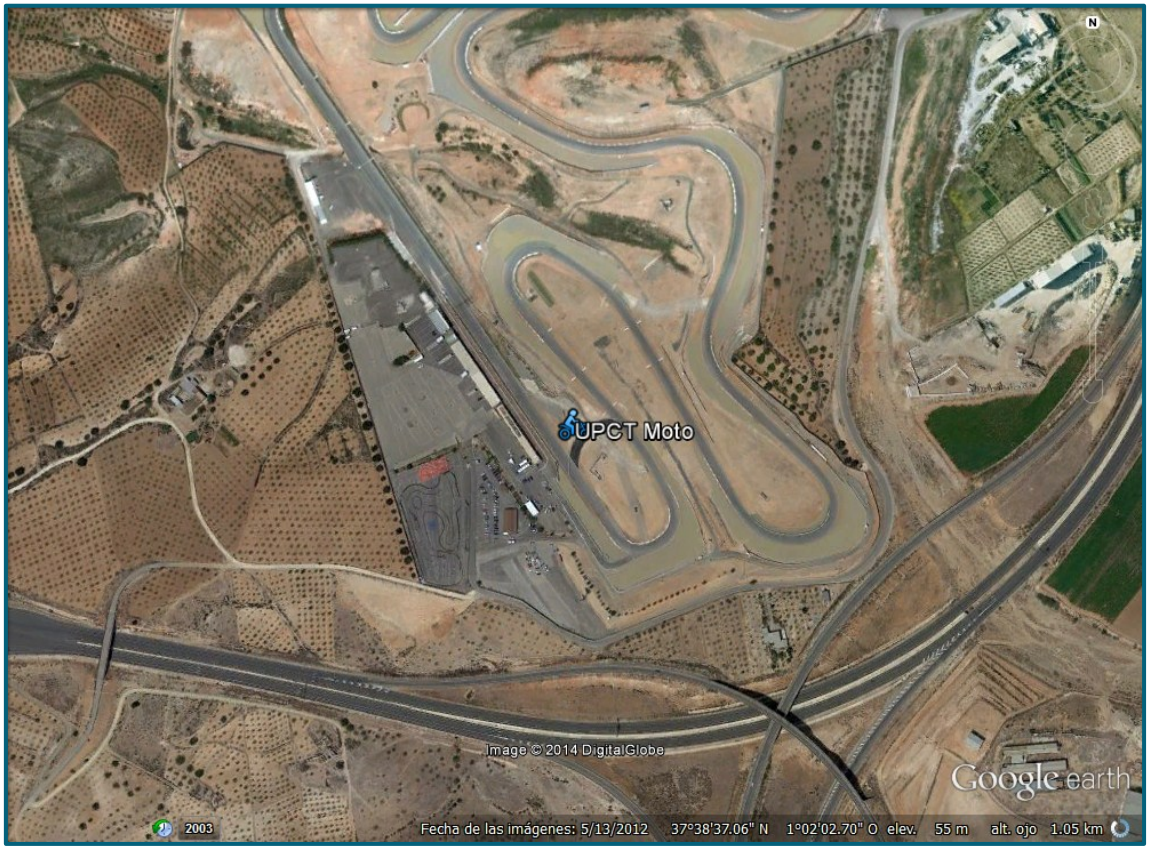
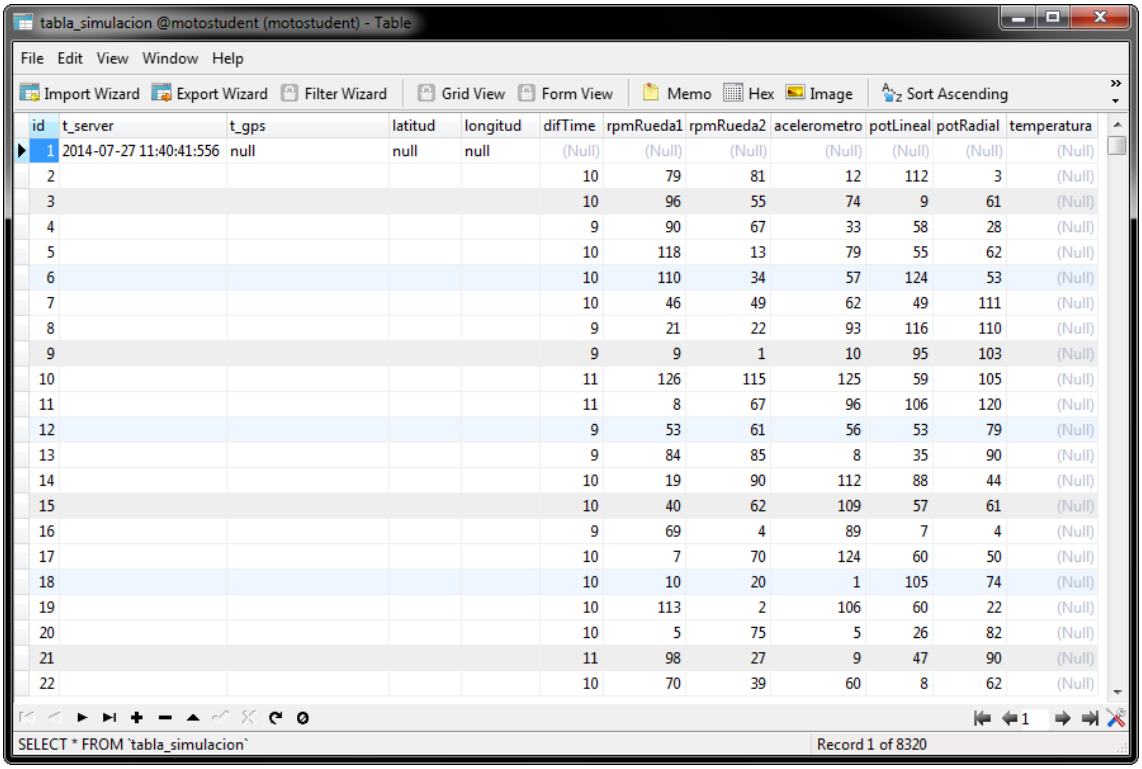


Figura 5.16. Posición en GoogleEarth tras 200 datagramas de datos

Procedemos a comparar los datos almacenados en la base de datos a los exportados a los documentos Hoja de Excel y a un fichero de texto plano:

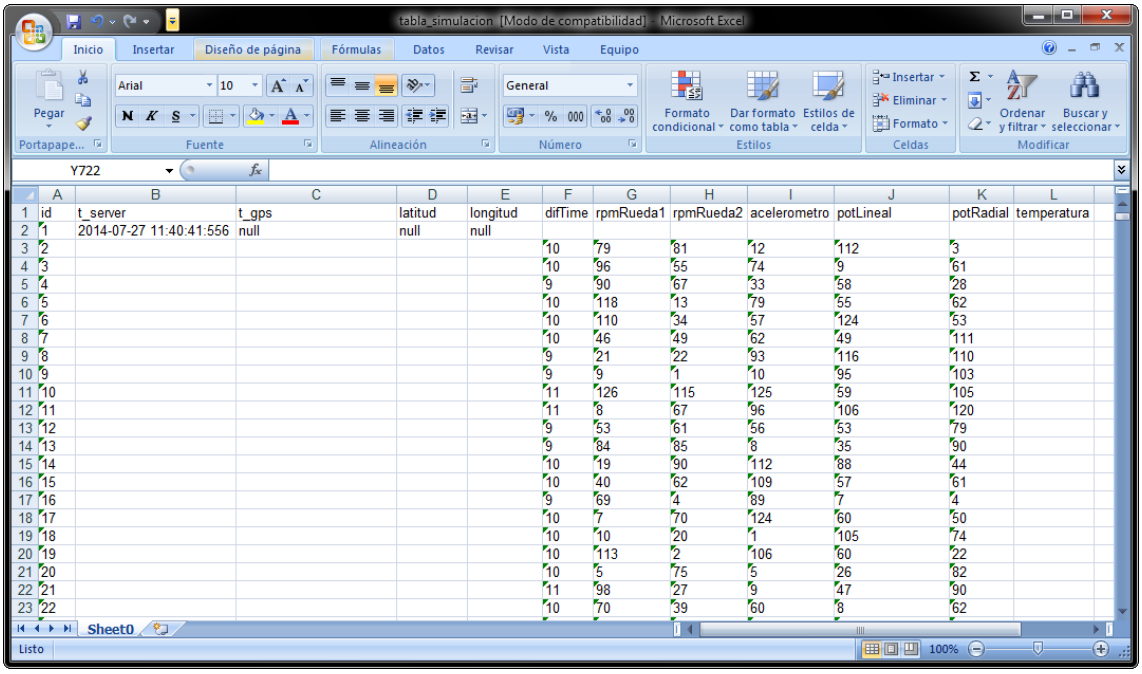
BASE DE DATOS



id	t_server	t_gps	latitud	longitud	difTime	rpmRueda1	rpmRueda2	acelerometro	potLineal	potRadial	temperatura
1	2014-07-27 11:40:41:556	null	null	null	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)
2					10	79	81	12	112	3	(Null)
3					10	96	55	74	9	61	(Null)
4					9	90	67	33	58	28	(Null)
5					10	118	13	79	55	62	(Null)
6					10	110	34	57	124	53	(Null)
7					10	46	49	62	49	111	(Null)
8					9	21	22	93	116	110	(Null)
9					9	9	1	10	95	103	(Null)
10					11	126	115	125	59	105	(Null)
11					11	8	67	96	106	120	(Null)
12					9	53	61	56	53	79	(Null)
13					9	84	85	8	35	90	(Null)
14					10	19	90	112	88	44	(Null)
15					10	40	62	109	57	61	(Null)
16					9	69	4	89	7	4	(Null)
17					10	7	70	124	60	50	(Null)
18					10	10	20	1	105	74	(Null)
19					10	113	2	106	60	22	(Null)
20					10	5	75	5	26	82	(Null)
21					11	98	27	9	47	90	(Null)
22					10	70	39	60	8	62	(Null)

Figura 5.17. Base de datos tras 200 datagramas de datos

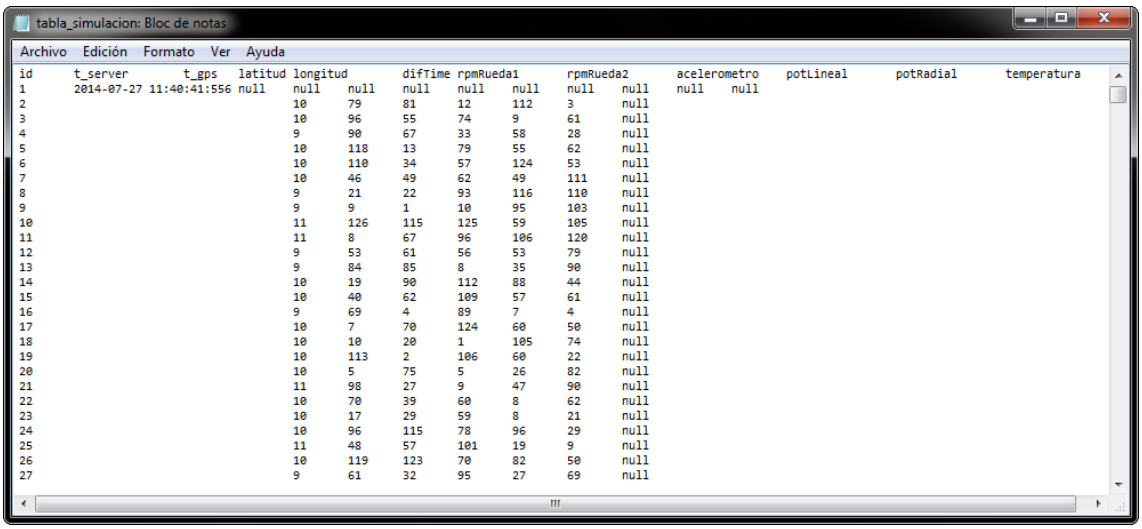
HOJA DE CÁLCULO EXCEL



id	t_server	t_gps	latitud	longitud	difTime	rpmRueda1	rpmRueda2	acelerometro	potLineal	potRadial	temperatura
1	2014-07-27 11:40:41:556	null	null	null							
2					10	79	81	12	112	3	
3					10	96	55	74	9	61	
4					9	90	67	33	58	28	
5					10	118	13	79	55	62	
6					10	110	34	57	124	53	
7					10	46	49	62	49	111	
8					9	21	22	93	116	110	
9					9	9	1	10	95	103	
10					11	126	115	125	59	105	
11					11	8	67	96	106	120	
12					9	53	61	56	53	79	
13					9	84	85	8	35	90	
14					10	19	90	112	88	44	
15					10	40	62	109	57	61	
16					9	69	4	89	7	4	
17					10	7	70	124	60	50	
18					10	10	20	1	105	74	
19					10	113	2	106	60	22	
20					10	5	75	5	26	82	
21					11	98	27	9	47	90	
22					10	70	39	60	8	62	

Figura 5.18. Hoja de Cálculo tras 200 datagramas de datos

FICHERO DE TEXTO PLANO:



id	t_server	t_gps	latitud	longitud	difTime	rpmRueda1	rpmRueda2	acelerometro	potLineal	potRadial	temperatura
1	2014-07-27 11:40:41:556	null	null	null	null	null	null	null	null		
2				10	79	81	12	112	3	null	
3				10	96	55	74	9	61	null	
4				9	90	67	33	58	28	null	
5				10	118	13	79	55	62	null	
6				10	110	34	57	124	53	null	
7				10	46	49	62	49	111	null	
8				9	21	22	93	116	110	null	
9				9	9	1	10	95	103	null	
10				11	126	115	125	59	105	null	
11				11	8	67	96	106	120	null	
12				9	53	61	56	53	79	null	
13				9	84	85	8	35	90	null	
14				10	19	90	112	88	44	null	
15				10	40	62	109	57	61	null	
16				9	69	4	89	7	4	null	
17				10	7	70	124	60	50	null	
18				10	10	20	1	105	74	null	
19				10	113	2	106	60	22	null	
20				10	5	75	5	26	82	null	
21				11	98	27	9	47	90	null	
22				10	70	39	60	8	62	null	
23				10	17	29	59	8	21	null	
24				10	96	115	78	96	29	null	
25				11	48	57	101	19	9	null	
26				10	119	123	70	82	50	null	
27				9	61	32	95	27	69	null	

Figura 5.19. Texto plano tras 200 datagramas de datos

Como podemos observar, al exportar en ambos casos obtenemos los mismos resultados almacenados en la base de datos. Aunque no se observe en las capturas realizadas, tanto en la Hoja de Cálculo como en el fichero de texto plano aparecen los 8320 registros de la tabla, con los cuales podremos trabajar con otras aplicaciones que lo deseen.

CAPÍTULO 6. "CONCLUSIONES Y FUTUROS TRABAJOS"

En este trabajo se ha podido observar cómo desarrollar de manera sencilla un software que nos permita gestionar un sistema de telemetría básico.

El sistema es utilizable por todos los equipos, siempre que se configure adecuadamente las condiciones de transmisión, como la codificación de la información y la adaptación de la misma a las condiciones que sean necesarias.

Después de gestionar las necesidades básicas para la moto de competición, en una línea futura es posible afinar dichos sistemas, adecuando la representación de los datos en gráficas más aptas para su visualización, así como la posibilidad de añadir funciones matemáticas que faciliten la toma de decisiones en tiempo real en función de los ratos recibidos desde los sensores instalados en la moto.

Figura 6.1. Ejemplo gráficas de telemetría en competición [H]

Por otro lado, en función de mejorar el funcionamiento de la aplicación, es posible aumentar el rendimiento de la posición GPS, precargando la ruta que la moto ha de trazar en un circuito dado, ofreciendo la posibilidad de analizar las trayectorias que realiza el vehículo superponiendo los datos registrados del GPS sobre *GoogleEarth*.

En esta aplicación, no estamos limitados a un circuito o una zona concreta de un mapa, si no que somos libres de situar la moto en cualquier parte del mundo, lo cual es una ventaja de este proyecto, sin embargo, una opción a futuro sería añadir esta opción a elegir, pues puede ser de mayor utilidad para la visualización y representación de los datos.

Figura 6.2. Ejemplo posicionamiento GPS en competición [H]

Por último, una mejora en el rendimiento de la aplicación sería implementar la parte de control del servidor y de los datos que reciben en código C[7], el cual su ejecución es más veloz, mientras que la interfaz gráfica se mantendría en un lenguaje multiplataforma como es JAVA.

[7] **Código C**, es un lenguaje orientado a la implementación de Sistemas Operativos, concretamente Unix. C es apreciado por la eficiencia del código que produce y es el lenguaje de programación más popular para crear software de sistemas, aunque también se utiliza para crear aplicaciones.

CAPÍTULO 7. "BIBLIOGRAFÍA"

[1] Información sobre *MotoStudent*

http://www.motostudent.com/

[2] Documentación *KML* (*Keyhole Markup Language*).

https://developers.google.com/kml/documentation/?hl=ES

[3] Página oficial de GoogleEarth.

http://www.google.es/intl/es/earth/index.html

[4] Documentación API Google Maps.

https://developers.google.com/maps/documentation/javascript/tutorial?hl=es-ES

[5] http://www.radiocomunicaciones.net/telemetria.html

[6] Carden, F., Jedlicka, R. P., Henry, R. , "Telemetry Systems Engineering", Artech House, 2002.

[7] Robinson, W., "Improvement in Electric Signaling Apparatus for Railroads", U.S. Patent No. 130661, Brooklyn, NY, August 20, 1872.

[8] United States National Weather Service, Division of the National Oceanic and Atmosphere Administration, January 3, 2001.

[9] http://www.gadgetos.com/noticias/telemetria-formula-1/

[10] http://www.a2r.com/la-telemetria-de-una-motogp-6413.html

[11] Tema 1_Introduccion a la domótica_V10_pdf, Universidad Politécnica de Cartagena, 2014.

[12] http://robotica.wordpress.com/about/

[13] http://www.enfermeriaencardiologia.com/revista/res2904.html

[14] A. Tourón, O. Radulovich, M. Aguero, l. Fidalgo, D. Krygier, M. Kovalsky, A. Hnilo, P. Diodati, "Short pulse solid state laser for telemetry".

[15] Pedro Celestino López Jiménez, "Desarrollo de un dispositivo de telemetría basado en la plataforma Arduino y Shield 3G + GPS", Universidad Politécnica de Cartagena, 2014.

[16] María Belén Pérez Muñoz, "Implementación de un sistema de radiocomunicaciones para la transmisión de la telemetría de una moto de carreras", Universidad Politécnica de Cartagena, 2014.

[17] Documentación NetBeans.

https://netbeans.org

[18] Documentación MySQL Server.

http://www.mysql.com

[19] Documentación Navicat.

http://www.navicat.com/

CAPÍTULO 8. "LISTA DE FIGURAS"

[A] Figura 2.1

http://cienciadebolsillo.com/historia/regulador-centrifugo-watt-revolucion-industrial-automata/gmx-niv30-con179.htm

[B] Figura 2.2

http://es.wikipedia.org/wiki/Circuito_de_vía

[C] Figura 2.3, 2.4, 2.5

http://www.gadgetos.com/noticias/telemetria-formula-1/

[D] Figura 2.6

http://www.a2r.com/la-telemetria-de-una-motogp-6413.html

[E] Figura 2.7

http://mercantilelectrico.blogspot.com.es/2012_11_01_archive.html

[F] Figura 2.8

http://img.alibaba.com/img/pb/816/412/236/1267688719040_hz_fileserver3_232456.jpg

[G] Figura 2.9

A. Tourón, O. Radulovich, M. Aguero, l. Fidalgo, D. Krygier, M. Kovalsky, A. Hnilo, P. Diodati, “Short pulse solid state laser for telemetry.”

[H] Figura 6.1, 6.2

http://www.csrmotorsport.com/2d-software/

ANEXO 1. GUÍA DE LA APLICACIÓN

Al hacer doble clip sobre el fichero `ProyectoMotoStudent.jar`, aparecerá la ventana principal del programa.

Figura A. 1. Ventana principal del programa MotoStudent UPCT

Para comenzar la configuración del programa se recomienda seguir la siguiente secuencia de pasos:

1. Acceder al menú `Visualizar`, y seleccionar aquellos parámetros que se deseen mostrar por pantalla. La selección de los mismos no implica que los no seleccionados no se almacenen en la base de datos.

Figura A. 2. Menú de selección de los parámetros a visualizar

2. Dentro del menú `Configuración`, se han de realizar varios pasos:

Figura A. 3. Menú de configuración de la aplicación

a. En el menú `Configuración > Servidor`, seleccionar número de puerto para que se establezca la configuración.

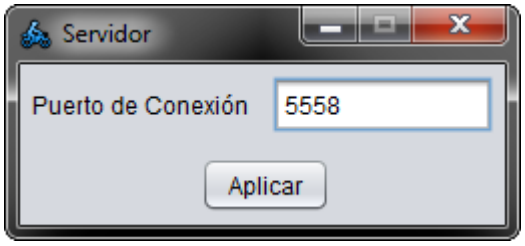


Figura A. 4. Configuración del puerto del servidor

b. Para configurar la base de datos, primero se ha de seleccionar la misma, a través del menú que está en `Configuración > Base de Datos`. Debemos de introducir el nombre de la base de datos y un usuario y contraseña que esté habilitado a utilizarla.

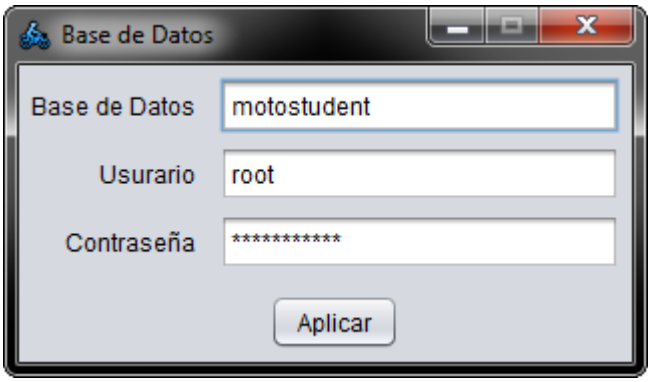


Figura A. 5. Menú de configuración de la base de datos

A continuación, seleccionaremos la tabla donde guardar los datos recibidos. La tabla puede ya encontrarse creada en el la base de datos así como se puede crear una nueva tabla. Por defecto se genera un nombre de tabla en realción a la fecha actual.

Figura A. 6. Menú para seleccionar el nombre de la tabla

c. Por último, si se desea, desde el menú `Configuración > GPS`, se seleccionará el nombre del fichero *.kml* que se generará con cada posición de la moto (notar que la aplicación sobreescribirá este fichero con cada nuevo dato recibido desde el GPS).

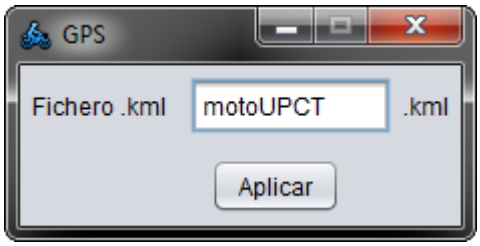


Figura A. 7. Menú para seleccionar el nombre del fichero KML

3. Por útlimo, desde el menú `Configuración` realizaremos la siguiente secuencia:
   a. Activar el servidor: `Configuración > Activar Servidor`.
   b. Activar la base de datos: `Configuración > Activar Base de Datos`.
   c. (Opcional) Activar el GPS: `Configuración > Activar GPS`.
   d. Activar la aplicación para que comience realmente la ejecución del programa: `Configuración > Activar`.

En el transcurso de la aplicación, podremos modificar qué parámetros deseamos visualizar sin parar la aplicación, simplemente entrando en el menú `Visualizar`.

Si deseamos exportar los datos de una tabla previamente almacenada o la misma que se está ejecutando, recomendamos detener la aplicación (`Configuración > Desactivar`). A continuación, seleccionando `Archivo > Exportar datos a…` podemos seleccionar el tipo de fichero a exportar.

Figura A. 8. Despegable del menú Archivo

La aplicación nos permite exportar la tabla actual a una Hoja de Cálculo o a un archivo en texto plano.

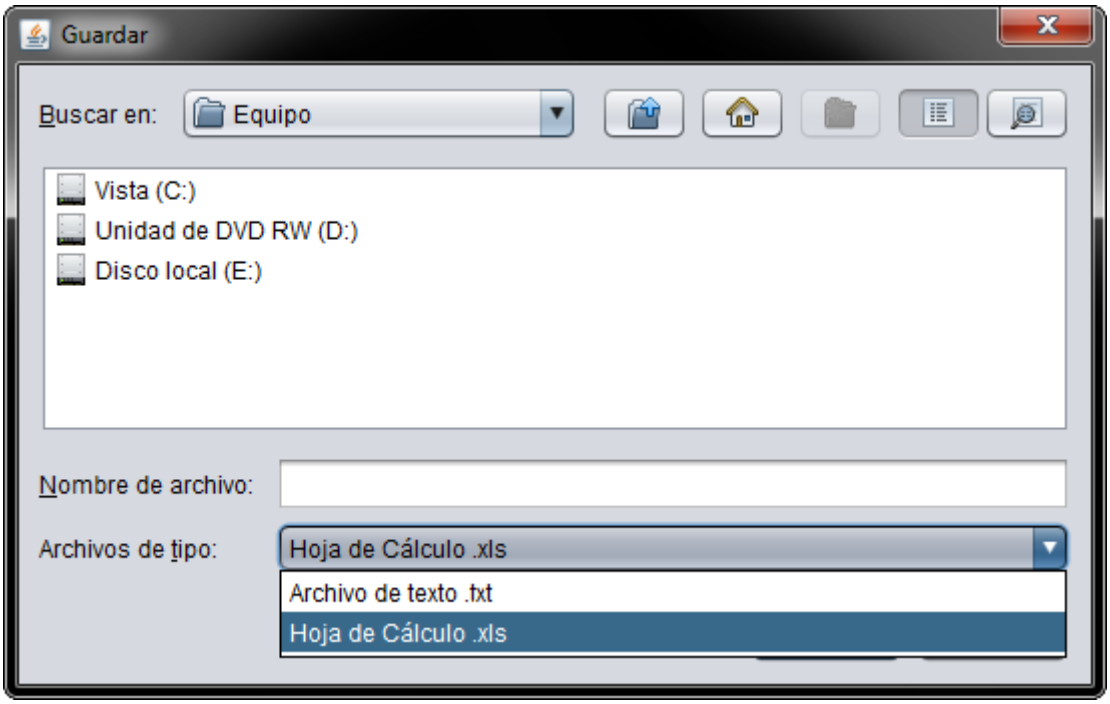


Figura A. 9. Menú para exportar la base de datos actual

Todas estas indicaciones del uso del programa se encuentran incluidas en el menú `Ayuda > Uso` de la aplicación, tal y como se puede ver en la Figura 4.12.