ESCUELA TÉCNICA SUPERIOR de INGENIERÍA del DISEÑO

UNIVERSIDAD POLITÉCNICA de VALENCIA

2º Master en Ingeniería Aeroespacial 2018/2019

Trabajo Final de Master

Desarrollo de software para adquisición y tratamiento de datos de un sistema multisensor embarcado en un USV

Realizado por:

Antonio Aledo Portugués

Tutorizado por:

Ángel Rodas Jordá

Fecha de entrega:

Valencia, 12 de septiembre 2019

RESUMEN:

El objeto del presente trabajo será a extraer e interpretar datos de las mediciones de posición de un **GPS** y de una **EcoSonda** mediante un programa desarrollado en lenguaje **Java** en el entorno de NetBeans para obtener un modelo digital de elevaciones del terreno (**MDT**). Este modelo representa la distribución espacial de la elevación de la superficie estudiada proyectada sobre un sistema de referencia terrestre, en concreto ETRS89, y es imprescindible para la correcta planificación de proyectos de ingeniería civil. Las superficies a medir serán cualquier tipo de formación de suelo que pueda haber debajo de una masa de agua.

Estas mediciones se realizan con un equipo montado en un barco o en un **USV** (Unmanned Surface Vehicle) como será nuestro caso, también se pueden realizar con imágenes satélite o datos LIDAR, pero estas técnicas son menos precisas. Con el tipo de sensor que tenemos, una **EcoSonda** monohaz, seremos capaces de medir profundidades que junto con las medidas del **GPS** posicionarán una **nube de puntos** con coordenadas x, y, z con una cierta tolerancia, cumpliendo la normativa internacional *IHO Standards for Hydrographic Surveys*, que posteriormente trataremos mediante algoritmos de triangulación para crear una superficie (**MDT**).

PALABRAS CLAVE:

EcoSonda, GPS, MDT, USV, Java, Nube de puntos

ABSTRACT:

The purpose of this work will be to extract and interpret data from the position measurements of a **GPS** and an **EchoSounder** through a program developed in **Java** language in the NetBeans environment to obtain a digital terrain elevation model (**MDT**). This model represents the spatial distribution of the surface elevation projected on a terrestrial reference system, specifically ETRS89, and it is essential for the proper planning of civil engineering projects. The surfaces to be measured will be any type of soil formation that may be under a body of water.

These measurements are made with equipment mounted on a ship or in a **USV** (Unmanned Surface Vehicle) as will be our case, they can also be made with satellite images or LIDAR data, but these techniques are less accurate. With the type of sensor we have, a single beam **EchoSounder**, we will be able to measure depths that, together with **GPS** measurements, will position a **pointcloud** with *x*, *y*, *z* coordinates with a certain tolerance, complying with the international IHO Standards for Hydrographic Surveys, which we will later treat using triangulation algorithms to create a surface (**MDT**).

KEYWORDS:

EchoSounder, GPS, MDT, USV, Java, Pointcloud

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE ECUACIONES

1. INTRODUCCIÓN

Conocemos como batimetría al estudio de las profundidades de masas de agua, se habla de masas de agua porque se realizan tanto en mares, como ríos, lagos, etc. Hay una gran cantidad de información que podemos obtener de las zonas de estudio dependiendo del tipo de sensor que utilicemos en la toma de datos. Podemos conocer desde las poblaciones de especies vegetales que pueblan las profundidades del mar hasta los espesores de las capas de residuos que se acumulan en los ríos o los calados de un puerto, tantas son las aplicaciones, que en las licitaciones que oferta el estado esta técnica tiene su propio CPV: 71351923 - Servicios de mediciones batimétricas.

Estas mediciones se realizan con un equipo montado en un barco o, como se realizará en nuestro caso, en un USV (Unmanned Surface Vehicle), también se pueden realizar con imágenes satélite o datos LIDAR, pero estas técnicas son menos precisas. Con el tipo de sensor que tenemos, una EchoSonda monohaz, seremos capaces de medir profundidades que junto con las medidas del GPS posicionarán una nube de puntos con coordenadas *x*, *y*, *z* con una cierta tolerancia que posteriormente trataremos mediante algoritmos de triangulación para crear una superficie (modelo digital de elevaciones) y obtener un curvado, es importante hacer saber que, principalmente, el uso final que se hará del sistema es la obtención de un curvado para estudiar calados de puertos cumpliendo la normativa internacional IHO Standards for Hydrographic Surveys.

Con este curvado se realizan estudios de líneas de pendiente, se obtienen perfiles, secciones y volumetrías que se utilizan en la planificación y ejecución de obras civiles. Se trabaja principalmente en ingeniería civil en el sistema de coordenadas UTM (Universal Transversal de Mercator).

2. MOTIVACIÓN

Este trabajo nace de la necesidad de resolver un problema real para una empresa que desarrolla su actividad en el ámbito de la ingeniería civil y ha brindado la oportunidad al alumno de realizar un trabajo final de máster con instrumentación física que ha exigido pruebas de campo y en la que los resultados se aplican directamente para la consecución de objetivos empresariales. No se puede citar ni hacer referencia a la empresa por contrato de confidencialidad firmado entre dicha empresa y la empresa a la que pertenece el alumno.

El problema a resolver fue que la citada empresa adquirió un USV para incorporarle un GPS y una EchoSonda, pero el software de esta última no funcionaba correctamente y no se tuvo un buen servicio de postventa (no se dio soporte) de este equipo, este software que traía la EchoSonda, por un error de programación, no integraba los datos proporcionados por el GPS necesarios para la correcta obtención de una nube de puntos con coordenadas *x*, *y*, *z* sobre la que obtener un MDT y sobre él un curvado.

Surgió entonces la necesidad de desarrollar un software que tuviera las mismas funcionalidades que el que incorporaba la EchoSonda, al menos al nivel de la aplicación para la que se iba a utilizar este sistema, la obtención de coordenadas del suelo marino.

3. OBJETIVOS

El objetivo que se persigue en la siguiente tesina es la creación de un programa íntegramente desarrollado en Java que mediante una serie de herramientas permitan la obtención de coordenadas del terreno y su posterior procesado para la creación de una superficie digital del terreno sobre la que además, se operará para generar las curvas de nivel de las que la empresa precisa en los estudios previos a las obras.

Los terrenos a medir serán cualquier tipo de formación de suelo cubierta por una masa de agua. El sistema propuesto va a abordar técnicas propias de las batimetrías y de los sistemas de información geográfica.

Se aporta como beneficio del sistema la integración que se menciona, realizándose todos los procesos necesarios sobre un mismo programa, ya que la metodología a seguir por esta empresa sería extraer la nube de puntos del software de la EchoSonda y posteriormente tratarlos en un sistema de información geográfica (teniendo que hacer uso de un programa adicional). Mediante el desarrollo propuesto se extraen los datos necesarios y se realiza el tratamiento de los mismos sobre la misma plataforma.

Los objetivos secundarios, que son los que desarrollan la tesina son los que se van a detallar a continuación:

- Lectura de paquetes del GPS y de la EchoSonda recibidos a través de los puerto serie del ordenador que ejecuta el software.
- Estudio, selección y tratamiento de los paquetes de datos de manera que se obtengan los datos con los que se va a trabajar [x, y, z].
- Configuración de la EchoSonda.
- Obteción de un *fichero.txt* que contenga todas las coordenadas de los puntos medidos una vez que se finaliza la sesión de toma de datos.
- Implementar, sobre la nube de puntos obtenida, un algoritmo para generar una triangulación de Delaunay que represente una superficie (MDT).
- Obtener curvados o isolíneas, trabajando sobre esta superficie, es decir líneas que se cierran, que no se cortan con otras y que tienen el mismo valor de cota a lo largo de su recorrido.
- Representar en una ventana en 2D la nube de puntos, la triangulación, el MDT y el curvado.
- Representar en 3D la nube de puntos, la triangulación, el MDT y las curvas de nivel.

- Realizar pruebas de campo en las fases de prueba y de implantación del proyecto para medir el progreso y para presentar resultados en una aplicación real.
- Comparar curvados obtenidos en aplicación real realizada en una zona cercana al puerto de Sagunto con datos existentes proporcionados por la empresa.

4. ESTRUCTURA DEL DOCUMENTO

Una vez introducida la temática sobre la que va a versar la tesina, las motivaciones que nos hacen llevarlo a cabo y los objetivos que se persiguen, el documento se va a estructurar como sigue.

En primer lugar se hará una descripción de los componentes de los que vamos a hacer uso. Presentamos aquí un diagrama de bloques del sistema que nos pone en perspectiva y nos facilita la comprensión del mismo, en el que se aprecian las conexiones y las comunicaciones entre componentes (sentido unidireccional o bidireccional). El USV se controla mediante una pixHawck 2 Cube que recibe información del GPS del barco que pocesa para ejecutar el sistema de control y actuar sobre la aleta (superficie de control). Por otro lado, dentro del casco está el ordenador que corre el software programado en Java sobre la plataforma Neatbeans y recibe datos de la EchoSonda y del GPS a través del portátil, el GPS necesita estar conectado vía Wi-Fi a un móvil para recibir las correcciones y el portátil también se conecta a este móvil para dotarle de internet y poder acceder a él vía TeamViewer.

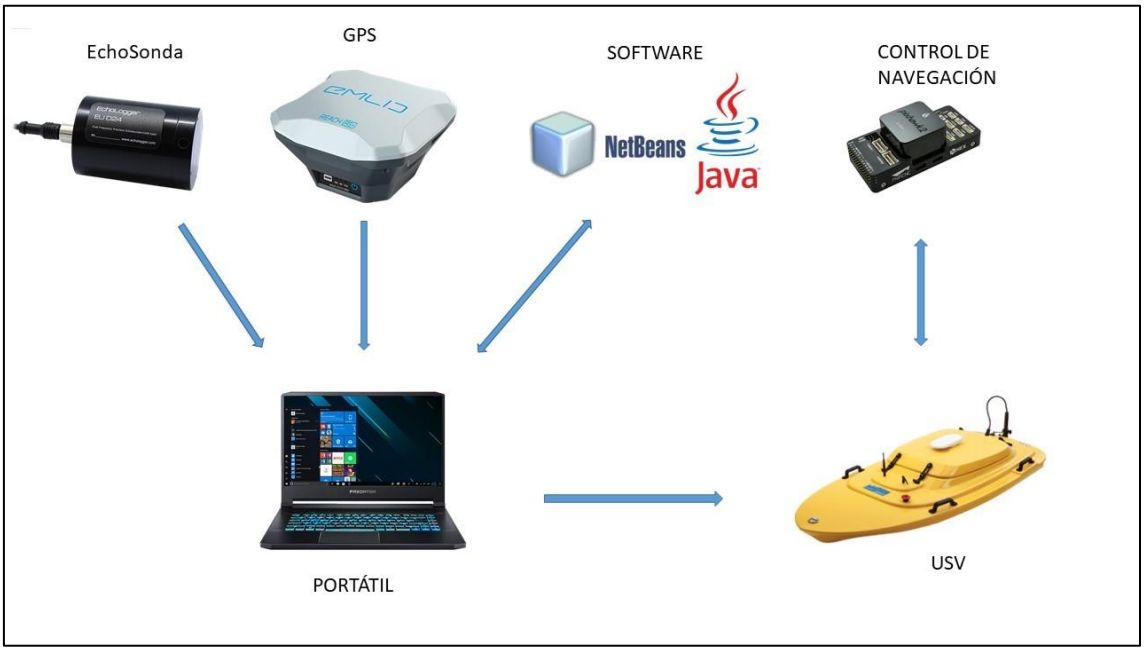


Figura 1 Diagrama de bloques del sistema

Seguidamente se presentará una descripción de la metodología que se desarrollará en una fase más teórica a nivel software y dos fases de pruebas del sistema y de implantación en una aplicación real. Veremos como trabajamos inicialmente para familiarizarnos con el hardware y posteriormente expondremos la recogida de datos tanto a nivel test en coche y en piscina, como a nivel de implantación en la navegación en Sagunto.

El documento se acompañará de un diagrama de gantt, situado después de la introducción a la metodología (apartado 6) donde se va a poder observar la temporización del proyecto, para finalmente acabar con las conclusiones y con una serie de propuestas de trabajos futuros que sigan en la línea de lo expuesto durante el trabajo.

5. HERRAMIENTAS Y EQUIPOS UTILIZADOS

Se van a presentar las herramientas de programación utilizadas y se van a caracterizar los equipos que componen el sistema de medida

5.1. NetBeans

Es un entorno de desarrollo integrado para la implementación y desarrollo de código basados principalmente en lenguaje Java. NetBeans es un producto libre y gratuito sin restricciones de uso. La funcionalidad de esta plataforma se basa en el desarrollo de software en conjuntos a los que llama módulos. El código creado en java es leído por un intérprete por lo que puede ejecutarse sobre cualquier plataforma.

5.2. GPS EMLID

El Sistema de posicionamiento Global (GPS) es un sistema de localización diseñado por el departamento de Defensa de los Estados Unidos para proporcionar estimaciones precisas de posición, velocidad y tiempo. No es objeto de este trabajo presentar la arquitectura del sistema o sus principios de funcionamiento. Sí lo es presentar de manera resumida el receptor de señal (EMLID) que se va a utilizar, así como los paquetes de información que proporciona (comunes a cualquier receptor GPS) relevantes para el desarrollo de este trabajo.

Figura 2 Receptor de señal GPS EMLID

La marca EMLID ha sacado al mercado estos receptores que funcionan con la misma precisión que un receptor de las marcas trimble, topcon o leica (top 3 del mercado) con

un precio de 800$, unas 10 veces por debajo del precio de los receptores que se han mencionado anteriormente. La principal diferencia con los otros receptores es que no incorpora la típica tableta de otras marcas que gestionan la configuración del equipo y almacenan datos, en este caso es nuestro propio móvil, con una aplicación desarrollada por la marca, el que hace las funciones de esa tableta conectando por vía Wi-Fi con el receptor. Otro modo de conexión y acceso a la configuración del receptor es por cable USB, otro aspecto que mejora desde nuestro punto de vista a los otros equipos haciendo posible la lectura e interpretación de los paquetes de manera libre sin tener que depender del software comercial que las otras marcas incorporan o de la app de la propia marca EMLID.

Se recibe una serie de paquetes NMEA 0183 (protocolo de comunicación de GPS) con distintos tipos de información, aquí presentamos el paquete que realmente nos interesa que es de donde sacaremos la información de las coordenadas x, y, z del receptor:

GGA Global Positioning System Fix Data. Time, Position and fix related data for a GPS receiver

```
                                                     11
       1         2       3 4        5 6 7  8   9  10 |  12 13  14   15
       |         |       | |        | | |  |   |   | |   | |   |    |
$--GGA,hhmmss.ss,llll.ll,a,yyyyy.yy,a,x,xx,x.x,x.x,M,x.x,M,x.x,xxxx*hh

 1) Time (UTC)
 2) Latitude
 3) N or S (North or South)
 4) Longitude
 5) E or W (East or West)
 6) GPS Quality Indicator,
    0 - fix not available,
    1 - GPS fix,
    2 - Differential GPS fix
 7) Number of satellites in view, 00 - 12
 8) Horizontal Dilution of precision
 9) Antenna Altitude above/below mean-sea-level (geoid)
10) Units of antenna altitude, meters
11) Geoidal separation, the difference between the WGS-84 earth
    ellipsoid and mean-sea-level (geoid), "-" means mean-sea-level below ellipsoid
12) Units of geoidal separation, meters
13) Age of differential GPS data, time in seconds since last SC104
    type 1 or 9 update, null field when DGPS is not used
14) Differential reference station ID, 0000-1023
15) Checksum
```

Figura 3 Paquete GPGCA del protocolo NMEA 0183

Nos interesan los campos:

- [2]: Double-Latitud en coordenadas geográficas, con la peculiaridad de que se obtienen en grados, minutos decimales cuyo cambio a grados decimales se expresa con las fórmulas que veremos a continuación, cambio que es necesario previo a la transformación a coordenadas UTM, con las que se creará el modelo digital de elevaciones y sistema de coordenadas con el que habituamente se trabaja en proyectos de ingeniería civil.

$$Long.Calculada = \left(\frac{Long.\ Inicial}{60}\right) * 100$$

Ecuación 1 Cambio de longitud en grados, minutos decimales a grados decimales

- [3]: Byte que Indica si la medida de latitud corresponde al hemisferio Sur o al Norte.
- [4]: Double-Latitud en coordenadas geográficas, seguimos el mismo criterio que con la longitud para poder transformar a coordenadas UTM.

$$Lat.Calculada = \left(\left(\frac{Lat.Inicial - floor(Lat.Inicial)}{60}\right) * 100\right) + floor(Lat.Inicial)$$

Ecuación 2 Cambio de latitud en grados, minutos decimales a grados decimales

- [5]: Byte que Indica si la medida se realiza al Este o al Oeste
- [9]: Double-Altitud sobre el geoide.

Con esta información, restando la medida de profundidad que obtendremos de la sonda y la diferencia de altura entre la posición del receptor GPS y la ubicación de la sonda, hallaremos el punto [*x*, *y*, z] buscado de la superficie debajo del agua, veremos este cálculo en el apartado 6.1.2.2

Como paso previo comprobamos que los paquetes de entrada del GPS se corresponden con lo visto en la bibliografía, para ello hacemos uso del programa PuTTY que nos permite leer estos paquetes indicando el puerto al que conectamos el equipo y su baud rate.

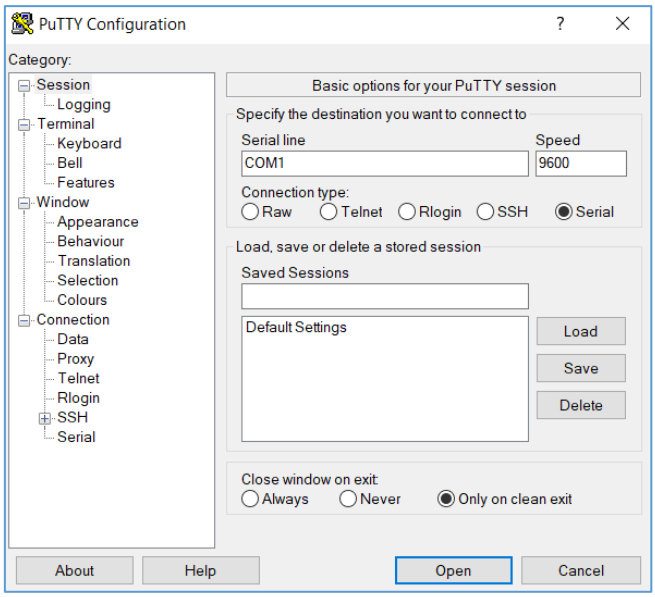


Figura 4 Configuración PuTTY

Abrimos la conexión serie y comprobamos que efectivamente se están recibiendo los paquetes que esperabamos recibir.

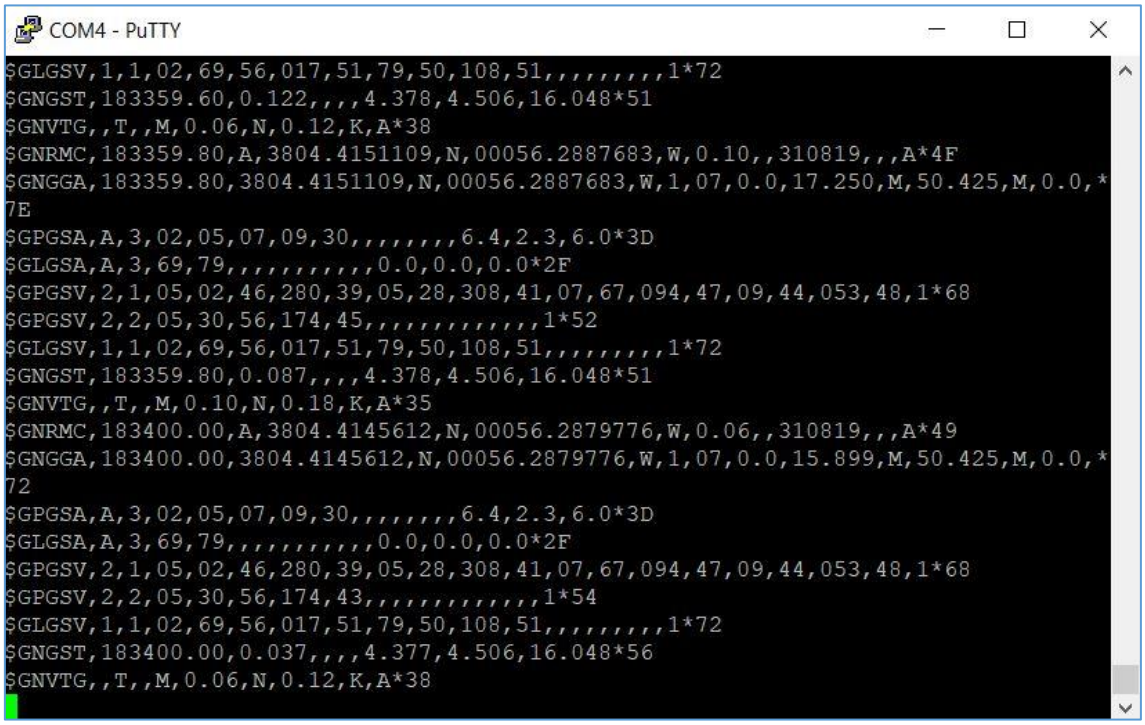


Figura 5 Paquetes GPS PuTTY

El GPS se conecta a la red ERVA de estaciones de referencia del Instituto Cartográfico Valenciano a través de un móvil con internet conectado al GPS vía WiFi, este recibe las correcciones que se aplican a la señal de modo que la precisión final obtenida sea centimétrica, hecho indispensable para que los datos adquiridos puedan emplearse con la finalidad que se desea ya que para los trabajos a los que se destina esta aplicación se necesitan estas tolerancias. Por ejemplo si queremos calcular un movimiento de tierras de una gran extensión a partir del curvado obtenido irnos a precisiones métricas resultaría en unos sobregastos que harían inviables las ejecuciones de obras.

5.3. EchoSonda Echologger DU24

Cabe destacar que llamamos EchoSonda al sistema completo de medida que se compone de dos elementos, un transductor que es la parte del equipo que va sumergida en el agua y envía y recibe ecos y calcula, entre otras cosas la profundidad, conociendo la velocidad de propagación del sonido en el agua (aprox 1500 m/s). El otro elemento es el sistema de recepción de la señal que en los casos habituales almacena los datos proporcionados por el transductor y los representa de manera que se vea la forma del fondo, este elemento es el que daba el problema y el que va a ser objeto de desarrollo de la presenta tesina.

En nuestro caso tenemos el transductor y el objetivo de este trabajo es diseñar un receptor personalizado a nuestras necesidades. El transductor de la Echosonda DU24 es el que se ve en la siguiente figura:

Figura 6 Echologger DU24

Este tipo de transductor es monohaz, quiere decir que se lanza un solo haz de sonido por lo tanto solo podemos medir un punto con cada eco. Hay otras sondas, llamadas multihaz que lanzan muchos haces a la vez cada uno con una inclinación diferente por lo que se puede barrer el fondo estudiado con un menor número de pasadas, optimizando la toma de datos, este tipo de sensores son, como se puede intuir, bastante más caros que con el que se va a trabajar.

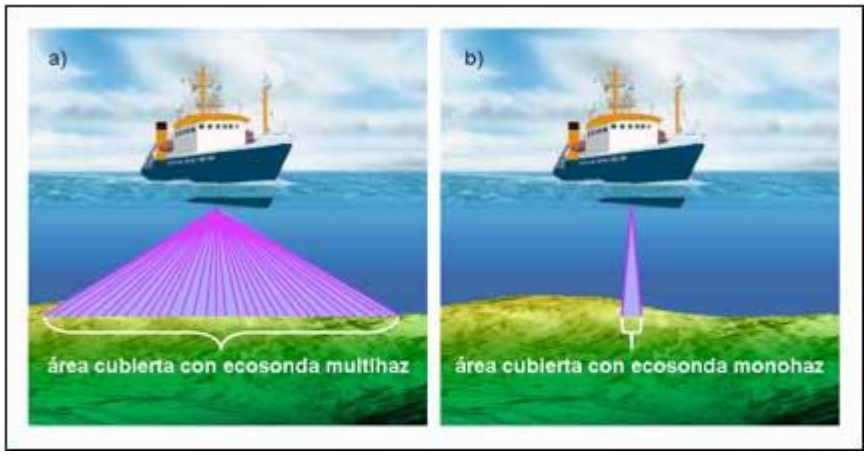


Figura 7 Monohaz vs multihaz

El transductor DU24 se puede configurar, para lanzar un haz con una frecuencia de 200 kHz o de 450 kHz, siendo la principal diferencia el alcance en profundidad que se puede

alcanzar siendo la de 200 Hz la que más abajo llega penalizando eso sí, la precisión de la medida. Llegando a realizar medidas a profundidades de 200 m.

Se presenta a continuación la tabla de especificaciones de la sonda:

Acoustic Frequency	200 kHz / 450 KHz
Beam width	10° / 5° Conical (-3dB)
Transmit Pulse Width	10µsec ~ 200µsec
Transmit Power	Max. 50 W, adjustable
TVG Control	Up to 60 dB,
Gain Control	-30 dB to +30 dB
Input Signal Attenuator (-20dB)	Activation time 0~300,000 µSec
Ranges	0.15 m ~ 200 m (200kHz), 0.15 m ~ 100 m (450kHz)
Repetition (Ping) Rate	100Hz max
Sampling Rate	Max 100kHz; adjustable, or auto mode (default)
Water Column Resolution	Appx. 7.5 mm @100kHz sampling
Altimeter Range Resolution	1.0 mm
Temperature sensor Resolution	0.1°C
Tilt sensor integrated	Dual-axis (Roll & Pitch) ±90°
	Inclination data resolution 0.1°
Synchronization	Outward / Inward (S/W selectable)
Analog Output Interface	1.25V, 2.5V, 5V, 10V max (S/W selectable)
	Distance or Envelope
Digital Output Interface	RS-232, RS-485 (selectable by select pin)
Communication Speed	4800 ~ 921600 baud (115200 baud default)
Data Output Format	Profile 12bit resolution ASCII Text
	Profile 10bit resolution ASCII Text
	Profile 12bit resolution Binary
	Profile 8bit resolution Binary (12bit compressed to 8bit)
	Altitude NMEA0183/ Altitude Simple
Multi node Internetworking(RS485 only)	Up to 32 units
Configuration and Data reading	Echologger Control Program or any Terminal program
Connector	ECT D24S: SEACON MCBH8MSS
	ECS D24S: SEACON MCBH8MSS
	EGT D24: Cable Gland
Power supply	10 ~ 75 VDC, 2W max, Internally isolated
Operation Temperature	-10°C +50°C
Operating Depth	1m, 100m, 1000m, 6000m
Housing	ECT D24S: Acetal (100m), Aluminum (1000m, 6000m)
	ECS D24S: Aluminum (100m, 1000m, 6000m)
	EGT D24: Acetal (1m) IP68
Dimensions	ECT D24S: D56 mm x L85 mm (without connector)
	ECS D24S: D55 mm x L70 mm (without connector)
	EGT D24: D56 mm x L80 mm (without cable gland)
Weight	ECT D24S: 280g (100m version)
	ECS D24S: 320g (100m, 1000m versions)
	EGT D24: 240g (without cable)
Other features	Compatible with many Geo-related commercail softwares (Hypack, HydroPro, Echoview etc.)
	GPS integrated data
	Multi-node networking

Figura 8 Especificaciones DU24

Y como en el caso del GPS presentamos los paquetes que nos da el transductor:

Refer to NMEA 0183 specification

Message	Meaning	Description	Format example
$SDDBT	Depth below transducer	Water depth referenced to the transducer's position. Depth value expressed in feet, meters and fathoms.	$xxDBT,FEET,f, METRES,M, FATHOMS,F*hh<0D><0A>
$SDDPT	Water Depth	Water depth relative to the transducer, the depth offset of the transducer, and maximum depth that the sounder can detect a sea-bed (all in meters only). Positive offsets provide distance from the transducer to the water line. Negative offsets provide distance from the transducer to the keel.	$xxDPT,DATA_METRES,OFFSET_METRES *hh<0D><0A>
$SDMTW	Mean Temperature of Water	Water temperature in degrees centigrade.	$xxMTW,TEMPERATURE,C*hh<0D><0A>
$SDZDA	Time and Date	UTC, day, month, year, and local time zone.	$xxZDA,hhmmss.ss,dd,mm,yyyy,hh,mm*hh<0D><0A>
$SDXDR	Transducer Measurements	Provide information about pitch/roll and Maximum Amplitude of Echo signal	$xxXDR,A,X.X,D,PTCH,A,X.X,D,ROLL*hh<0D><0A> $xxXDR,A,X.XX,P,EMA*hh<0D><0A>

Figura 9 Paquetes Echologger DU24

De donde nos quedaremos con el valor que hay en la posición 4 del mensaje $SDDBT que nos indica la profundidad en metros. También nos interesa el campo $SDXDR para tomar los valores de pitch y roll del USV con el fin de corregir la profundidad que arroja la EchoSonda y calcular el valor zenital. Se realiza la misma comprobación previa que con el GPS con el programa PuTTY, con la peculiaridad de que este equipo solo trabaja con una velocidad de transmisión de 115200 baudios, por lo que es necesario definir este baud rate en las propiedades del dispositivo:

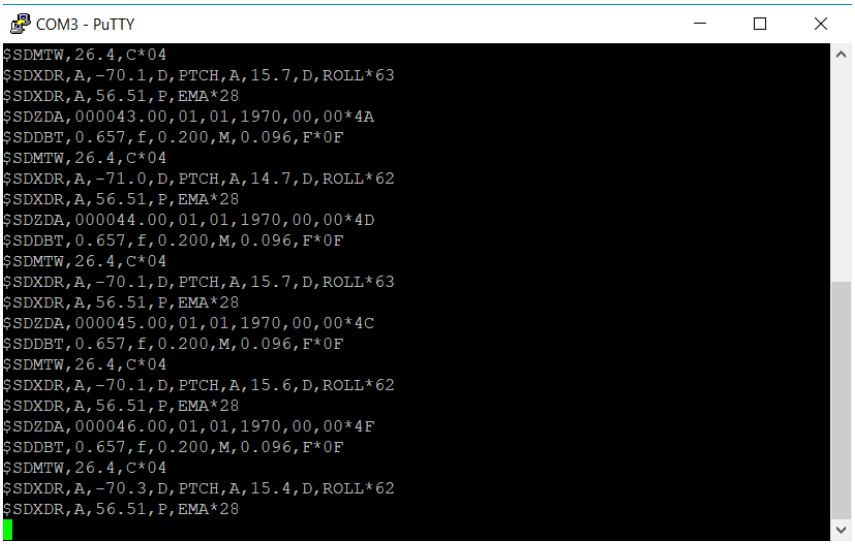


Figura 10 Paquetes Echosonda DU24 PuTTY

5.4. USV HarbourScout Platform

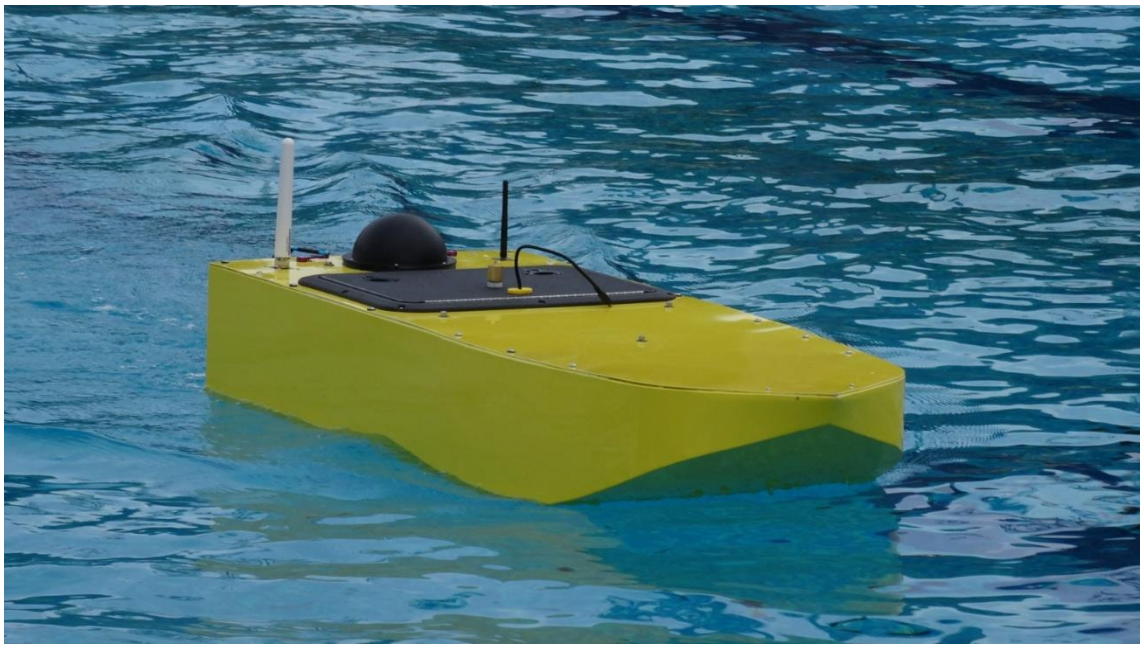


Figura 11 USV HarbourScout

El sistema de toma de datos puede ir montado en cualquier plataforma que sea capaz de seguir una ruta de navegación, en nuestro caso se utilizará un casco fabricado en aluminio y con un acabado con pintura marina para minimizar los efectos de corrosión con una controladora típica de uso en RPAS, la PixHawk 2 Cube que controla la

dirección mediante un servo de 20 kg que mueve las aletas, y la velocidad mediante los ESC de los dos motores.

Figura 12 PixHawk 2 Cube

A esta controladora se le pasará un plan de navegación con el programa mission planner en el que se fijarán los nudos a los que navegará el barco de tal manera que las medidas que se tomen cumplan con los estándares de la IHO, en el apartado 6.1.2.3 se verán estos estándares y se calculará la velocidad a la que navegará el barco que estará en función de la velocidad de envío de datos de los sensores a la CPU del ordenador de abordo que ejecutará nuestro programa y en función de la tolerancia que se especifique en estos estándares. Para la estación de tierra se cuenta con una taranis 9XD para el control manual y con un ordenador para el envío y monitorización del plan de navegación.

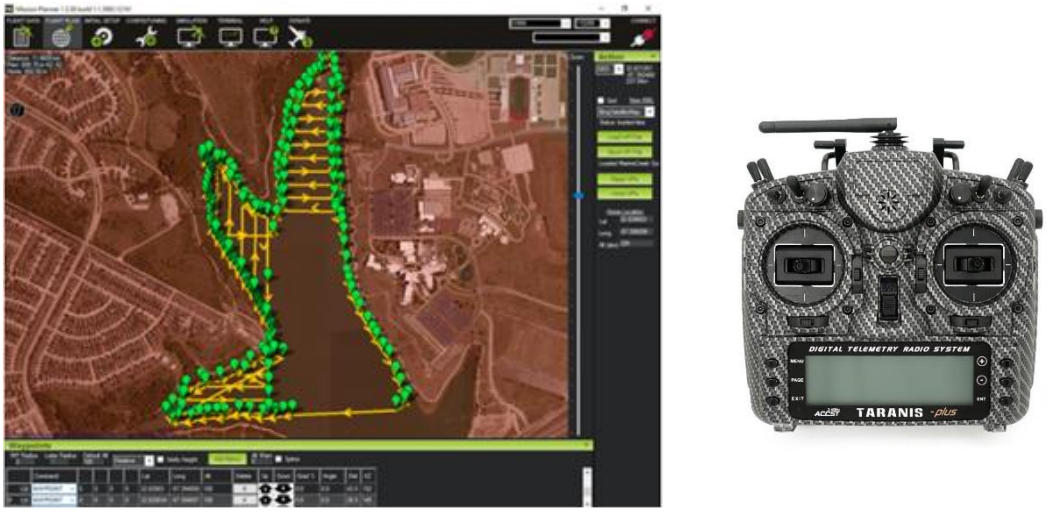


Figura 13 Mission planner + Estación de tierra

Adicionalmente al problema con el transductor de la EchoSonda, que conlleva la propuesta de solución que se desarrolla en esta tesina, el equipo USV recibido no cumplía las expectativas previstas y tuvo que ser sometido a una serie de modificaciones.

El primer paso que se realizó fue desmontar por completo el casco para poder comprobar los componentes que incorporaba el mismo. Aquí detectamos ciertos déficits que pasamos a mejorar.

- Cambio de todos los tornillos del USV por tornillos de acero inoxidable.
- Adquisición de adaptadores para conectar el portátil a la toma de corriente del USV.
- Compra de 2 baterías de 16000 mhA con las que se alcanzan hasta 4 horas de autonomía.
- Empalmes y cableados necesarios para la toma de corriente de las baterías, ya que solo disponíamos de un conector.
- Adecuación de un soporte impreso en 3D para la colocación de la sonda, ya que solo disponíamos de la cavidad, a pesar de que el fabricante solicitó las medidas de la EchoSonda para incorporar un soporte adecuado.
- Colocación de un soporte para el GPS, coincidiendo con la vertical de la sonda.
- Amarres para el ordenador de a bordo.
- Sellamiento de todo de equipo con silicona náutica

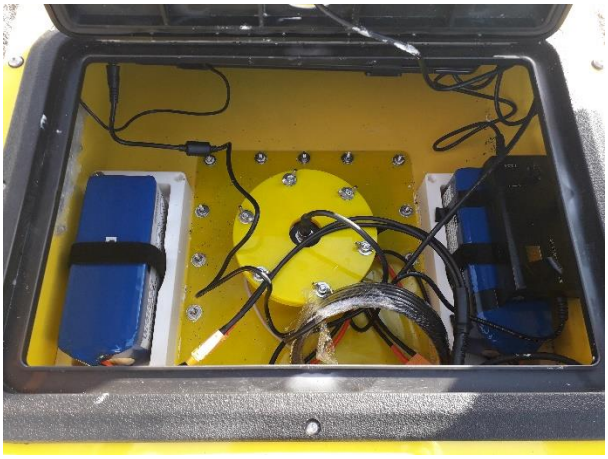


Figura 14 Interior modificado

6. METODOLOGÍA

Para el correcto desarrollo del proyecto seguiremos una metodología claramente diferenciada en tres fases:

- **Fase de desarrollo**: se ha generado un programa realizado en lenguaje Java sobre la plataforma de desarrollo NeatBeans que cubre las deficiencias de la EchoSonda adquirida por la empresa contratante y que a su vez aporta facilidad de uso y optimización de recursos reuniéndo en un solo programa todas las funcionalidades necesarias para llegar a los resultados deseados. Java es una potente herramienta de desarrollo que se alimenta de librerías de libre acceso, en este proyecto se han utilizado varias clases y librerías haciéndo una comprensión de las mismas y adaptándolas a nuestro caso de uso, no se pretende hacer una apropiación de partes de código que desarrollan el proyecto y en todo momento se indicará qué librerías y clases han sido utilizadas y adaptadas.

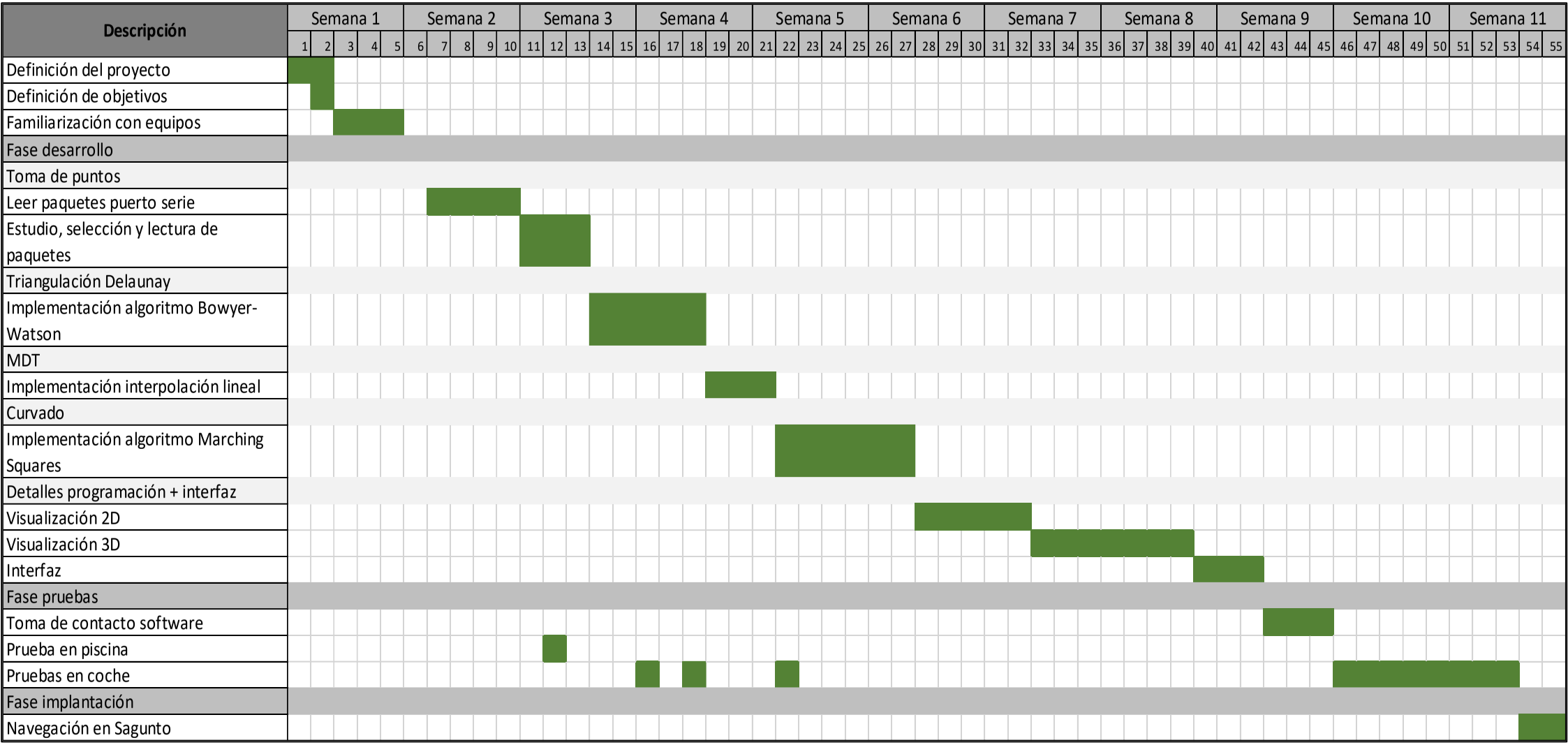
  En la exposición de esta fase en la memoria se harán las explicaciones teóricas necesarias para comprender que estamos haciendo, por qué y que resultados esperamos obtener, se acompañará la teoría con las partes del código que la desarrollan. Se presentarán también los detalles del programa que se consideren necesarios para tener una comprensión global del mismo y se añadirá un pseudocódigo de consulta donde se plasmarán las principales ideas anteriormente expuestas sobre código. Se diferenciarán por tanto 4 bloques:

  - Toma de puntos: en la que se explicará una serie de consideraciones técnicas en las que se ha tenido que trabajar para asegurar la correcta utilización de los datos y se hará un breve repaso del sistema de coordenadas UTM sobre el que se va a trabajar.
  - Triangulación de Delaunay + MDT: Se explicará el algoritmo empleado para realizar la triangulación y el algoritmo de creación de la superficie continua (MDT) a partir de la triangulación.
  - Isolíneas o curvas de nivel: Se explicará que son, cómo se utilizan en ingeniería civil y el algoritmo con el que se obtienen.
  - Detalles de programación: Haremos un repaso de los detalles de visualización y de interacción con el usuario.
- **Fase de pruebas**: En paralelo a la fase de desarrollo y para depurar esta misma se han realizado pruebas del sistema sin tener que montarlo necesariamente en

el USV, se han realizado tomándo puntos montados en un coche, con las consideraciones que esto conlleva y que se presentarán en el apartado 6.2.3. También se han realizado pruebas en un entorno controlado en una piscina ya con todo el equipo montado, en un escenario muy similar al de uso real.

- **Fase de implantación**: En esta fase se van a exponer los resultados extraídos de un trabajo de aplicación real que se realizó en una zona de mar cercana a las instalaciones del puerto de Sagunto de la que se disponen datos previos y con los que se verá una comparativa.

Se muestra a continuación un diagrama de Gantt con la temporización de cada una de estas tareas. Se cuentan semanas de 5 días.


Descripción
Semana 1
Semana 2
Semana 3
Semana 4
Semana 5
Semana 6
Semana 7
Semana 8
Semana 9
Semana 10
Semana 11
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55
Definición del proyecto
Definición de objetivos
Familiarización con equipos
Fase desarrollo
Toma de puntos
Leer paquetes puerto serie
Estudio, selección y lectura de paquetes
Triangulación Delaunay
Implementación algoritmo Bowyer-Watson
MDT
Implementación interpolación lineal
Curvado
Implementación algoritmo Marching Squares
Detalles programación + interfaz
Visualización 2D
Visualización 3D
Interfaz
Fase pruebas
Toma de contacto software
Prueba en piscina
Pruebas en coche
Fase implantación
Navegación en Sagunto

6.1. Fase de desarrollo

Esta fase va a estar subdividida en 4 bloques claramente diferenciados que ya se han expuesto en el apartado anterior, cada bloque tendrá su componente teórico acompañado por la parte de código que desarrolla esta teoría. En esta fase se pretende que quede claro que está haciendo nuestro programa, cómo lo está haciendo y cuáles son los resultados que se esperan obtener, a modo de resumen se va a explicar cómo partiendo de distintos sensores de medida, se extraerá una nube de puntos que procesaremos para crear una superficie contínua sobre la seremos capaces de construir isolíneas que marquen las cotas de terreno deseadas por el usuario.

6.1.1 Apariencia del programa

Antes de hacer un desglose del programa Java y de los métodos y clases que lo desarrollan vamos a ver una pequeña presentación para tener una conciencia global del mismo y saber dónde nos encontramos en cada momento.

El proyecto tiene 3 clases principales:

- TomaPuntos: que se extiende un JFrame (Ventana) y se ejecuta cuando deseamos comenzar a tomar puntos durante la navegación.
- Visualización2D: también extiende JFrame, crea objetos y tiene métodos para representar en 4 paneles de una ventana los puntos obtenidos, la triangulación generada, el MDT y el curvado.
- Visualización3D: que hace uso de la API jMonkey Engine que es un motor de juegos 3D de código abierto creado para desarrolladores Java, con el que vamos a representar en 3D los resultados obtenidos.

Estas clases principales son complementadas con otras 10 clases:

- Clases CPaint: redibuja los gráficos cada vez que se produce un evento (cuando se mueve la pantalla o se minimiza y se maximiza).
- CalculadoraCoordenadas: contiene métodos que aplica las consideraciones matemáticas necesarias para que las coordenadas obtenidas sean válidas.
- Delaunay: triangula la nube de puntos.
- LeeDatos: clase de tipo thread, trabaja tomando los datos del GPS y de la EchoSonda durante la navegación.
- IsoCell + MarquingSquares: crean un array de General Paths que representan las curvas de nivel.

- SerialPortExample: clase que se encarga de detectar el puerto de entrada de datos y de establecer una comunicación con este.

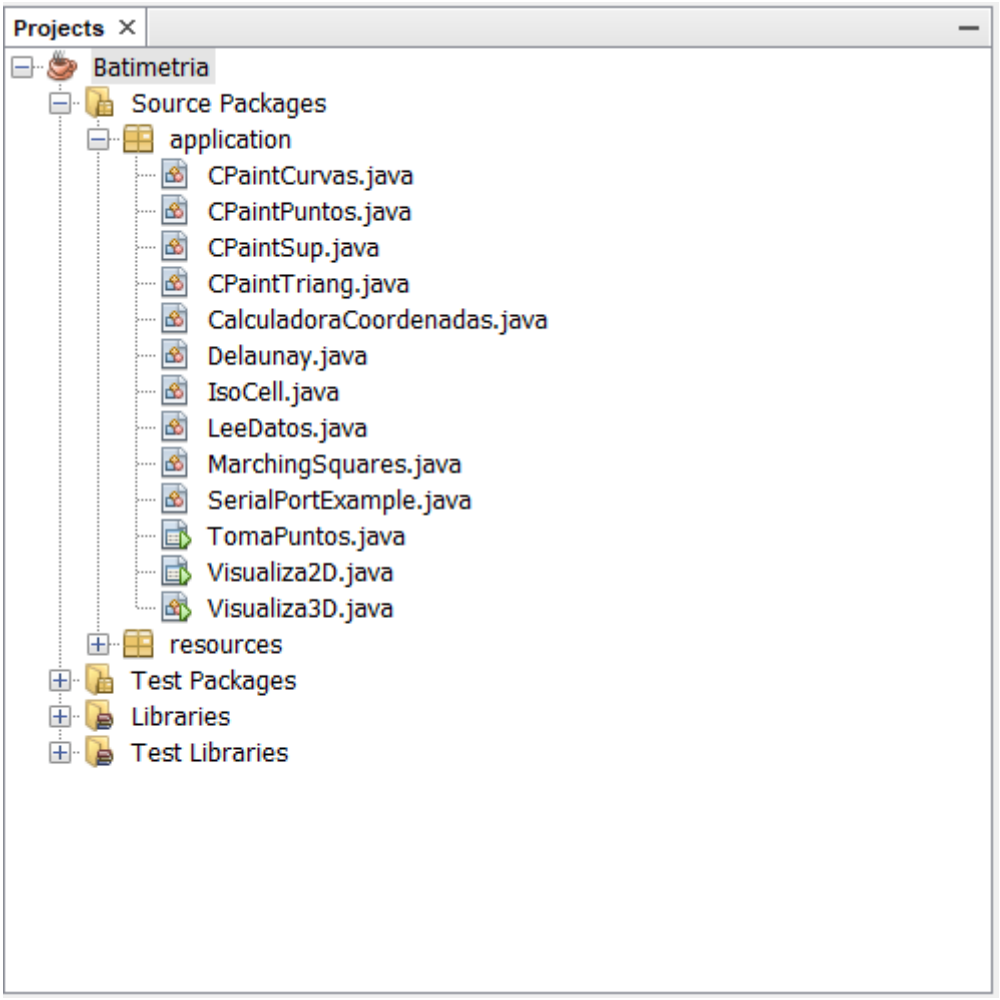


Figura 15 Proyecto Batimetria

6.1.2 Toma de puntos

La toma de puntos se realiza conectándo los equipos al ordenador de a bordo, este sistema va montado sobre la plataforma que navega barriendo la sobre la superficie a medir con pasadas paralelas. La EchoSonda y el receptor GPS deben estar debidamente alineados con la finalidad de que el punto *z* que tome la EchoSonda se corresponda en *x* y en *y* con el punto almacenado por el GPS. Conociendo la diferencia de altura entre receptor GPS y EchoSonda quedará perfectamente definido el punto buscado teniendo en cuenta las consideraciones técnicas que vamos a ver.

6.1.2.1. Formato GPS

El proyecto pretende ser lo más generalista posible en cuánto a la libertad de uso de sensores que sean capaces de enviar paquetes con el protocolo NMEA 0183, tanto el mensaje $SDDBT que da la sonda como el $GPGGA del GPS están incluidos en este protocolo.

En el caso del GPS se tuvo el problema de que no se encuentran los paquetes de GPS $GPGGA que nos dan latitud, longitud y elevación, en un principio se pensó que era necesario cambiar la configuración del dispositivo desde la aplicación de móvil que se mencionó en la descripción del EMLID. Se ha probado esto y no se consiguen más que los paquetes que se obtienen de la constelación rusa GLONASS ($GNGGA) donde se ha detectado que la longitud no se corresponde con la longitud proyectada sobre el sistema de coordenadas internacional WGS84, para solucionar esto se configuró el equipo para que envíe paquetes en un formato propio que da coordenadas geodésicas para la prueba en piscina, es decir, la longitud y la latitud se proyectan sobre el WGS84 y el valor de elevación se toma a partir del elipsoide, diferencia con las coordenadas geográficas que miden la elevación sobre el geoide y que son las que podemos ver en google earth. Haciendo uso del protocolo NMEA no tendremos este problema.

El programa se presenta finalmente con la lectura del protocolo NMEA para que quede lo más generalista posible, además después de las pruebas en piscina nos cercioramos de que aunque exista esa diferencia al medir de la constelación GLONASS, las correcciones diferenciales que nos llegan de la red ERVA hacen que las medidas sean correctas. Si se leyera de la constelación GPS el paquete comenzaría por $GPGGA. Hay que destacar que la longitud y la latitud se expresan en grados, minutos decimales, que deben ser transformados a grados decimales y posteriormente a coordenadas UTM. Las elevaciones se miden sobre el geoide.

6.1.2.2. Sistema de coordenadas UTM

Vamos a trabajar con proyección UTM porque es *la proyección empleada en la cartografía española desde que nos la dieron los Americanos en los años 50* (FERNÁNDEZ-COPPEL, I. A., 2001) y por ende es la proyección que se lleva utilizando desde entonces en la mayoría de proyectos de ingeniería civil en nuestro país.

La proyección UTM pertenece a las llamadas proyecciones cilíndricas ya que se hace uso de un cilíndro para proyectar los puntos de la superficie terrestre, además es una proyección conforme que mantiene el valor de los ángulos entre puntos proyectados. La tierra se divide en husos de 6º de longitud que contienen todos los puntos comprendidos entre dos meridianos, estos husos tienen un “meridiano central” equidistante 3º de los extremos de cada uso por los que pasa tangencialmente el cilindro de proyección. Por convección se generan siempre valores positvos de *X* y de *Y* como se puede apreciar en las siguientes imágenes:

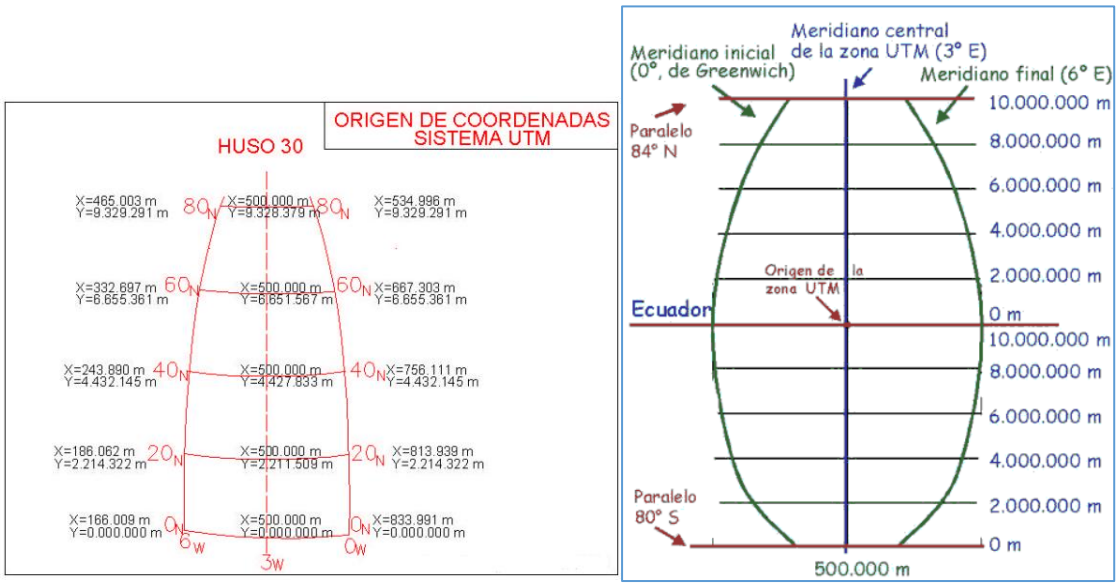


Figura 16 Y postitivas (izq.) X positivas (dcha.)

El GPS nos da medidas de los puntos en coordenadas geodésicas LLH que arrojan valores de longitud y latitud en grados, minutos decimales. Una vez obtenidas las coordenadas gedésicas LLH expresadas en grados decimales con las fórmulas vistas en el apartado 3.2 es necesario transformarlas al sistema de coordenadas UTM (x,y) para poder operar con ellas en postproceso. El sistema de referencia de estas coordenadas es el ecuador para el eje de abscisas y el eje de ordenadas se define con el “meridiano central” que pasa por el centro de cada huso horario, en el caso del puerto de Sagunto nos encontramos en el huso 30

Para el cálculo de la cota terreno final Z_terreno se tienen en cuenta también varias consideraciones:

- Valor de Geoide en la zona de estudio. El GPS toma por defecto un valor homogéneo de Geoide de 51.06 metros, probablemente está trabajando con algún modelo extranjero o anterior al oficial en España, el egm08_rednap. Por ello averiguamos el valor medio de Geoide para nuestra zona descargándolo de los servicios cartográficos del IGN (Instituto Geográfico Nacional) o del ICV (Instituto Cartográfico Valenciano) y tendremos en cuenta la diferencia existente entre ambos. Esto supone un trabajo previo a realizar y es un valor que deberemos introducir como parámetro de entrada al programa. En el anexo 1 (manual de usuario), se detallará como proceder.

  En el caso de la prueba final de implantación en el puerto de Sagunto, nos encontramos en una zona con un valor geoidal sobre el elipsoide de referencia de 50.19 metros por lo que la diferencia entre ambos es de 0.87 metros.

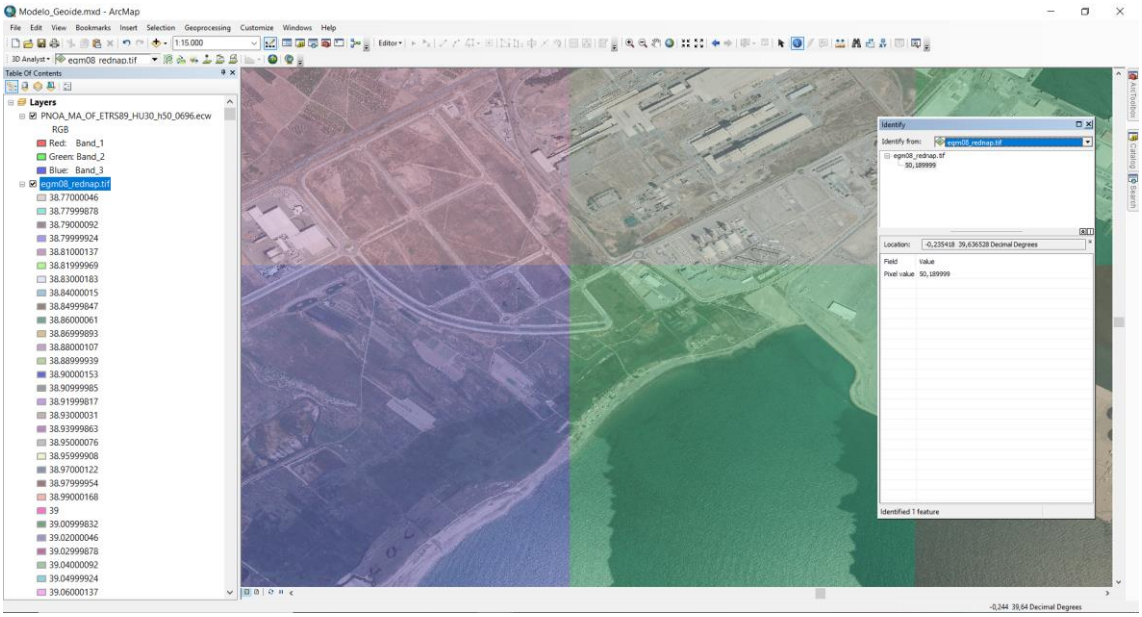


Figura 17 Modelo de Geoide EGM08 visualizado en programa GIS

- Offset entre la sonda y la base del receptor GPS. El GPS toma su dato en *Z* en el tronillo de su base, y la sonda desde la parte inferior de la misma. Ambos se encuentran en la misma perpendicular a una distancia de 22.5 centímetros, valor que deberemos restar para el correcto cálculo de la cota deseada.

Finalmente nuestra fórmula para el cálculo de la cota terreno es:

$$Z_{terreno} = Z_{GPS} - Offset - Z_{SONDA} + Diferencia\ Geoidal$$

Ecuación 3 Cálculo cota terreno

6.1.2.3 Normativa Internacional batimetrías

Se muestra el cuadro de requisitos que establece la norma internacional para batimetrías impuestas por la IHO (Organización Hidrográfica Internacional) de donde se obtendrá la velocidad necesario del barco teniendo en cuenta la velocidad de muestreo de la sonda, que es la condicionante ya que manda paquetes cada segundo.

Reference	Order	Special	1a	1b	2
Chapter 1	**Description of areas.**	Areas where under-keel clearance is critical	Areas shallower than 100 metres where under-keel clearance is less critical but *features* of concern to surface shipping may exist.	Areas shallower than 100 metres where under-keel clearance is not considered to be an issue for the type of surface shipping expected to transit the area.	Areas generally deeper than 100 metres where a general description of the sea floor is considered adequate.
Chapter 2	**Maximum allowable THU 95% *Confidence level***	2 metres	5 metres + 5% of depth	5 metres + 5% of depth	20 metres + 10% of depth
Para 3.2 and note 1	**Maximum allowable TVU 95% *Confidence level***	a = 0.25 metre b = 0.0075	a = 0.5 metre b = 0.013	a = 0.5 metre b = 0.013	a = 1.0 metre b = 0.023
Glossary and note 2	***Full Sea floor Search***	Required	Required	Not required	Not required
Para 2.1 Para 3.4 Para 3.5 and note 3	***Feature Detection***	Cubic *features* > 1 metre	Cubic *features* > 2 metres, in depths up to 40 metres; 10% of depth beyond 40 metres	Not Applicable	Not Applicable
Para 3.6 and note 4	**Recommended maximum Line Spacing**	Not defined as *full sea floor search* is required	Not defined as *full sea floor search* is required	3 x average depth or 25 metres, whichever is greater For bathymetric lidar a spot spacing of 5 x 5 metres	4 x average depth
Chapter 2 and note 5	**Positioning of fixed aids to navigation and topography significant to navigation. (95% *Confidence level*)**	2 metres	2 metres	2 metres	5 metres
Chapter 2 and note 5	**Positioning of the Coastline and topography less significant to navigation (95% *Confidence level*)**	10 metres	20 metres	20 metres	20 metres
Chapter 2 and note 5	**Mean position of floating aids to navigation (95% *Confidence level*)**	10 metres	10 metres	10 metres	20 metres

Figura 18 Estándares mínimos para batimetrías

Nuestro caso de uso (principalmente medidas de calados de puertos) es el 1a o 1b. Cumplimos con creces los errores mínimos de medida ya que se pueden tomar datos cada 0.1 segundos

De la norma internacional para batimetrías se extrae la información relevante que necesitamos para conocer la velocidad de navegación de un barco, nuestro caso más habitual de estudio será en puertos, por lo que se supone que estamos en el caso 1a de aguas de profundidad inferior a 100 metros donde la toma de puntos se tiene que realizar como mínimo cada 2 metros, luego por el teorema de muestreo de *nysquit* las

tomas de puntos deben efectuarse cada metro por lo que la velocidad a la que tiene que avanzar el barco es de 1 m/s si se toman medidas cada segundo si trabajamos con la configuración estándar de la sonda. Más adelante veremos como cambiar esta configuración. Por otro lado se cumple con creces la condición de no sobrepasar los errores mínimos. En la medida horizontal, la precisión del GPS es centimétrica cuando se están exigiendo 5 metros + un 5% de la profundidad como mínimo. Para la medida vertical hay que hacer un cálculo con las constantes a y b que se proporcionan, pero la resolución altimétrica de la sonda es de un mm como podemos ver en la figura 8.

$$error\ vertical\ admitido = \pm\sqrt{a^2 + (b * d)^2}$$

Ecuación 4 Error vertical admitido (TVU)

El parámetro *d* representa la profundidad expresada en metros en cada instante de medida, aún así el milímetro de resolución en altimetría nos asegura en todo momento cumplir las especificaciones de la norma.

La ruta de navegación del barco consiste en pasadas paralelas de (3*media de profundidad metros) o si la media es mayor de 25 (3*25 metros), así teniendo en cuenta que la autonomía de las baterías de 32.000 mAh es de aproximadamente 4 horas (14400 seg) se podrá realizar la medición de aproximadamente 55 hectáreas por ruta, como se ha dicho, con la configuración estándar de la sonda.

$$\frac{14400[m]}{2} * 75[m] \approx 55[hectáreas]$$

Ecuación 5 Hectáreas cubiertas por juego de baterías

6.1.2.4. Código

Vamos a ver ahora como trabaja el programa para ser capaz de hacer la recogida y almacenamiento de los puntos. Se parten de dos clases utilizadas como soporte:

- La clase *SerialPortExample*: La clase escanea todos los puertos COM en busca de sentencias NMEA 0183. Implementa un método para definir el puerto serie por el que se está comunicando el aparato de medida. En su constructor hay que pasarle el puerto que aparece en el administrador de dispositivos y el baud rate

del dispositivo que se puede consultar en las propiedades dentro del administrador de dispositivos. El método *getSerialPort()* es el siguiente:

```
public SerialPort getSerialPort() {

    Enumeration<?> e = CommPortIdentifier.getPortIdentifiers();
    CommPortIdentifier id = null;
    SerialPort sp = null;

    while (e.hasMoreElements()) {
        id = (CommPortIdentifier) e.nextElement();
        if (id.getName().equals(this.name)) {
            break;
        }
    }

    try {
        sp = (SerialPort) id.open("SerialExample", 30);
        sp.setSerialPortParams(this.baud, SerialPort.DATABITS_8,
                SerialPort.STOPBITS_1, SerialPort.PARITY_NONE);
        sp.disableReceiveTimeout();
        sp.enableReceiveThreshold(1);
    } catch (PortInUseException ex) {
        Logger.getLogger(SerialPortExample.class.getName()).log(Level.SEVERE, null, ex);
    } catch (UnsupportedCommOperationException ex) {
        Logger.getLogger(SerialPortExample.class.getName()).log(Level.SEVERE, null, ex);
    }

    return sp;

}
```

Figura 19 Método getSerialPort()

- La clase *TomaPuntos*: Que extiende Jframe y nos permite visualizar las coordenadas y poner botones de manejo sobre un panel con el que interactúa el usuario.

En el constructor de la clase *TomaPuntos* incializamos nuestra ventana donde visualizaremos los puntos que se van tomando, mediante un display de las coordenadas *x*, *y*, *z*. En este constructor también se crean dos objetos de la clase SerialPortExample que tiene métodos para obtener el puerto del ordenador por el que se están leyendo los datos del GPS y de la EchoSond y un objeto calculadora que hará cálculos sobre las coordenadas.

```
public TomaPuntos() {
    initComponents();
    calculadora = new CalculadoraCoordenadas();
    gps = new SerialPortExample("COM6", 115200);
    sonda = new SerialPortExample("COM10", 115200);

}
```

Figura 20 Constructor TomaPuntos

Una vez ejecutada la clase principal *TomaPuntos* se nos abre la siguiente ventana:

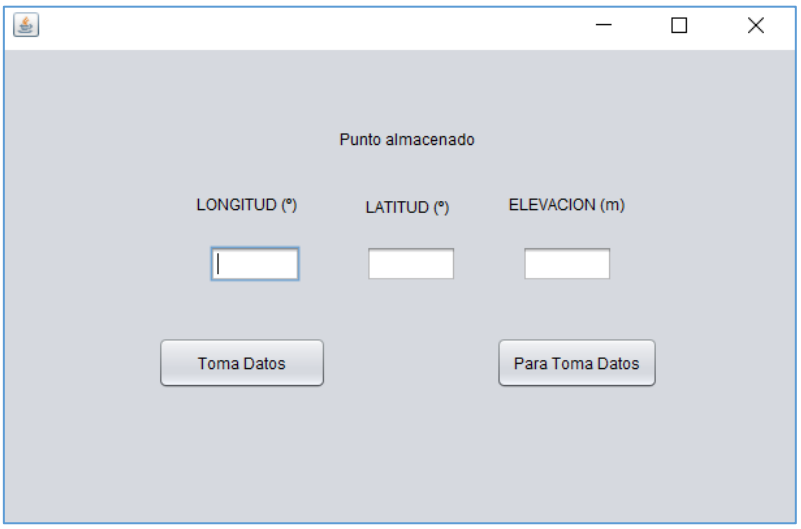


Figura 21 Ventana TomaPuntos

El botón Toma Datos es un Jbutton que al pulsarlo crea un objeto *lector* de la clase *LeeDatos* que extiende Thread

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:

    lector = new LeeDatos(this, this.gps, this.sonda);

}
```

Figura 22 JbuttonActionPerformed

A este objeto de la clase Thread se le pasa como parámetros de entrada la propia clase *TomaPuntos* (this) ya que tendrá que escribir los datos que vaya leyendo de los sensores en la clase *TomaPuntos* para que esta pueda operar con ellos y se le pasan los dos objetos de la clase S*erialPortExample* ya que necesitarán implementar el método getSerialPort(); de esta clase para poder leer de los sensores cuando se ejecute el thread.

En la clase *LeeDatos* se definen una serie de variables con las que se va a trabajar para conseguir el objetivo de la clase que es leer las cadenas de datos de entrada, seleccionar los paquetes qunecesarios de GPS y EchoSonda y enviarlos a la clase *TomaPuntos*.

```
22  public class LeeDatos extends Thread {
23
24      public boolean running = false;
25      public int contador = 0;
26      public TomaPuntos tomapuntos;
27      public SerialPortExample gps;
28      public SerialPortExample sonda;
29      public SerialPort sp_gps;
30      public SerialPort sp_sonda;
31      public BufferedReader is_gps = null;
32      public BufferedReader is_sonda = null;
33      public String is_gps_str;
34      public String is_sonda_str;
35
36
37      public LeeDatos(TomaPuntos tomapuntos, SerialPortExample gps,
38              SerialPortExample sonda) {
39          running = true;
40          this.tomapuntos = tomapuntos;
41          this.gps = gps;
42          this.sonda = sonda;
            this.start();
44      }
```

Figura 23 Variables LeeDatos + constructor

En el constructor de esta clase se pone la variable de un thread que hace que continúe el bucle (running) a true que previamente estaba definida como booleano false y se asignan los parámetros de entrada a las variables que se han definido al principio de la clase.

Se muestra a continuación el método run del thread cuyo objetivo es ir llenando los ArrayList de *x*, *y* y *z* de GPS y *z* de la sonda y pasándoselos a la clase principal, se recogen también los valores de interés hora y flag, este último indica la calidad de la señal GPS, siendo este valor de 5 cuando la precisión es centimétrica.

```
54      @Override
55      public void run() {
56
57          sp_gps = gps.getSerialPort();
58          sp_sonda = sonda.getSerialPort();
59
60          while (running) {
61
62              try {
63
64                  is_sonda = new BufferedReader(new InputStreamReader(sp_sonda.getInputStream()));
65
66                  while ((is_sonda_str = is_sonda.readLine()) != null) {
67
68                      String[] values_sonda = is_sonda_str.split(",");
69
70                      if (values_sonda[0].equals("$SDDBT")) {
71                          System.out.println(is_sonda_str);
72                          tomapuntos.z_sonda.add(Double.parseDouble(values_sonda[3]));
73
74                      }
75
76                      if (values_sonda[0].equals("$SDXDR")) {
77                          System.out.println(is_sonda_str);
78                          tomapuntos.ptch.add(Double.parseDouble(values_sonda[2]));
79                          tomapuntos.roll.add(Double.parseDouble(values_sonda[6]));
80                          break;
81                      }
82
83                  }
84
```

Figura 24 Método run de LeeDatos

```
85                      is_gps = new BufferedReader(new InputStreamReader(sp_gps.getInputStream()));
86
87                      while ((is_gps_str = is_gps.readLine()) != null) {
88
89                          System.out.println(is_gps_str);
90
91                          String[] values_gps = is_gps_str.split(",");
92
93                          if (values_gps[0].equals("$GNGGA")) {
94
95                              tomapuntos.lon_gps.add(Double.parseDouble(values_gps[4]) / 100);
96                              tomapuntos.lon_position.add(values_gps[5]);
97                              tomapuntos.lat_gps.add(Double.parseDouble(values_gps[2]) / 100);
98                              tomapuntos.lat_position.add(values_gps[3]);
99                              tomapuntos.z_gps.add(Double.parseDouble(values_gps[9]));
100
101                             tomapuntos.flag.add(values_gps[6]);
102                             tomapuntos.hora.add(values_gps[1]);
103
104                             break;
105                         }
106                     }
107
108                 } catch (IOException e) {
110                     e.printStackTrace();
111                 }
112
113                 tomapuntos.Display(tomapuntos.lat_gps.get(contador),
114                         tomapuntos.lon_gps.get(contador), tomapuntos.z_gps.get(contador)
115                         - tomapuntos.h - tomapuntos.z_sonda.get(contador));
116
117                 contador = contador + 1;
118
119             }
        }
```

Figura 25 Método run de LeeDatos (cont.)

Se obtienen los puertos por los que se van a leer los datos de la sonda y del GPS mediante el método getSerialPort() de la clase *SerialPortExample* (líneas 59-60), posteriormente se lee la cadena que entra mediante las conchas que se han visto en clase que convierten la entrada en un string (líneas 66-67 y 90-91), el motivo de hacer

el while es que se continúe la lectura y no se quede solo en la primera línea. En el caso de la EchSonda el while tiene el mismo motivo, pero se hace un if para coger valores solo de la línea que nos interesa, la que nos da la profundidad, es el mensaje que empieza por $SDDBT y la que nos da valores de pitch y roll es $SDXDR, el break se utiliza en los dos casos para salir de la lectura de ese puerto una vez que tenemos los datos que queremos y pasamos a almacenar el siguiente valor, ya sea de sonda o de GPS. El envío de datos del GPS es mucho más rápido que el de la sonda, por eso se coloca la lectura de la EchoSonda primero, en cuanto se hace el break se lee del GPS, si lo hicieramos al revés no aseguraríamos tomar medidas en los mismos tiempos, aunque esto ocurriría únicamente en la primera toma de datos.

Finalmente el método run finaliza llamando al método de la clase principal Display que sacará por la ventana los valores de longitud, latitud y profundidad calculada teniendo la consideración de que esta profundidad todavía no está corregida con los valores de pitch y roll porque no queremos que se realicen más operaciones en el interior del thread. Además se incrementa el contador de modo que se apunte al índice pertinente de los ArrayList.

Los valores almacenados en los ArrayList se utilizan en cada ciclo del run para hacer el display:

```
public void Display(Double lat, Double lon, Double z) {

    lonjText.setText(Double.toString(lat));
    latjText.setText(Double.toString(lon));
    elevjText.setText(Double.toString(z));

}
```

Figura 26 Métodos Display

Una vez se detiene la toma de datos poniendo running a false mediante el método lector.stopit() el jbutton correspondiente lanza los métodos transformaCoordenadas() y creaFicheroUTM.

```
145     private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
146         // TODO add your handling code here:
147
148         lector.stopit();
149         transformaCoordenadas();
150         creaFicheroUTM();
151
152     }
```

Figura 27 JButton2ActionPerformed

El método transformaCoordenadas() del objeto calculadora pasa de coordenadas geográficas decimales a coordenadas UTM, se podrá consultar el proceso matemático en el Anexo II dónde estará todo el código de esta parte, no se desarrolla en este apartado ya que únicamente se aplican fórmulas matemáticas que carecen de interés para la tesina. Se calculan todas las *x* y las *y* en la proyección UTM y las almacena en un ArrayList.

```
private void transformaCoordenadas() {

    for (int i = 0; i < lat_gps.size(); i++) {


        calculadora.calculo(lat_gps.get(i), lat_position.get(i), lon_gps.get(i), lon_position.get(i));

        x_utm_gps.add(calculadora.X);
        y_utm_gps.add(calculadora.Y);

    }
}
```

Figura 28 Método transformaCoordenadas()

Una vez realizado este cálculo se crea un fichero con coordenadas UTM haciendo uso del método creaFicheroUTM();

```
211     private void creaFicheroUTM() {
212
213         FileWriter fichero = null;
            PrintWriter pw = null;
215
216         try {
217             fichero = new FileWriter("UTM.txt");
218             pw = new PrintWriter(fichero);
219
220             for (int i = 0; i < z_gps.size(); i++) {
221
222                 calculadora.recalculo_z(z_sonda.get(i), ptch.get(i), roll.get(i));
223
224                 z_terreno.add(z_gps.get(i) - h - calculadora.z_OK + geoid_diff);
225
226                 if (z_sonda.get(i) != 0 && z_sonda.get(i) != 0.2) {
227
228                     pw.println(x_utm_gps.get(i) + "," + y_utm_gps.get(i) +
229                             "," + z_terreno.get(i)+ "," + flag.get(i)+ "," +
230                             hora.get(i)+ "," + z_gps.get(i)+ "," + z_sonda.get(i));
231                 }
232             }
233
            } catch (Exception e) {
                e.printStackTrace();
236         } finally {
237             try {
238                 // Nuevamente aprovechamos el finally para
239                 // asegurarnos que se cierra el fichero.
240                 if (null != fichero) {
241                     fichero.close();
242                 }
            } catch (Exception e2) {
                e2.printStackTrace();
245             }
246         }
247         System.out.println("Fichero creado");
248     }
```

Figura 29 Método creaFicheroUTM()

Se genera un *fichero* “UTM.txt” que es un objeto de la clase *FileWriter*, es necesario también crear un escritor (*pw*) de la clase *PrintWriter*. Se hace un bucle for del tamaño de los ArrayList rellenados por el thread *lector* (se pone z_gps en la línea 220, pero cualquier Array de los obtenidos en la lectura valdría ya que todos tienen el mismo tamaño), el objeto calculadora recalcula la *z* de la sonda (valor de profundidad de la EchoSonda) teniendo en cuenta el pitch y el roll del USV en cada instante de tiempo (Figura 29 línea 222) y finalmente se obtiene la *z* real del terreno con las consideraciones vistas en el apartado 6.1.2.2 (línea 224).

```
    public void recalculo_z(Double z_sonda, Double pitch, Double roll){

        z_roll = Math.abs(Math.cos(roll)) * z_sonda;
        z_OK = Math.abs(Math.cos(pitch)) * z_roll;

    }

}
```

Figura 30 Método recalculo_z

Tras las primeras pruebas nos percatamos de que en ocasiones la sonda arrojaba medidas incorrectas con valor 0, por lo que no se escriben las líneas que contengan esa medida de la EchoSonda (if de la línea 226).

En este método tuvimos el problema de que se ponía un retorno de carro al escribir la línea ya que se desconocía que el método println ya crea por defecto ese retorno de carro, después a la hora de leer el fichero se leían líneas en blanco cuando pensabamos que se debían leer valores de coordenadas.

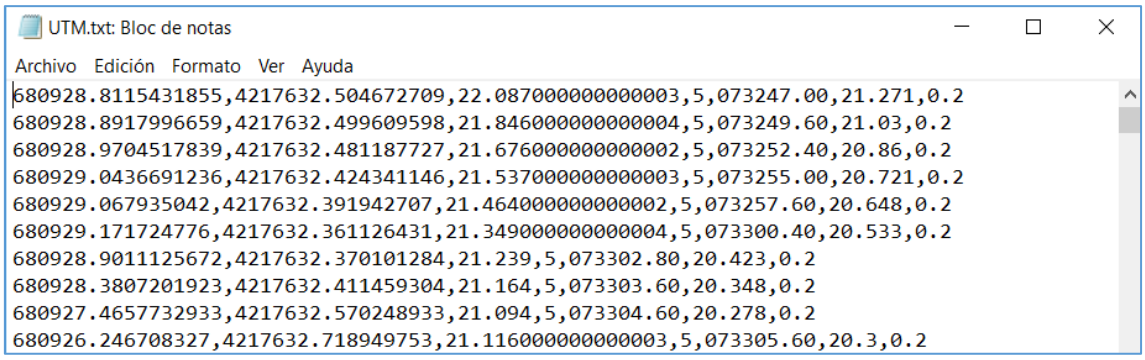


Figura 31 Ejemplo fichero

Vemos finalmente sobre un editor de texto (en este caso el bloc de notas), como queda el fichero que contiene la nube de puntos del que haremos uso en postproceso para obtener los resultados deseados.

6.1.3 Triangulación + MDT

El siguiente paso de la tesina es, crear a partir de la nube de puntos obtenida, una triangulación de Delaunay con el objetivo de obtener a partir de esta un MDT, que se define como *una estructura numérica de datos que representan la distribución espacial de una variable cuantitativa y continua* (FELICÍSIMO, A. M., 2014), la superficie topográfica real se aproxima a una superficie matemática discreta formada por superficies planas triangulares definidas a partir de los puntos de coordenadas obtenidos (nube de puntos).

Los algoritmos que se usan se basan en la triangulación de Delaunay con la que se construye una triangulación óptima para la representación del terreno. Estos algoritmos crean triángulos lo más regulares posibles y dónde la longitud de los lados se minimiza

por lo que intuitivamente se aprecia que la red de triángulos generada será la que ofrezca una imagen más fiel del terreno real al permitir una interpolación coherente de los triángulos en los que no existirá gran variación de alturas entre sus vértices.

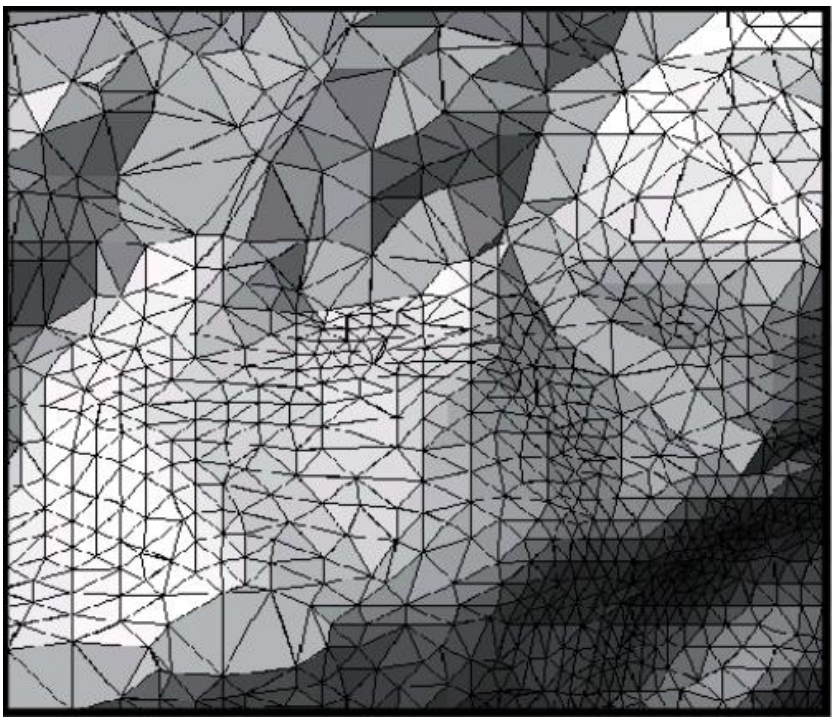


Figura 32 Ejemplo MDT

6.1.3.1. Triangulación de Delaunay

Existe más de una posible combinación de triángulos a partir una nube de puntos, las propiedades fundamentales de toda triangulación es que no puede existir ningún punto inconexo en la nube ni ningún vértice dentro de otro triángulo existente en la triangulación. A la hora de representar el terreno *una triangulación T1, es mejor que otra T2, cuando el menor ángulo de los triángulos de T1 es mayor que el menor ángulo de los triángulos de T2* (PRIEGO DE LOS SANTOS, J. E.; PORRES DE LA HAZA, M. J., s.f.). Partiendo de esta base se hace una triangulación de Delaunay porque por sus propiedades, que veremos en este apartado, es la que crea una red de triángulos irregulares (TIN) lo más regulares posibles. Vemos en la siguiente imagen una red triangulada por este algoritmo (dcha.) frente a otra triangulada con otro algoritmo (izq.)

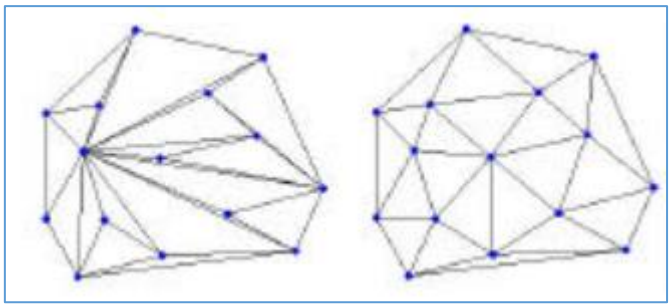


Figura 33 Triangulación Delaunay vs otra tiangulación

Vamos a estudiar a continuación las propiedades de este tipo de triangulación y posteriormente se expondrá el algoritmo utilizado en el desarrollo de nuestro programa. Se añadirá el código que veremos en el siguiente apartado.

Toda triangulación de Delaunay de una nube de puntos D, siendo D = *{d1,d2,d3,…,dn}* cumple la siguiente condición:

- Círculo circunscrito vacío: Tres puntos pertenecientes a D son vértices de un mismo triángulo, si y solamente si, su círculo circunscrito no contiene ningún otro punto del conjunto D.

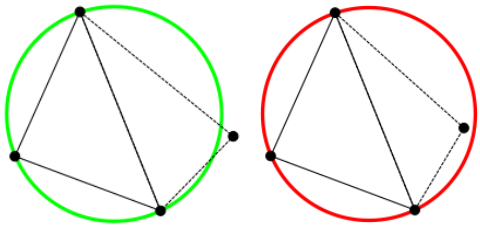


Figura 34 En verde se cumple propiedad 1. En rojo se incumple

Cumpliendo esta condición se maximiza el ángulo mínimo de la triangulación.

6.1.3.2 Algoritmo Incremental

Para construir la triangulación se sigue un algoritmo incremental a partir de una triangulación D inicial envolvente (SuperTriangle) que engloba toda la nube de puntos.

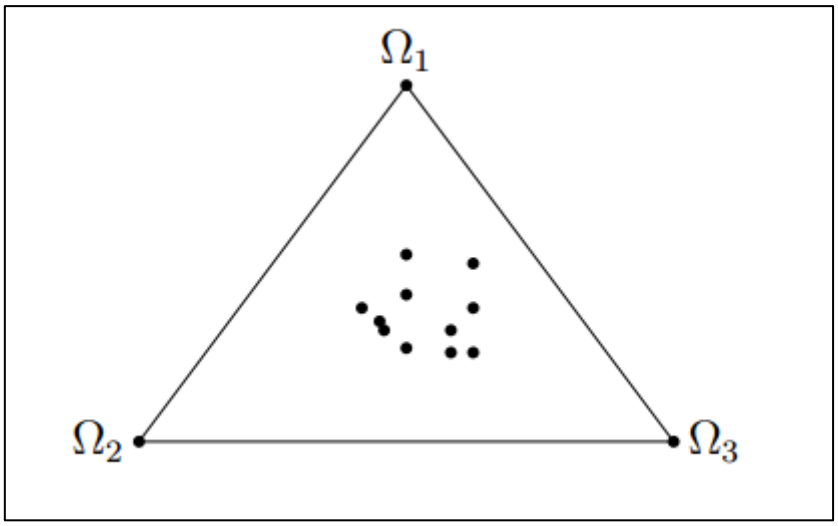


Figura 35 Triángulo ficticio inicial

El algoritmo empleado ha sido el incremental de Bowyer-Watson que procede de la siguiente manera:

1. Creación del superTriangle inicial: que se crea a una distancia 3 veces mayor a los valores máximos absolutos en *x* e *y* de la nube de puntos, esta es una construcción inicial *ampliamente utilizada en la aplicación de los algoritmos de triangulación incrementales* (HARIJAONA RAZAFINDRAZAKA, F., 2009).
2. Sea P un conjunto de puntos y D (P_r) una triangulación de $P_r \subset P$ en la etapa r. La triangulación D (P_r + 1) se obtiene insertando un punto p_r, tomado aleatoriamente de P, en D (P_r).
3. A continuación se buscan todos los triángulos en los que p_r se encuentra dentro de su circúlo circunscrito. Esta condición se comprueba mediante el predicado geométrico Incircle (A, B, C, D) que devuelve true si el punto D se encuentra contenido por el circuncírculo de A, B, C.

$$\text{Incircle}(A,B,C,D) \iff \det \begin{pmatrix} x_A & y_A & x_A^2+y_A^2 & 1 \\ x_B & y_B & x_B^2+y_B^2 & 1 \\ x_C & y_C & x_C^2+y_C^2 & 1 \\ x_D & y_D & x_D^2+y_D^2 & 1 \end{pmatrix} > 0$$

Figura 36 Condición geométrica Incircle

Esta condición se cumple si el camino ABC se recorre en sentido antihorario, que, en términos de coordenadas, puede expresarse numéricamente de la siguiente forma:

$$\mathsf{CClockwise}(A,B,C) \iff \det \begin{pmatrix} x_A & y_A & 1 \\ x_B & y_B & 1 \\ x_C & y_C & 1 \end{pmatrix} > 0.$$

Figura 37 Condición sentido antihorario

4. Se eliminan todos estos triángulos de la solución. Quedando una cavidad convexa contenedora del punto.
5. Se une el punto mediante aristas con todos los vecinos que forman la cavidad convexa (figura 38 dcha.).
6. El último paso del algoritmo es eliminar las aristas que contengan en sus extremos puntos del triángulo inicial (SuperTriangle).

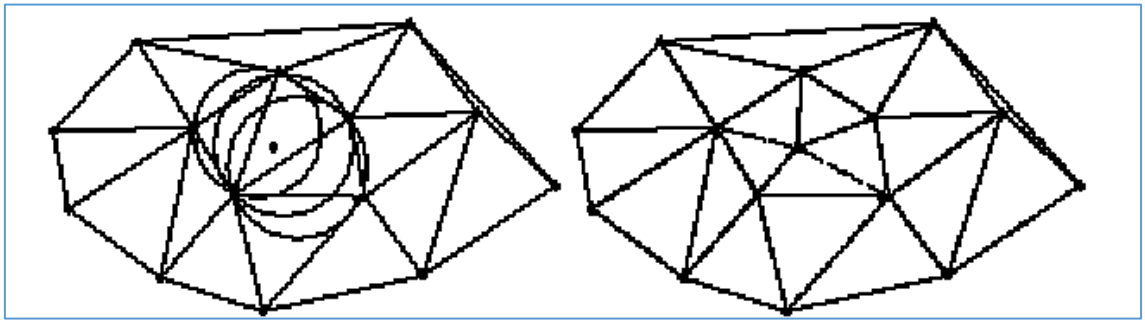


Figura 38 Algoritmo Bowyer-Watson

6.1.3.3 Código

El proyecto Java tiene su clase propia *Delaunay* cuyo objetivo es leer de un fichero una serie de puntos *{x, y, z}* y aplicar el algoritmo de triangulación de Bowyer-Watson. Esta clase ha sido obtenida del repositorio de GitHub, el código fuente de libre acceso proporcionado por el usuario slemenik, pertenece a un proyecto de un curso avanzado de computación gráfica de la facultad de Informática de la Universidad de Liubliana, Eslovenia. Tiene 3 métodos en total, uno que lee de fichero y almacena la lista de puntos, el que realiza el algoritmo de triangulación y otro que almacena y saca por

pantalla la lista de vértices de la triangulación y la lista de caras con sus vértices correspondientes, también puede crear un *archivo.obj* que se puede visualizar en 3D. Esta lista de caras la utilizaremos más adelante para la representación gráfica de la triangulación.

```
public class Delaunay {

    public ArrayList<String> faceList;
    public List<Point> pointList;
    public HashMap<Integer, String> vertexMap;//id=>"v x y z"
//    public List<Triangle> triangulation;
    public HashSet<Triangle> triangulation;


    public Delaunay() {

        faceList = new ArrayList<>();
        vertexMap = new HashMap<>();//id=>"v x y z"
        pointList = new ArrayList<>();
//        triangulation = new LinkedList<>();
        triangulation = new HashSet<>();

        System.out.println("Started...");
        pointList = getParsedData();
        HashSet<Triangle> triangulation = getTriangulation(pointList);
        makeOBJ(triangulation, true);
        System.out.println("Finished.");
```

Figura 39 Clase Delaunay con su constructor

El primer método al que se llama es a getParseData();

```
//1. Point cloud data
private  List<Point> getParsedData() {
    //.asc only - format: x;y;z\n (e.g. ARSO Lidar DMR)
    List<Point> pointList = new ArrayList<>();
    try (BufferedReader br = new BufferedReader
    (new FileReader(new File("UTM_ok.txt")))) {
        String line;
        int id = 1;
        while ((line = br.readLine()) != null) {
            String[] coordinatesString = line.split(",");
            Point p = new Point(Double.parseDouble(coordinatesString[0]),
                                Double.parseDouble(coordinatesString[1]),
                                Double.parseDouble(coordinatesString[2]),
                                id++);
            pointList.add(p);
        }
    } catch (Exception e){
        System.out.println("izjema" + e);
    }

    return pointList;

}
```

Figura 40 Método getParseData();

La salida del método es una lista de objetos de puntos, estos objetos contienen las coordenadas *x, y, z* y un identificador como último campo, se hace uso de la concha que permite leer los datos binarios del fichero y pasarlos a string. Estos string se parsean dividiendo por comas y en cada línea de lectura (bucle while) se crea un objeto de la clase punto, pasando el string a double.

El siguiente método al que se llama es el que realiza la función que realmente define a esta clase, el getTriangulation(), al que se le pasa la lista de puntos obtenida con el método anterior y devuelve un HashSet de objetos triángulos. Un triángulo se crea en el constructor pasándole tres vértices y además crea en el propio constructor 3 aristas (*edge*) que unen los vérices.

```
public static class Triangle{

    Point p1,p2,p3;
    Edge e1, e2, e3;
    int id;

    Triangle(Point p1, Point p2, Point p3, int id)  {
        this.id = id;

        //point are ALWAYS from lowest to highest
        Point[] pointsArray = new Point[]{p1,p2,p3};
        Arrays.sort(pointsArray);
        this.p1 = pointsArray[0];
        this.p2 = pointsArray[1];
        this.p3 = pointsArray[2];

        this.e1 = new Edge(p1, p2, this);
        this.e2 = new Edge(p2, p3, this);
        this.e3 = new Edge(p3, p1, this);
    }
```

Figura 41 Objeto Triangle

Vamos a ver sobre el código como se implementan los 6 pasos mencionados en la explicación del algoritmo:

- Paso 1: supertTriangle

```
//3D surface reconstruction.
private  HashSet<Triangle> getTriangulation(List<Point> pointList){

    //Bowyer-Watson algorithm
    int triangleID = 1;
    List<Triangle> triangulation = new LinkedList<>();
    double M = getMaximumAbsoluteCoordinate(pointList);
    Triangle superTriangle = new Triangle(  new Point(3*M, 0,0, -1),//-1
            new Point(0,3*M, 0, -2),//-2
            new Point( -3*M, -3*M, 0, -3), -1);//-3
    triangulation.add(superTriangle);
    HashSet<Triangle> solution = new HashSet<>();
```

Figura 42 getTriangulation("args")_1

Se inicializa además una lista de objetos triángulos (*triangulation*) que va a servir para ir apuntando a los triángulos que resultan válidos (como se ha explicado el algoritmo consiste en ir quitando triángulos no válidos para dejar cavidades convexas y rellenar estas con triángulos válidos) y se inicializa el HashSet que se devolverá como argumento de salida (*solution*).

- Paso 2: Inserción de puntos. Se realiza un bucle for que recorre la lista de puntos. El método getTriangulation("args") finaliza cuando ya no quedan puntos por insertar, al finalizar este bucle, en la siguiente línea se retorna el HashSet<Triangle> solution y concluye el método.
- Los siguientes 4 pasos se explicarán sobre la misma imagen del código:

```
88          for (Point point : pointList) {
89  //            System.out.println();
90              if (point.id % 10000 == 0)System.out.println("no está" + point);
91              HashSet<Edge> edgelstAppearance = new HashSet<>();
92              HashSet<Edge> polygon = new HashSet<>();
93
94              Iterator<Triangle> i = triangulation.iterator();
95              while (i.hasNext()) {
96                  Triangle triangle = i.next();
97                  if (inCircle(point, triangle.p1, triangle.p2, triangle.p3)) {
98                      i.remove();// Quitas de la triangulación
99                      //el triángulo contenedor
100                     solution.remove(triangle); // Quitas de la solución
101                     // el triángulo contenedor
102                     if (edgelstAppearance.contains(triangle.e1)) {
103                     //En caso de que ya exista una arista compartida,
104                     //esta se borra para crear la cavidad
105                         polygon.remove(triangle.e1);
106                     } else {//1st appearance
107                         edgelstAppearance.add(triangle.e1);
108                     // Si es la primera vez que aparece esta arista se apunta,
109                     //si no vuelve a aparecer es porque no está compartida con
110                     //BadTriangles y se usa para generar un nuevo triángulo
111                         polygon.add(triangle.e1);
112                     }
113
114                     if (edgelstAppearance.contains(triangle.e2)) {
115                         polygon.remove(triangle.e2);
116                     } else {
117                         edgelstAppearance.add(triangle.e2);
118                         polygon.add(triangle.e2);
119                     }
120
121                     if (edgelstAppearance.contains(triangle.e3)) {
122                         polygon.remove(triangle.e3);
123                     } else {
124                         edgelstAppearance.add(triangle.e3);
125                         polygon.add(triangle.e3);
126                     }
127                 }
128             }
```

Figura 43 getTriangulation("args")_Pasos 3, 4 ,5 y 6

```
130                 for (Edge edge : polygon) {
131                     Triangle newTriangle =
132                             new Triangle(point, edge.p1, edge.p2, triangleID++);
133                     triangulation.add(newTriangle);
134                     if (hasNoSuperTrianglePoint(newTriangle, superTriangle)) {
135                         solution.add(newTriangle);
136                     }
137                 }
138             }
139
140             return solution;
141         }
```

Figura 44 getTriangulation("args")_Pasos 3, 4 ,5 y 6 (cont.)

Se entra en el bucle for de inserción de puntos y tras comprobar que existe el punto se crean dos HashSet nuevos en cada iteración de este bucle:

- *edge1stAppearance*: cuya misión es apuntar las aristas de los triángulos que contienen al punto en su circuncírculo.
- *polygon*: que únicamente se apunta las aristas de los triángulos contenedores que aparecen por primera vez. Observar que pueden existir triángulos contenedores que compartan aristas (Figura 38 izq.).

Se crea un iterador de la lista de triángulos válidos. En la siguiente línea (95) se abre un bucle while que se ejecuta mientras existan elementos de la lista *triangulation.* Se pasa ahora a comprobar la condición geométrica *inCircle*, línea 97, donde se sigue dentro de esa condición (es decir el punto cae dentro del circuncírculo del triángulo sobre el que se está iterando) si el determinante es mayor que 0, este método nos devuelve un booleano a true.

```
private static boolean inCircle(Point pt, Point v1, Point v2, Point v3) {

    double ax = v1.x;
    double ay = v1.y;
    double bx = v2.x;
    double by = v2.y;
    double cx = v3.x;
    double cy = v3.y;
    double dx = pt.x;
    double dy = pt.y;

    double  ax_ = ax-dx;
    double  ay_ = ay-dy;
    double  bx_ = bx-dx;
    double  by_ = by-dy;
    double  cx_ = cx-dx;
    double  cy_ = cy-dy;
    double det=  (
                    (ax_*ax_ + ay_*ay_) * (bx_*cy_-cx_*by_) -
                    (bx_*bx_ + by_*by_) * (ax_*cy_-cx_*ay_) +
                    (cx_*cx_ + cy_*cy_) * (ax_*by_-bx_*ay_)
    );

    if (ccw ( ax,  ay,  bx,  by,  cx,  cy)) {
        return (det>0);
    } else {
        return (det<0);
    }

}

private static boolean
    ccw(double ax, double ay, double bx, double by, double cx, double cy) {
    return (bx - ax)*(cy - ay)-(cx - ax)*(by - ay) > 0;
}
```

Figura 45 Método inCircle

El triángulo sobre el que se está iterando pasa a ser no válido automáticamente y se borra de la lista *triangulation* y del HashSet *solution.* Los if_else que siguen se encargan de rellenar los HashSet *edge1stappearance* y *polygon*. Cuando finaliza el bucle while (ya se ha iterado sobre todos los triángulos de la lista *triangulation*) se recorren las aristas que contiene polygon (quedan las aristas que forman la cavidad convexa), es el bucle for de la línea 130 y se van creando triángulos que rellenan la cavidad convexa uniendo los vértices de cada arista con el punto que ha sido insertado en el paso 1 (Figura 38 dcha.), esto se puede ver en línea 132, finalmente si los nuevos triángulos formados no tienen vértices que pertenezcan al superTriangle inicial estos se añaden al HashSet *solution*.

Por último se llama al método makeOBJ(“args”) que crea un ArrayList de cadenas llamado *faceList* que contiene el grafo de las caras, donde cada línea determina que puntos de la lista de puntos son vértice de cada triángulo.

```
143     //Output and Implementation
144     private  void makeOBJ(HashSet<Triangle> triangles, boolean writeToFile) {
145
147         for (Triangle triangle : triangles) {
148
149             if (!vertexMap.containsKey(triangle.p1.id))
150                 vertexMap.put(triangle.p1.id, "v "
151                     + triangle.p1.x +" " + triangle.p1.y + " " + triangle.p1.z );
152             if (!vertexMap.containsKey(triangle.p2.id))
153                 vertexMap.put(triangle.p2.id, "v "
154                     + triangle.p2.x + " " + triangle.p2.y + " " + triangle.p2.z );
155             if (!vertexMap.containsKey(triangle.p3.id))
156                 vertexMap.put(triangle.p3.id, "v "
157                     + triangle.p3.x + " " + triangle.p3.y + " " + triangle.p3.z );
158
159             String facesString = String.format("f %d %d %d",
160                     triangle.p1.id, triangle.p2.id, triangle.p3.id);
161             if (!faceList.contains(facesString)) {
162                 faceList.add(facesString);
163             }
164         }
165         PrintWriter writer = null;
166         if (writeToFile) {
167             try {
168                 writer = new PrintWriter("output.obj", "UTF-8");
169             } catch (Exception e) {
170                 e.printStackTrace();
171             }
172         }
```

Figura 46 Método makeOBJ

6.1.3.4 Pseudocódigo

Mostramos el pseudocódigo generado para explicar el algoritmo Bowyer-Watson que tiene como salida la triangulación que representa el terreno.

```
1. Entrada: nube de puntos x, y, z
2. Generación SuperTriangle
3.   Generación   List   triangulation   (inicialmente   solo   contiene
SuperTriangle)
4. Generación List solution (inicialmente vacía)
5. for 1 : n (número de puntos)

6.    while (triangulation.iterator) Se recorren los triángulos válidos

7.          if (inCircle) Punto contenido en un circuncírculo, condición
            errónea
8.          Se borra triángulo de la solución (lista triangulation)
9.          Se genera la cavidad convexa, apuntando las aristas que la
            contienen
10.         end

11.   end

12.   for 1 : edge (número de aristas)
13.   Se crean tantos triángulos como aristas forman la cavidad
14.   Se añaden estos triángulos a la List solution si no tienen vértices
      que pertenezcan a SuperTriangle
15.   end

16. end
```

6.1.3.5 Obtención MDT: interpolación lineal

Una vez conocidos los triángulos que forman la triangulación de Delaunay (vértices con *x, y, z* definidos) hay que estimar los valores de cota de todos los puntos del terreno estudiado, a nivel digital, esto quiere decir que hay estimar el valor de z a todos los pixeles de la pantalla cuya *x* e *y* se puede extrapolar de a los valores *x* e *y* de la nube de puntos conocida. Se realiza para ello una interpolación lineal mediante el siguiente procedimiento:

1. Se cambia de dominio la nube de puntos de modo que los límites del Jpanel (gráfico) $\{pix\_x_{min} = 0;\ pix\_x_{max} = ancho_{JPanel};\ pix\_y_{min} = 0;\ pix\_y_{max} = alto_{JPanel}\}$ donde se van a representar los puntos, la triangulación, el MDT y el curvado se correspondan con los valores máximos y mínimos de la longitud y la latitud de la nube de puntos.
2. Una vez realizado el cambio de dominio, se crea una lista de triángulos a partir de la lista de caras que arroja la clase Delaunay
3. Se crea a partir de esta lista, otra lista de objetos de la clase Shape que tiene el método contains() con el que vamos a comprobar si un punto se encuentra dentro del triángulo que consultemos.
4. Se halla el plano de ese triángulo, cálculo que se almacena para no tener que repetirlo cada vez que encontramos un pixel en un triángulo del que ya se ha calculado su plano. La ecuación del plano $P\ x + Q\ y + R\ z\ +\ S\ = 0$ que pasa por los vértices del triángulo $A = (A_x, A_y, A_z), B = (B_x, B_y, B_z)$ y $C = (C_x, C_y, C_z)$ se obtiene de desarrollar el determinante:

$$\begin{vmatrix} x - A_x & y - A_y & z - A_z \\ B_x - A_x & B_y - A_y & B_z - A_z \\ C_x - A_x & C_y - A_y & C_z - A_z \end{vmatrix} = 0$$

Ecuación 6 Determinante cálculo coeficientes plano

5. Se obtiene el valor de *z* interpolado evaluando en la ecuación del plano, donde *x* e *y* toman los valores de los pixeles correspondientes al punto evaluado.

$$z = \frac{S - P * pix_x - Q * pix_y}{R}$$

Ecuación 7 Interpolación de z

6.1.3.6 Código

Vamos a ver ahora la parte del código que desarrolla esta interpolación lineal. Esta interpolación no va a tener una clase propia que la desarrolle como hemos visto que hacíamos para la triangulación. Se incluye en la clase *Visualiza2D* que se encarga de mostrar por pantalla los resultados obtenidos. Esta clase se explicará con más detalles en el apartado 6.1.5.1. Vamos a centrarnos en este apartado únicamente en las líneas de código dedicadas a la interpolación lineal.

- Paso 1: este paso es fundamental no solo para realizar este cálculo, si no para hacer las representaciones gráficas en 2D y 3D que es uno de los objetivos del principales de la tesina. Lo ejecuta el método cambiaDominio("args") que implementa la ecuación 7 de escalado en la que se pasa de un valor p_x dentro de una escala de tamaño ($x_{max} - x_{min}$) a un valor de pixel dentro de la escala del tamaño del gráfico que usemos para representar los puntos:

$$pixel_x = \frac{p_x - x_{max}}{x_{min} - x_{max}} * (Size_{JPanel})$$

Ecuación 8 Ecuación de escalado

```
public int[] cambioDominio(int anchoImagen, int altoImagen, double longitud,
        double longitud_max, double longitud_min, double latitud,
        double latitud_max, double latitud_min, double altitud,
        double altitud_max, double altitud_min) {

    int x_img, y_img, gray;
    int escala_gris = 65535;

    x_img = (int) Math.round(((longitud - longitud_max) /
            (longitud_min - longitud_max)) * (anchoImagen - 1));
    y_img = (altoImagen - 1) - (int) Math.round(((latitud - latitud_max) /
            (latitud_min - latitud_max)) * (altoImagen - 1));
    gray = escala_gris - (int) Math.round(((altitud - altitud_max) /
            (altitud_min - altitud_max)) * escala_gris);

    int[] escalado = {x_img, y_img, gray};

    return escalado;

}
```

Figura 47 Método cambioDominio

Para el valor de *z* se sigue el mismo procedimiento asociando las cotas a una escala de grises en la que el valor 0 se representa completamente negro equivale a la cota más baja y la cota más alta se representa con el valor 65535, pintado completamente de

blanco. Este método devuelve los valores transformados al dominio del JPanel y se utilizará en la representación de los puntos.

- Paso 2: Hacemos un bucle for que recorre la *faceList* de salida de la clase Delaunay, previamente se ha creado un objeto delaunay en el constructor de la clase en la que nos encontramos (*Visualiza2D*). Dentro del bucle for leemos en cada pasada los vértices que forman una cara, a estos vértices les pasamos el método de escalado que acabamos de ver y creamos un objeto de la clase *Triangle* (que define un triángulo mediante sus tres vértices) que se almacena en un ArrayList de triángulos definido como null en el inicio de la clase *Visualiza2D*.

```
372         for (int i = 0; i < delaunai.faceList.size(); i++) {
373
374             //System.out.println(delaunai.faceList.size());
375
376             String linea = delaunai.faceList.get(i);
377             String[] value = linea.split(" ");
378
379             int vertex_1 = Integer.parseInt(value[1]);
380             int vertex_2 = Integer.parseInt(value[2]);
381             int vertex_3 = Integer.parseInt(value[3]);
382
383             escalado1 = cambioDominio(anchoImagen, altoImagen,
384                     ventana.lon_gps.get(vertex_1-1), extremo_lon[1],
385                     extremo_lon[0], ventana.lat_gps.get(vertex_1-1),
386                     extremo_lat[1], extremo_lat[0], ventana.z_terreno.get(vertex_1-1),
387                     extremo_z[1], extremo_z[0]);
388             escalado2 = cambioDominio(anchoImagen, altoImagen,
389                     ventana.lon_gps.get(vertex_2-1), extremo_lon[1],
390                     extremo_lon[0], ventana.lat_gps.get(vertex_2-1),
391                     extremo_lat[1], extremo_lat[0],
392                     ventana.z_terreno.get(vertex_2-1), extremo_z[1], extremo_z[0]);
393             escalado3 = cambioDominio(anchoImagen, altoImagen,
394                     ventana.lon_gps.get(vertex_3-1), extremo_lon[1],
395                     extremo_lon[0], ventana.lat_gps.get(vertex_3-1),
396                     extremo_lat[1], extremo_lat[0],
397                     ventana.z_terreno.get(vertex_3-1), extremo_z[1], extremo_z[0]);
398
399             g2.setPaint(Color.black);
400
401             g2.drawLine(escalado1[0],escalado1[1],escalado2[0],escalado2[1]);
402             g2.drawLine(escalado1[0],escalado1[1],escalado3[0],escalado3[1]);
403             g2.drawLine(escalado3[0],escalado3[1],escalado2[0],escalado2[1]);
404
405             Triangle triangulo = new Triangle(escalado1[0],escalado1[1],
406                     escalado1[2],escalado2[0],escalado2[1],escalado2[2],
407                     escalado3[0],escalado3[1],escalado3[2]);
408
409             triangulos.add(triangulo);
410         }
```

Figura 48 Rellenado del ArrayList<Triangle> triangulos

Las líneas de código de la 399 a la 403 se usan para la representación y no nos interesa explicarlas de momento.

- Paso 3: creamos objetos de la clase Shape para comprobar si un pixel está dentro del triángulo que evaluaremos, método contains();.

```
449             for (int i = 0; i < triangulos.size(); i++) {
450
451                 GeneralPath t = new GeneralPath();
452
453                 t.moveTo(triangulos.get(i).x1, triangulos.get(i).y1);
454                 t.lineTo(triangulos.get(i).x2, triangulos.get(i).y2);
455                 t.lineTo(triangulos.get(i).x3, triangulos.get(i).y3);
456                 t.lineTo(triangulos.get(i).x1, triangulos.get(i).y1);
457                 t.closePath();
458
459                 triangulos_draw.add(t);
460
461             }
```

Figura 49 Generación del ArrayList<Shape> triangulos_draw

Un GeneralPath extiende Path2D que a su vez implementa la clase Shape que contiene el método que deseamos utilizar. Creamos el GeneralPath *t* y recorremos la lista de triángulos manejado por un dibujante, cuando recibe la orden moveTo el dibujante levanta el lápiz del papel y se dirige al primer vértice de un triángulo, con las órdenes lineTo desplaza el lápiz sobre el papel recorriendo todos los vértices del triángulo, finalmente el Shape se almacena en un ArrayList de objetos de la clase Shape inicializado a null al principio de la clase. Este ArrayList se denomina *triangulos_draw* y será sobre el que se opere para comprobar si el pixel de estudio está contenido en un triángulo.

- Paso 4: se inicializan dos bucles for que van a recorrer todos los pixeles del gráfico por filas y por columnas. A su vez cada pixel tiene que recorrer la lista de triángulos “preguntando” si está contenido dentro del triángulo al que pregunta (método contains()) y calculando el plano de este triángulo. Se asume que es necesario hacer algo para que no se realice esta búsqueda por fuerza bruta. Se realizan dos acciones para minimizar el tiempo de cálculo:

    1. Cada vez que salta el método contains(), el triángulo que contiene al pixel se almacena como *triangle_old*, cuando se avanza al siguiente pixel, el

primer triángulo en ser consultado es *triangle_old* ya que es muy probable que el siguiente pixel se encuentre en este.

2. Además cada vez que un triángulo es “visitado” por primera vez este se anota como llave en un HashMap *myMap* al que se asocia como valor un objeto de la clase plano que ha calculado este plano. Por lo que cada vez que se “visite” ese triángulo se hará uso del método get() del HashMap para obtener las constantes del plano de ese triángulo.

```
463             for (int pix_x = 0; pix_x < anchoImagen; pix_x++) {
464
465                 for (int pix_y = 0; pix_y < altoImagen; pix_y++) {
466
467                     if (triangle_old != null && triangle_old.contains(pix_x, pix_y)) {
468
469                       z_pix = -(myMap.get(triangle_old).p * pix_x +
470                               myMap.get(triangle_old).q * pix_y + myMap.get(triangle_old).s)/
471                               myMap.get(triangle_old).r;
472
473
474                       obi_sup.setRGB(pix_x, pix_y, z_pix);
475
476                       Pixel_MDT pixel = new Pixel_MDT(pix_x, pix_y, z_pix);
477
478                       pixeles_MDT.add(pixel);
479
480                     }
481
482                     for (int i = 0; i < triangulos_draw.size(); i++) {
483
484                         if (triangulos_draw.get(i).contains(pix_x, pix_y)) {
485
486                             if (!myMap.containsKey(triangulos_draw.get(i))){
487
                                Plano plano = ec_plano(triangulos.get(i));
489
490                             myMap.put(triangulos_draw.get(i),plano);
491
492                             }
493
494                             z_pix = -(myMap.get(triangulos_draw.get(i)).p * pix_x +
495                                     myMap.get(triangulos_draw.get(i)).q * pix_y +
496                                     myMap.get(triangulos_draw.get(i)).s)/
497                                     myMap.get(triangulos_draw.get(i)).r;
498
499
500                             triangle_old = triangulos_draw.get(i);
501                             vertices_old = triangulos.get(i);
```

Figura 50 Recorrido de todos los pixeles del gráfico

El plano de cada triángulo solo se calcula, por tanto, una vez por cada triángulo, línea 488. El método ec_plano al que se le pasa el triángulo que toca como entrada calcula las constantes del plano.

```
public Plano ec_plano (Triangle triangulo) {

        int a1 = triangulo.x2 - triangulo.x1;
    int b1 = triangulo.y2 - triangulo.y1;
    int c1 = triangulo.gray2 - triangulo.gray1;
    int a2 = triangulo.x3 - triangulo.x1;
    int b2 = triangulo.y3 - triangulo.y1;
    int c2 = triangulo.gray3 - triangulo.gray1;
    int p = b1 * c2 - b2 * c1;
    int q = a2 * c1 - a1 * c2;
    int r = a1 * b2 - b1 * a2;
    int s = (-p * triangulo.x1 - q * triangulo.y1 - r * triangulo.gray1);

    Plano plano = new Plano (p, q, r, s);

    return plano;
}

public class Plano {

    int p, q, r, s;

    public Plano(int p, int q, int r, int s) {

        this.p = p;
        this.q = q;
        this.r = r;
        this.s = s;
    }
}
```

Figura 51 Método ec_plano y clase Plano

Las líneas de código que faltan hasta que se cierra el bucle se utilizan para la representación del MDT y son como las líneas de la 474 a la 478, se verá en el apartado 6.1.5.1.3 donde se hará referencia a esta figura.

- Paso 5: Por último queda indicar las líneas de código que implementan la ecuación 6 para la interpolación del valor de cota. Líneas de la 469 a la 471 en el caso de que el pixel esté dentro de *triangle_old* y se sigue el mismo procedimiento de las líneas 494 a la 497 para el caso de que el triángulo contenedor no sea el mismo que en el paso anterior del bucle for.

6.1.3.7 Pseudocódigo

Como se ha comentado al introducir la metodología se crea un pseudocódigo para facilitar la comprensión del algoritmo utilizado al lector. En este caso se presenta el pseudocódigo generado para explicar la interpolación lineal de los triángulos que tiene el objetivo de obtener como resultado un valor de cota en todas aquellas zonas que forman la superficie, pero de las que no tenemos información.

```
1. Entrada: Lista de caras de triángulos conocidos sus vértices
2. Cambio de dominio de los vértices de los triángulos (De coordenadas
UTM a pixeles     de pantalla)
3. Se crea una lista de Shapes triángulos que se convierten a General
Path
4. Se crea un HashMap que va llenandose con los planos calculados para
cada triángulo
5. Se genera triangle_old que será el primer triángulo en el que se
comprobará si el punto está contenido
6. for 0 : n° columnas JPanel (iteranción sobre pixeles)

7.    for 0 : n° filas JPanel (iteración sobre pixeles)

8.          if punto contenido en triangle_old
9.          Se obtiene el valor de z del pixel sobre el que se está
            iterando llamando a las constantes del plano previamente
            calculado de triangle_old
10.         Se pinta el pixel con el valor de z correspondiente en la
            posición x e y sobre la que se está iterando
11.         end

12.         for 0 : el tamaño del array de triángulos
13.               if el punto está contenido en otro triángulo que no
            sea triangle_old

14.               if todavía no se ha creado el plano y añadido al
                  HashMap
15.                     Se crea el plano y se añade al HashMap
16.                     end

17.               Se obtiene el valor de z del pixel sobre el que se
                  está iterando llamando a las constantes de ese
                  triángulo
18.               Se asocia la variable triangle_old a este último
                  triángulo
19.               Se pinta el pixel con el valor de z correspondiente en
                  la posición x e y sobre la que se está iterando
20.               end

21.         end

22.     end

23. end
```

6.1.4 Curvas de nivel o Isolíneas

Una vez obtenida la digitalización del terreno es necesario obtener de esta la información que realmente va a ser de utilidad para el usuario final del sistema, esa información son las curvas de nivel o isolíneas que indican las coordenadas obtenidas que tienen el mismo nivel de cota consultado. Este nivel de cota se fija por el usuario final dependiendo de la aplicación o del tipo de estudio que se vaya a realizar en base a estas curvas de nivel. A nivel de comprensión simple y a falta del desarrollo que viene

a continuación, la obtención de curvas de nivel se trata de unir los puntos que se han interpolado para la obtención del MDT que tienen el mismo valor de elevación.

Su obtención se basa en el clásico algoritmo de Marquing Squares que se explicará a continuación, y se desarrolla en el proyecto de Neatbeans por las clases *MarquingSquares* e *Isocell.* Se han utilizado estas clases desarrolladas por Mike Markowski, haciendo una comprensión previa del algoritmo y realizando las modificaciones necesarias para poder utilizarlas en nuestro proyecto.

6.1.4.1 Algoritmo Marquing Squares

El algoritmo Marquing Squares hace un filtrado de la imagen (MDT), que tiene una estructura de datos tipo grid, esto quiere decir que se tiene un valor de *z* correspondiente a cada pixel (*x, y*). Este filtrado consiste en una umbralización donde todos los píxeles de la imagen que superen el valor *z* de consulta se ponen a 1 y el resto a 0.

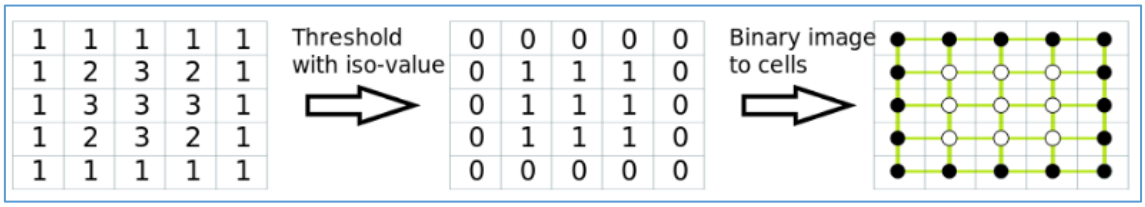


Figura 52 Umbralización: valor de consulta de z = 2

Posteriormente se crean celdas cuadradas cuyos vértices son los centros de los píxeles (el algoritmo no trabaja con píxeles únicos, si no que previamente hace agrupaciones de píxeles a los que da un valor medio, cada número que vemos en la figura 52 representa 3x3 píxeles), cada vértice tiene un valor binario de 0 o 1. A cada celda se le asigna un número binario de 4 dígitos recorriendo los vértices de cada celda en sentido horario con lo cuál podemos tener $2^4 - 1 = 15$ casos posibles que se tienen almacenados en una tabla de consulta y dependiendo de en que caso nos encontremos en cada celda se dibujará un tipo de línea. Se va a entender esto viendo la siguiente figura, donde los puntos blancos representan el valor 0 y los puntos negros el valor 1 (Los colores de los vértices están intercambiados entre la figura 52 y la 53, la primera celda de arriba a la izquierda se corresponde con el caso 2 con con el caso 13).

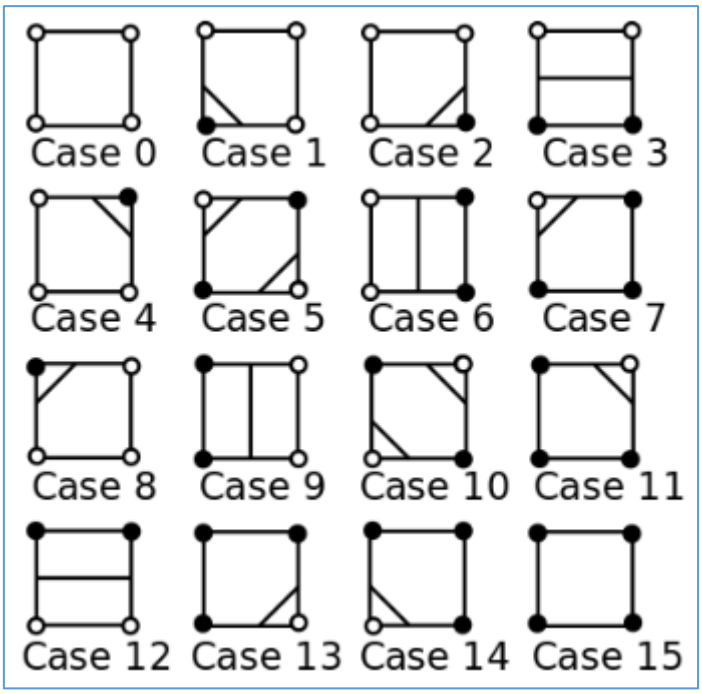


Figura 53 Tabla de consulta

Una vez asignados valores a cada celda el algoritmo trabaja recorriendo las celdas y pintando según el caso en que se encuentre cada celda, para la comprensión de este punto volvemos a visualizar la figura 52 sobre la que se han pintado las líneas correspondientes a cada caso (otra vez los colores entre los casos de la figura 53 y lo que vemos en la siguiente imagen están intercambiados).

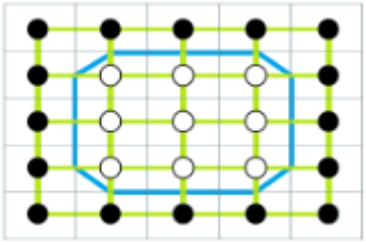


Figura 54 Isolínea

Vemos, por ejemplo, como la primera celda de arriba a la izquierda se corresponde con el caso nº 2 (teniendo en cuenta el cambio de colores). Por último, se consultan los valores originales y se hace una interpolación lineal de manera que la isolínea pase por el pixel del grupo de píxeles originales que más se acerca al valor de consulta.

6.1.4.2 Problema en condiciones de contorno

Esta clase está originalmente pensada para una imagen que esté completa (todos los píxeles contienen información). En nuestro caso el MDT no va a ocupar toda la pantalla por lo que los espacios en blanco van a ser tomados como valores por debajo del valor de consulta y vamos a tener un problema de contorno apareciendo líneas donde no deben aparecer. La superficie que se está representando es una rampa.

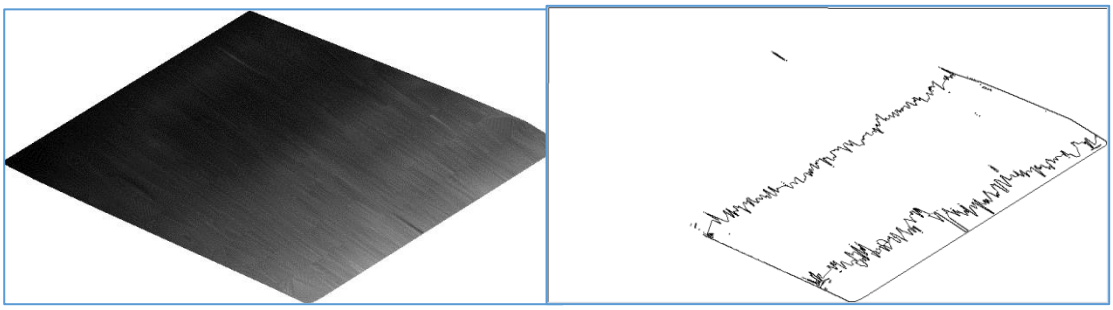


Figura 55 Ejemplo MDT

Nótese que al hacer la umbralización, en los bordes del MDT vamos a tener celdas como los casos que hemos visto (distintas de todo unos o todo ceros) donde no queremos que se pinten isolíneas porque no se corresponden con la realidad. Las únicas isolíneas que se deberían pintar son las que cruzan el MDT.

Para solucionar este hecho, se realiza una erosión, que es una de las dos operaciones fundamentales de la morfología matemática en el procesamiento de imágenes. Primero se recortan varias bandas de píxeles de la imagen de la izquierda (se ha comprobado que con 10 son suficientes) y luego se usa esta como filtro para la imagen de la derecha dibujando posteriormente sólo las líneas que están contenidas por la imagen recortada. Obteniendo el siguiente resultado:

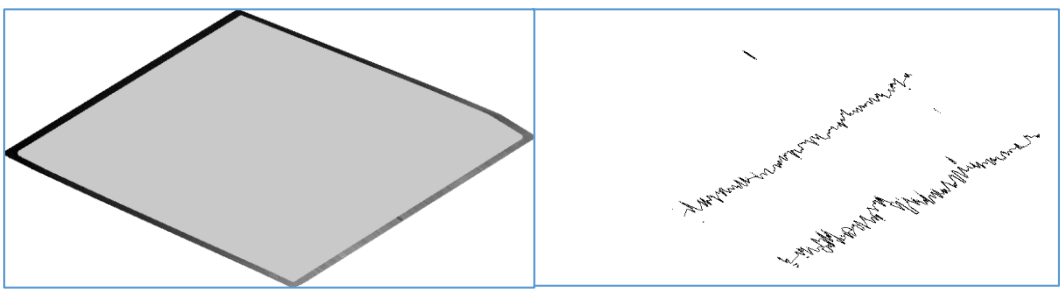


Figura 56 Filtro utilizado (en gris) para borra curvas de nivel que no son reales

Esta erosión que realiza el método erosion(); se pasa dos veces a la imagen inicial ya que para algunos niveles de consulta todavía aparecía algún problema. La erosión consiste en hacer un recorrido por los píxeles de la imagen fila por fila, se pone un booleano a false, cuando se encuentra el primer pixel de la fila se cambian los valores de los siguientes 10 píxeles y se pone el booleano a true, mientras haya dato y el booleano esté a true no se hace nada, en el momento en que se vuelve a encontrar un valor no válido, se pone el booleano a false y se eliminan lo 10 píxeles anteriores al actual, este algoritmo se repite en cada fila de píxeles de la imagen.

```
732             //EROSION 10 CAPAS
733             puntero_pix = 0;
734
735             for (int i = 0; i < altoImagen; i++) {
736
737                 for (int j = 0; j < anchoImagen; j++) {
738
739                     if (pixeles_MDT.get(puntero_pix).pixel_z != valor_nulo &&
740                             izq == false) {
741
742                         banda = puntero_pix + 9;
743
744                         for (int k = 0; k < 10; k++) {
745
746                             pixeles_MDT.remove(puntero_pix + k);
747                             pixeles_MDT.add((puntero_pix + k),
748                                     new Pixel_MDT(j + k, i, valor_nulo));
749
750                         }
751
752                         izq = true;
753
754                     }
755
756                     if (pixeles_MDT.get(puntero_pix).pixel_z == valor_nulo &&
757                             izq == true && puntero_pix > banda) {
758
759                         for (int k = 0; k < 10; k++) {
760
761                             pixeles_MDT.remove(puntero_pix - k);
762                             pixeles_MDT.add((puntero_pix - k),
763                                     new Pixel_MDT(j - k, i, valor_nulo));
764
765                         }
766
767                     }
768
769                     puntero_pix = puntero_pix + 1;
770                 }
771                 izq = false;
772             }
```

Figura 57 Método erosion()

Conociendo que es necesario hacer esta erosión para obtener las curvas correctamente, será necesario tener en cuenta este hecho y hacer un plan de navegación que cubra un poco más de terreno por los laterales del que se planea medir.

6.1.4.3 Pseudocódigo

Estas clases se consideran bastante más tediosas de explicar por su extensión y detalles (no es tan compacta como la clase delaunay), por lo que en este caso se considera suficiente el haber explicado la teoría y se va a exponer a continuación el pseudocódigo correspondiente.

```
1. Entrada: Estructura de datos tipo grid. Array multidimensional con
los valores de z de cada pixel ordenados por filas y columnas (píxeles)
2. Reordenación de píxeles en subgrupos de 3x3

3. for 0: n° datos

4.    if z_pixel distinto de valor nulo (pixel no contenido en la
      superficie) && < valor de consulta
5.    z_pixel = 0
6.    end

7.    if z_pixel distinto de valor nulo (pixel no contenido en la
      superficie) && > valor de consulta
8.    z_pixel = 1
9.    end

10. end

11. Generación de celdas (15 casos posibles) en función del valor de
      z_pixel

12. for 0 : n° celdas
13.   Consulta del tipo de celda en el que nos encontramos
14.   Interpolación sobre los dos lados por los que pasa la línea
15.   Dibujado de linea dentro de la celda entre los puntos calculados
      en la línea anterior
16. end

17. Método Erosión (Morfología matemática) del MDT 2 veces: 10 bandas
      de píxeles
      booleano izq = false

      if izq = false && pixel de consulta != valor nulo
      Se quitan los siguientes 10 píxeles de la fila y y se pone booleano
      a true
      end

      if izq = false && pixel de consulta = valor nulo
      Se quitan los anteriores 10 píxeles de la fila y y se pone booleano
      a false
      end

18. Redibujado de isolíneas que estén contenidas en el MDT
```

6.1.5 Visualización e interacción con el usuario

6.1.5.1 Representación 2D

La representación en 2D de los resultados lo realiza la clase *Visualiza2D* que extiende JFrame, por tanto es una ventana. Tiene 4 métodos que pinan sobre 4 paneles los resultados.

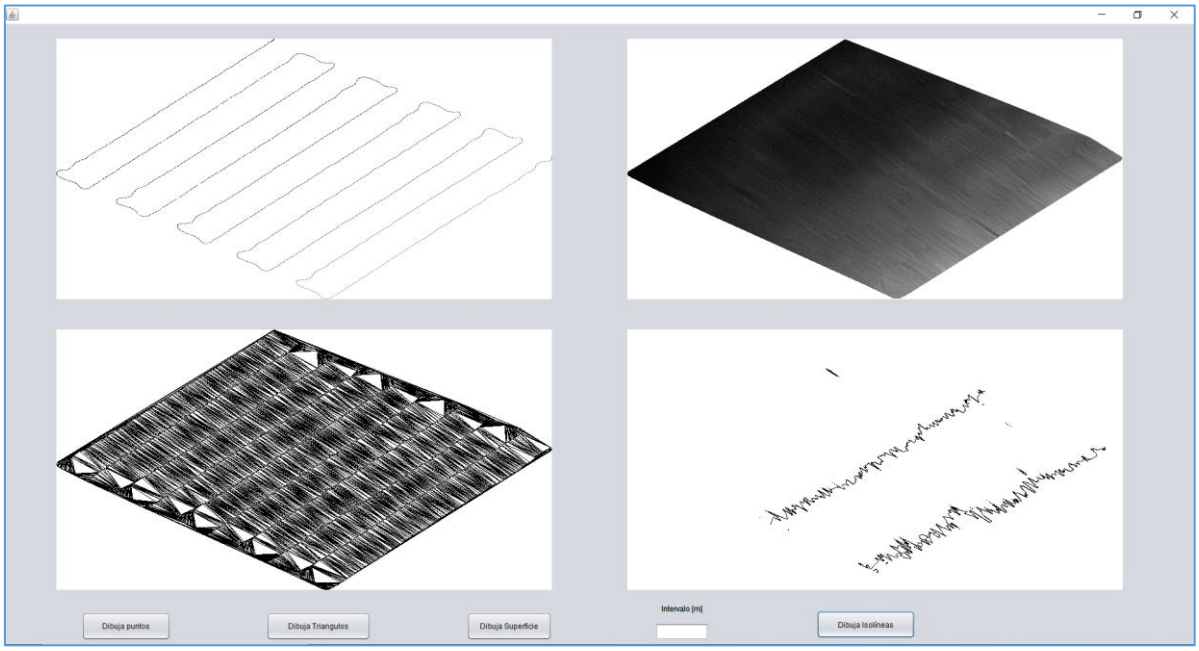


Figura 58 Ventana de resultados gráficos

6.1.5.1.1 Puntos

El método nuevaImagenPuntos() al que se le pasa como entrada el panel donde vamos a representar los puntos, crea una imagen *obi_puntos* (Figura 58 arriba, izq.) sobre la que se modifican los valores de los pixeles correspondientes a la nube de puntos, previo cambio de dominio (método cambioDominio() que ya se ha visto en el apartado 6.1.3.5).

```
324     public void nuevaImagenPuntos(JPanel lienzo) {
325
326         int altoImagen = lienzo.getHeight();
327         int anchoImagen = lienzo.getWidth();
329         int [] escalado = null;
330         obi_puntos = new BufferedImage(anchoImagen, altoImagen, BufferedImage.TYPE_USHORT_GRAY);
331
332         if (obi_puntos == null) {
333             JOptionPane.showMessageDialog(lienzo, "No hay una imagen para dibujar");
334             return;
335         }
336
337         Graphics2D g2 = obi_puntos.createGraphics();
338         g2.setBackground(Color.white);
339         g2.clearRect(0, 0, anchoImagen, altoImagen);
340
341         for (int i = 0; i < ventana.lon_gps.size(); i++) {
342
343             escalado = cambioDominio(anchoImagen, altoImagen,
344                     ventana.lon_gps.get(i), extremo_lon[1], extremo_lon[0],
345                     ventana.lat_gps.get(i), extremo_lat[1], extremo_lat[0],
346                     ventana.z_terreno.get(i), extremo_z[1], extremo_z[0]);
347
348             obi_puntos.setRGB(escalado[0], escalado[1], escalado[2]);
349
350         }
351
352         Graphics g = lienzo.getGraphics();
353         g2 = (Graphics2D) g;
354         g2.drawImage(obi_puntos, null, 0, 0);
355
356     }
```

Figura 59 Método nuevaImagenPuntos("args")

Primero se hace un get de los valores de alto y ancho del lienzo que son necesarios para el cambio de dominio. Se genera la imagen *obi_puntos*, es una imagen en modo de color de escala de grises de la clase BufferedImage que se construye con el alto y el ancho del lienzo y el tipo de datos (o modo de color) USHORT_GRAY. La escala de grises la vamos a utilizar para tener una visión de “profundidad” en 2D. El mismo escalado que se hace con la *x* y con la *y* lo realizamos con la cota, asociándola a un valor entre 0 y $2^{16} - 1 = 65535$ que es el número máximo que se alcanza utilizando los 16 bits del TYPE_USHORT_GRAY (línea 330). El 0 se va a representar 100% con el gris más claro que representa este tipo, profundidad máxima de la nube de puntos, y por el contrario el valor máximo de cota representado por 65535 al 100% negro. Es necesario generar un objeto de la clase BufferedImage porque nos va a permitir atacar directamente a cada pixel mediante el método de esta clase setRGB().

Sin embargo necesitamos un objeto de la clase Graphics para sacarlo por pantalla en un JPanel, para ello se genera un gráfico *g2* que se asocia a la imagen *obi_puntos* (línea 337) y se limpia la imagen (se pone a blanco, líneas 338, 339).

Se realiza el escalado, es decir, tenemos los valores el valor de latitud y longitud de cada punto asociado a un valor de fila y columna de la imagen y la cota asociada a la escala de grises y se hace el setRGB que va pintando los puntos sobre la imagen (Figura 58 arriba izq.). Por último hay q asociar el gráfico generado al lienzo para sacar por pantalla, que es el jPanel1 (líneas 352, 353, 354). Este método se llama en el constructor de la clase y se almacenan los datos para que la clase Cpaint correspondiente pueda repintar cada vez que modificamos la ventana creada (aciones de mover, minimizar o maximizar ventana)

Es necesario implementar previamente el método parametrosJPanel(); que busca los máximos y mínimos de *x*, *y*, *z* de la nube de puntos que necesita el método cambiaDominio().

6.1.5.1.2 Triángulos

El método que representa los triángulos es nuevaImagenTriangulos(), para representar los triángulos se va a proceder de manera similar al método anterior, se va a recorrer la lista de triángulos que sale de la clase Delaunay con un bucle for y se va a ir pintando rectas (líneas 410-413) que unen los vértices (previo cambio de dominio) de cada triángulo.

```
410                 g2.setPaint(Color.black);
411                 g2.drawLine(escalado1[0],escalado1[1],escalado2[0],escalado2[1]);
412                 g2.drawLine(escalado1[0],escalado1[1],escalado3[0],escalado3[1]);
413                 g2.drawLine(escalado3[0],escalado3[1],escalado2[0],escalado2[1]);
414
415                 Triangle triangulo = new Triangle(escalado1[0],escalado1[1],
416                         escalado1[2],escalado2[0],escalado2[1],escalado2[2],
417                         escalado3[0],escalado3[1],escalado3[2]);
418
419                 triangulos.add(triangulo);
```

Figura 60 Pintado de triángulos

Se ha creado previamente el objeto g2 de la clase Graphics2D haciendo un createGraphics() de la BufferedImage obi_triang. El método drawLine dibuja una recta entre las coordenadas de los vértices (*x1, y1, x2, y2*). Se aprovecha para almacenar los triángulos escalados en un ArrayList de objetos Triangle. Este método se llama también en el constructor de la clase *Visualiza2D*.

6.1.5.1.3 MDT

El método nuevaImagenSuperficie() es el que representa este resultado y trabaja exactamente igual que el método que representa los puntos. En esta ocasión, y como se ha visto en el apartado 6.1.3.5 se tiene el valor de todos los pixeles que están contenidos por los triángulos de la red TIN y es necesario atacar al pixel con el método setRGB(); (líneas 474 a la 478 de figura 50) por lo que se hace uso del objeto de la clase BufferedImage *obi_sup.* Método también llamado en el constructor.

6.1.5.1.4. Curvas de nivel

El método nuevaImagenIsolineas(); se encarga de representar las isolineas que se han creado como General Paths y que se dibujan con el método draw del Graphics2D *g2* asociado a la BufferedImage obi_iso.

En este método se han tenido que añadir líneas de código que no tenemos en los anteriores para solventar el problema de condiciones de contorno que se menciona en el apartado 6.1.4.2. Hay que recorrer los General Pahts de salida de la clase *MarquingSquares* de modo que se cree un nuevo General Path que será el que finalmente se representará, que se apunte únicamente las líneas que caigan dentro de nuestro filtro. Es necesario conocer previamente como se han generado estos paths. Un path se crea mediante 3 operaciones básicas:

- moveTo: punto de partida de dibujo o nuevo punto al que dirigirse sin pintar mientras se dirige a él.
- lineTo: siguiente punto al que se dirige el puntero que dibuja, dibujando una linea recta entre el punto en el que se encontraba y al que apunta lineTo.
- closePath: termina el dibujado de ese del camino.

Un path se compone de muchas órdenes lineTo y moveTo que forman segmentos. Se hace un bucle for en el que consulta que tipo de segmento estamos visitando y en el que se va a crear nuestro nuevo path *isos_ok*. Se muestra como pseudocódigo el algoritmo de creación de nuestro path isos_ok que consideramos solución.

```
1. Entrada: Path de salida de clase MarquingSquares llamémoslas isolines
2. Para cada nivel de consulta se procede como sigue:

3. for 0 : nº segmentos del camino
4. Los segmentos de isolines se visitan dos veces, una por cada extremo

5.    if tipo extremo = moveTo && coordenadas dentro del filtro
6.    Se hace un moveTo a esas coordenadas isos_ok = moveTo (coordenadas)
7.    end

8.    if tipo extremo = moveTo && coordenadas fuera del filtro
9.    No apuntamos nada en nuestro path final isos_ok
10.   Una booleana (move) cambia para saber que el lineTo del siguiente
      extremo que se consulte y que esté dentro del filtro tiene que ser
      un moveTo en nuestro path isos_ok.
11.   end

12.   if tipo extremo = lineTo && coordenadas fuera del filtro
13.   No se hace nada
14.   end

15.   if tipo extremo = lineTo && coordenadas dentro del filtro
16.   Se hace un lineTo a esas coordenadas a no ser que la booleana move
      haya cambiado, por lo que este va a ser el primer punto que caiga
      dentro del filtro y hay que hacer un moveTo y volver a cambiar la
      booleana move.
17.   end

18. end
```

En un inicio se pensó en solicitar por pantalla el intervalo para la búsqueda de las curvas de nivel, pero por experiencia propia sabemos que habitualmente nos solicitan las curvas de nivel cada metro a cotas de valor entero, rara vez nos solicitan las curvas de nivel con un intervalo más pequeño. Se ha preferido dejar en el software original este intervalo de 1 metro y en el manual de usuario se explicará como configurar el método nuevaImagenIsolineas(); para que saque curvas de nivel con el intervalo deseado.

6.1.5.2 Representación 3D

La clase *Visualización3D*, pensada para ejecutar offline, hace uso de la API JME para mostrar una escena o espacio virtual sobre el que se representan los resultados en 3D.

La filosofía de trabajo de este motor 3D está basada en 3 conceptos:

- Mundo virtual: espacio contenedor de las geometrías y los nodos (fondo gris figura 61) con un sistema de coordenadas representado en la esquina superior de la figura 61.
- Nodo: ente invisible al que está unido (o “pegado”) una o más geometrías u otros nodos (cruces figura 61) siendo estos sus “hijos”. Que tiene la propiedad de transladarse, escalarse y rotar afectando a la geometría o nodo al que está unido, siendo el vector de unión el mismo (flechas figura 61 permanecen iguales).
- Geometría: ente visible que está unido a un nodo “padre” y que tiene la propiedad de transladarse, escalarse y rotar sin la necesidad de afectar al nodo “padre” en más aspectos que en el vector de unión, digamos que tiene estos grados de libertad por sí mismo y que el vector de unión con el nodo se modifica cuando se hace uso de estos grados de libertad (flechas figura 61 cambian en dirección y tamaño).

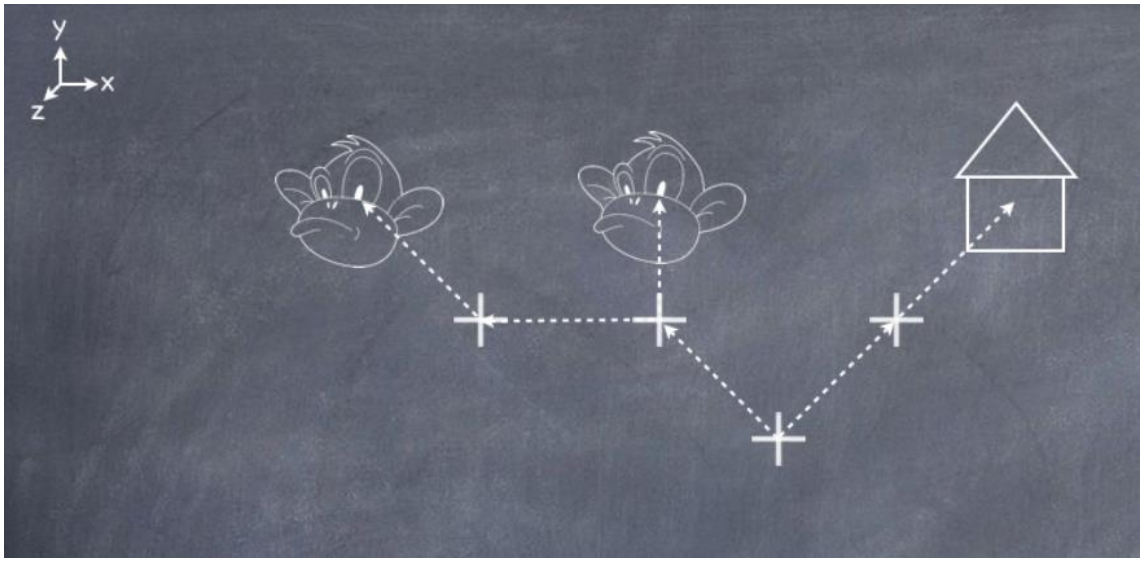


Figura 61 Esquema motor JME

Nuestra clase *Visualiza3D* extiende simpleApplication convirtiéndose en un mundo virtual. En el constructor se crea una lista de objetos Vector3f de nombre *lineVerticies* que va a posicionar los puntos en el mundo virtual y que almacena la nube de puntos. Hay que considerar que los ejes del sistema de coordenadas UTM y del sistema de coordenadas de la escena de jme no son coincidentes por lo que nuestras latitudes o coordenadas *y* se introducen como *z* negativas, mientras que las cotas se introducen como coordenadas y en la escena.

Finalmente dentro del constructor se llama al método plotPoints(“args”) que es el que se encarga de visualizar y vamos a ver como trabaja.

```
107     public void plotPoints(Vector3f[] lineVertices, ColorRGBA pointColor){
108         Mesh mesh = new Mesh();
109         mesh.setMode(Mesh.Mode.Points);
110         mesh.setBuffer(VertexBuffer.Type.Position, 3,
111                 BufferUtils.createFloatBuffer(lineVertices));
112         mesh.updateBound();
113         mesh.updateCounts();
114 
115 //          mesh.setMode(Mesh.Mode.Lines);
116 //          mesh.setBuffer(VertexBuffer.Type.Position, 3,
117 //                BufferUtils.createFloatBuffer(lineVerticies));
118 //          mesh.updateBound();
119 //          mesh.updateCounts();
120 
121 //          mesh.setMode(Mesh.Mode.Triangles);
122 //          mesh.setBuffer(VertexBuffer.Type.Position, 3,
123 //                BufferUtils.createFloatBuffer(lineVerticies));
124 //          mesh.updateBound();
125 //          mesh.updateCounts();
126 
127         n = new Node("puntos");
128 
129         Geometry geo = new Geometry("point",mesh);
130         Material mat = new Material(assetManager, "Common/MatDefs/Misc/Unshaded.j3md");
131         mat.setColor("Color", pointColor);
132         geo.setMaterial(mat);
133         geo.center();
134 
136         Vector3f cam_position = null;
137         cam_position = new Vector3f(-250f, 96.97773f, 200.50664f);
138         cam.setLocation(cam_position);
139 
140         n.attachChild(geo);
141 
142         inputManager.addMapping("x", new KeyTrigger(KeyInput.KEY_X));
143         inputManager.addMapping("y", new KeyTrigger(KeyInput.KEY_Y));
144         inputManager.addMapping("z", new KeyTrigger(KeyInput.KEY_Z));
145         inputManager.addMapping("c", new KeyTrigger(KeyInput.KEY_C));
146         inputManager.addMapping("t", new KeyTrigger(KeyInput.KEY_T));
147         inputManager.addMapping("q", new KeyTrigger(KeyInput.KEY_Q));
148         inputManager.addListener(analogListener, new String[]{"x","y","z","c","t","q"});
149 
150         rootNode.attachChild(n);
```

Figura 62 Método plotPoints

La clase mesh se utiliza para guardar datos de tres modos, puede almacenar puntos, líneas o triángulos, se utilizarán los tres modos de la clase para representar puntos y MDT con el primer modo, las curvas de nivel con el segundo modo y la triangulación con el modo triángulos.

Se crea un Buffer donde se indica que se le van a pasar datos del tipo Position, que requiere 3 números flotantes que representan la posición de un vértice (o punto en este

caso) y se le pasa la lista previamente creada Vector3f que contiene objetos que representan vértices con tres números flotantes (línea110-111).

Para los datos del modo *mesh.mode.Lines* se pasa una lista similar teniendo en cuenta que va a crear una línea cada dos vértices que lea, por lo que todos los datos excepto el primero y el último tienen que estar por duplicado. Para los del modo *mesh.mode.Triangles* se pasa una lista de triángulos donde cada 3 líneas encontramos los 3 vértices correspondientes a cada triángulo y se crea un triángulo cada 3 vértices que lee.

Seguidamente se crea un nodo (línea 127) en las coordenadas (0,0,0) por defecto y se convierte la mesh generada a un objeto de la clase geometría, es importante remarcar que en la línea 133 trasladamos la geometría al origen del mundo virtual de modo que al operar sobre el nodo, como veremos a continuación, no perdemos de vista la geometría. Es más sencillo operar de esta forma que buscar dónde está situada nuestra geometría dentro del mundo y colocar un nodo cercano sobre el que operar.

Seguidamente colocamos la cámara en una posición en la que se vea adecuadamente toda la geometría, de manera experimental se ha llegado a la conclusión de que tiene que estar alejada de la coordenada (0, 0, 0) aproximadamente un 50% del valor del tamaño del dominio UTM para latitudes y longitudes y un 2000% del valor del tamaño del dominio de las cotas.

Asociamos la geometría al nodo (línea 139) y hacemos un rootNode del nodo que quiere decir que ponemos el nodo en la escena, obviamente al tener asociada la geometría esta aparece en escena.

En las líneas 141-144 se manejan los eventos que se introducen por teclado para hacer rotar el nodo “padre”. Se genera un objeto de la clase AnalogListener que se mantiene a la escucha, cuando las teclas x, y o z son pulsadas rota el nodo alrededor del eje que se pulse, se mantiene el criterio de ejes de la escena. Las teclas c, t y q se utilizan para girar en el sentido contrario.

```
153     public AnalogListener analogListener = new AnalogListener() {
154
155         public void onAnalog(String name, float value, float tpf) {
156
157             if (name.equals("x")) {
158
159                 n.rotate( tpf, 0, 0);
160
161             }
162
163             if (name.equals("c")) {
164
165                 n.rotate( -tpf, 0, 0);
166
167             }
168
169
170             if (name.equals("y")) {
171
172                 n.rotate(0, tpf, 0);
173
174             }
175
176             if (name.equals("t")) {
177
178                 n.rotate(0, -tpf, 0);
179
180             }
181
182             if (name.equals("z")) {
183
184                 n.rotate(0, 0, tpf);
185
186             }
187             if (name.equals("q")) {
188
189                 n.rotate(0, 0, -tpf);
190
191             }
192
193         }
194     };
```

Figura 63 Objeto que escucha del teclado y actúa sobre el nodo

Finalmente mostramos un ejemplo de la apariencia de la escena con una mesh de puntos:

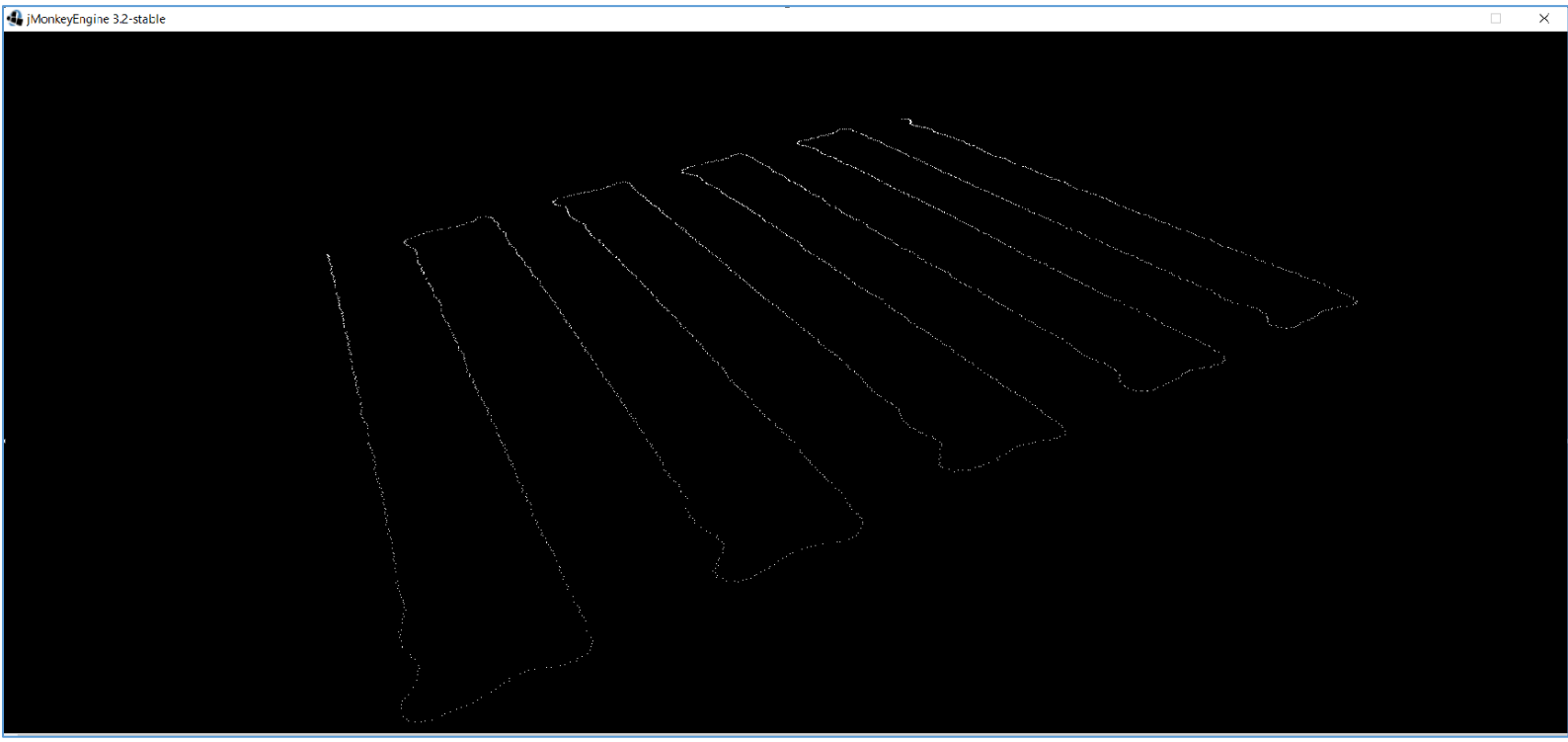


Figura 64 JME scene

6.1.5.3 Clase Cpaint

Estas clases nos van a permitir mover, minimizar o maximizar la ventana sin que perdamos la información gráfica que contienen los paneles de la ventana. Cambia el paradigma del graficado, en lugar de crear un gráfico y llamar al panel donde se pinta, con lo que perderíamos información cada vez que refrescamos la pantalla porque ya no se está creando el gráfico, lo que se hace es almacenar un gráfico (en este caso las imágenes de tipo BufferedImage que hemos ido llenando con los distintos métodos de la clase *Visualiza2D*) en memoria que será llamado por el objeto de la clase Cpaint correspondiente cada vez que refresquemos la ventana.

Para poder seguir este proceso es necesario customizar el JPanel correspondiente a cada uno de los paneles y cambiar el código para que cuando se inicialice el initcomponents() de la clase *Visualiza2D* no se cree un JPanel normal, si no un objeto de la clase Cpaint, que no deja de ser un JPanel customizado.

Para ello en la pestaña de diseño de la clase *Visualiza2D* pulsamos botón derecho sobre el JPanel y elegimos la opción Customize code. Dentro de esta herramienta creamos un new Cpaint (en el caso de la figura es el correspondiente a los puntos) al que le pasamos

la imagen que tenemos almacenada en disco que ha graficado los puntos en este caso. La manera de proceder es idéntica para triángulos, MDT y curvas.

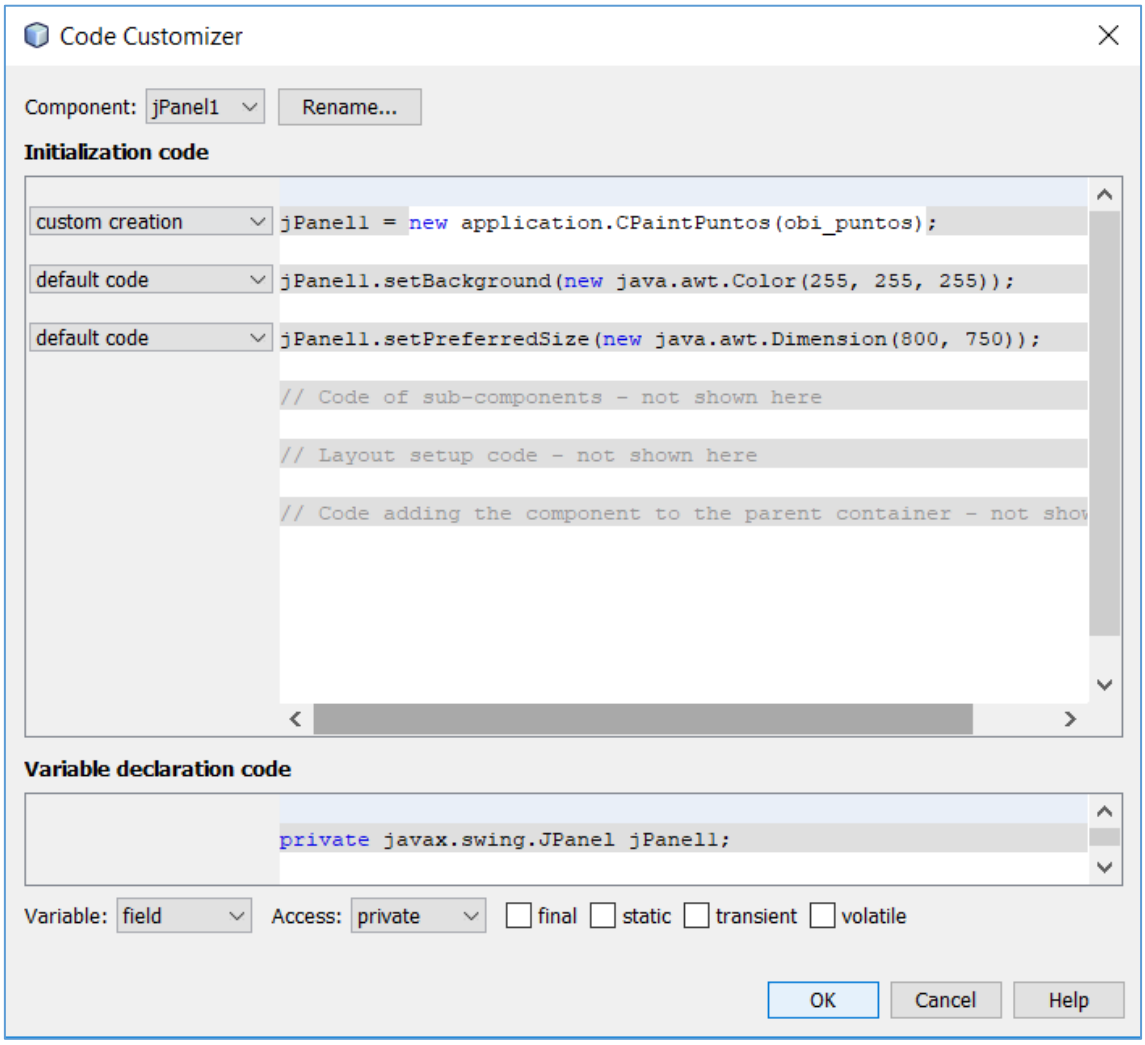


Figura 65 Customize code

Como peculiaridad de estas clases CPaint cabe mencionar que es necesario cambiar el valor de una variable booleana para que de tiempo a inicializarse a la ventana antes de que se haga el pintado de las diferentes imágenes. Por lo que al pulsar sobre el botón Dibuja Puntos (Figura 58), se lanza el siguiente evento que cambia la booleana y posteriormente repinta la imagen obi_puntos. Cada botón tiene asociada una booleana que controla que se haga un pintado sucesivo, es necesario pintar por este orden: puntos, triángulos, MDT y finalmente curvado.

```
270     private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
271
272         try {
273
274             ((CPaintPuntos) jPanel1).setPintar(true);
275             ((CPaintPuntos) jPanel1).repaint();
276
277             control_triang = true;
278         } catch (Exception ex) {
279             Logger.getLogger(Visualiza2D.class.getName()).log(Level.SEVERE, null, ex);
280         }
281
282     }
```

Figura 66 Evento pulsar botón Dibuja Puntos

6.1.5.4 Exportación de curvas de nivel

Para exportar las curvas de nivel se hace uso de una librería desarrollada por el MIT (jdxf library) que escribe la cabecera del archivo que vamos a usar en CAD y que no es objeto de este trabajo llegar a tal nivel de detalle acerca del formato de un archivo. El formato dxf se creó como un archivo para dibujos de diseño asistido por ordenador.

En el apartado 6.1.5.1.4 se ha explicado como obteníamos las curvas de nivel a las que hemos llamado *isos_ok*, estas curvas se representan en el JPanel con coordenadas pixel, por lo que para obtener las medidas reales hacemos uso una vez más del cambio de dominio y generamos las curvas de nivel *isos_cad*, se tiene en cuenta que las coordenadas *pixel y* son negativas, origen en esquina superior izquierda, por lo que es necesario cambiar el signo de estas coordenadas pasadas a utm en *isos_cad.* Finalmente las líneas de código que generan el archivo dxf son las siguientes:

```
905             DXFDocument dxfDocument = new DXFDocument("Example");
906             DXFGraphics dxfGraphics = dxfDocument.getGraphics();
907
908             for (int i = 0; i < levels.length; i++) {
909     //              g2.draw(isos_ok[i]);
910                 dxfGraphics.draw(isos_cad[i]);
911             }
912
913             String stringOutput = dxfDocument.toDXFString();
914             String filePath = ("src/resources/curvas2.dxf");
                FileWriter fileWriter = new FileWriter(filePath);
916             fileWriter.write(stringOutput);
917             fileWriter.flush();
918             fileWriter.close();
```

Figura 67 Escritura archivo dxf

El método toDXFString() es el que añade la cabecera que hace que se combierta en este tipo de archivo, el resto es un escritor de la clase FileWriter que ya hemos utilizado en el proyecto para escribir el archivo de datos que se obtiene de EchoSonda y GPS.

6.1.5.5 Optimización de cálculos

Se pretende optimizar el tiempo de cálculo de los procesos más pesados, que son el cálculo del MDT en el que hay que recorrer todos los píxeles y consultar dentro de que triángulo está y el cálculo de la erosión en el que hay que recorrer también todos los píxeles de la imagen consultando si los píxeles pertenecen a la imagen y además si son cercanos al borde.

El primero de los procesos se ha optimizado en la medida de los algoritmos que hemos sido capaces de imaginar e implementar. Pasamos por los siguientes estadios de cálculo:

(1) Hacer un estudio de todos los pixeles por fuerza bruta consultando por cada píxel la toda la lista de triángulos para ver en cuál estaba contenido y calcular una vez allí el plano y después interpolar el valor de z conocido el plano y los tres vértices del triángulo contenedor. En el cálculo de los resultados que se presentan en el apartado 6.3.1 en este estadio se toma 60 segundos en realizar el cálculo del MDT.
(2) Se genera la figura de triangle_old, este triángulo se apunta cada vez que se hace una interpolación y es el primero que se consulta de la lista de triángulos en cada iteración de pixel ya que los triángulos son grandes en comparación con el salto que se da de pixel a pixel, por lo que es probable que el siguiente pixel consultado esté dentro del triángulo anterior. En este estadio el MDT tardaba 15 segundos menos.
(3) Se genera un HashMap que guarda los planos calculados y que evita hacer el cálculo del plano cada vez que hay que hacer una interpolación. Finalmente este estadio mejoraba el proceso 3 segundos.

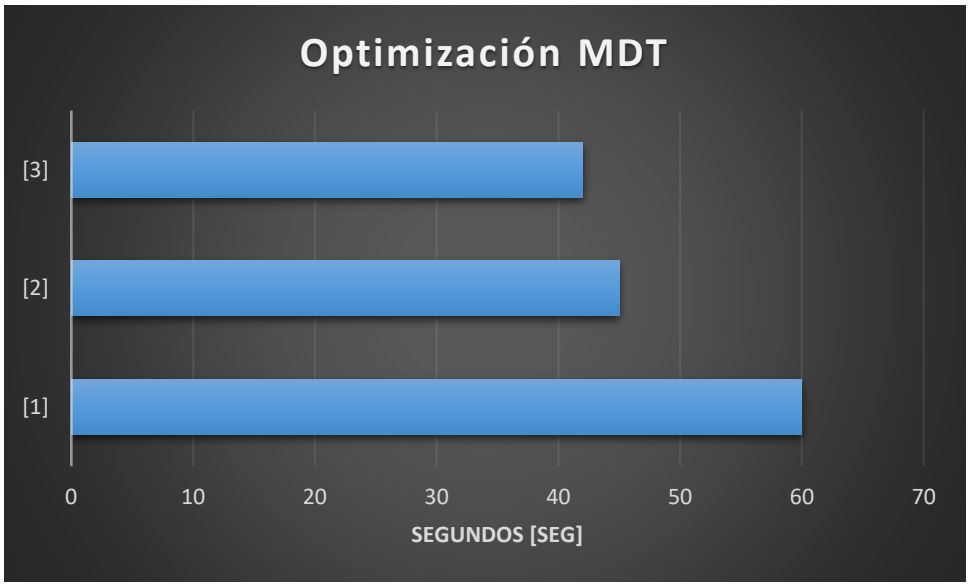


Figura 68 Tiempos de ejecución MDT

El segundo proceso se deja para un trabajo futuro, no hemos llegado a tiempo de implementar un algoritmo para optimizar el cálculo de la erosión. Si que se ha ideado un plan de acción que se podría aplicar también al proceso anterior.

La idea sería dividir la imagen en tantos trozos como procesos simultáneos sea capaz de soportar el ordenador (multithreading), para el proceso anterior no habría mayor problema que realizar estas divisiones. En el caso de la erosión habría que coger incrementos de los trozos de manera que al recortar y posteriormente volver a unir no se quedasen huecos que hayan sido recortados, habría que coger incrementos de manera que al hacer las erosiones de cada trozo por separado no se pierda información de los trozos originales (sin contar el incremento). Con un ordenador con un procesador i7 por ejemplo se podría llegar a dividir hasta por 32 (cantidad de threads simultáneos capaz de ejecutar el procesador) el tiempo de ejecución, actualmente el programa tarda alrededor de 60 segundos en hacer cada erosión.

6.2 Fase pruebas

Habiendo visto como trabaja el sistema, vamos a ser capaces de entender y de justificar una serie de tests que se han realizado a nivel de usuario final con el objetivo de comprobar que las herramientas y equipos utilizados funcionan acorde con sus

especificaciones, y de depurar el código y de confirmar que todas las aplicaciones teóricas están bien implementadas y que todo funciona según lo previsto.

6.2.1 Toma de contacto con el hardware

Antes de comenzar el proyecto propiamente dicho se buscó, por activa y por pasiva, la manera de hacer que el programa que incorporaba la EchoSonda fuera capaz de leer los datos del GPS, junto con el director del proyecto se buscó una posible solución o encontrar el problema que presentaba el software de la EchoSonda, se barajaron varias posibiliddes por las que pudiera fallar entre ellas que no estuviera adaptado a las nuevas versiones de sistema operativo, podrían tener algún modo de trabajo anticuado, se probó a ejecutarlo en Windows 7 por si la solución fuese simplemente hacer un downgrading del sistema operativo, pero no resultó satisfactoria con windows 7. La empresa koreana que vendió esta EchoSonda (EchoLogger) no ofreció más soluciones que una actualización de firmware que no resolvió el problema.

Se pone, entonces, en marcha el proyecto. Nuestro punto de partida es poder leer las cadenas entrantes del protocolo NMEA 0183 a través de NeatBeans, hecho que se consigue gracias a la clase SerialPort. Esta clase, a la que se ha tenido que realizar una modificación para hacerla funcionar es parte de una API marina que según su descripción en GitHub “contiene librerías para codificar y decodificar datos proporcionados por diversos dispositivos electrónicos marinos como GPS, EchoSondas e intrumentos meteorológicos”. Esta modificación consistió en cambiar el timeout de espera de escucha de los puertos serie, este era muy pequeño y no le daba tiempo a leer del puerto que consultaba, tuvimos que elevar este timeout en la clase *SerialPortExample*. También es posible que los programadores del software tuviesen este mismo problema y no son capaces de leer el puerto del GPS.

Una vez alcanzado este logro, los primeros pasos que se realizaron estuvieron enfocados a familiarizarse con los dispositivos de recogida de datos (GPS y EchoSonda).

Figura 69 Pruebas en terraza

Para trabajar con el GPS es necesario hacer una revisión previa de la configuración del mismo a través de su app. También es necesario conectarse a la red ERVA para recibir correcciones que den precisión centimétrica a las medidas tomadas.

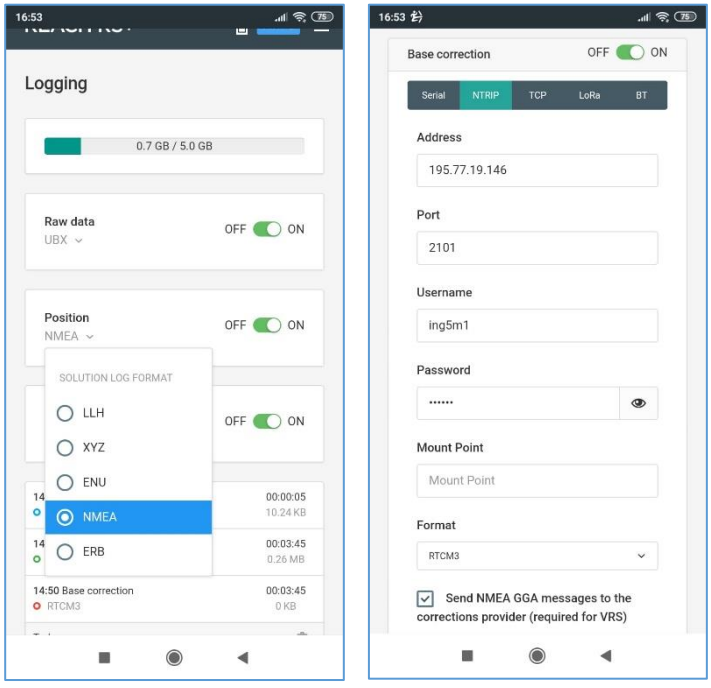


Figura 70 Elección de formato (izq.) Conexión a red ERVA (dcha.)

El formato LLH es el formato propio de la aplicación que se eligió para la prueba de piscina, mientras que el NMEA es el que se ha utilizado de ahí en adelante. Esto se configura desde la opción Logging del menú que hay arriba a la derecha de la aplicación.

Por otro lado para conectarse a la red ERVA es necesario disponer de un usuario y una contraseña que ha de solicitarse al ICV (Instituto Cartográfico Valenciano).

La EchoSonda por su parte tiene el método de configuración que veremos en el apartado 6.2.3.1. De cara al alcance de esta tesina nos interesa poder cambiar la frecuencia de emisión de la señal (actualmente 1 Hz, tomamos valores cada segundo).

6.2.2 Prueba en piscina

Paralelamente a la familiarización con los equipos y con los primeros pasos de desarrollo del Software, se hizo una prueba en piscina. Se eligió la piscina municipal de Els Poblets como localización de la primera prueba en la que se montaba todo el sistema, simulando una aplicación real. Navegando eso sí de forma manual, a diferencia de la aplicación real en la que le enviamos la ruta a través de Mission Planner.

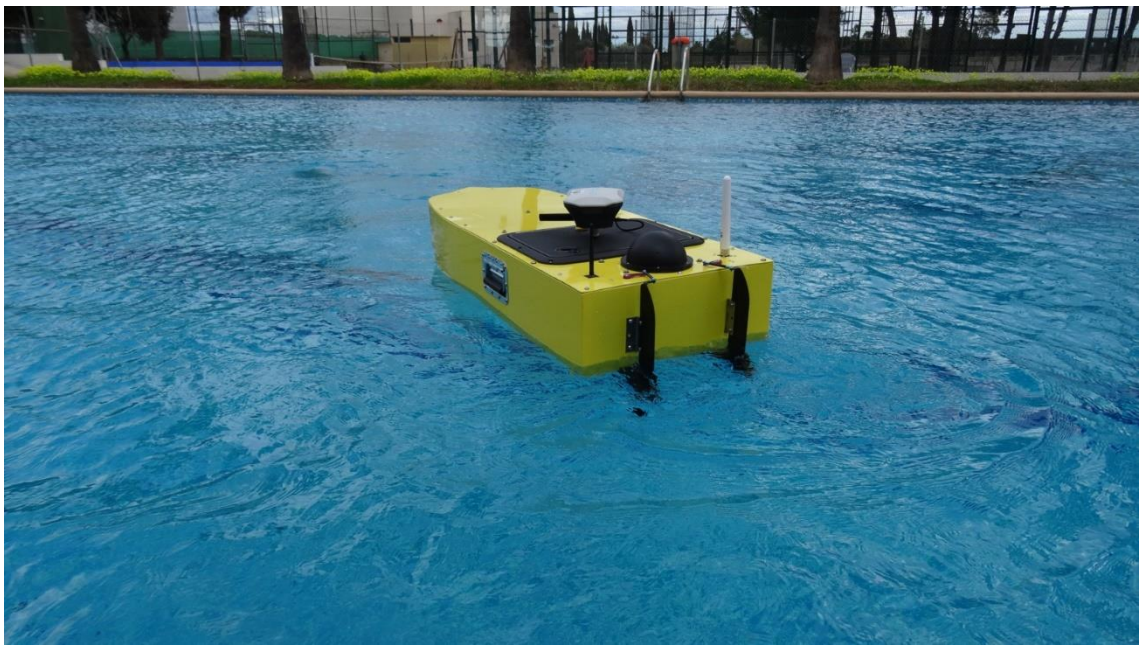


Figura 71 Sistema batimétrico navegando en piscina

Nos dimos cuenta unos días antes de la prueba en la piscina, que se recibían coordenadas de la constelación GLONASS y no de GPS (apartado 6.1.2.1) y se decidió probar con con un formato propio del sistema EMLID que proporcionaba longitud, latitud (LLH) que dan medidas en grados centígrados y altura geoidal.

Figura 72 Preparativos para la prueba

En las pruebas de oficina no tuvimos errores, pero en la prueba en la piscina hubo un error en la toma de medidas. Por lo que no hay resultados que mostrar de este día porque con ese tipo de coordenadas, en ese momento, nos dimos cuenta de que salen, de vez en cuando, cadenas donde hay más espacios en blanco de los que deberían, por lo que al hacer el parseado de la cadena de entrada, cuándo se debe leer un *double,* entra un espacio en blanco que nos arroja un error de tipo Null Pointer, el programa falla y se para y no llega a generar el fichero.

Al comprobar este error y decidir pasar otra vez a coordenadas geográficas empezó a llover y decidimos finalizar la prueba sin realizar medidas ya que la finalidad principal de esta prueba era comprobar la flotabilidad del USV. El programa todavía necesitaba aspectos que mejorar y esta prueba de flotabilidad coincidió muy prematuramente en el tiempo con el desarrollo del software (ya estaba programada antes de recibir la EchoSonda y comprobar que no funcionaba su software).

Adicionalmente, se añaden las conclusiones de la prueba:

- Se encontró agua en el interior del USV, por lo que no era completamente estanco, pudimos detectar la zona por la que entraba agua y se colocó silicona marina para completar el sellado.
- En los días posteriores corroboramos que aunque se midiera con la constelación GLONASS las mediciones eran válidas al estar modificadas por las correcciones diferenciales de la red ERVA.

6.2.3 Primera prueba de Software

Para las primeras pruebas de software, una vez depurados los fallos que evitaron que pudieramos obtener datos de la piscina, se ha simulado una navegación con muchas pasadas paralelas. Se montó parte del sistema (ordenador + GPS + EchoSonda) y fuimos moviéndonos en coche en el recinto ferial Los Huertos de Orihuela.

Figura 73 Parte del sistema montado en coche

La EchoSonda está programada para calcular el time of fligth de la señal y a partir de ahí arrojar una medida de profundidad, como su objetivo es medir sobre agua tiene una velocidad del sonido configurada de 1500 m/s que es la velocidad a la que se desplaza el sonido en el agua. En el aire esta velocidad es de 343.2 m/2. En un principio se supuso

que se obtendrían medidas erróneas por esta diferencia, pero la sonda arrojaba siempre el valor mínimo que puede medir, que es 0.2.

Este hecho junto con la frecuencia tan alta de emisión de un haz de sonido (se necesitan medir más puntos), nos llevó a plantearnos que era necesario acceder a la configuración de la EchoSonda. Por lo demás el software funcionaba tal y como se esperaba de él. Se hace un print de las cadenas para verlas por pantalla mientras se sigue la ruta y se está haciendo el display de las coordenadas en la ventana creada.

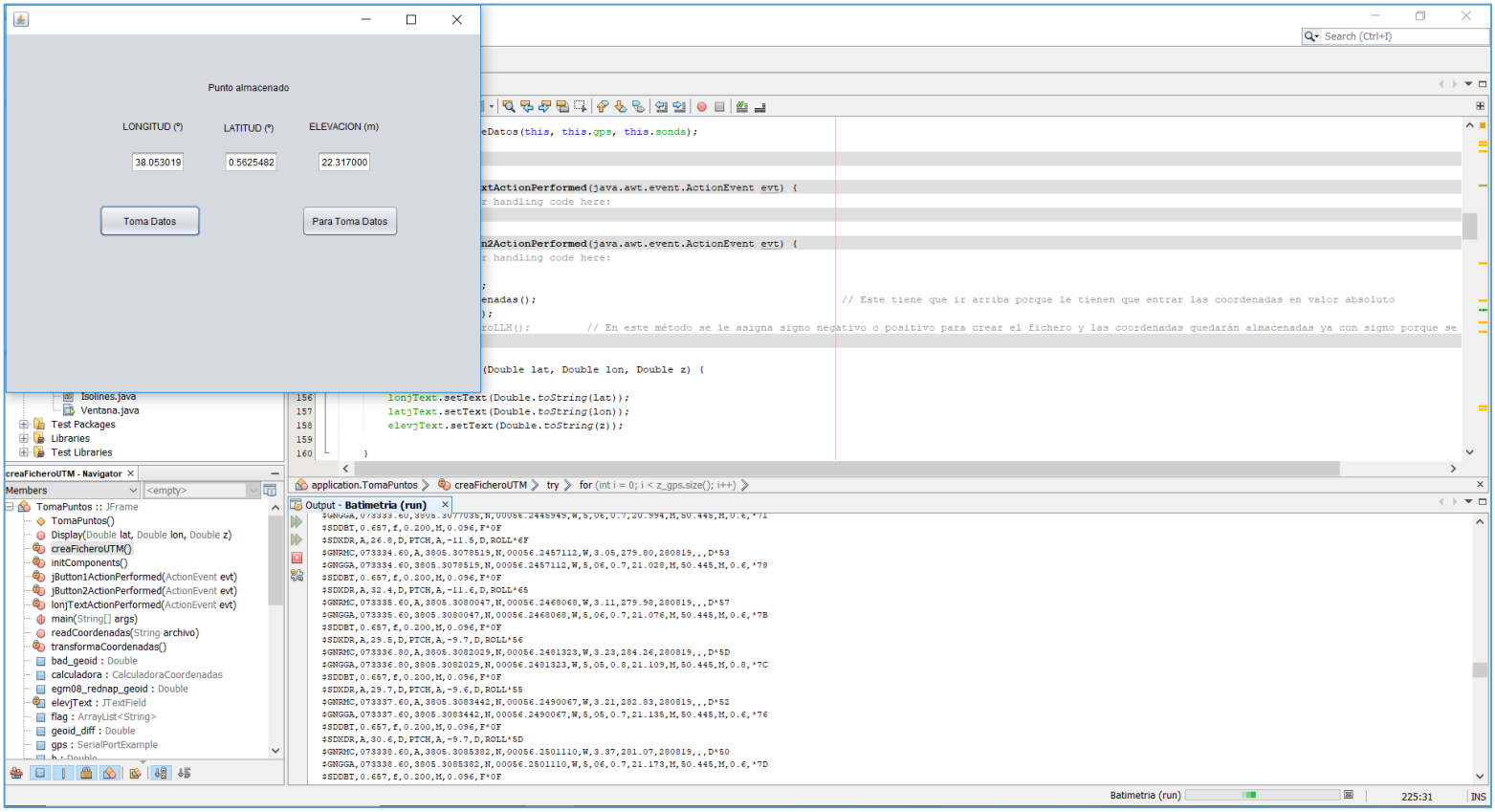


Figura 74 Pantallazo de NeatBeans ejecutándose

Para presentar los datos de cara a la memoria se importan a Excell desde la pestaña datos con la herramienta *Desde texto.* En un principio y para continuar testeando el programa antes de realizar la configuración de la EchoSonda se pusieron valores inventados, pero coherentes, para comprobar las otras herramientas del software (Triangulación, MDT, curvado, Visualización…)

ID	Longitud_UTM [metros]	Latitud UTM [metros]	Cota terreno [metros]	Flag	Hora GPS	z_GPS	Profundidad_EchoSonda
1	680928.8115	4217632.505	22.087	5	73247	21.271	0.2
2	680928.8918	4217632.5	21.846	5	73249.6	21.03	0.2
3	680928.9705	4217632.481	21.676	5	73252.4	20.86	0.2
4	680929.0437	4217632.424	21.537	5	73255	20.721	0.2
5	680929.0679	4217632.392	21.464	5	73257.6	20.648	0.2
6	680929.1717	4217632.361	21.349	5	73300.4	20.533	0.2
7	680928.9011	4217632.37	21.239	5	73302.8	20.423	0.2
8	680928.3807	4217632.411	21.164	5	73303.6	20.348	0.2
9	680927.4658	4217632.57	21.094	5	73304.6	20.278	0.2
10	680926.2467	4217632.719	21.116	5	73305.6	20.3	0.2
11	680924.8408	4217632.856	21.172	5	73306.6	20.356	0.2
12	680923.0462	4217633.117	21.211	5	73307.8	20.395	0.2
13	680921.8415	4217633.289	21.227	5	73308.6	20.411	0.2
14	680920.3006	4217633.541	21.228	5	73309.6	20.412	0.2
15	680918.7499	4217633.777	21.251	5	73310.6	20.435	0.2
16	680917.1645	4217634.026	21.298	5	73311.6	20.482	0.2
17	680915.2272	4217634.3	21.336	5	73312.8	20.52	0.2
18	680913.986	4217634.508	21.282	5	73313.6	20.466	0.2
19	680912.3752	4217634.768	21.27	5	73314.6	20.454	0.2
20	680910.7703	4217635.009	21.285	5	73315.6	20.469	0.2

Figura 75 Primeros 20 datos recogidos

También para la memoria y para que resulte “más visual” de cara al lector se introducen los datos a un programa de información geográfica donde podemos ver la ruta seguida sobre una ortofoto.

Figura 76 Ruta primera prueba

Para poder cargar los datos en programa de información geográfica fue necesario concatenar en Excell los valores de *x*, *y*, *z* separados por una coma y en el programa se hace uso de la función Create Feature Class / From XY Table.

La principal conclusión que se sacó de esta prueba fue que era necesario acceder a la configuración de la EchoSonda para modificar los valores que se han mencionado arriba. Se verá en el apartado siguiente.

Los resultados de esta prueba fueron los siguientes:

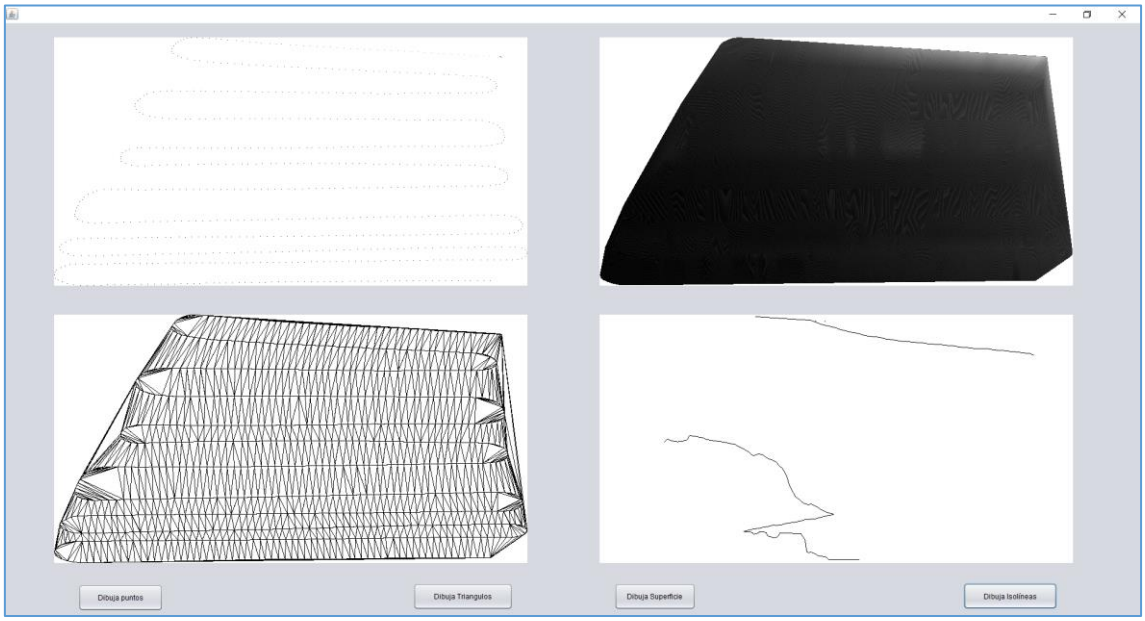


Figura 77 Resultados Orihuela

Cabe destacar que la superficie que no tenía un desnivel excesivo, como se aprecia en las curvas, (la de arriba representa 22 metros sobre el elipsoide y la de abajo 23 metros), tampoco se pudieron tomar medidas reales ya que la EchoSonda está diseñada para medir en agua y la velocidad del sonido con la que calcula está fijada en 1500 m/s (muy alejada de los 343.2 m/s de propagación en el aire). Como veremos en el siguiente apartado no es posible configurar la EchoSonda para medir en aire.

6.2.3.1 Configuración EchoSonda

Como ya se ha comentado, ha sido necesario manipular la configuración estándar de la EchoSonda para adaptarla a nuestros requerimientos, se intentó hacer desde el

software propio de la EchoSonda, pero tampoco funcionaba en este sentido, por lo que se tuvo que ir al manual de usuario de la EchoSonda para consultar los nombres de ciertos parámetros para poder acceder a ellos desde un programa externo.

Output settings

Command	Description	Values	Items
#samplfreq	Set output sampling frequency	6250~100000	Hz
#medianflt	Set median filter value	3~21	N/A
#moveavgflt	Set SMA filter	2~12	N/A
#nmearate	Set NMEA output rate	0~1	seconds
#nmeadpzero	Set NMEA DPT/DBT zero value output	0/1	N/A
#nmeadptoff	Set NMEA DPT offset	±50	meters

Figura 78 Ejemplo de parámetros

En el manual se especifican todos los parámetros de configuración, se comprueba que la velocidad de propagación del sonido no se puede fijar por debajo de los 1000 m/s, por lo que no podremos hacer pruebas con medidas fieles al terreno en tierra. Queda pues, cambiar la frecuencia de emisión de datos NMEA. Podemos ver que el parámetro que regula esta frecuencia es #nmearate. Para cambiar la configuración se procede de manera similar a como leemos los paquetes de la sonda. Se genear un escritor pw (PrintWriter), en lugar de un lector (BufferedReader) y se escribe mediante el comando que podemos ver en la línea 76 de la siguiente figura.

```
68              sp_sonda = sonda.getSerialPort();
69
70              try {
71                  pw = new java.io.PrintWriter(sp_sonda.getOutputStream());
72              } catch (IOException ex) {
73                  Logger.getLogger(LeeDatos.class.getName()).log(Level.SEVERE, null, ex);
74              }
75
76              pw.println("#nmearate = 0.1"); //escribir aquí las sentencias
77              pw.close();
```

Figura 79 Configuración parámetro #nmearate

6.3 Fase implantación

Llegados al punto en el que el Software realizaba su función sin fallos y en el que además ya habíamos asegurado la flotabilidad del USV no quedaba más que poner el sistema en funcionamiento en una prueba de campo real. Para ello se eligió el lateral sur del puerto de Sagunto zona de la que la empresa contratante del proyecto tenía

datos batimétricos con los que íbamos a ser capaces de realizar una comparación para medir la fiabilidad del sistema.

6.3.1 Navegación en Sagunto

Así pues, llevamos todo lo necesario a campo para hacer una prueba de aplicación real, se detalla a continuación el listado de equipos:

- USV
- GPS
- EchoSonda
- Portátil de tierra
- Portátil de a bordo
- Equipo de radio para comunicar portátil con barco (plan de navegación)
- Equipo de radiocontrol Frisky Taranis X9D
- 2 baterías de 16000 mAh
- Móvil de a bordo para dar internet a GPS y portátil
- Generador eléctrico: para baterías de portátil
- Herramientas de montaje
- Prismáticos para no perder de vista el barco
- Barca imchable: Por si había alguna emergencia

Eramos 3 personas, más una que se encargaba de documentar fotográficamente, se tuvo el sistema listo para navegar en aproximadamente media hora.

La preparación incluye:

- Inflado del barco inchable de emergencia
- Colocación de EchoSonda con gomas aislantes + tornillería
- Colocación de baterías dentro del casco
- Abrir NeatBeans y Teamviewer en ordenador de a bordo y colocarlo dentro del casco
- Conectar el GPS con el móvil vía WiFi para recibir correcciones diferenciales e introducir el movil en el casco dentro de una bolsa aislante y colocar el GPS como se ve en figura 80 abajo izq.
- Hacer todas las conexiones del sistema:
    - Ordenador conectado a baterías

- USV conexión con baterías
- EchoSonda conexión con portátil
- GPS conexión con portátil

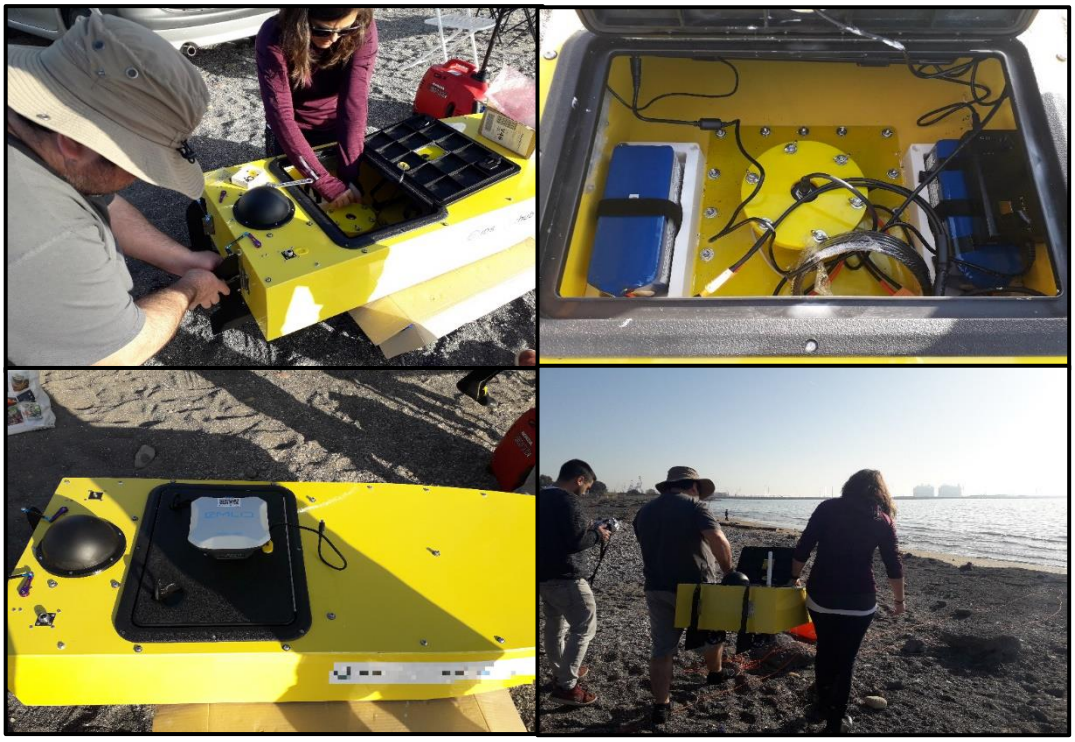


Figura 80 Preparativos

La duración aproximada de las baterías es de 4 horas, teniendo esto en cuenta y previendo posibles fallos se programó un plan de navegación de 40 minutos, por si fuera necesario reiniciarlo en alguna ocasión.

Con nuestra experiencia con el uso de drones, tenemos bien sabido que es conveniente que el plan de navegación o vuelo empiece lo más alejado posible y termine lo más cercano posible al “home point” o punto de partida y regreso para que si hay problemas de batería y el drone debe finalizar el plan antes de lo previsto, pueda ser visto donde “cae”, o en el caso de la navegación para que resulte más fácil llegar hasta el casco. Aunque en este caso nos cercioramos de no perder de vista nunca el USV. Se programó una ruta muy cercana a la tierra.

Figura 81 Navegando

La ruta de navegación se establece en el programa mission planner teniendo en cuenta lo dicho previamente y los requierimientos mínimos del estándar internacional de batimetrías (apartado 6.1.2.3).

En el menú de mission planner vamos al aparatado Flight Plan, buscamos sobre el mapa la zona de navegación, hacemos click derecho en uno de los vértices de nuestro plan de navegación (tenemos datos batimétricos de una gran extensión de terreno, por lo que no hay que hacer un estudio previo de donde se quiere tomar datos exactamente, simplemente elegimos una zona al azar), y generamos un polígono con la herramienta dibujar polígono. Una vez dibujado volvemos a pulsar botón derecho y elegimos la opción Auto Wp → Survey (Grid). En as pestañas Simple y Grid Options podemos fijar la distancia entre pasadas y el ángulo que forma la malla siendo el norte 0º.

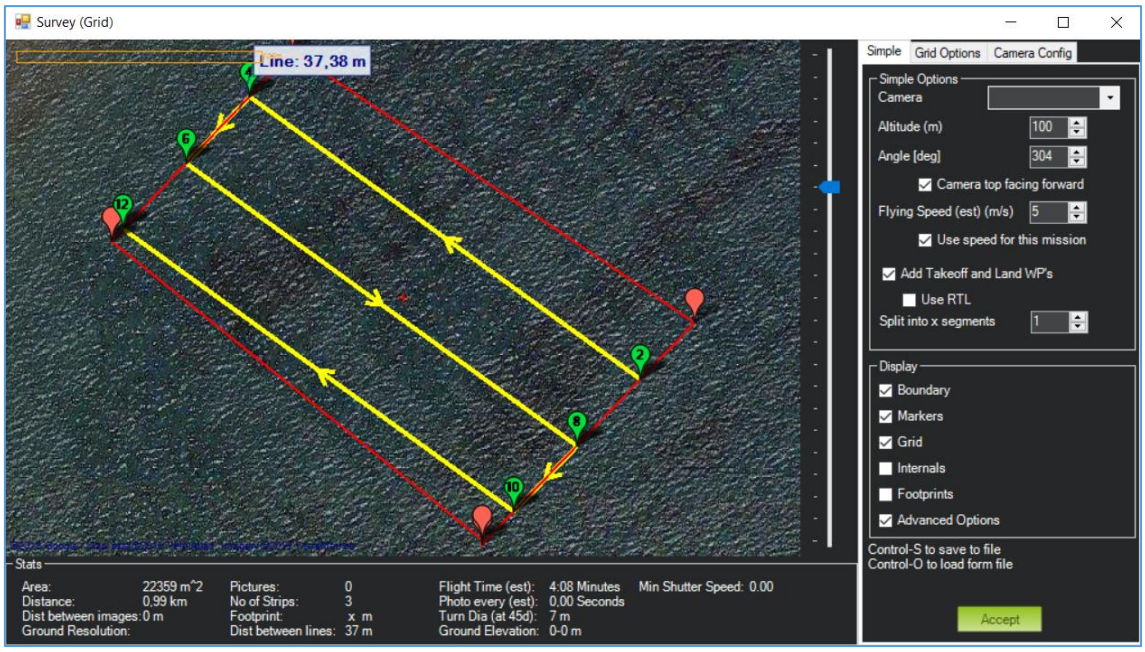


Figura 82 Diseño de ruta de navegación

Posteriormente se guarda la ruta, se conecta con el barco (botón de enchufe blanco y rojo, arriba a la derecha en el programa) y se le carga el plan, todo esto sin salir del submenú Fligth Plan (no se añade imagen de esto, es muy intuitivo y se ven perfectamente los botones conectar, guardar y cargar). En el momento en que está cargado el plan podemos activarlo poniendo una pestaña de la estación de tierra en posición de navegación automática.

Figura 83 Ruta de navegación programada

A partir de este momento podríamos ejecutar a través de Teamviewer el programa de toma de puntos, para tener datos extra mientras se navega hasta el punto de inicio del plan, o podríamos esperar a llegar a este punto para ejecutar el programa.

En este caso se cogieron puntos desde que el barco entró al agua, aunque para la visualización en la memoria y de cara a presentar resultados se han seleccionado únicamente los puntos pertenecientes al plan de navegación para que se aprecien bien las pasadas. Vemos una imagen del programa información geográfica igual que se ha hecho en la Figura 76.

Figura 84 Puntos obtenidos representados sobre ortofoto

Antes de pasar a comentar los resultados obtenidos se va a mencionar una serie de conclusiones que obtuvimos el día de navegación en mar abierto y que mejorarían la toma de datos:

- La principal modificación que sería necesaria en el sistema para que la toma de datos fuera óptima sería incorporar un gimbal a la EchoSonda porque en días de oleaje moderado (con oleaje fuerte no saldríamos a medir) la toma de medidas con este sistema se vería comprometida al estar rebotando el haz en un punto de coordenadas *x* e *y* demasiado alejado del punto que está midiendo el GPS.
- El plan de navegación se realizó paralelo a la línea de costa, pero sería más efectivo y seguro hacerlo de forma perpendicular, tomando mayor muestreo de datos de la pendiente y evitando el oleaje lateral que podría provocar un vuelco.

- El tiempo que tomó esta prueba fueron 40 minutos cubriendo 10 hectáreas, se realizó con la configuración estándard de la EchoSonda, teniendo en cuenta que las baterías duran 4 horas y que el factor limitante es la velocidad que puede alcanzar el USV, que es de 5 m/s si tomamos medidas cada 0.2 segundos para que se cumpla el teorema de *nysquit* (se toma una medida cada metro), según los cálculos realizados en el apartado 6.1.2.3 las hectáreas que se pueden cubrir con una ruta de navegación optimizada son 275 hectáreas. No se tiene en cuenta la vida de las baterías a una velocidad diferente, se calcula como si durásen 4 horas, sería necesario hacer otra prueba de esto.

A continuación vamos a comentar los resultados obtenidos de esta implantación del sistema en la prueba de aplicación real en la ladera sur del puerto de Sagunto. Vemos primero una muestra de la toma de datos efectuada en la siguiente figura.

ID	Longitud_UTM [metros]	Latitud_UTM [metros]	Cota terreno [metros]	Flag	Hora GPS	z_GPS	Profundida EchoSonda
1	737269.5936	4390810.899	-5.022	5	93319	-0.775	4.271
2	737268.7769	4390810.459	-4.901	5	93320	-0.735	4.181
3	737268.0483	4390810.076	-5.078	5	93321	-0.738	4.413
4	737267.3786	4390809.434	-5.001	5	93322	-0.742	4.321
5	737266.8555	4390808.775	-4.982	5	93323	-0.626	4.327
6	737266.4146	4390807.995	-4.976	5	93324	-0.674	4.228
7	737266.0736	4390807.227	-4.989	5	93325	-0.725	4.268
8	737265.7613	4390806.444	-4.877	5	93326	-0.736	4.346
9	737265.463	4390805.641	-4.922	5	93327	-0.732	4.416
10	737265.1097	4390804.808	-5.102	5	93328	-0.682	4.333
11	737264.7549	4390803.996	-4.706	5	93329	-0.648	4.458
12	737264.4881	4390803.218	-4.755	5	93330	-0.792	4.395
13	737264.2071	4390802.33	-4.812	5	93331	-0.634	4.352
14	737263.7495	4390801.635	-4.778	5	93332	-0.738	4.454
15	737263.1918	4390800.93	-4.881	5	93333	-0.747	4.399

Figura 85 15 primeros datos de la recogida de puntos

Nótese que la cota terreno no es directamente ya que se realizan correcciones previas a la escritura del fichero que se han comentado en el apartado 6.1.2.2.

Los resultados gráficos que obtuvimos en postproceso mediante la clase *Visualiza2D* ya explicada son los siguientes:

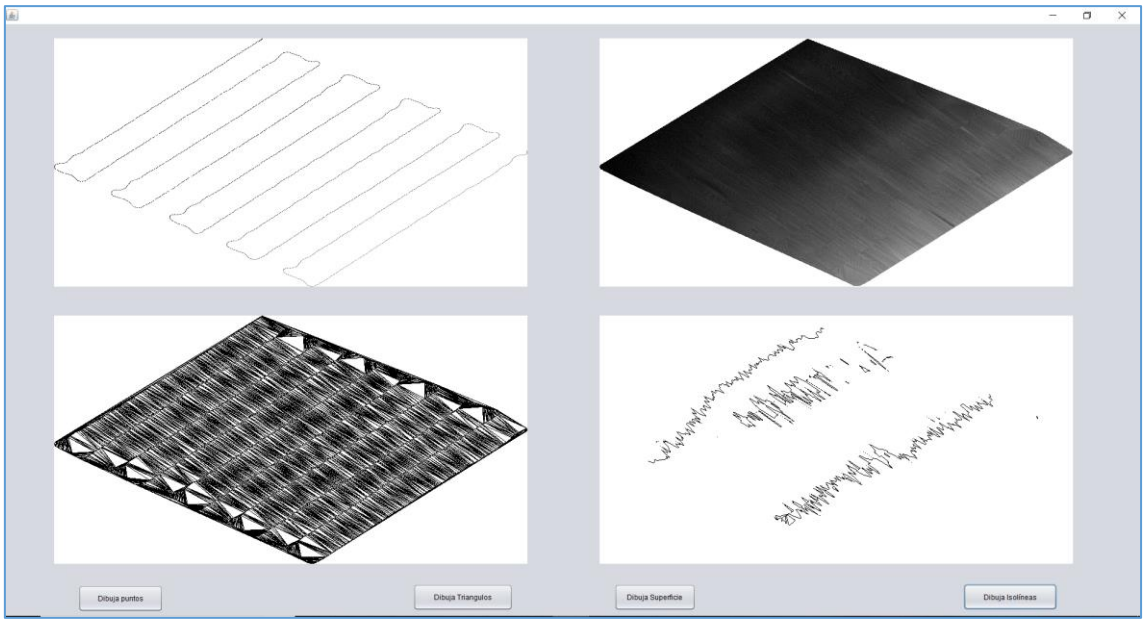


Figura 86 Resultados gráficos prueba Sagunto

Los resultados se ciñen bastante a lo esperado, al tener una nube de puntos tan poco densa por el tipo de sonda que utilizamos (se hará un comentario sobre esto en las conclusiones), los triángulos distan bastante de ser triángulos “regulares”, aún así reflejan bastante bien el tipo de superficie que estábamos midiendo (el suelo marino cercano a la línea de costa, que puede verse como una rampa).

El problema de tener una nube de puntos tan poco densa es que la interpolación lineal de la que se obtiene el MDT no es todo lo precisa que debería ser, por tanto vemos por ejemplo en el nivel de cota -4 (imagen abajo izq. por orden de aparición de izquierda a derecha: cota -3, -4, -5) como la isolínea que se calcula pega muchos saltos, se van encontrando valores de -4 en triángulos muy separados entre sí. Sabemos pues, que esta isolínea no se ha calculado correctamente. Las otras dos isolíneas si que parecen asemejarse más a la realidad del terreno, aunque también cuentan con muchos saltos de triángulo en triángulo.

A continuación introducimos el archivo *curvas.dxf* en el programa de información geográfica que estamos utilizando para dar vistosidad a los resultados, previamente se tiene que guardar el archivo *curvas.dxf* como un archivo .dwg para que puede ser leído por este programa. Se añade estas curvas.dwg (isolíneas negras) a una vista en la que se ha incorporado una ortofoto de la zona y el shape de las curvas de nivel (lisolíneas

naranjas) que nos proporciona la empresa, calculadas mediante técnicas batimétricas en el año 2017.

Figura 87 Comparación de resultados

Vemos como las isolíneas que predecíamos que estaban bien calculadas (-3 y -5) se asemejan mucho a las que nos proporciona la empresa. Las isolíneas proporcionadas por la empresa están calculadas mediante técnicas batimétricas consolidadas y con el uso de software comercial. Y aunque vemos que existe una mínima variación entre ellas, el patrón de comportamiento del curvado se mantiene, por lo que consideramos que la toma y el tratamiento de los datos ha sido correcto.

7 CONCLUSIONES

Una vez alcanzados todos los objetivos de la tesina y habiendo quedado ampliamente satisfecho con los resultados obtenidos se pueden sacar una serie de conclusiones acerca del mismo.

Como se comentaba en la introducción del presente proyecto, se ha podido hacer un trabajo de ingeniería en en un ámbito de aplicación real para una empresa muy importante en el sector de la ingeniería civl que ha motivado al alumno a la consecución de los objetivos marcados en un inicio.

Además el software se ha desarollado en su totalidad en lenguaje Java profundizando en su conocimiento y haciendo uso de esta gran herramienta de acceso libre con la que se ha combinado desarrollo propio con clases que han servido de apoyo tanto en la programación como en la "investigación algorítmica" realizada para conseguir lograr una fiel representación del terreno.

A nivel Hardware extraemos una conclusión que ya intuíamos y es que con la sonda monohaz queda una nube de puntos muy pobre, con esa baja cantidad de datos no se puede extraer información que se pueda aplicar en un trabajo real ya que el terreno no se va a representar de manera fiel a la realidad. Se hace saber a la empresa que es necesario instalar una sonda multihaz en las siguientes fases del proyecto.

Cabe mencionar el alcance de esta técnica con el uso de un USV frente a las técnicas tradicionales en la que se monta todo el equipo sobre un barco. La prueba de implantación se realizó en mar abierto, pero por la naturaleza del propio USV, consideramos y recomendamos a la empresa que este sistema debe ser utilizado en ambientes cerrados (como puertos o lagos), por la sensibilidad al oleaje y donde el tiempo de ejecución no sea un factor determinante o exclusivo para poder acceder a contratos con empresas privadas o con la administración.

Por último, mencionar que se nos contrató para realizar este tipo de trabajos y al surgir el problema tuve que crear este software para la recogida y tratamiento de datos, no pretende ser lanzado como producto y yo seré el principal usuario de este sistema. Si se quisiese comercializar habría que depurar la parte gráfica y de interfaz del mismo, así como tratar de compactar lo máximo posible el código.

8 TRABAJOS FUTUROS

Siguiendo en la línea de lo realizado en esta tesina se van a proponer una serie de trabajos complementarios que se van a proponer en orden de supuesta menor a mayor complejidad:

- Multihtreading para optimización de procesos (apartado 6.1.5.5)
- Mejora de la interfaz gráfica de manera que adquiera un aspecto más “profesional”, se ha trabajado de forma primitiva en Java en cuanto a interfaz se refiere ya que no era alcance ni objetivo de este proyecto el desarrollo de una interfaz más “artística”.
- Integrar el modelo de geoide en el programa para que no haya que hacer la comprobación previa de la zona donde se va a navegar e introducir manualmente el dato del valor geoidal correspondiente al modelo de geoide usado en España.
- Suavizado de curvas de nivel.
- Seguir la misma filosofía de extracción y tratamiento de datos aplicada a una sonda multihaz como a la que se hace referencia en el apartado 6.1.2.
- Estudiar y profundizar en conocimientos batimétricos para poder desarrollar otro tipo de aplicaciones que puedan ser interesantes para empresas del sector de aguas.
- A nivel hardware sería importante incorporar un gimbal para realizar una correcta medida de datos. Esto estaba fuera del presupuesto inicial y la empresa decidió posponerlo a una fase más avanzada del proyecto.

9 PRESUPUESTO

Se va a realizar el cálculo del coste que ha supuesto el desarrollo de la tesina, teniendo en cuenta tanto las horas de trabajo de alumno y tutor como el material utilizado para poder llevarlo a cabo.

CONCEPTO	Unidades	Precio	Total
1. USV: HarborScout	1	8135 €	8135 €
2. EchoSonda: Echologger EU D24	1	3290 €	3290 €
3. GPS: Emlid Reach RS+	1	775 €	775 €
4. Material para modificaciones USV	1	400 €	400 €
5. Portátil para campo	1	1000 €	1000 €
6. Horas Ingeniero Aeronáutico Junior	350	12 €	4200 €
7. Horas Tutor	40	40 €	1600 €
TOTAL			**19400 €**

10 BIBLIOGRAFÍA

FERNÁNDEZ COPPEL, I. A., *La proyección UTM. (Universal Transversa Mercator),* 2001, Recuperado de http://www.cartesia.org

ARENS, C. A., *The Bowyer-Watson algorithm. An efficient implementation in a database environment,* Department of Geodesy, Faculty of Civil Engineering and Geosciences, Delft University of Technology, s.f.

HARIJAONA RAZAFINDRAZAKA, F., *Delaunay Triangulation Algorithm and Application to Terrain Generation*, International Institute for Software Technology, United Nations University, Macao, 2009.

PRIEGO DE LOS SANTOS, J. E. y PORRES DE LA HAZA, M. J., *LA TRIANGULACIÓN DE DELAUNAY APLICADA A LOS MODELOS DIGITALES DEL TERRENO*, Departamento de Ingeniería Cartográfica, Geodesia y Fotogrametría, Universidad Politécnica de Valencia, s.f.

RUIZ TORRES, B., C*ontour map representation through augmented reality*, Master thesis, Master in Science in Telecomunication Engineering & Management, Universitat Politécnica de Catalunya, 2012.

LEGRÁ LOBAINA, A. A.; ATANES BEATÓN, D. M. y GUILARTE FUENTES, C. Contribución al método de interpolación lineal con triangulación de Delaunay. Minería y Geología, v 30, 2014, pp 58 – 72.

MORENO DURÁN, J. L. y ORDÓÑEZ PÉREZ, S., *Diagramas de Voronoi de alcance limitado*, Departamento Matemática Aplicada II, Universitat Politécnica de Catalunya, 2009.

FELICÍSIMO, A. M., Modelos digitales del terreno. Introducción y aplicaciones en las ciencias ambientales, Oviedo, Pentalfa Ediciones, 2014.

Enlaces de interés:

Clases utilizadas para obtener curvas de nivel. Recuperado de http://udel.edu/~mm/code/marchingSquares/

Proyecto utilizado en la construcción de la triangulación. Recuperado de https://github.com/slemenik/delauney-triangulation

MarineAPI para lectura de puertos. Recuperado de http://www.java2s.com/Code/Jar/r/Downloadrxtxcommjar.htm

Librería para escritura de archivos dxf. Recuperado de https://jsevy.com/wordpress/index.php/java-and-android/jdxf-java-dxf-library/

https://github.com/

https://www.oracle.com/es/java/

https://wiki.jmonkeyengine.org

ANEXO I : MANUAL DE USUARIO

A continuación se va a hacer una breve explicación de como usar el programa del proyecto para un usuario que no sabe nada acerca del mismo, pero si consideramos que tiene conocimientos mínimos del entorno de NetBeans para reducir la carga de información básica y de imágenes.

Será necesario tener descargado en el ordenador el programa NetBeans y, preferiblemente, la última actualización de java disponible, así como tener conectados la sonda y el GPS por el cable USB. Será necesario incluir en la carpeta de jdk de java que se encontrará en archivos de programa los siguientes archivos:

- En el jre > lib > ext se incluirá el archivo RXTXcomm.jar: que contiene las librerías que permiten la comunicación por los puertos serie.
- En el jre > lib > ext se incluirá la librería openmap.jar.
- En el jre > bin se incluirá rxtxSerial.dll

Posteriormente se abre la aplicación de NetBeans desde el icono del escritorio y se abre el proyecto *Batimetría*. Antes de ejecutar el programa es necesario hacer dos pasos previos:

- Iremos al administrador de dispositivos y anotaremos los puertos (COM) en los que están la sonda y el GPS, se anota también el baud rate de cada uno de los dispositivos, siendo el de la sonda 115200 por defecto y que no se debe cambiar.

- Dentro del paquete *application* en la clase *TomaPuntos* es necesario poner los puertos que previamente hemos consultado en el administrador de dispositivos.

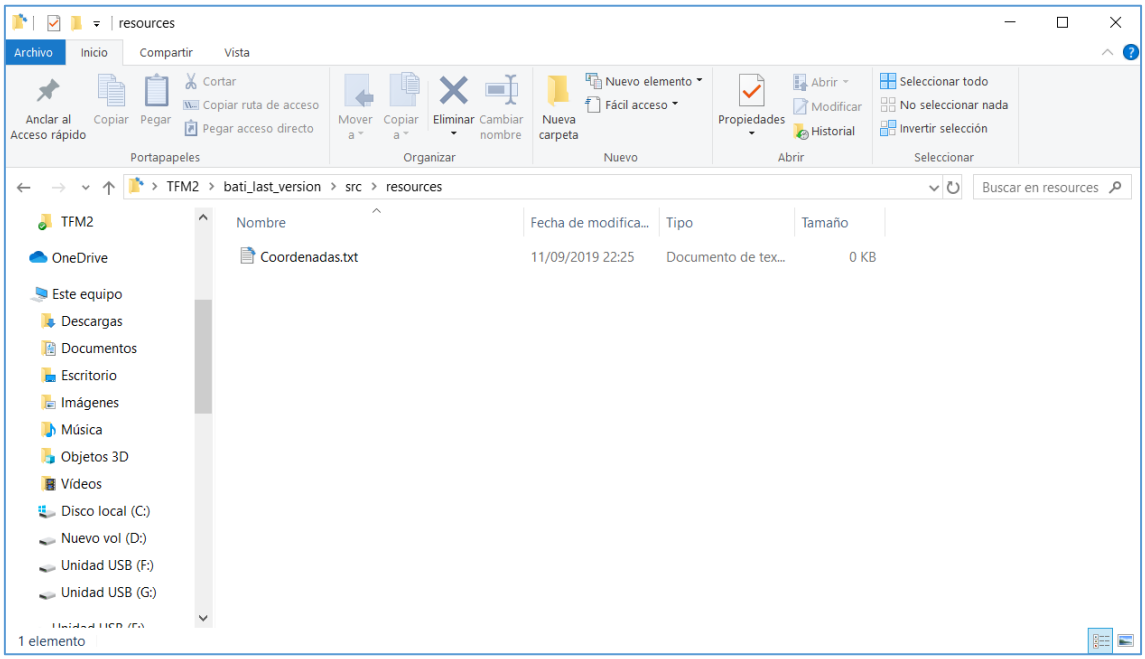
```
56      public TomaPuntos() {
57          initComponents();
58          calculadora = new CalculadoraCoordenadas();
59          gps = new SerialPortExample("COM4", 115200);
60          sonda = new SerialPortExample("COM3", 115200);
61
62      }
```

- Es necesario poner el dato del valor del geoide en la clase *TomaPuntos* en la línea 49. Para ello hay que cargar el modelo de geoide que se adjunta como

recurso del proyecto en un Sistema de Información Geográfica (ArcGis por ejemplo) y consultar el valor de la zona donde se va a operar.

Habiendo hecho esto ya podemos ejecutar el programa, se nos abrirá una ventana y pulsaremos el botón Toma Datos.

Una vez finalizado el plan de navegación y con el USV de vuelta pulsaremos esta vez en el botón Para Toma Datos y se nos guardará un fichero en la carpeta resources, dentro de la carpeta src del proyecto con el nombre de Coordenadas.txt (nombre genérico, falta terminar de definir en que formato se van a sacar las coordenadas y nombrar el fichero en consecuencia).



Una vez llegado a este punto podremos cerrar el programa. Ya que el proceso de generación de la superficie puede dilatarse bastante en el tiempo.

Las clases *Visualiza2D* y *Visualiza3D* ejecutan el código para obtener la superficie, están pensadas para no tener que esperar en campo a que se generen.

Para ejecutar estas clases será necesario tener el archivo que ha sido guardado por la clase *TomaPuntos* en la misma carpeta donde se guardó.

Y habrá que añadir el path siguiente al código "src/resources/Coordenadas.txt" en la línea 181 de la clase *TomaPuntos*, en la 57 de la clase *Delaunay* y en la 342 de la clase *Visualiza2D*. Vemos el ejemplo de como se haría en la clase *TomaPuntos*.

```
181            File fichero = new File("src/resources/Coordenadas.txt");
```

Como apunte, en la clase *Visualiza2D* se deja por defecto un intervalo de cálculo de curvas de nivel de 1 metro, se puede modificar el intervalo escribiendo el valor deseado en metros en intervalo e intervalo_apoyo, pero dejando el signo negativo.

```
599            int puntero_levels = 0;
600            double intervalo = -1;
601            double intervalo_apoyo = -1;
```

Dicho, esto queda ejecutar la clase, esperar que se calculen los resultados y posteriormente pulsar por orden de izquierda a derecha los botones que aparecen abajo en la ventana emergente.