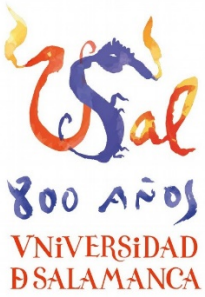
UNIVERSIDAD DE SALAMANCA

Escuela Politécnica Superior de Ávila



MÁSTER EN GEOTECNOLOGÍAS CARTOGRÁFICAS EN INGENIERÍA Y ARQUITECTURA

Trabajo Final de Máster

Complemento de QGIS para la optimización del control de calidad visual de las ortoimágenes del Instituto Cartográfico Valenciano (ICV).

Autor:

Francisco Javier Vaquero Cáceres

Tutor:

David Hernández López

Julio, 2021

ÍNDICE

ÍNDICE DE FIGURAS

Resumen

Se implementa un complemento de QGIS para un control de calidad visual de las ortofotos de la Comunidad Valenciana que gestiona el Institut Cartogràfic Valencià (ICV). A partir de una base de datos Spatialite plantilla, donde se encuentran todas las capas necesarias para dicho control, el complemento realiza una copia de dicha plantilla para trabajar sobre ella y genera una conexión a esa base de datos dentro de QGIS.

Se crean relaciones entre las tablas de la base de datos para conseguir barrer todas las hojas 1:5000 de las ortofotos con un flujo de trabajo ordenado y exhaustivo. El barrido se hace a partir de niveles de zoom a escala 1:2000 y 1:1000.

Finalmente, el resultado de este trabajo es obtener una capa de errores tanto geométricos como radiométricos encontrados en las ortofotos revisadas.

Palabras clave: Ortofotografías, Control de calidad, Barrido, ICV, QGIS

1. Introducción

1.1 Ámbito de trabajo

Desde el año 2018 el Instituto Cartográfico Valenciano manda a concurso un vuelo fotogramétrico digital de 22 cm cada año y la posterior realización de las ortofotos de 25cm sobre la superficie de toda la comunidad valenciana.

El vuelo a realizar debe cumplir las precisiones necesarias para su posterior elaboración de ortofotografías de 25 cm y la restitución de cartografía a escala 1:5.000. Además, se deberá tener en cuenta que se empleará este vuelo para la actualización de los modelos digitales de elevaciones a partir de estereocorrelación en las zonas de cambios.

El ámbito geográfico para todas las fases de los trabajos y los productos resultantes son las hojas completas 1:5.000 de la Comunitat Valenciana, representadas en el gráfico que figura en este documento. Se trata de un total de 3098 hojas 1:5.000 según el corte establecido en base al REAL DECRETO 1071/2007, de 27 de julio, por el que se regula el sistema geodésico de referencia oficial en España.

Todos los trabajos se llevan a cabo bajo la supervisión del Instituto Cartográfico Valenciano. El ICV suministrará los gráficos de las hojas CV05 que hay que realizar en el huso 30 y también el gráfico de las hojas CV05 que se deben entregar en el huso 31.

La superficie total de la ortofoto a generar será de 2.479.070 hectáreas.

A continuación, se muestra de forma gráfica el ámbito de las ortofotos que se generan cada año:

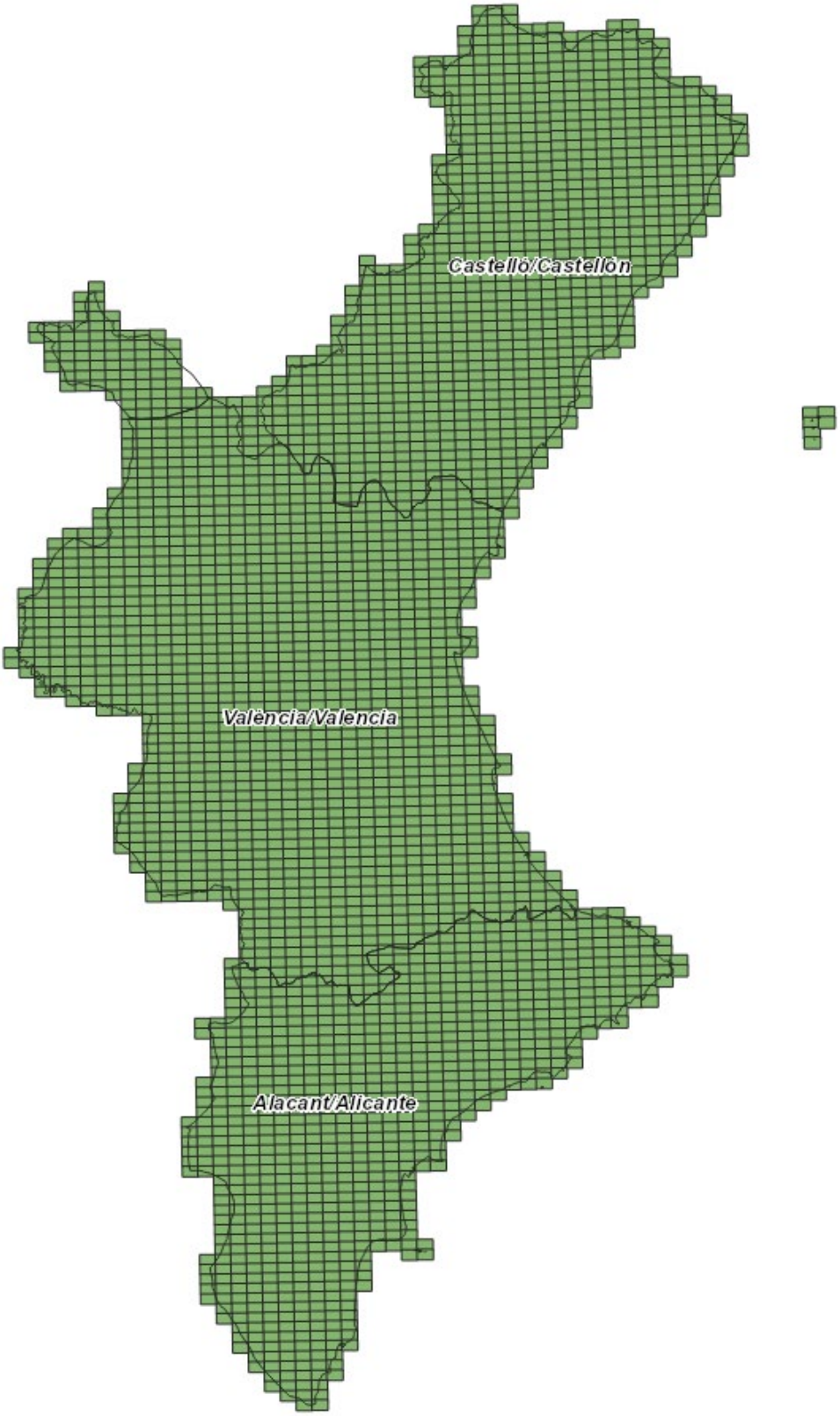


FIGURA 1. DISTRIBUCIÓN DE HOJAS CV05 DEL HUSO 30 PARA LA REALIZACIÓN DE LA ORTOFOTO.

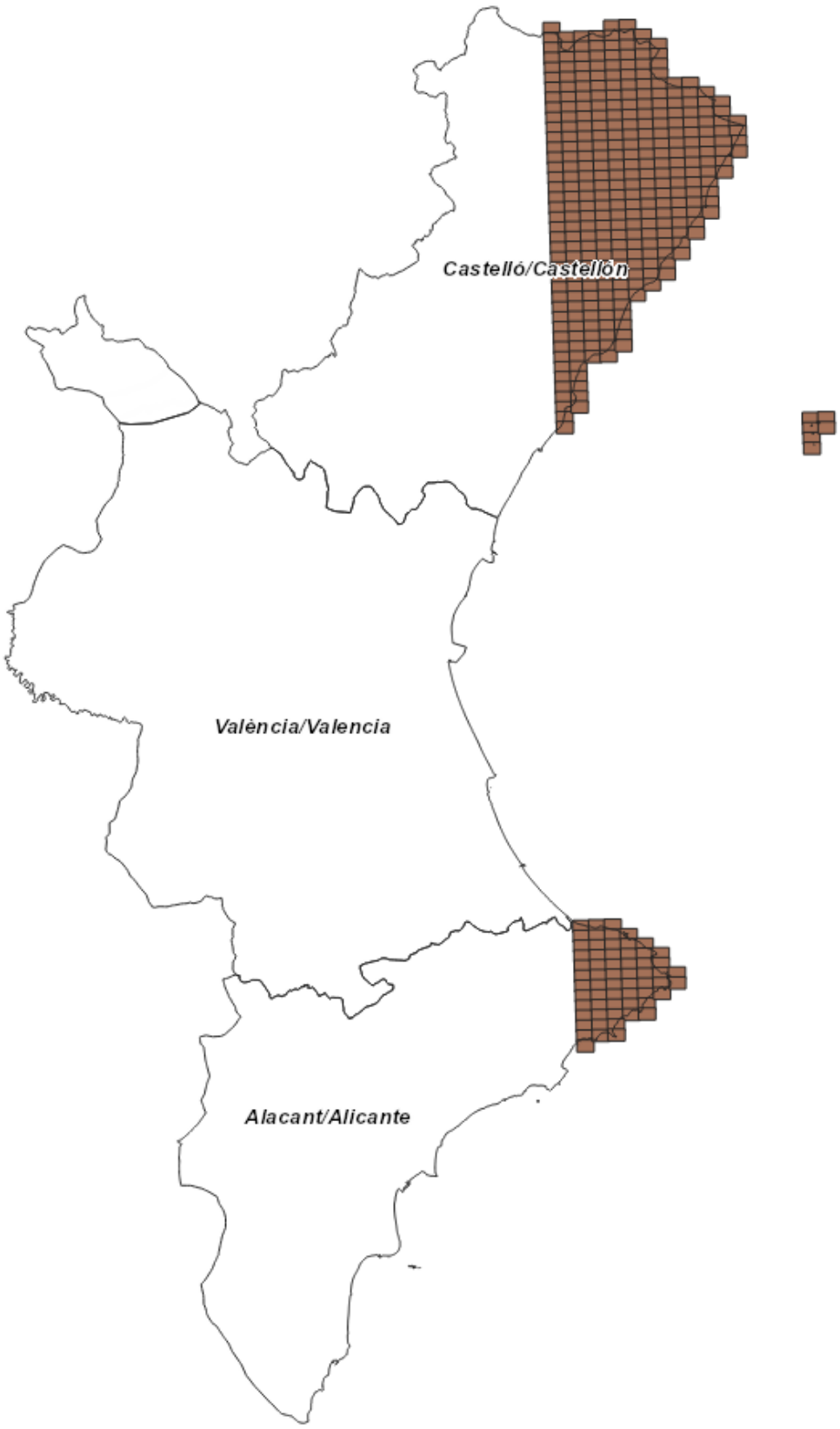


FIGURA 2. DISTRIBUCIÓN DE HOJAS CV05 DEL HUSO 31 PARA LA REALIZACIÓN DE LA ORTOFOTO

1.2 Plazo de ejecución de los trabajos

Los trabajos serán realizados de acuerdo con el calendario del Plan de Trabajo del adjudicatario, aceptado por la dirección técnica, pero siempre cumpliendo los siguientes hitos de entregas. En el caso de este último año 2020:

-Vuelo del año 2020 y todos sus productos generados antes del 30 de junio del 2020.

-Aerotriangulación vuelo 2020 y Ortofoto rápida 2020 antes del 31 de agosto de 2020.

- Modelo digital de elevaciones 2020 y ortofoto rigurosa 2020 antes del 1 de diciembre de 2020.

La empresa va entregando a la dirección técnica el material y la documentación de las diferentes fases para permitir que el control de calidad se vaya realizando paralelamente a la ejecución del trabajo. La frecuencia de dichas entregas parciales las indicará la dirección técnica. La empresa mantendrá permanentemente informada a la dirección técnica de la evolución y posibles incidencias de los trabajos.

1.3 El vuelo fotogramétrico y ortofoto digital

La ejecución de un proyecto fotogramétrico requiere de una precisa planificación del mismo y más concretamente del vuelo fotogramétrico, que garantice que la cobertura fotográfica de la zona a levantar es la adecuada para satisfacer el objetivo del trabajo cumpliendo al mismo tiempo las especificaciones reflejadas en el pliego de condiciones.

El proyecto de vuelo es el conjunto de cálculos previos a la realización de un vuelo fotogramétrico, mediante los cuales se organizan las operaciones que permiten conseguir el fin propuesto bajo unas condiciones establecidas previamente. El objetivo del vuelo es el de cubrir una determinada zona con imágenes que cumplen los porcentajes de recubrimiento longitudinal y transversal especificados, sobrevolando la zona a una altitud determinada en función de la escala deseada y de la distancia principal de la cámara. Para conseguirlo, el avión deberá volar a una altitud constante, siguiendo una ruta predeterminada, y a una velocidad constante que permita realizar disparos a intervalos regulares que se correspondan con recorridos iguales.

En la planificación del vuelo hay que adoptar una serie de decisiones previas que condicionan las características del vuelo (escala del mapa a realizar, formato de los fotogramas, proyección del mapa, elipsoide de referencia, etc.) que se determinan, fundamentalmente, en función del propósito en el que se utilizarán las fotografías aéreas.

Así pues, para la obtención de planos se precisarán buenas condiciones métricas de las fotografías por lo que habrá que utilizar cámaras métricas calibradas, películas con granulometría fina, tiempos de exposición cortos y emulsiones con alta resolución. Cuando se trata de mapas topográficos es aconsejable utilizar cámaras gran angulares o super gran angulares, para obtener una amplia relación base-altura (B/H).

Los condicionantes que influyen en la calidad de la fotografía, tanto en su aspecto geométrico como fotográfico que se consideran en esta fase son los siguientes:

- Aspectos geométricos:

- Certificado de calibración de la cámara, que facilita los parámetros de orientación interna (distancia principal, punto principal, distorsiones).
- Ground Sample Distance (GSD)
- Recubrimientos longitudinales y laterales.
- Seguridad de un recubrimiento total en toda la zona.
- Arrastre de la imagen sobre la fotografía.
- Horas útiles de tomas fotográficas.

- Condiciones fotográficas:

- Contraste fotográfico de la película.
- La calidad de la imagen.
- La homogeneidad de tonalidad.
- La ausencia de nubes.
- Longitud e intensidad de las sombras.

Como el vuelo es para toda la comunidad Valencia, se adopta que los ejes de vuelo están orientados Este-Oeste, numerando las pasadas según este criterio, así como las fotografías de cada pasada. (Pérez Álvarez, Juan Antonio, 2001)

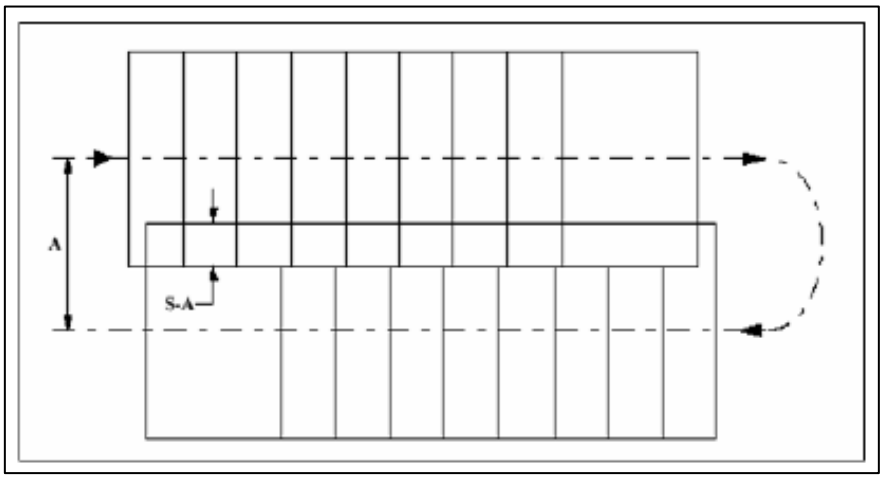


FIGURA 3. EJES DE VUELO CON ORIENTACIÓN W-E Y E-W(F.J. HERMOSILLA Y S. LÓPEZ CUERVO)

Una ortofoto contiene la riqueza informativa de una foto con la métrica convencional de un mapa o plano. Para ello es necesario corregir en la foto original los desplazamientos debidos a la inclinación del eje óptico de la cámara en el momento de la toma (rectificación) y el desplazamiento radial debido al relieve de la superficie topográfica fotografiada.

Por lo tanto, una ortofoto es un mapa hecho a partir de fotos, de manera que todos los detalles naturales o artificiales comprendidos en el terreno están en una perfecta posición planimétrica (x, y) con arreglo a la proyección empleada en la representación.

La idea de conseguir en una foto aérea eliminar de alguna manera las deformaciones debidas a la inclinación del eje óptico y al relieve es muy antigua. Lo primero que se fabricaron fueron unos instrumentos muy sencillos llamados "rectificadores" que permitían corregir los efectos de la perspectiva debidos a la falta de verticalidad del eje de toma utilizando tres "puntos de control", en terreno llano el procedimiento anterior era perfecto, pero en terreno ondulado se hace necesario corregir también los desplazamientos debidos al relieve y para esto se crearon unos instrumentos llamados "ortoproyectores", basados en restituidores fotogramétricos analógicos y posteriormente en los analíticos.

Pero ha sido con la fotogrametría digital con la que la producción de ortofotos ha cobrado gran importancia, de hecho, hasta ahora la fotogrametría digital ha estado más enfocada a la creación de ortofotos y actualmente hay que reconocer que la mayoría de los sistemas digitales se están utilizando para su producción mediante un proceso rutinario debido a su simplicidad.

Esto ha hecho que sea la tarea fotogramétrica más extendida actualmente y las series cartográficas más utilizadas hoy en día son las ortofotos como complemento a los mapas de línea en zonas o países muy desarrollados y como cartografía básica en aquellas zonas más desfavorecidas.

Para la confección de ortofotos digitales a escala 25.000 o mayor se han utilizado fotos convencionales e imágenes de satélite para escalas inferiores al 25.000, no obstante, todo esto es orientativo dado el continuo desarrollo de la información procedente de vehículos espaciales.

Teniendo en cuenta que la confección de un mosaico de ortofotos a partir de fotos aéreas es más difícil que cuando las imágenes proceden de satélites, ya que debido a la gran apertura de campo y a las características ópticas del objetivo se producen modificaciones en la radiometría de los píxeles en función del ángulo que forma cada rayo incidente respecto al eje óptico.

Si se trata de ortofotos en color, las dificultades de la homogeneización radiométrica se acrecientan considerablemente.

La disponibilidad cada vez más frecuente del MDT ha permitido trabajos de actualización y puesta al día a partir de métodos monoscópicos empleando ortofotos como fuente de información, así como la posibilidad de integrar ortofotos digitales en los SIG.

Las ventajas de la ortoproyección digital sobre la ana1ógica tradicional son múltiples, y entre ellas podemos citar:

- Mejora en la calidad de la imagen del producto resultante gracias a la eliminación del complicado proceso óptico de los equipos ana1ógicos y su sustitución por las estaciones fotogramétricas de imágenes digitales.

- Mayor facilidad para el tratamiento de las fotografías en color

- Posibilidad de realización de mosaicos digitalmente.

Para generar las ortofotos se necesita una imagen con su orientación interna conocida (OI): *f, x0, y0,* con la orientación externa conocida (OE): *X0, Y0, Z0, ω,* φ , *κ* y con un MDT conocido.

Se define una ortofoto sin colores, por sus dimensiones resolución. Se trata de encontrar el color de cada píxel. (F.J. Hermosilla y S. López Cuervo)

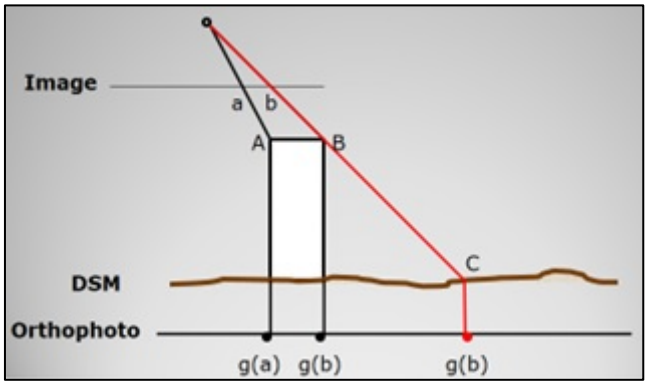


FIGURA 4. IMAGEN – MDT – ORTOFOTO. LOS PIXELES IMAGEN SE ORTOPROYECTAN UNO A UNO, PASANDO POR EL MDT. (MEHAD HAGGAG, 2018)

2. Estado del arte

QGIS es una aplicación profesional de SIG que está construida sobre Software Libre y de Código Abierto (FOSS), licenciado bajo GNU - General Public License . Es un proyecto oficial de Open Source Geospatial Foundation (OSGeo). Funciona sobre Linux, Unix, Mac OSX, Windows y Android y soporta numerosos formatos y funcionalidades de datos vector, datos ráster y bases de datos. El último lanzamiento a junio de 2021 es *QGIS 3.20*

La necesidad de crear un complemento en QGIS con un flujo de trabajo como el que se va a describir en el presente trabajo, viene porque a día de hoy el control de calidad de las ortofotos no se realiza de forma exhaustiva y visual.

Existen métodos que las empresas privadas adaptan para llevar dicho control. En estos trabajos es muy sencillo que se puedan cometer errores groseros por el trabajo repetitivo y monótono que conlleva el control de calidad.

Actualmente, no existe ningún complemento en QGIS que haga un barrido ordenado por hojas 5.000 donde vayas pasando hoja tras hoja con diferente zoom para encontrar posibles errores que puedan originarse por diferentes motivos.

La idea de utilizar diferentes zooms viene a la hora de la detección visual del error. Es posible que un error radiométrico a un zoom grande no se vea y sin embargo con un zoom más pequeño y una vista general sí. En el caso de errores geométricos pasa, al contrario, con un zoom pequeño no llegas a apreciar ese error y si te acercas con un zoom más grande se llega a ver.

FIGURA 5. EJEMPLO DE ERROR RADIOMÉTRICO A ESCALA GRANDE (ANEXO II)

El trabajo es un plugin específico para las ortofotos de la comunidad valenciana, adaptado a un flujo de trabajo pensado en el Institut Cartográfic Valencià que es el organismo y dirección técnica que gestiona todo el proyecto. En futuras versiones, lo ideal

es extrapolar el trabajo realizado a otras zonas más pequeñas o incluso a otras zonas fuera de la comunidad Valenciana.

Como ya se ha comentado, es un trabajo monótono y visual para la detección de errores geométricos y radiométricos y las empresas que se dedican a ese control utilizan software donde se pueda manejar imágenes de gran tamaño y de forma fácil e intuitiva.

Global Mapper es un software de sistemas de información geográfica desarrollado por Blue Marble Geographics, idóneo entre otras cosas para la visualización de rasters y la creación de modelos digitales. Donde también se puede hacer un barrido de los raster con un zoom determinado por el usuario, pero la posibilidad de comer equivocaciones y no llevar un correcto orden es alta.

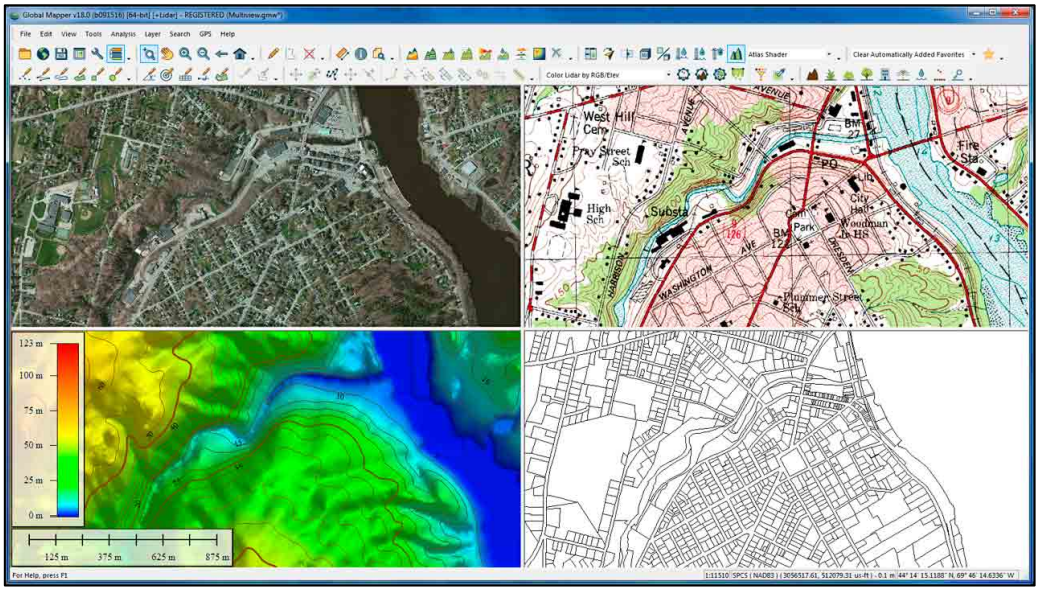


FIGURA 6. IMAGEN DEL SOFTWARE GLOBAL MAPPER DESARROLLADO POR BLUE MARBLE GEOGRAPHICS

El software GIS compite con los productos de ESRI, Geomedia, Manifold System, Mapinfo GIS… entre otros. Dichos softwares son todos de licencia comercial.

FIGURA 7. LOGO SOFTWARE ARCGIS PRO DE ESRI

FIGURA 8. LOGO SOFTWARE GEOMEDIA DE HEXAGON

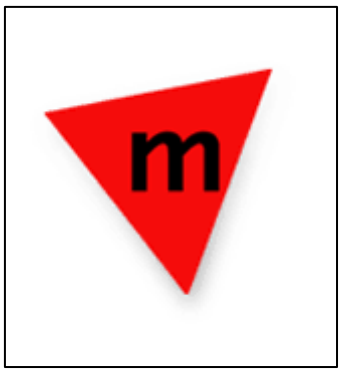


FIGURA 9. LOGO SOFTWARE MANIFOLD SYSTEM GIS

FIGURA 10. LOGO SOFTWARE MAPINFO GIS

Dirección de información, URL de los softwares citados:

Global Mapper: https://www.bluemarblegeo.com/global-mapper/

ArcGIS PRO: https://www.esri.com/en-us/home

Geomedia: https://www.hexagongeospatial.com/products/power-portfolio/geomedia

Manifold System: https://www.manifold.net/

MapInfo: https://www.precisely.com/product/precisely-mapinfo/mapinfo-pro

3. Objetivos del trabajo

El objetivo principal de dicho trabajo es implementar un complemento de QGIS que, a partir de un modelo de datos y los tipos de errores, tanto geométricos como radiométricos que puedan aparecer, se pueda gestionar y optimizar el control de revisiones de la ortofoto.

Como es habitual en el control visual de los productos cartográficos, el nivel de zoom es muy importante para hacer un barrido, es por eso que se ha pensado que la mejor manera de hacer un control es permitirle al usuario que pueda interactuar con la escala de zoom del software QGIS de manera rápida y sencilla con tan solo pulsar unos botones.

El modelo de datos es una base de datos Spatialite plantilla y en el cual se integran las cuadriculas que se van a utilizar para el control de calidad de la ortofoto. La idea es que el usuario que vaya a hacer el barrido tenga preparada toda la información y tener un mecanismo de trabajo sencillo. Las cuadriculas añadidas han sido creadas a partir de la cuadricula oficial 5000 de ortofotos proporcionado por el Institut Cartogràfic Valenciá.

El usuario podrá marcar todos los errores que encuentre, indicar el tipo de error, en qué hoja 5000 se encuentra y finalmente exportarlos en un archivo Shapefile.

4. Medios empleados

5.1. Medios empleados en la creación del complemento.

Para la creación del complemento se han utilizado los siguientes softwares:

La plataforma de trabajo y entorno se utiliza el software *QGIS* en la versión 3.10-A Coruña en la versión de largo plazo y el lenguaje de programación utilizado es *Python* en su versión 3.8.

FIGURA 11. LOGO DE QGIS 3.10- A CORUÑA

El software que se ha utilizado para la edición de los ficheros que componen el complemento ha sido *Pycharm* en la versión 2020.1.5 en el cual utiliza el intérprete de *Python.* Para su desarrollo el complemento utiliza las siguientes librerías:

-*QGIS.Core*. Es la biblioteca principal de *QGIS.* Contiene la mayoría de los objetos no basados en la interfaz de usuario que puede utilizar para crear un complemento o una aplicación independiente. Esta biblioteca contiene cosas como capas (*QgsMapLayer, QgsVectorLayer*), registro de capas (*QgsMapLayerRegistry),* objetos de geometría (Q*gsGeometry*), objetos de características (*QgsFeature*).

-*QGIS.PyQt*. Biblioteca gráfica de *Python* adaptación de las versiones originales en C++ de Qt.

-*PyQt5*. Conjunto completo de enlaces de *Python* para *Qt v5*. Se implementa como más de 35 módulos de extensión y permite que *Python* se utilice como un lenguaje de desarrollo de aplicaciones alternativo a *C ++* en todas las plataformas compatibles, incluidas iOS y Android.

–*QGIS.utils*. Es un conjunto de utilidades que utiliza la aplicación para hacer cosas basadas en Python. Algunas de estas cosas son configurar complementos, agregar manejo de errores de Python, etc.

Para la creación del complemento se utiliza el complemento *"Plugin Builder"* de *QGIS*. Sirve para la creación de la estructura inicial y de los archivos necesarios para el correcto funcionamiento. Se ha utilizado otro complemento de *QGIS "Reload Plugin"*, se encarga de compilar de nuevo el complemento después de cada modificación en el código y poder probarlo dentro de *QGIS.* Y para la creación de la interfaz gráfica se ha utilizado el programa *"Qt Designer with QGIS 3.10.14 custom widgets",* programa que se instala junto a QGIS y te permite acceder a widgets (en este caso a botones) de *PyQt* y de *PyQGIS.*

5.2. Medios empleados en la creación del modelo de datos.

La parte fundamental del complemento es definir una base de datos fija para que el trabajo se lleve correctamente. Este proceso lo podemos definir en 3 fases:

5.2.1. Creación de una base de datos Spatialite.

Para crear un archivo Spatialite en QGIS, hay que ir a al menú de *Capa/ Crear capa/Nueva capa Spatialite:*

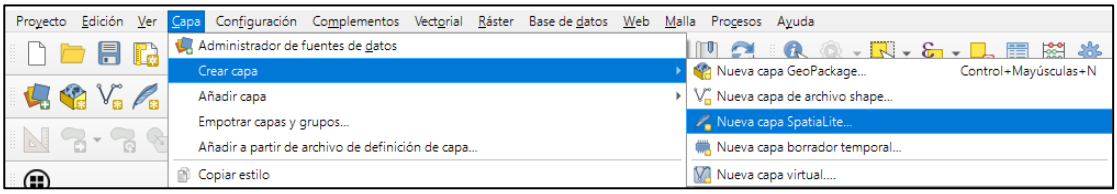


FIGURA 12. PASO 1 CREACIÓN BASE DE DATOS

Sale un desplegable y tan solo hay que indicar el directorio y el nombre de la base de datos:

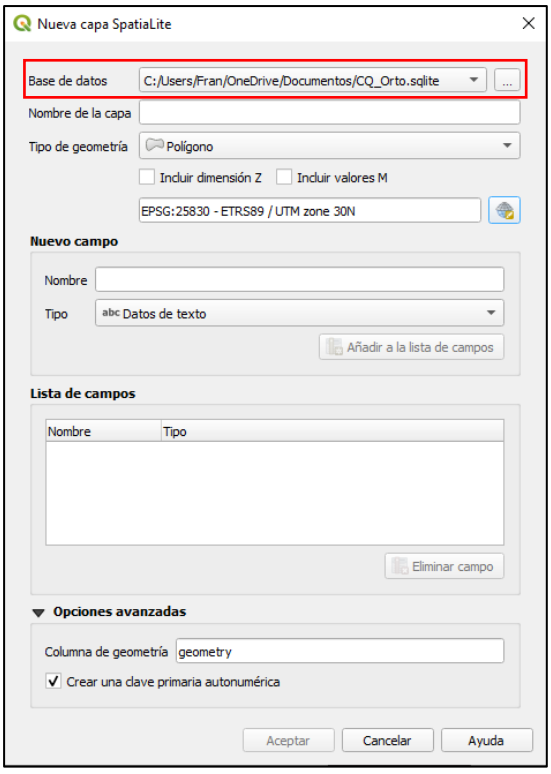


FIGURA 13. PASO 2 CREACIÓN BASE DE DATOS

Directamente crea un archivo spatialite vacío listo para que se puedan añadir los elementos que se desee.

5.2.2. Creación de las capas con FME Desktop.

FME Desktop es la herramienta ETL espacial más usada y flexible para la integración, procesamiento y control de calidad de información geoespacial. La herramienta permite integrar una gran cantidad de formatos de datos diferentes de una manera rápida y eficiente dentro de procesos FME, llamados espacios de trabajo, reestructurándolos a medida y transformándolos en nuevos modelos de datos especificados por el usuario. Es la manera más sencilla para automatizar procesos de gestión de datos espaciales y análisis espacial.

Las siglas ETL son la abreviatura de Extract, Transform and Load (Extraer, Transformar y Cargar). Este tipo de herramientas son las que permiten diseñar procesos para mover datos desde múltiples fuentes, reformatearlos, limpiarlos, y cargarlos en otra base de datos o sistema de almacenamiento de datos. Se convirtieron en un concepto popular durante los años 1970. FME Desktop no sólo es un ETL, sino que permite la manipulación de datos con geometrías, y es por ello por lo que se sitúa en la herramienta fundamental para profesionales de sistemas de información geográfica.

Estos procesos o espacios de trabajo son capaces de automatizar operaciones para la transformación estructural y de contenido de tablas y geometrías, de manera que evita que se tengan que repetir tareas manuales una y otra vez, sin necesidad de programar. Los espacios de trabajo se crean a través de una interfaz (FME Workbench) donde se pueden añadir o eliminar objetos de FME simplemente arrastrándolos y soltándolos.

FME Desktop permite una gran cantidad de operaciones con nuestros datos como, por ejemplo:

Convertir datos de un formato a otro: FME dispone de más de 300 formatos diferentes que facilitan la integración de los datos. Entre ellos se incluyen tanto formatos espaciales como CAD, SIG, bases de datos, ráster, BIM… así como formatos de datos no espaciales como CSV, DBF, etc.

Transformación de datos: Permite la transformación tanto del contenido de los datos (es decir, cambiar o añadir valores de atributos mediante cálculos numéricos o manipulación de cadenas de caracteres, incluyendo también las modificaciones geométricas) como de la estructura de los datos (añadir, quitar o renombrar atributos). Dispone de más de una galería con más de 475 transformadores predefinidos para el procesamiento de los datos geoespaciales.

Fusión de datos: Es posible unificar diferentes tablas en solo una o fusionar tablas de datos a su geometría cuando se encuentren separadas

Filtrado de datos: La operación contraria a lo anterior expuesto. Desde una tabla de origen se puede obtener más de una filtrando por algunos de los atributos. (Martí, C, &, Asociación Geoinnova, 2018).

El modelo de datos consta de las siguientes capas:

- Cuadriculas_ortofoto_mdt_5000_25830_icv_round_10m: Se corresponde a la cuadricula oficial del Instituto Cartográfico Valenciano, con escala 1:5000 de las ortofotos de la Comunidad Valenciana.
- Cuadriculas_zoom2000_rev: Capa de tipo polígono que corresponda con un zoom a una cuadricula equivalente en QGIS a 1:2000
- Cuadriculas_zoom1000_rev: Capa de tipo polígono creada para que se corresponda con un zoom a una cuadricula equivalente en QGIS a 1:1000. Por cada cuadricula 1:2000 corresponden hasta 4 cuadriculas de 1:1000
- errores: Capa de tipo polígono vacía creada para almacenar los distintos errores que nos podamos encontrar en la revisión de las ortofotos.

A partir de la cuadricula oficial de la ortofoto 5000 proporcionado por el ICV se han creado con el software FME Desktop las dos siguientes cuadriculas.

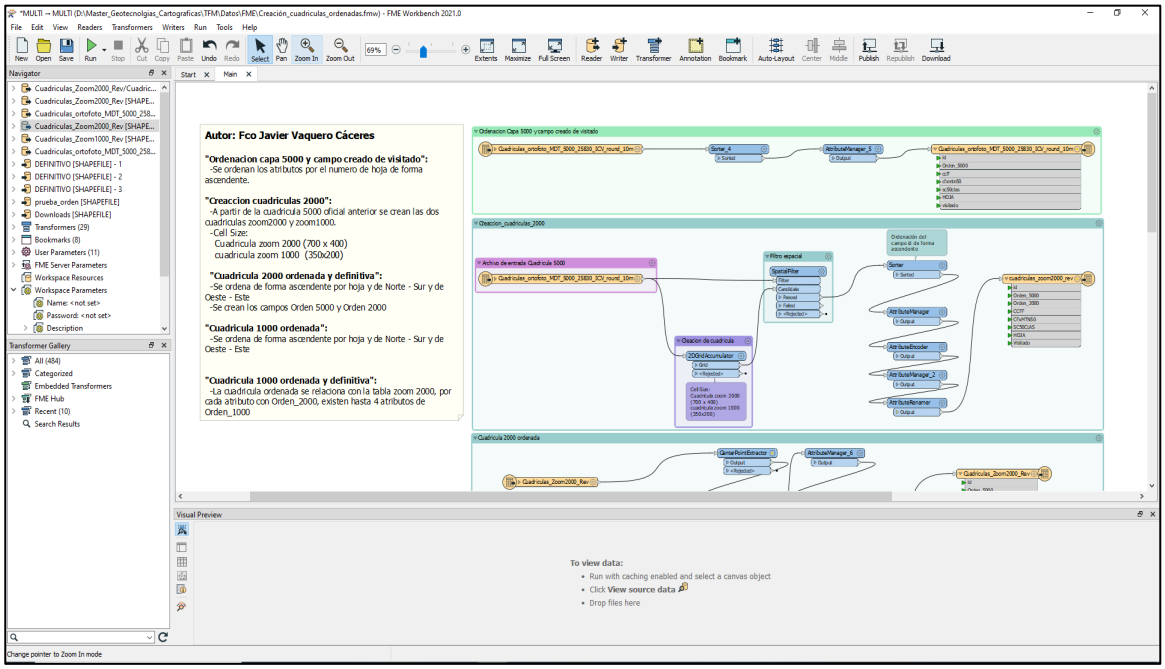


FIGURA 14. INTERFAZ GRÁFICA DEL PROYECTO DE FME DESKTOP UTILIZADO

El proceso que se ha seguido se divide en dos partes:

-Primera parte. Creación de las dos cuadriculas a partir de la 5000 con el transformador de FME "2dGridAccumulator".

Se ha calculado para un monitor estándar de 24" el tamaño de la cuadricula con una escala alrededor de 1:2000 tiene que ser 700 x 400 y para la cuadricula con escala alrededor 1:1000 un tamaño de celda de 350 x 200. Son los parámetros que se meten en el transformador:

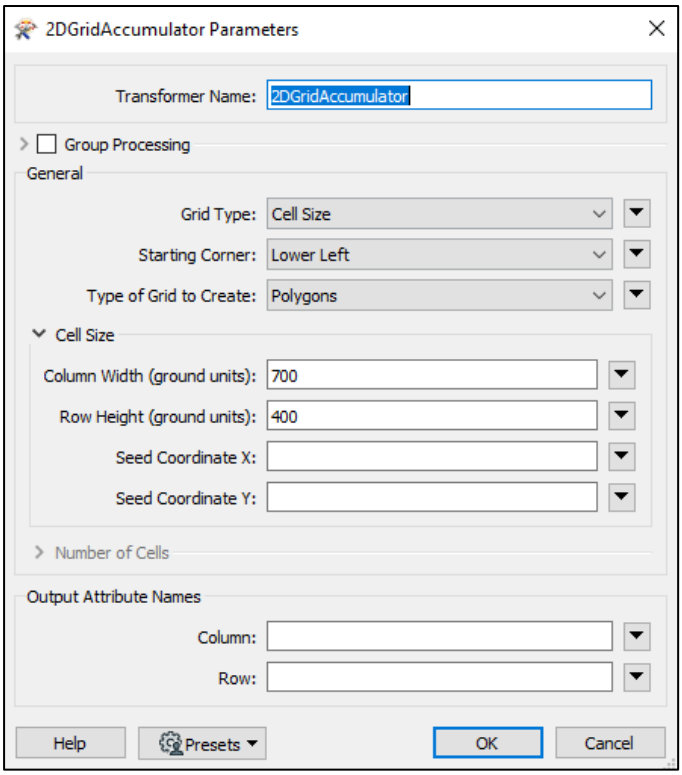


FIGURA 15. TRANSFORMADOR CREACIÓN DE CUADRICULA 2DGRIDACCUMULATOR

El resultado final queda que para una hoja 5000, existen varias hojas 2000 dentro de ella. En la figura siguiente se puede ver la cuadricula 5000 en color naranja y las cuadriculas amarillas las que corresponden con las hojas con escala 1:2000.

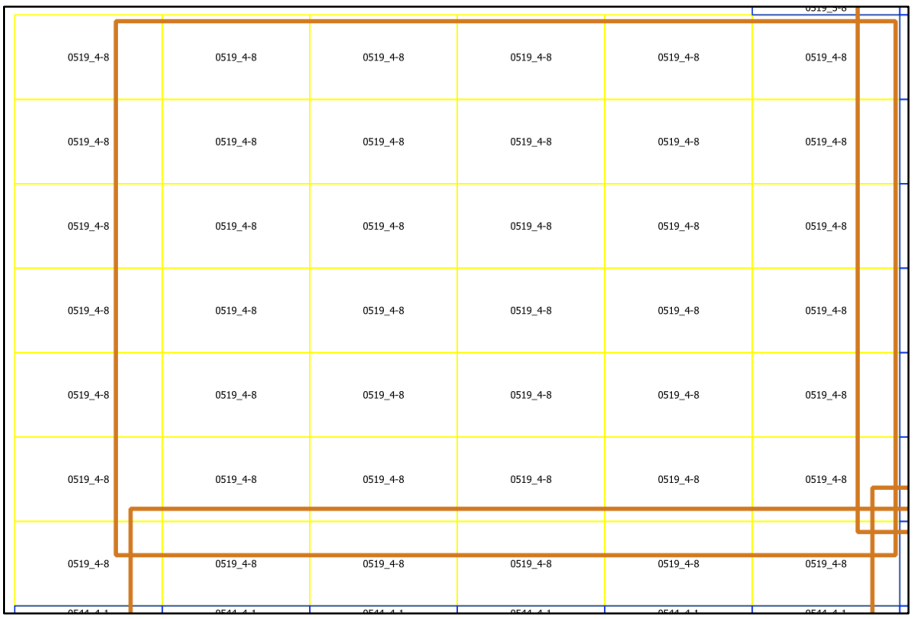


FIGURA 16. ILUSTRACIÓN DE CUADRÍCULAS 5000 Y 2000

Y para una hoja 2000 existen hasta 4 hojas 1000. En la figura siguiente, las cuadriculas magentas corresponden a las hojas 1000 y la azul a las cuadriculas 2000.

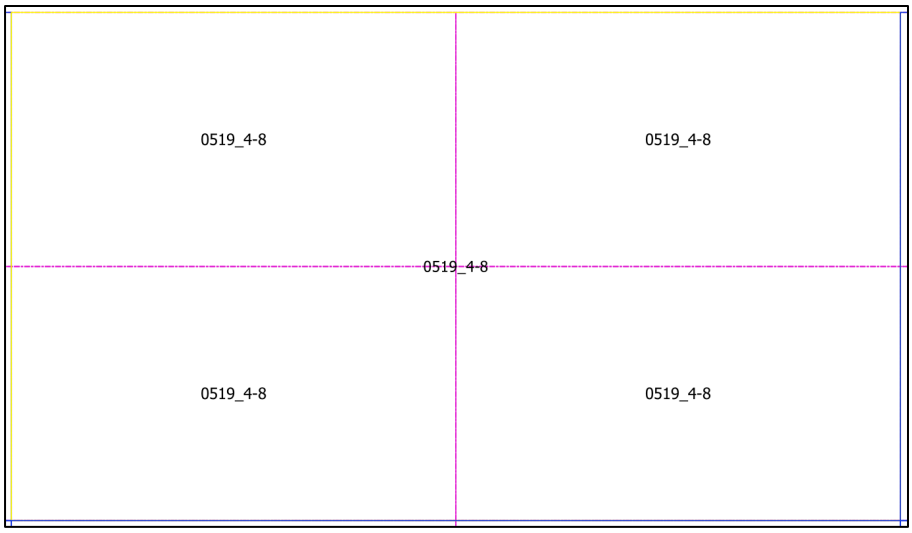


FIGURA 17. ILUSTRACIÓN DE CUADRÍCULAS 2000 Y 1000

-Segunda parte. Relación de cuadrículas entre sí.

Una vez tenemos las cuadriculas creadas geométricamente, se relacionan de forma lógica con campos en común. Por tanto, se establece que la *Cuadriculas_ortofoto_mdt_5000_25830_icv_round_10m* tiene los siguientes campos:

Ogc_fid: Id creado automáticamente

Orden_5000: Campo creado de tipo entero y es una copia del campo ogc_fid para trabajar con las relaciones entre las siguientes tablas. Están ordenadas en orden ascendente por el número de la hoja. Ejemplo la hoja 0519_4-8 llevara en este campo el valor 1 y la hoja 0935_5-2 llevara el valor 3098

Ccff: Campo de tipo texto por defecto de la cuadrícula,

Cfxmtn50: Campo de tipo texto por defecto de la cuadrícula,

Sc50clas: Campo de tipo texto por defecto de la cuadrícula,

Hoja: Campo de tipo texto por defecto de la cuadrícula, corresponde con el numero de la hoja.

visitado: Campo creado de tipo entero donde 0 significa que el campo no ha sido visitado y 1 que sí se ha visitado

Cuadricula Cuadriculas 5000

Ogc_fid	Orden_5000	ccff	cfxmtn50	sc50clas	hoja	visitado
1	2	228-160	4-8	0519	0519_4-8	0
2	2	229-160	5-8	0519	0519_5-8	0
:	:	:	:	:	:	:

FIGURA 18. IMAGEN GRÁFICA DE LA TABLA DE ATRIBUTOS CUADRICULA 5000

La *Cuadriculas_zoom2000_rev* tiene los siguientes campos:

Ogc_fid: Id creado automáticamente

Orden_5000: Campo de tipo entero creado y heredado de la cuadricula 5000 y asignado por relación espacial.

Orden_2000: Campo creado de tipo entero y se crea con el fin de relacionarla con la cuadricula anterior. Es decir, en una cuadricula con *Orden_5000=1*, habrá varias cuadriculas. A la hora de asignar el valor, el primer valor se corresponderá con la cuadricula que pertenece al mismo *Orden_5000* y tiene como coordenada del centro de la cuadricula la X mínima y la Y máxima, el siguiente valor será la siguiente cuadricula con X mínima e Y máxima y así vamos rellenando el campo *Orden_2000*.

Hoja: Campo de tipo texto por defecto de la cuadricula, corresponde con el numero de la hoja.

visitado: Campo creado de tipo entero donde 0 significa que el campo no ha sido visitado y 1 que sí se ha visitado

Cuadricula Cuadriculas Zoom_2000

Ogc_fid	Orden_5000	Orden_2000	hoja	visitado
1	1	1	0519_4-8	0
2	1	2	0519_4-8	0
3	1	3	0519_4-8	0
:	:	:	:	:

FIGURA 19. IMAGEN GRÁFICA DE LA TABLA DE ATRIBUTOS CUADRICULA 2000

La *Cuadriculas_zoom1000_rev* tiene los siguientes campos:

Ogc_fid: Id creado automáticamente

Orden_5000: Campo creado y heredado de la cuadricula 5000 y asignado por relación espacial.

Orden_2000: Campo creado y heredado de la cuadricula 2000 y asignado por relación espacial.

Orden_1000: Campo creado de tipo entero y se crea con el fin de relacionarla con la cuadricula anterior. Es decir, en una cuadricula con *Orden_5000=1*, habrá varias cuadriculas de *Orden_2000* y a su vez habrá hasta cuatro cuadriculas con escala zoom 1000. A la hora de asignar el valor, el primer valor se corresponderá con la cuadricula que pertenece al mismo *Orden_2000* y tiene como coordenada del centro de la cuadricula la X mínima y la Y máxima, el siguiente valor será la siguiente cuadricula con X mínima e Y máxima y así vamos rellenando el campo *Orden_1000*.

Hoja: Campo de tipo texto por defecto de la cuadricula, corresponde con el numero de la hoja.

visitado: Campo creado de tipo entero donde 0 significa que el campo no ha sido visitado y 1 que sí se ha visitado

Cuadricula Cuadriculas Zoom_1000

Ogc_fid	Orden_5000	Orden_2000	Orden_1000	hoja	visitado
1	1	1	1	0519_4-8	0
2	1	1	2	0519_4-8	0
3	1	2	1	0519_4-8	0
4	1	2	2	0519_4-8	0
5	1	2	3	0519_4-8	0
6	1	2	4	0519_4-8	0
:	:	:	:	:	:

FIGURA 20. IMAGEN GRÁFICA DE LA TABLA DE ATRIBUTOS CUADRICULA 1000

La manera de asignar los valores a los campos del Orden_5000, Orden_2000 y Orden_1000 se establece para que a la hora de hacer ese barrido se ajuste a un barrido parecido a un vuelo fotogramétrico, de W a E y no perderse.

Y, por último, la capa *errores* tiene los siguientes campos:

Ogc_fid: Id creado automáticamente de tipo entero.

tipo: Campo de tipo texto, se corresponderá con un campo donde aparece un desplegable y se indica el tipo de error con los códigos oficiales (Anexo II) que se utilizan en el Institut Cartogràfic Valenciá

Hoja: Campo de tipo texto, campo rellenable donde hay que indicar manualmente la hoja donde se encuentra el error.

corregido: : Campo de tipo texto, se corresponderá con un campo donde aparece un desplegable y se indica “Sí” o “No”. Con la idea de cuando se reporten los errores a la empresa y vuelvan las ortofotos corregidas indicarlo en un campo y tener constancia de ello.

errores

Ogc_fid	tipo	hoja	corregido
1	"Tipo de error"	"Hoja"	SI/NO
:	:	:	:

FIGURA 21. IMAGEN GRÁFICA DE LA TABLA DE ATRIBUTOS ERRORES

5.2.3. Añadir las capas a la base de datos creada.

La manera de añadir las anteriores capas a la base de datos ha sido dentro del entorno de trabajo *QGIS*, exportar las capas al mismo archivo Spatialite con nombre *"CQ_Orto.sqlite"*. El proceso en QGIS ha sido el siguiente:

-Botón derecho a la capa que queremos añadir a la base de datos/Exportar/Guardar objetos como…

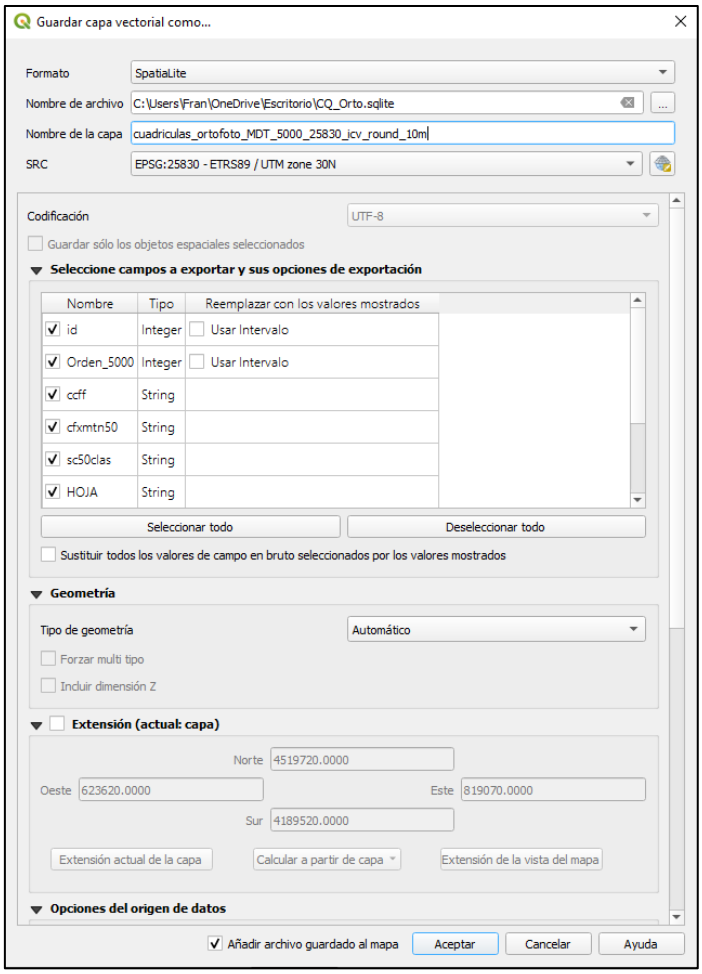


FIGURA 22. EJEMPLO EXPORTAR CUADRICULA OFICIAL 5000 EN QGIS A SPATIALITE

Y así tenemos todas las capas que necesitamos para crear el complemento. Se guarda el archivo *"CQ_Orto"* en una carpeta llamada *SQLITE*, dentro del complemento creado. Para que cuando se le ordene que cree un nuevo proyecto, coja ese archivo y haga una copia. Pero todo esto se explicará en el apartado siguiente de Desarrollo del complemento de forma detallada.

5. Desarrollo del complemento

En este apartado se va a explicar paso a paso la forma de crear el complemento y sus funciones.

6.1. Creación del plugin con "Plugin Builder"

Accionamos el complemento "Plugin Builder" para crear el complemento en cuestión. Aparece el siguiente cuadro donde tenemos que completar, nombre del plugin, nombre de la clase, una descripción corta y datos de contacto.

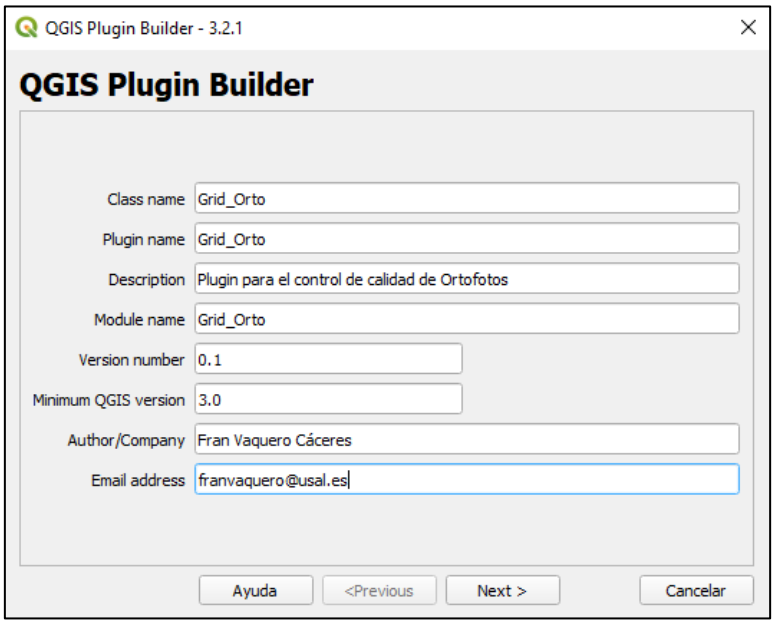


FIGURA 23. CREACIÓN DEL COMPLEMENTO CON QGIS PLUGIN BUILDER 1/8

En el siguiente dialogo, nos pide completar un resumen sobre el plugin. Esta información aparecerá cuando cualquier usuario busque el complemento a través del "buscador de complementos" dentro de QGIS.

Dado que el complemento es para un trabajo específico, hay que indicarlo correctamente en este mensaje para que quede claro.

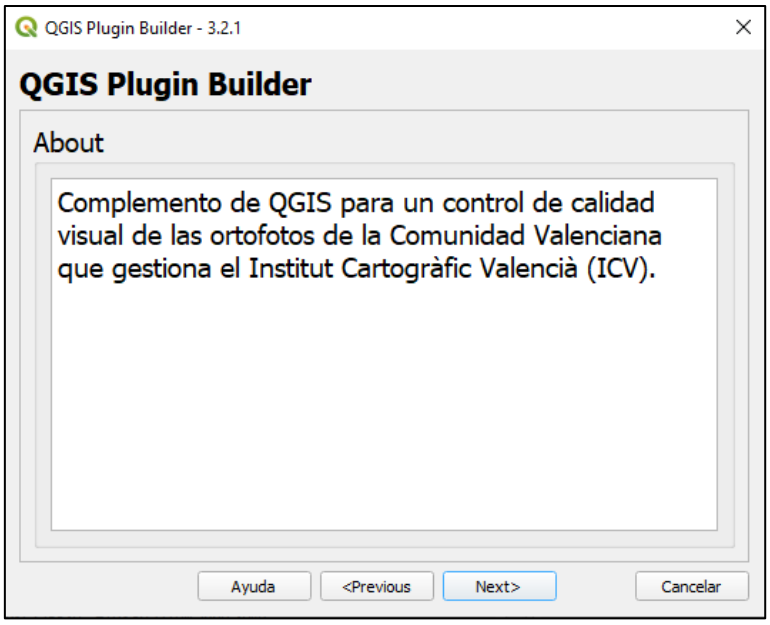


FIGURA 24. CREACIÓN DEL COMPLEMENTO CON QGIS PLUGIN BUILDER 2/8

En el siguiente dialogo nos pide que tipo de plantilla de plugin queremos, he indicado en el campo "Template" *Tool button with dock widget* para que el plugin se quede en un panel y selecciono izquierda como zona para anclarlo. Cómo menú elijo Plugins.

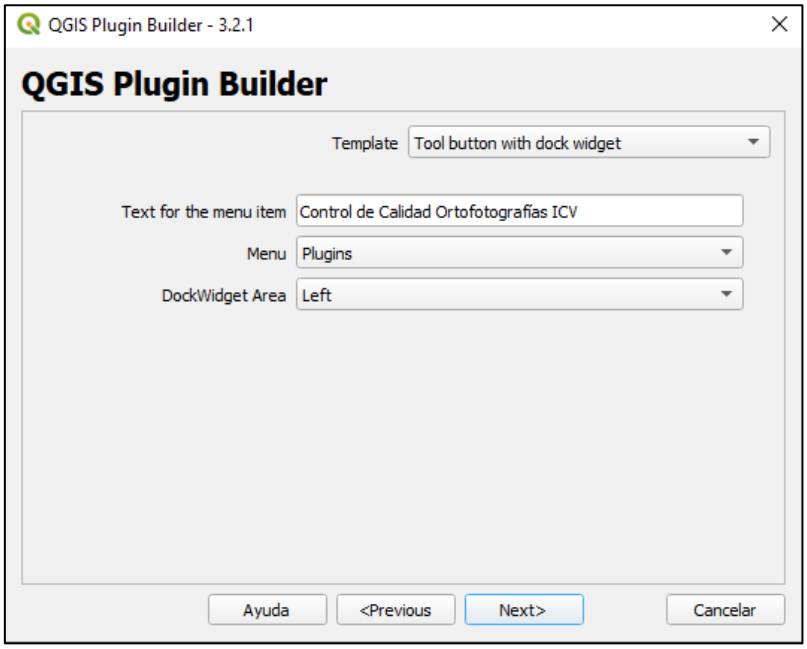


FIGURA 25. CREACIÓN DEL COMPLEMENTO CON QGIS PLUGIN BUILDER 3/8

En el siguiente dialogo, dejo marcadas todas las opciones:

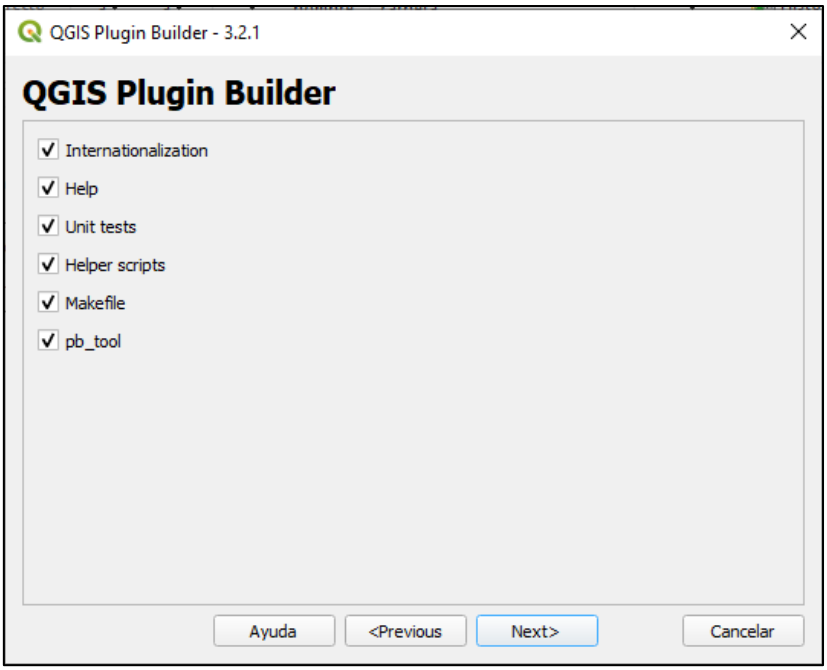


FIGURA 26. CREACIÓN DEL COMPLEMENTO CON QGIS PLUGIN BUILDER 4/8

En el siguiente nos pide la dirección del repositorio del plugin, indico mi repositorio de Git-Hub creado que en el apartado siguiente de este informe indico como lo creo. También indico la página oficial del ICV junto a unas etiquetas. Esta información servirá a los usuarios para encontrar el plugin deseado.

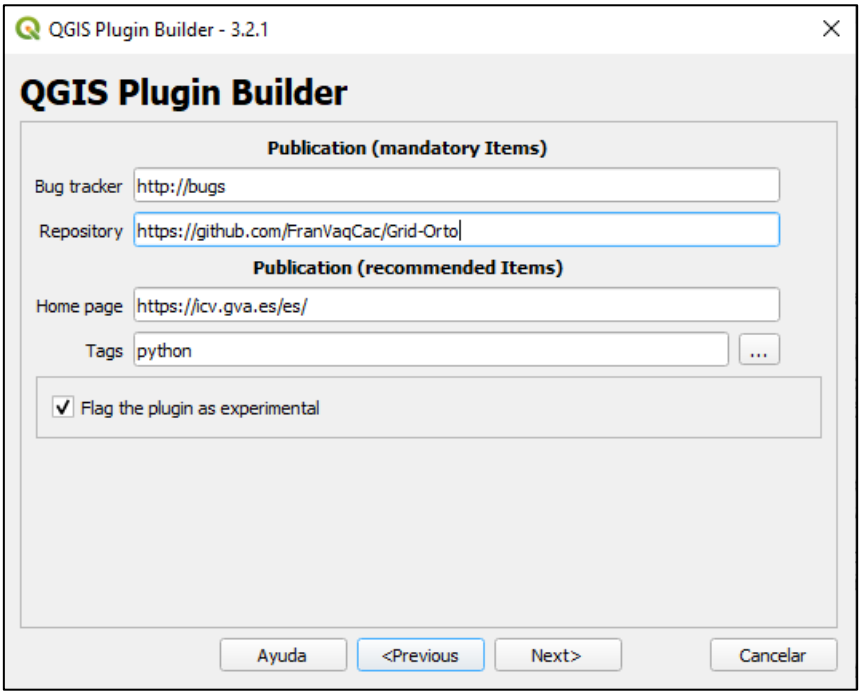


FIGURA 27. CREACIÓN DEL COMPLEMENTO CON QGIS PLUGIN BUILDER 5/8

Finalmente, por defecto nos indica donde va a almacenar el plugin creado que es donde se almacenan todos los plugin instalados. A esa misma dirección podemos acceder a través de QGIS, dándole a la pestaña de Configuración/ Perfiles de Usuario/Abrir la carpeta del perfil activo.

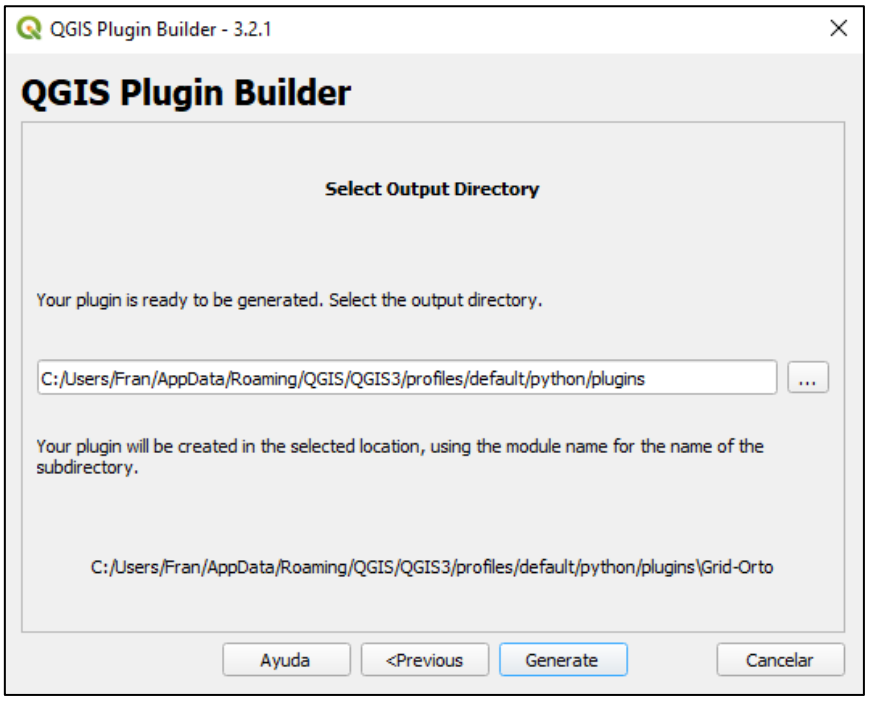


FIGURA 28. CREACIÓN DEL COMPLEMENTO CON QGIS PLUGIN BUILDER 6/8

Nos crea el Plugin, si no encuentra el compilador de recursos en el PATH, nos indica que tenemos que compilarlo manualmente.

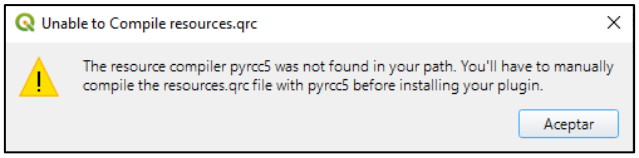


FIGURA 29. CREACIÓN DEL COMPLEMENTO CON QGIS PLUGIN BUILDER 7/8

Después se nos presenta la siguiente ventana en la que nos indica los pasos a seguir para la compilación, nos indica como cambiar el icono de invocación que aparecerá en la barra de herramienta, nos informa del fichero que contiene la interfaz de usuario y donde podemos consultar ayuda para el desarrollo de complementos.(Moya, A, 2020)

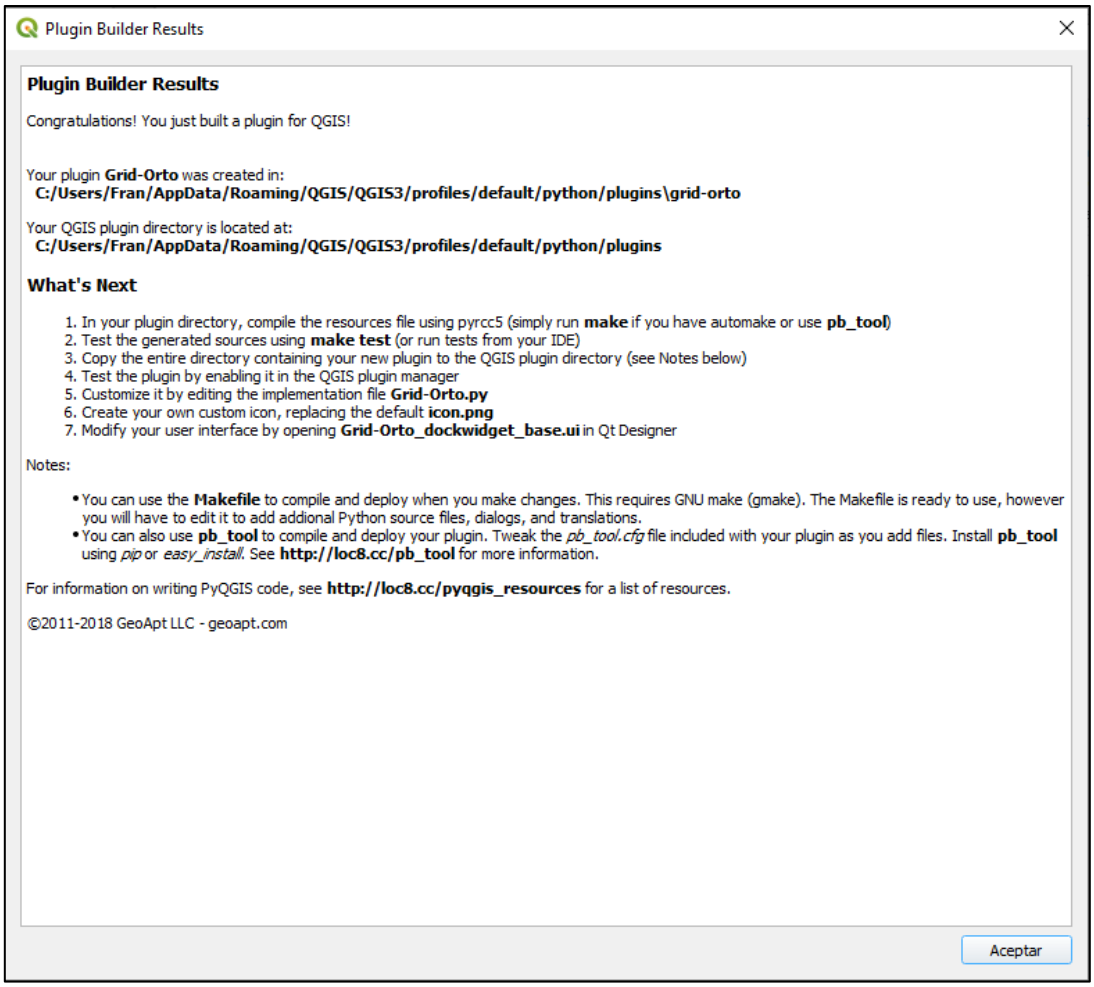


FIGURA 30. CREACIÓN DEL COMPLEMENTO CON QGIS PLUGIN BUILDER 8/8

Nos ha creado una carpeta con el nombre del módulo dentro de la ruta indicada, como hemos seleccionado la ruta por defecto para los complementos en el perfil actual, vamos a ver los archivos y carpetas creadas:

-*Grid_Orto_dockwidget.py* .- Código fuente del panel, en el será donde insertemos todas las líneas de código ya que es en ese panel en el que añadiremos todos los controles de la interfaz.

-*Grid_Orto_dockwidget_base.ui* .- Fichero de configuración de la interfaz, en este caso es el fichero que editaremos con el programa "Qt Designer for QGIS"

-*_init_.py* .- Fichero en el que se define el objeto "*iface*" de la clase *Qgsinterface.*

-*Grid_Orto.py* .- Constructor principal del complemento, en este fichero estarían definidas todas las ventanas que se utilizan, en este caso, sólo tenemos un panel principal.

-*metadata.txt* .- Textos de la información que ofrece el complemento en el gestor de complementos de QGIS e información acerca del mismo.

-*resources.qrc* .- Fichero que contiene referencias a otros recursos utilizados, como, por ejemplo, el fichero de icono en formato .png .

6.2. Creación de la interfaz del Plugin

Para la creación de la interfaz del complemento usamos un módulo que viene integrado con la instalación de QGIS, el programa *Qt Designer with QGIS 3.10.14 custom widgets.* En él abrimos el archivo *Grid_Orto_dockwidget_base.ui,* donde va almacenar toda la interfaz del plugin arrastrando sobre el panel todas las herramientas empleadas y son las siguientes:

- *QPushButton*: Botones para dar funcionalidad al flujo de trabajo . Por ejemplo, crear nuevo proyecto, abrir capas, activar zoom… etc.
- *QLabel*: Sirven para añadir textos informativos e imágenes.
- *QGroupBox:* Delimitador para meter dentro varios botones con igual aplicación. Por ejemplo, los botones que actúan para hacer zoom.
- *ToolBox*: Para crear diferentes pestañas, diferenciar y ordenar el plugin. En este caso existen 4 pestañas: Proyecto, CQ Orto, Exportar y About.
- *QTextBrowser*. Widget típico para meter información de contacto. En el caso del complemento se ha añadido información de contacto profesional para que los usuarios puedan contactar con el desarrollador si es necesario.

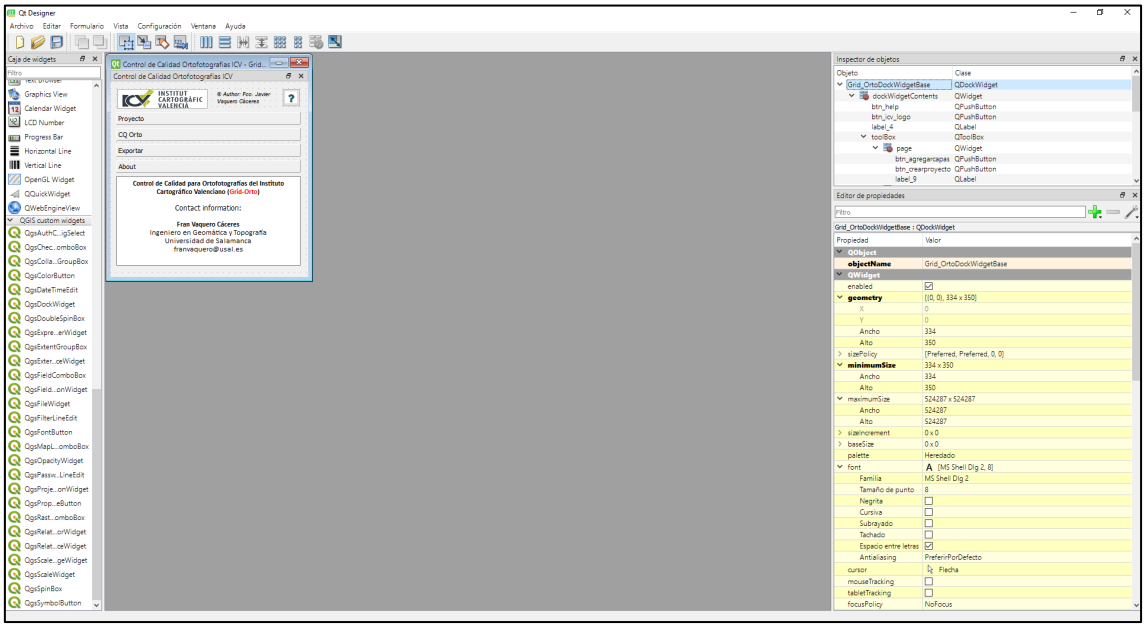


FIGURA 31. INTERFAZ DEL SOFTWARE QT DESIGNER

Para cada botón del complemento se ha puesto un nombre sencillo de recordar para cuando se llamen en el código se sepa de manera sencilla cual es. Ese se cambia dentro de la columna de información de la derecha en el apartado de "*ObjectName*".

Las imágenes de los botones como el logo del Instituto Cartográfico Valenciano y el botón de ayuda se cargan en el apartado de *QAbstractButton/icon* y también en el código fuente hacemos referencia a ellos. El tamaño que ocupa el plugin se edita dentro de *QWidget/ geometry/minimumSize,* los valores de alto y ancho. El componente gráfico del complemento no es demasiado grande porque se ha dado prioridad a que no ocupe

mucho en la pantalla porque el objetivo principal es el control visual de ortofotos y cuanto menos ocupe mayor visibilidad tendrá el usuario.

El archivo .ui *("user interface")* tiene una estructura de tipo XML. Donde va almacenando todos los parámetros que en el programa Qt Designer introducimos:

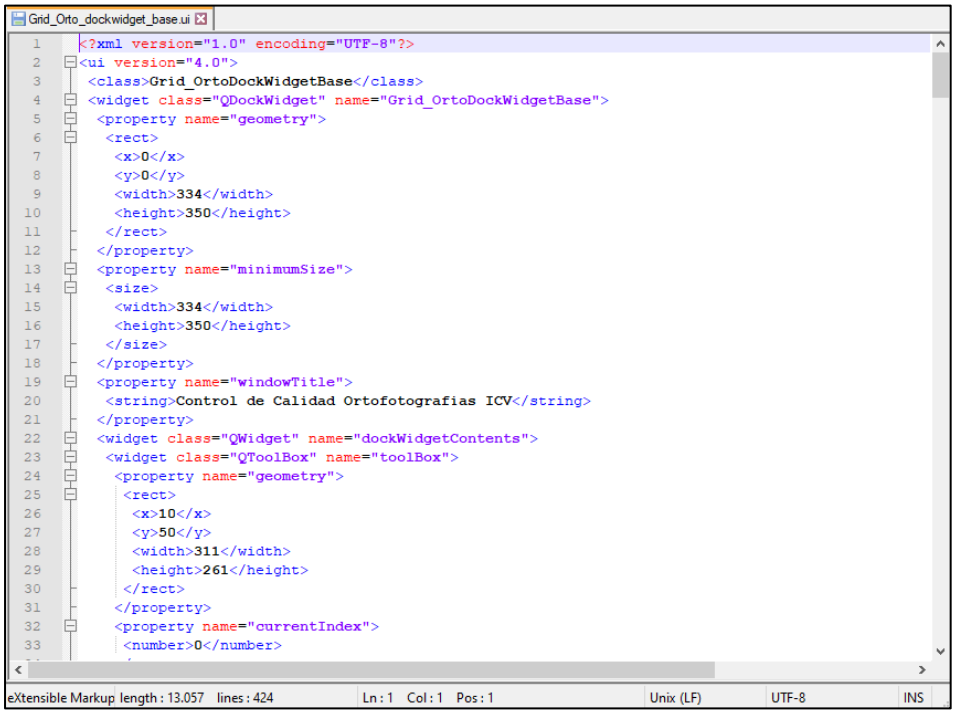


FIGURA 32. ARCHIVO .UI DEL COMPLEMENTO

Lista la interfaz gráfica levantamos *QGIS* y si cargamos el complemento a través del menú, aparece un botón con el icono que le hemos indicado anteriormente.

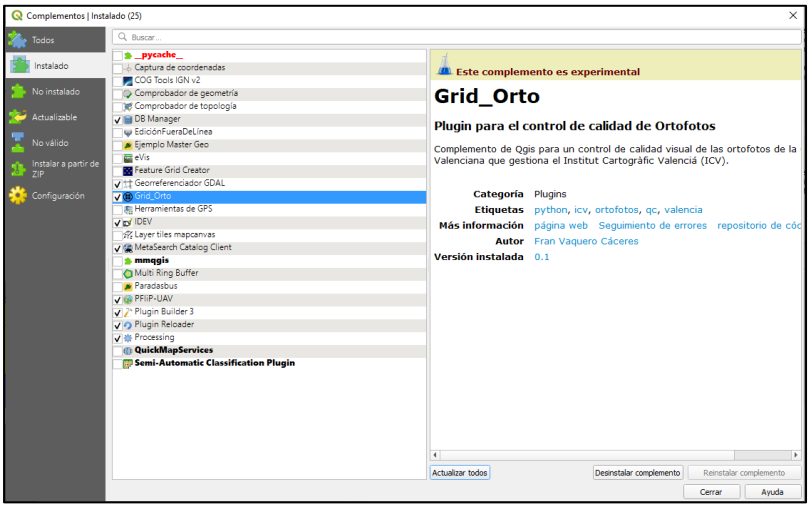


FIGURA 33. CARGA DEL COMPLEMENTO CON EL ADMINISTRADOR DE COMPLEMENTOS DE QGIS

Además del botón de invocación, también se crea una entrada en el menú "*Complementos*" / "*Grid_Orto*" / "*Grid_Orto*" , este sería su aspecto visual:

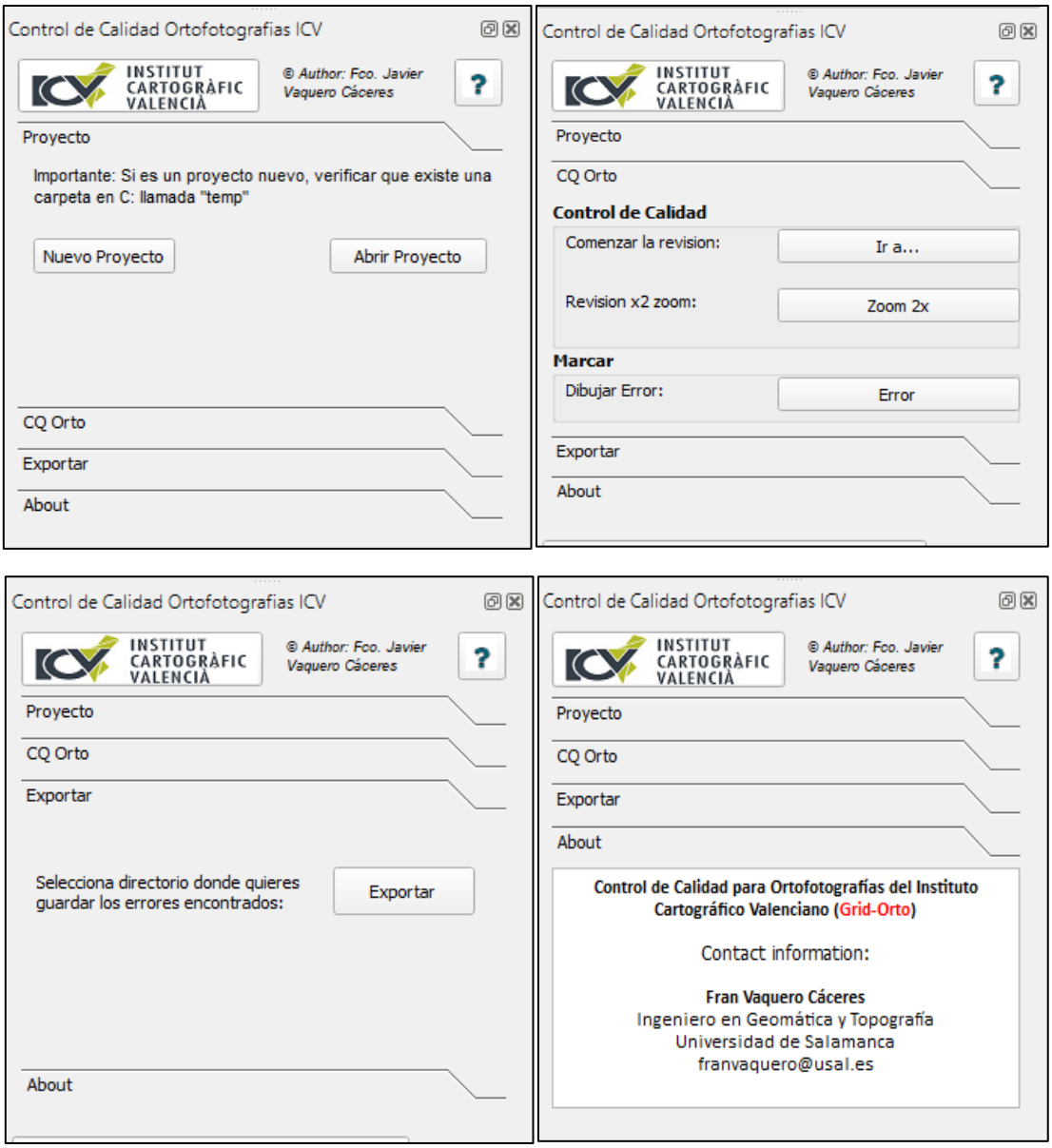


FIGURA 34. PANEL DEL COMPLEMENTO DENTRO DE QGIS

6.3. Código del complemento

Para la programación del complemento se ha utilizado el software *PyCharm* 2020.1. *PyCharm* es un IDE, es decir, no solo es un editor de código, sino que también tiene un depurador, un intérprete y otras herramientas que nos ayudarán a crear y exportar los programas que creemos. *PyCharm* tiene un intérprete en el editor de código que nos ayudará a saber o conocer los posibles errores del código en tiempo real, algo que ha hecho que *Python* y *PyCharm* sean elegidos por muchos usuarios que comienzan a programar.

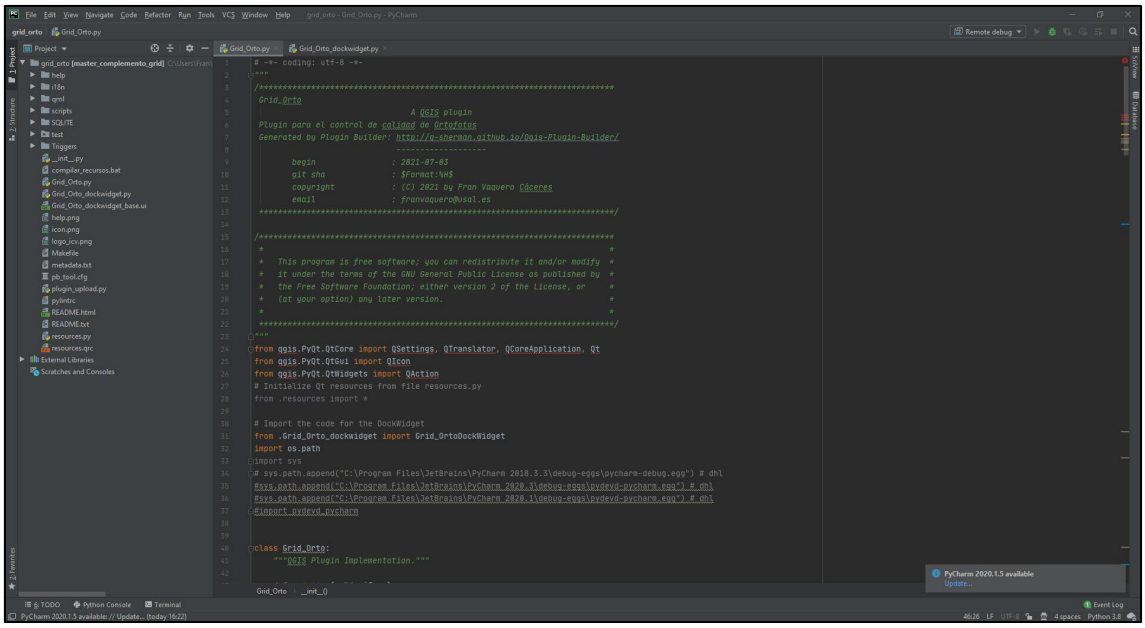


FIGURA 35. INTERFAZ SOFTWARE PYCHARM 2020.1.

El *Plugin Builder* como se ha comentado antes, crea el plugin vacío y listo para editar y trabajar con él. Nos crea todos los archivos que necesitamos para que el complemento funcione correctamente en el software *QGIS*. El fichero *Grid_Orto.py* es el fichero principal en cuanto a configuración, en él se crea la clase principal de la que va a depender todos los demás objetos.

Crea la *clase Grid_Orto*, y dentro de la clase se definen los objetos. Como podemos ver en el código siguiente crea el objeto *iface* que es una instancia a las funcionalidades de la interfaz de *QGIS*, con esto se consigue que acceder a los componentes gráficos del entorno de trabajo (carga de capas, control de la visualización, consulta a objetos, …):

```
    class Grid_Orto:
"""QGIS Plugin Implementation."""

def __init__(self, iface):
    """Constructor.

    :param iface: An interface instance that will be passed to this class
        which provides the hook by which you can manipulate the QGIS
        application at run time.
```

```
        :type iface: QgsInterface
        """
```

Todo el código que se ha creado ha sido en el fichero de *Grid_Orto_dockwidget.py* que es el fichero donde recoge todas las funcionalidades que tiene y donde contiene todos los widgets metidos en el archivo *Grid_Orto_dockwidget_base.ui.*

6.3.1 Importación de librerías y configuración inicial

La configuración inicial del plugin se basa en la importación de librerías que necesitas y usa el complemento y las posibles variables globales vamos a necesitar llamar durante el funcionamiento del programa:

```
import os
import webbrowser
from qgis.core import *
from qgis.utils import *
from qgis.PyQt import QtGui, QtWidgets, uic
from PyQt5.QtCore import *
from PyQt5.QtWidgets import QFileDialog, QMessageBox, QInputDialog
```

Las librerías que se usan son: *qgis.core, qgis.utils, qgis.PyQt, PyQt5.QtCore y PyQt5.QtWidgets*. Son módulos de *PyQt5* y de la API de *QGIS* que se llaman con *QGIS.* Se invoca los módulos *os* y *webbrowser* que son componentes básicos de *Python.*

El módulo *os* de *Python* permite realizar operaciones dependientes del Sistema Operativo como crear una carpeta, listar contenidos de una carpeta, conocer acerca de un proceso, finalizar un proceso, etc. Lo utilizamos para saber direcciones de directorios y archivos.

El módulo *webbrowser* invoca una interfaz que permite abrir documentos de la web. Bajo la mayoría de circunstancias, simplemente invocando la función *open()* desde este módulo se realizará la acción correcta. En nuestro caso la utilizaremos solo una vez para invocar la página web del Instituto Cartográfico Valenciano.

Seguidamente creamos las variables *pluginPath*, porque nos hace falta tener la dirección del plugin en una variable ya que se invoca a carpetas dentro de él para crear un nuevo proyecto.

```
pluginsPath = QFileInfo(QgsApplication.qgisUserDatabaseFilePath()).path()
pluginPath = os.path.dirname(os.path.realpath(__file__))
pluginPath = os.path.join(pluginsPath, pluginPath)
```

También creamos las variables de posición que nos van a ayudar a crear el barrido de la ortofoto para el control visual, más adelante se explicará más detallado.

```
posicion = 1
posicion2000 = 1
posicion1000 = 1
```

Y, por último, carga la configuración del fichero de la interfaz gráfica *Grid_Orto_dockwidget_base.ui*

```
FORM_CLASS, _ = uic.loadUiType(os.path.join(
    os.path.dirname(__file__), 'Grid_Orto_dockwidget_base.ui'))
```

6.3.2 Definición clase principal y funciones

Se implementa la clase principal llamada *Grid_OrtoDockWidget*, de ella va a nacer todas las funcionalidades del complemento y cargará todos los widgets utilizados y enlazados con el código creado. Es decir, para cada widget le corresponderá alguna función escrita bajo código.

```
class Grid_OrtoDockWidget(QtWidgets.QDockWidget, FORM_CLASS):

    closingPlugin = pyqtSignal()
```

Seguido se define la primera función *"def _init_"* es el llamado "constructor" del plugin, es una función clave que siempre se va a ejecutar cuando se haga instancia en el entorno de trabajo *QGIS*. Dentro de él podemos ver algunas funciones de configuración. El *setupUi* y el *SetupPlugin* sirven para la configuración del plugin, se carga *iface* para tener el control de las funcionalidades de *QGIS y el windowTitle* se corresponde con el título que aparece al pasar con el ratón por encima del icono dentro de *QGIS* en este caso, *"Grid_Orto"*.

```
def __init__(self,
             iface,
             parent=None):
    """Constructor."""
    super(Grid_OrtoDockWidget, self).__init__(parent)
    # Set up the user interface from Designer.
    # After setupUI you can access any designer object by doing
    # self.<objectname>, and you can use autoconnect slots - see
    # http://doc.qt.io/qt-5/designer-using-a-ui-file.html
    # #widgets-and-dialogs-with-auto-connect
    self.setupUi(self)
    self.setupPlugin()  # Appearance initialization
    self.iface = iface
    self.windowTitle = "Grid_Orto"
    self.getSpatialiteConnections()
```

Por último, *getSpatialiteConnections()* sirve para gestionar la conexión con la base de datos que vamos a trabajar. Más adelante se utilizará para cuando se pulse en crear un proyecto nuevo, automáticamente se añada en QGIS una conexión a la base de datos Spatialite (con todas las capas dentro) creada.

En *QGIS* si vamos a *Capa/Administrador de fuentes de datos/Spatialite* podemos acceder a las conexiones que tiene actualmente el entorno, ahí aparecerá en una pestaña, la dirección y a que archivo está conectada.

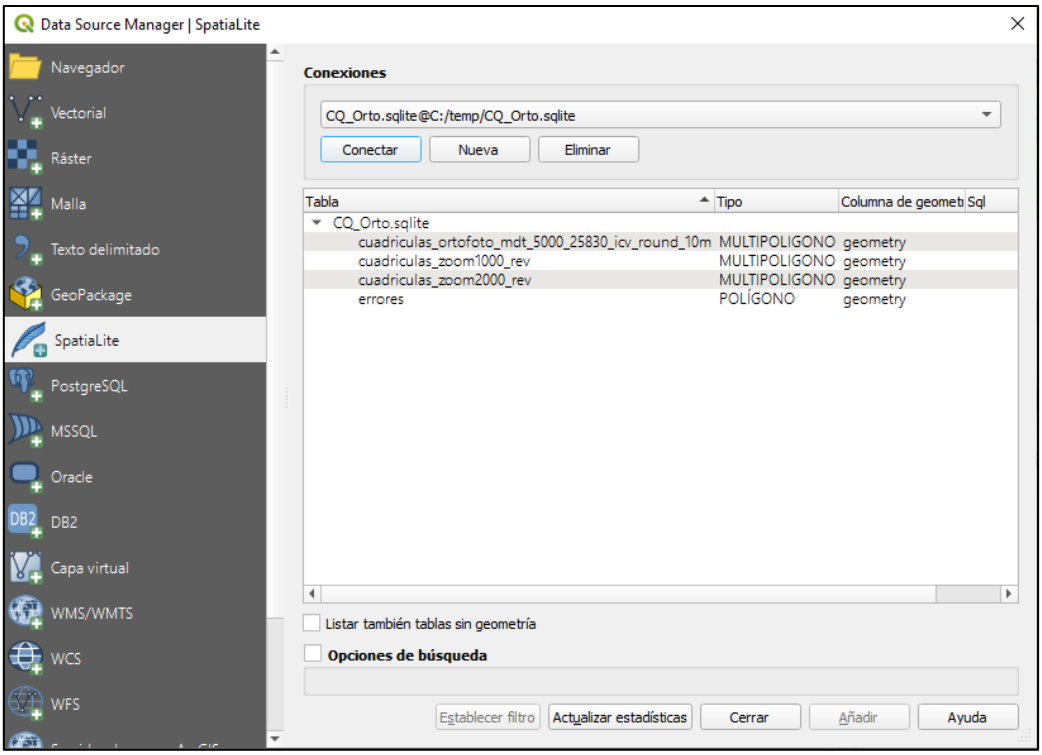


FIGURA 36. VENTANA DE ADMINISTRADOR DE FUENTES DE DATOS DE QGIS CON CONEXIÓN A LA BASE DE DATOS SPATIALITE

Seguimos con el código y en esta función constructora es donde se vinculan los widgets con las funciones creadas. La manera de integrar las funciones que se quería conseguir era con botones y son los siguientes:

```
""" Dashboard widget event connectors """
self.btn_crearproyecto.clicked.connect(self.crear_proyecto)
self.btn_agregarcapas.clicked.connect(self.capas)
self.btn_zoom2000.clicked.connect(self.zoom2000)
self.btn_zoom1000.clicked.connect(self.zoom1000)
self.btn_error.clicked.connect(self.error)
self.btn_help.clicked.connect(self.openHelp)
self.btn_aceptar.clicked.connect(self.aceptar)
self.btn_icv_logo.clicked.connect(self.icvweb)
self.btn_ayudaerror.clicked.connect(self.helperror)
```

Vamos a ver a continuación, paso a paso y en el orden del flujo de trabajo que hay que seguir, las funciones para cada botón. En las siguientes figuras vamos a identificar cada botón con el nombre del plugin.

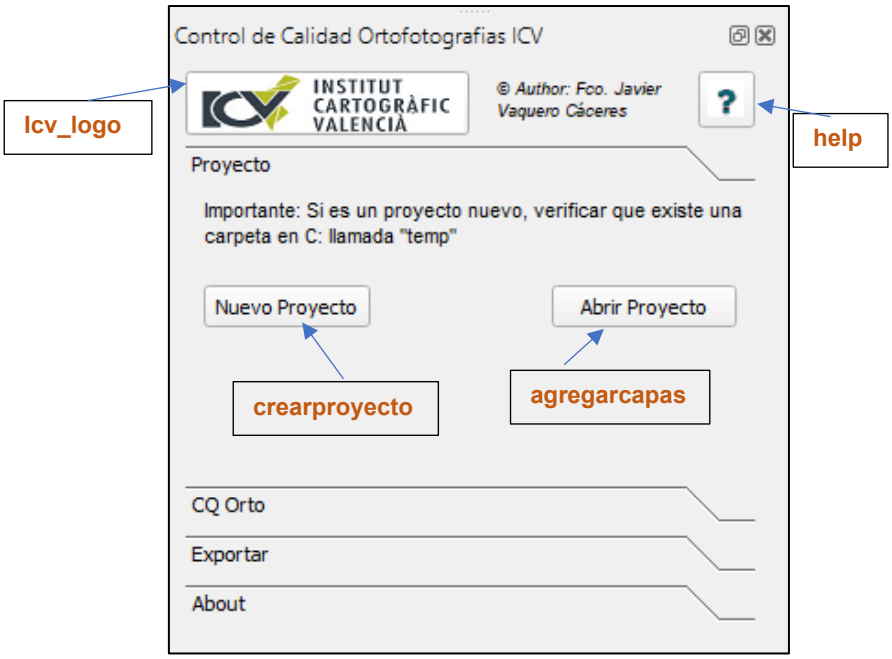


FIGURA 37. RELACIÓN DE BOTONES DE CÓDIGO CON BOTONES DEL COMPLEMENTO. PESTAÑA PROYECTO

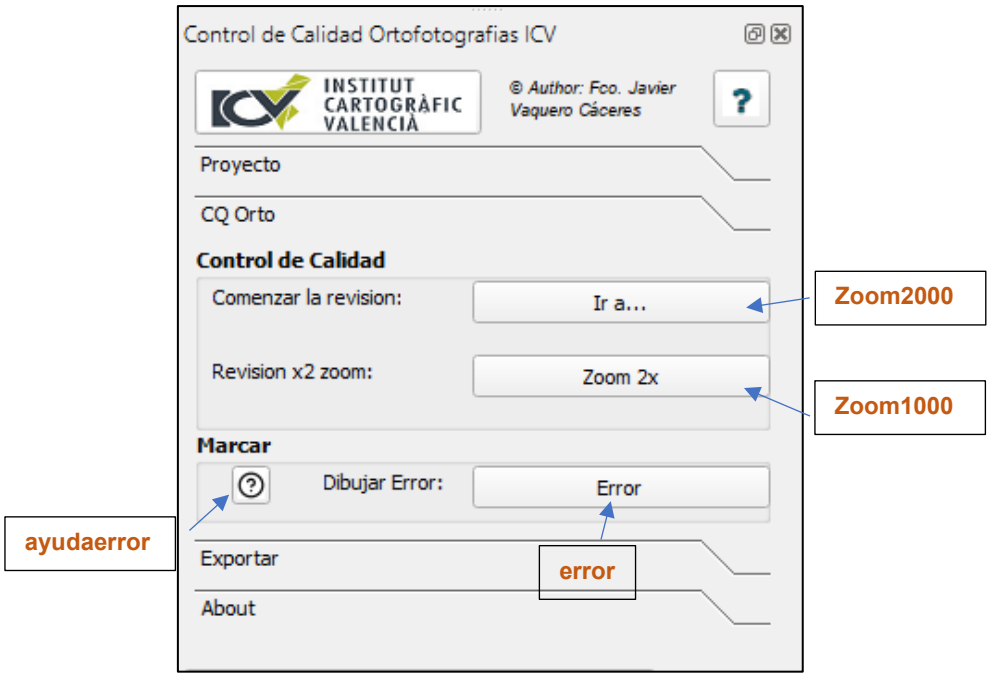


FIGURA 38. RELACIÓN DE BOTONES DE CÓDIGO CON BOTONES DEL COMPLEMENTO. PESTAÑA CQ ORTO

FIGURA 39. RELACIÓN DE BOTONES DE CÓDIGO CON BOTONES DEL COMPLEMENTO. PESTAÑA EXPORTAR

Antes de meternos en código, explicamos las cuatro funciones primeras después del constructor que tienes que ver con la interfaz y con elementos externos al complemento.

```
def setupPlugin(self):
    baseDirectory = os.path.dirname(os.path.realpath(__file__))  # Plugin absolute path
    self.btn_icv_logo.setIcon(QtGui.QIcon(os.path.dirname(os.path.realpath(__file__)) +
'/logo_icv.png'))
    self.btn_help.setIcon(QtGui.QIcon(os.path.dirname(os.path.realpath(__file__)) +
'/help.png'))
    self.btn_ayudaerror.setIcon(QtGui.QIcon(os.path.dirname(os.path.realpath(__file__)) +
'/errores.png'))
```

SetupPlugin sirve para añadir los iconos a los tres botones que tienen icono. Con la función *"setIcon"* después de hacer instancia al propio botón (self."nombre del boton") se añade un comando de la librería *"Qtgui"* donde indicamos el directorio donde se encuentra el archivo *.png* que le queremos añadir junto al nombre y extensión del archivo. Y así quedan definidos los botones *icv_logo* con el logo oficial del Instituto Cartográfico Valenciano, el botón *help* con un logo de un interrogante y el botón *ayudaerror* con un logo de otro interrogante diferente.

La siguiente función pertenece a una llamada web para que al accionar el botón nos lleve a la página web indicada. En nuestro caso a la página web oficial del Instituto Cartográfico Valenciano. Se consigue con el comando *webbrowser.open* definida al inicio del archivo.

```
def icvweb(self):
    webbrowser.open('http://www.icv.gva.es/es/inicio')
```

Las dos siguientes funciones se corresponden a una llamada de un archivo *.pdf* en este caso, que al pulsar nos abre un documento guardado dentro del propio plugin en una carpeta llamada "help".

El botón *help* llama a un archivo llamado *Flujo_Trabajo.pdf* donde explica el flujo de trabajo que hay que seguir y el botón *helperror* llama a un archivo *Ayuda_tipos_errores.pdf* donde nos muestra los tipos de errores que pueden aparecer en una ortofoto.

```
def openHelp(self):
    pathInfoDoc = os.path.dirname(__file__) + '/help/Flujo_Trabajo.pdf'
    os.startfile(pathInfoDoc)

def helperror(self):
    pathInfoDoc = os.path.dirname(__file__) + '/help/Ayuda_tipos_errores.pdf'
    os.startfile(pathInfoDoc)
```

Con la librería *os* todo esto es capaz, en el código anterior se hace instancia a un comando *startfile*(dirección y nombre del documento) para abrir con el lector de archivos de *pdf* por defecto que se tenga en el ordenador. En los **Anexos I** y **Anexo II** podemos consultar estos documentos.

"*os.path.dirname(__file__)"* busca directamente la dirección de Windows donde se encuentra el plugin instalado. Esta dirección es la misma de cuando accedemos en *QGIS/Configuración/Perfiles de usuario/Abrir la carpeta del perfil activo/Python/Plugins/Grid_Orto*

6.3.3. Creación de proyecto y de conexión Spatialite en *QGIS*

Al trabajar con bases de datos se necesita hacer una instancia en QGIS y crear una conexión *sqlite*. Para ello se define una función *getSpatialiteConnections*, donde con la clase *Qsettings* nos proporciona configuraciones independientes de la plataforma QGIS. Por tanto, con esta clase le decimos que en el administrador de fuente de datos Spatialite/Conexiones si se detecta una nueva conexión "path" que lo almacene y lo guarde con el nombre *connectionName.*

```
def getSpatialiteConnections(self):
    self.connections = {}
    settings = QSettings()
    settings.beginGroup('/SpatiaLite/connections')
    list_str_keys = settings.allKeys()
    paths = []
    for key in list_str_keys:
        if key != 'selected':
            paths.append(settings.value(key))
    connectionNames = settings.childGroups()
    cont = 0
    for connectionName in connectionNames:
        path = paths[cont]
        self.connections[connectionName] = paths[cont]
        cont = cont + 1
```

Pero la clase anterior no funciona sin la siguiente función que pertenece al botón de "Crear Proyecto".

```
def crear_proyecto(self, event):
    ruta_sqlite = 'SQLITE/CQ_Orto.sqlite'
    plugin_dir = pluginPath
    ruta_guardar_sqlite = 'C:/temp/CQ_Orto.sqlite'
    path_TemplateDb = os.path.normcase(os.path.join(plugin_dir, ruta_sqlite))

    # db_template.open()
    if QFile.exists(ruta_guardar_sqlite):
        if not QFile.remove(ruta_guardar_sqlite):
            strError = "Existe el archivo en:\n" + ruta_guardar_sqlite
            msgBox = QMessageBox(self)
            msgBox.setIcon(QMessageBox.Information)
            msgBox.setWindowTitle(self.windowTitle)
            msgBox.setText("Error:\n" + strError)
            msgBox.exec_()
            return
    QFile.copy(path_TemplateDb, ruta_guardar_sqlite)

    if not QFile.exists(ruta_guardar_sqlite):
        strError = "No existe el archivo en:\n" + ruta_guardar_sqlite
        msgBox = QMessageBox(self)
        msgBox.setIcon(QMessageBox.Information)
        msgBox.setWindowTitle(self.windowTitle)
        msgBox.setText("Error:\n" + strError)
        msgBox.exec_()
        return
```

Cuando se acciona el botón, se crea una copia del modelo de datos en la dirección *ruta guardar sqlite*. En esta versión del plugin dejamos una dirección fija *"C:/temp"*, en las próximas versiones es donde hay que trabajar para permitirle al usuario dónde guardar ese modelo de datos. Esa copia se hace gracias al comando *QFile.copy* ("ruta donde está el archivo a copiar", "ruta donde quieres que lo guarde").

Se establece una condicional para informarle al usuario, si el directorio existe (*if QFile.exists(ruta_guardar_sqlite):)* y no la base de datos se hace la copia. En el caso que ya exista el archivo *sqlite* creado se informará al usuario diciendo que existe ese archivo generado.

Si no existe el directorio donde se va a guardar lanza un mensaje de error *("No existe el archivo en:\n" + ruta guardar sqlite).*

```
    connectionName = QFileInfo(ruta_guardar_sqlite).fileName()
    con = [connectionName, ruta_guardar_sqlite]
    QSettings().setValue("SpatiaLite/connections/%s/sqlitepath" % (con[0]), con[1])
    self.iface.reloadConnections()
    self.getSpatialiteConnections()
    msgBox = QMessageBox(self)
    msgBox.setIcon(QMessageBox.Information)
    msgBox.setWindowTitle(self.windowTitle)
    msgBox.setText("Process completed successfully")
    msgBox.exec_()
```

Llegamos al paso que se genera esa copia, se le informa al usuario con el anterior mensaje : *msgBox.setText("Process completed successfully").* En este momento es cuando con *Qfileinfo* coge el directorio donde está la copia y el fichero copiado (en nuestro caso se generará un archivo en *C:/temp/ CQ_Orto.sqlite*), recarga el repositorio de conexiones y llama a la función *getSpatialiteConnections* para guardar la conexión actual.

Tenemos el modelo de datos copiado de la base de datos plantilla y con una conexión Spatialite, solo queda abrir esas capas que se encuentran dentro. Para eso se crea la función *"capas",* enlazada con el botón *"Abrir Proyecto".*

```
def capas(self, event):
    layer = iface.addVectorLayer("/temp/CQ_Orto.sqlite", "Capas para la revision", "ogr")
    if not layer or not layer.isValid():
        print("Layer failed to load!")
```

Con el comando *iface.addVectorLayer* se añade el archivo *sqlite*, se añade un nombre al grupo de capas y la librería a la que pertenece *"ogr".*
Al cargar las capas, se añaden estilos creados anteriormente para cada capa.

```
# Asignación de estilos a las cuadriculas
ruta_qml5000 = 'qml/cuadriculas_ortofoto_mdt_5000_25830_icv_round_10m.qml'
ruta_qml2000 = 'qml/cuadriculas_zoom2000_rev.qml'
ruta_qml1000 = 'qml/cuadriculas_zoom1000_rev.qml'
ruta_qmlerrores = 'qml/errores.qml'

plugin_dir = pluginPath
qml_path5000 = os.path.normcase(os.path.join(plugin_dir, ruta_qml5000))
qml_path2000 = os.path.normcase(os.path.join(plugin_dir, ruta_qml2000))
qml_path1000 = os.path.normcase(os.path.join(plugin_dir, ruta_qml1000))
qml_patherrores = os.path.normcase(os.path.join(plugin_dir, ruta_qmlerrores))
```

Se definen las direcciones de cada archivo qml, los archivos se encuentran en la carpeta dentro del plugin llamada "qml".

```
for layer in QgsProject.instance().mapLayersByName("CQ_Orto
cuadriculas_ortofoto_mdt_5000_25830_icv_round_10m"):
    layer.loadNamedStyle(qml_path5000)
```

Por ejemplo, para la capa *cuadriculas_ortofoto_mdt_5000_25830_icv_round_10m* y el comando *loadNamedStyle* se le asigna el fichero .qml (archivo de estilos de *QGIS*) "qml_path5000". En el apartado "Creación de estilos de capas" que se encuentra más adelante se detalla cómo se han creado esos estilos.

También se han insertado "labels" con el numero de la hoja para que en todo momento el usuario sepa en que hoja 5000 se encuentra. Se han insertado en las dos capas creadas: "cuadriculas_zoom1000_rev" y "cuadriculas_zoom2000_rev". Vamos a ver como se añaden solamente en una capa (En el anexo II se encuentra el código completo).

```
        # Labels
    layer_settings = QgsPalLayerSettings()
    text_format = QgsTextFormat()
    text_format.setSize(8)

    layer_settings.fieldName = "hoja"
    layer_settings.enabled = True
```

```
layer_settings = QgsVectorLayerSimpleLabeling(layer_settings)
layer.setLabelsEnabled(True)
layer.setLabeling(layer_settings)
layer.triggerRepaint()
```

Se accede con el comando *QgsPalLayerSettings* a las propiedades de la capa y con *fieldName* se indica que campo quieres contemplar para poner las etiquetas. Para cambiar el tamaño de la fuente se accede con el comando *QgsTextFormat()* y con *setSize* se indica con un numero el tamaño. Este procedimiento se repite para la otra capa.

6.3.4. Creación de estilos de capas

Como se ha comentado en el apartado anterior se han tenido que crear archivos *.qml* que son archivos de estilos de *QGIS* para asignarlos en código y que al cargar esa base de datos las capas que la integran aparezcan con los estilos.

Para las tres capas de cuadriculas se crearon los estilos de forma categorizada por el campo "visitado", con la idea de cuando se vaya haciendo el control, una vez visitada una cuadricula, el campo "visitado" cambia de 0 a 1 y cambiara de color. Con esto nos podemos alejar y ver de forma global lo que hemos revisado de forma muy sencilla.

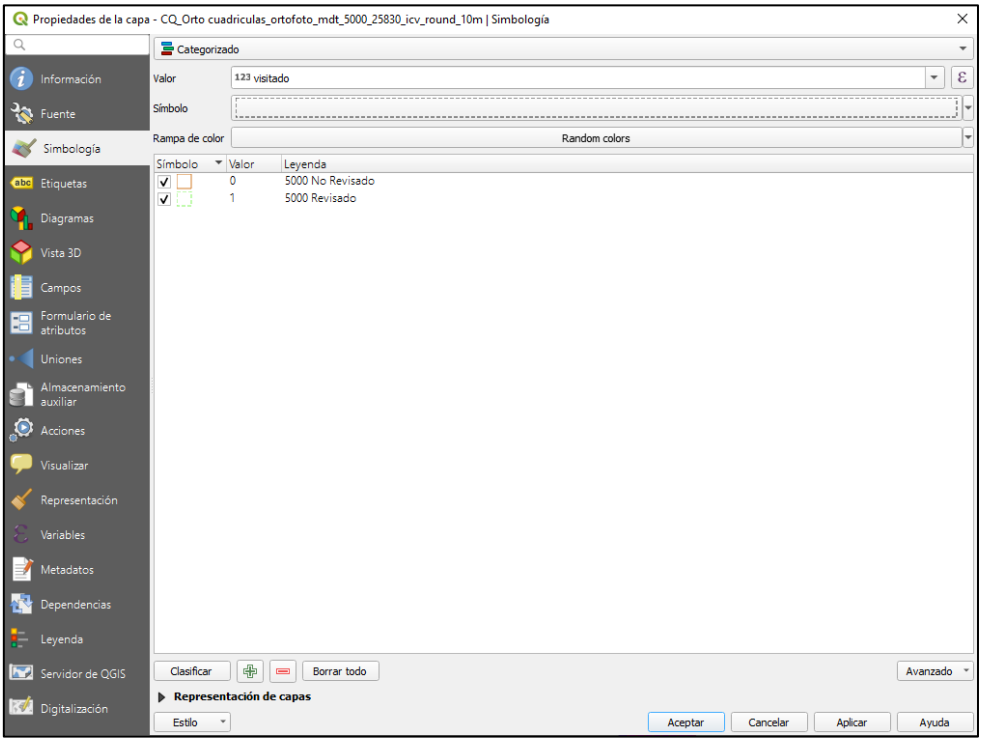


FIGURA 40. ESTILO CAPA CUADRÍCULAS_ORTOFOTO_MDT_5000

Una vez definitivo, se guarda desde la pestaña *Estilo/Guardar Estilo/Guardar* archivo de estilos QML de QGIS y se añade el directorio donde se quiere guardar y el nombre. Los archivos *qml*, los he llamado igual que el nombre de cada capa y el directorio dentro del plugin en una carpeta llamada "qml".

A continuación, muestro imágenes de los demás estilos:

Para la capa cuadriculas_zoom2000_rev:

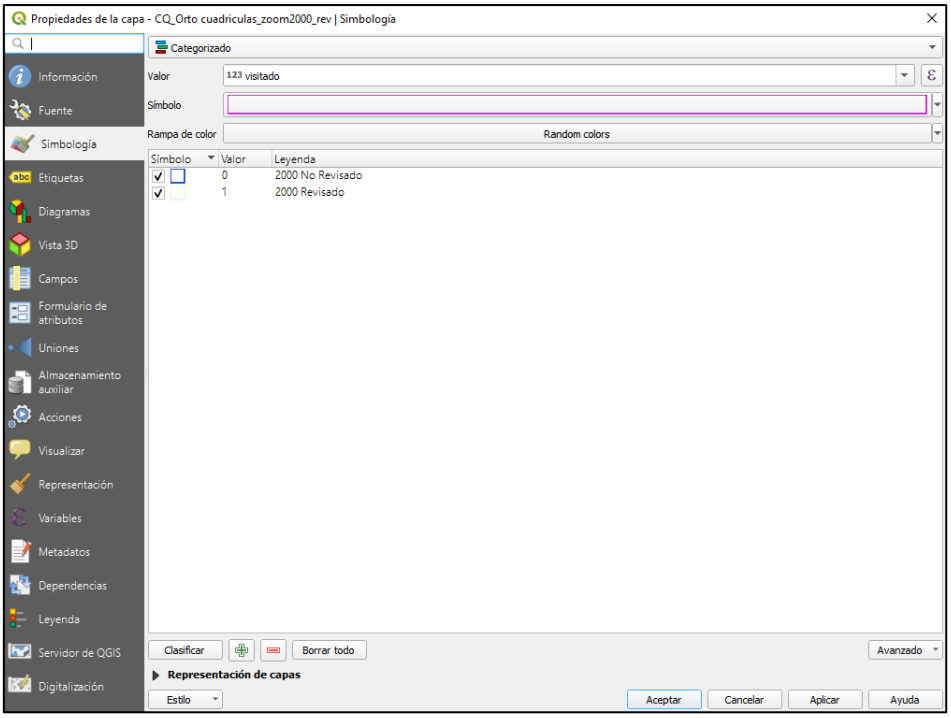


FIGURA 41. ESTILO CAPA CUADRÍCULAS_ZOOM2000_REV

Para la capa cuadriculas_zoom1000_rev:

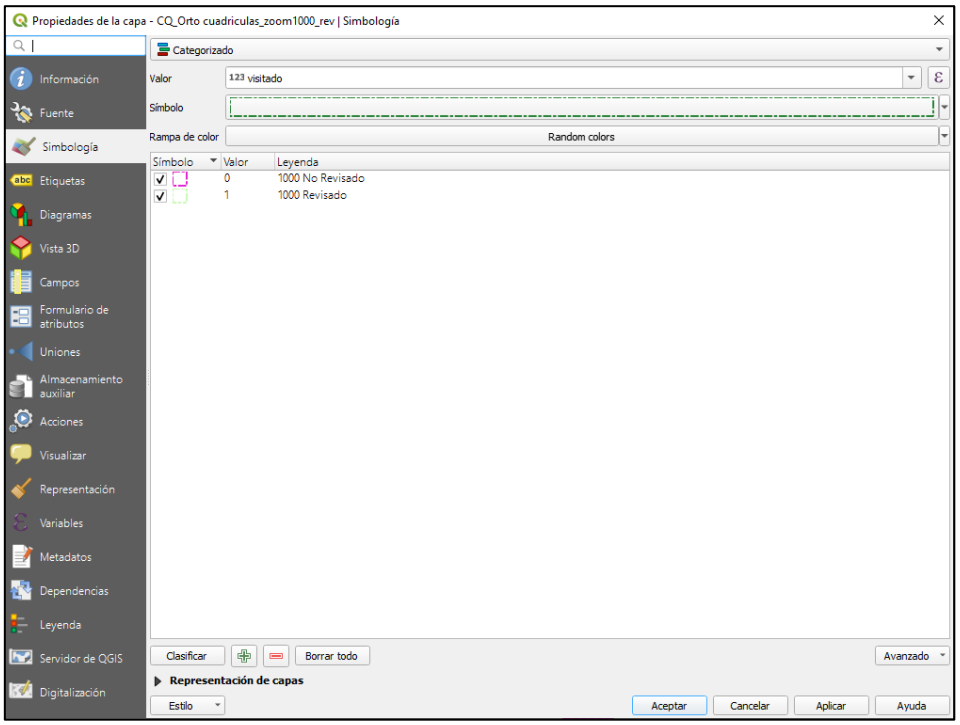


FIGURA 42. ESTILO CAPA CUADRÍCULAS_ZOOM1000_REV

En la capa errores, se han creado un categorizado diferente. Esta capa estará compuesta por polígonos creados por el usuario donde tendrán que indicar el tipo de error en un desplegable. Por ello, se ha categorizado por el campo "tipo":

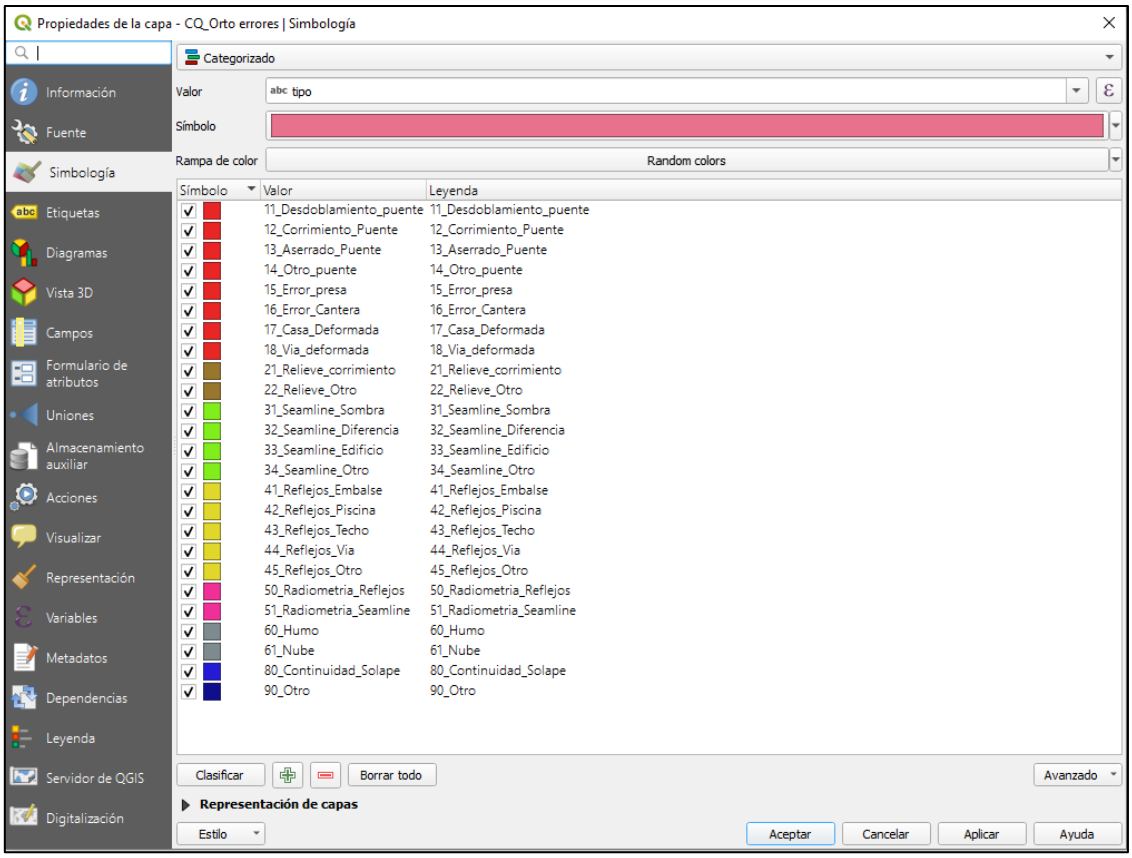


FIGURA 43. ESTILO CAPA ERRORES

6.3.5. Código del método de trabajo.

El código que viene a continuación se corresponde con el flujo de trabajo que tiene que seguir el usuario. Se compone de tan solo tres botones situados en la pestaña "CQ Orto": *Ir a…, Zoom 2x* y *Error*.

Partimos de que las capas están cargadas en el panel de *QGIS,* si accionamos el primer botón *Ir a…,* lo que hace es hacer zoom al primer elemento de la capa *cuadriculas_zoom2000_rev*, que se corresponde con la primera hoja 5000 ordenada de forma ascendente: "*0519 4-8"*. Selecciona el campo con Orden_5000 igual a 1 y Orden_2000 igual a 1 de su tabla de atributos, si se vuelve a accionar pasa al siguiente zoom de la cuadricula con los campos Orden_5000 igual a 1 (corresponde a la misma hoja 5000) y el campo Orden_2000 igual a 2, y así se va haciendo el barrido por toda la hoja. En el código queda así:

```
def zoom2000(self):
    global posicion
    global posicion2000
    layer = QgsProject.instance().mapLayersByName("CQ_Orto
cuadriculas_zoom2000_rev")[0]
    iface.setActiveLayer(layer)
```

Se define la función y se llama a las variables globales *posición y posicion2000*, ambas tienen valor 0. Con *QgsProjectinstance*, se hace instancia a la cuadricula "CQ Orto cuadriculas_zoom2000_rev y con el comando *setActiveLayer* se activa.

```
if layer != None:
    layer.startEditing()
    layer.selectByExpression("\"Orden_5000\"=" + str(posicion))
else:
    str_msg = "No se ha encontrado capa activa"
    iface.messageBar().pushMessage("Info", str_msg, level=Qgis.Warning, duration=15)
```

Teniendo la capa activa, se activa el modo de edición y se hace una primera selección por expresión con el comando *selectbyExpresion* ("campo de la tabla de atributos" = variable global posición, en nuestro caso =1). Así filtramos de todas las hojas 5000, nos quedamos con la primera y se selecciona.

```
        if layer.selectedFeatureCount() > 0:
         layer.selectByExpression("\"Orden_5000\"=" + str(posicion) + " and
\"Orden_2000\"=" + str(
            posicion2000) + "and \"visitado\"=" + str("0"), QgsVectorLayer.IntersectSelection)
         posicion2000 = posicion2000 + 1
         features2000 = layer.selectedFeatures()
         iface.mapCanvas().zoomToSelected(layer)
        else:
         posicion2000 = posicion2000 + 1
         msg = "No hay nada seleccionado"
         QMessageBox.information(iface.mainWindow(), "Selección ", msg)
```

Entramos en los condicionantes, si en la expresión anterior, el número de *features* es mayor que cero, se procede a la segunda selección por expresión. Se selecciona la *feature* con el campo *Orden 5000* igual al número de la posición y con el campo *Orden_2000* igual a la *posición2000*, intersecando en la expresión anterior. Con esto conseguimos quedarnos con un única *feature* que coincide con ese filtro (Si es la primera vez que accionamos el botón, *Orden_5000* = 1 y *Orden_2000* = 1) y con el comando *mapCanvas.zoomToSelected* se hace zoom a esa *feature*. Esa *feature* se guarda en la variable *features2000*, que nos servirá para ir actualizando campos.

Se genera un contador con la variable *posicion2000*, para que cuando se accione de nuevo el botón sume uno a la *posicion2000* y se haga zoom.

```
         if layer.selectedFeatureCount() == 0:
         msg = "Hoja 5000 terminada"  # Podria poner que hoja es la que se ha terminado
         QMessageBox.information(iface.mainWindow(), "Selección ", msg)
         posicion = posicion + 1
         posicion2000 = 1

         if layer.selectedFeatureCount() > 0:
        feature = features2000[0]
        feature["visitado"] = "1"
        layer.updateFeature(feature)
        # layer.commitChanges()
        # Para guardar automaticamente los cambios es: layer.commitChanges()

         else:
           msg = "No hay nada seleccionado"
          QMessageBox.information(iface.mainWindow(), "Selección ", msg)
```

Dentro del anterior condicionante, se tiene en cuenta el número de selecciones. En el primer condicionante si la selección es cero (*selected.FeatureCount ()==0*) se manda un mensaje de que la hoja 5000 termina (significa que todas las *features* con el mismo *Orden_5000* ya se han visitado) con el comando *QMessageBox.information.* Entonces, la *posicion2000* vinculada al campo *Orden_2000* vuelve a ser 1 y a la posición vinculada con el campo *Orden_5000* se le suma 1. Para que, si se vuelve accionar el botón, una vez terminada la hoja 5000, haga zoom a la siguiente *feature* que corresponderá con el campo *Orden 5000* igual a 2 y *Orden 2000* igual a 1.

Al mismo tiempo que se van visitando cuadriculas con escala zoom 2000 con el comando *feature = feature2000[0]* se genera instancia al campo seleccionado y con *feature["visitado"]= 1,* se va rellenando el campo visitado con un 1. El comando *updateFeature* lo que hace es actualizar ese campo y que todo lo anterior se genere.
A continuación, muestro una imagen gráfica del entorno *QGIS* con la tabla de atributos de la capa al accionar el botón *Ir a…* donde se puede apreciar todo lo comentado.

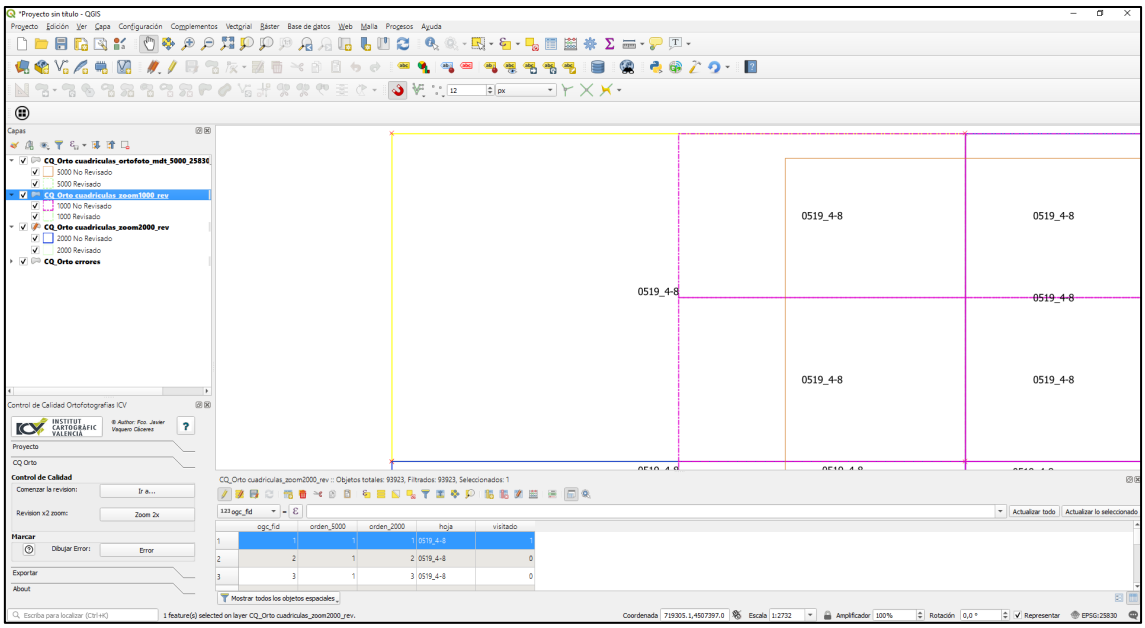


FIGURA 44. ACCIÓN DEL BOTÓN IR A… EN QGIS

El siguiente botón, *Zoom 2x*, va vinculada y hace instancia con la siguiente capa *cuadriculas zoom1000 rev*, donde hace un zoom a escala alrededor de 1:1000. La idea de este botón es cuando se va visitando zooms de cuadriculas 2000 y se quiere hacer a esa misma cuadricula un zoom mayor, accionar el botón de *Zoom 2x*. El modelo de datos está preparado que para una cuadricula 2000 existan hasta cuatro cuadriculas zoom 1000, por tanto, en el campo *Orden 1000* de su tabla de atributos serán valor de 1 a 4 como máximo.

En cuanto al código es muy similar al anterior botón solo se añade otra selección más:

```
def zoom1000(self):
    global posicion
    layer = QgsProject.instance().mapLayersByName("CQ_Orto
cuadriculas_zoom1000_rev")[0]
```

```
iface.setActiveLayer(layer)
```

Igual que el anterior, pero se activa la capa *CQ_Orto cuadriculas_zoom1000_rev*

```
if layer != None:
    layer.startEditing()
    global posicion2000
    layer.selectByExpression('\"Orden_5000\"=' + str(posicion))

    if layer.selectedFeatureCount() > 0:
        global posicion1000
        layer.selectByExpression('\"Orden_2000\"=' + str(posicion2000 - 1),
QgsVectorLayer.IntersectSelection)
        layer.selectByExpression('\"Orden_1000\"=' + str(posicion1000),
QgsVectorLayer.IntersectSelection)
        iface.mapCanvas().zoomToSelected(layer)
        posicion1000 = posicion1000 + 1
        features1000 = layer.selectedFeatures()
```

Entonces, filtrando primero por la posición del campo *Orden_5000*, es aquí donde se llama a la variable global *posicion1000* con valor igual a 1 y se proceden a las selecciones por expresión. La primera teniendo en cuenta el valor del campo *Orden_2000* que se le resta 1 a la *posición2000* actual (si no se resta una unidad el valor que coge es el valor que tiene guardado en el anterior botón) y otra selección filtrando por el campo *Orden_1000*.

Igual que en el anterior se le suma 1 a la *posicion1000* cuando se vuelva a accionar y se guarda la *feature* con zoom en la variable *features1000.*

```
if layer.selectedFeatureCount() == 0:
    msg = "Hoja 2000 terminada"
     QMessageBox.information(iface.mainWindow(), "Selección ", msg)
    layer.selectByExpression('\"Orden_2000\"=' + str(posicion2000 - 1),
                    QgsVectorLayer.IntersectSelection)
    posicion2000 = posicion2000
    posicion1000 = 1
```

Teniendo en cuenta el número de selecciones por expresión anterior, cuando *selectedFeatureCount() ==0* se manda un mensaje de que la hoja 2000 termina (significa que todas las *features* con el mismo *Orden_5000 y Orden_2000* ya se han visitado) con el comando *QMessageBox.information.* Entonces, la *posicion1000* vuelve a ser 1 y la *posicion2000* es el resultado de una selección por expresión por el campo Orden_2000 donde se le vuelve a restar 1 a la variable global posicion2000. Se hace esto, para que cuando termine la hoja 2000, si se pulsa de nuevo el botón, se quede en el mismo valor *Orden_2000* y no vaya recorriendo otro valor hasta perderse.

Un ejemplo práctico de esto último, si estamos en el *feature* con *Orden_5000* = 1 y *Orden_2000* = 3 y se acciona el botón, ira recorriendo los features de la capa cuadriculas_zoom1000_rev con *Orden_1000* = 1 hasta 4 (y mismos campos Orden_5000 = 1 y Orden_2000 = 3) hasta aparecer un mensaje de “hoja 2000 terminada” , si se vuelve a accionar recorrerá las mismas *features* anteriores.

```
        if layer.selectedFeatureCount() > 0:
            feature = features1000[0]
            feature.setAttribute("visitado", "1")
            feature["visitado"] = "1"
            layer.updateFeature(feature)
            # layer.commitChanges()

        # else:
        #     msg = "No hay nada seleccionado"
        #     QMessageBox.information(iface.mainWindow(), "Selección ", msg)


    else:
        msg = "No hay nada seleccionado"
        QMessageBox.information(iface.mainWindow(), "Selección ", msg)


else:
    str_msg = "No se ha encontrado capa activa"
```

Al mismo tiempo que se van visitando cuadriculas con escala zoom 1000 con el comando *feature = feature1000[0]* se genera instancia al campo seleccionado y con *feature["visitado"]= 1,* se va rellenando el campo visitado con un 1. El comando *updateFeature* lo que hace es actualizar ese campo y todo lo anterior se genere.
A continuación, muestro una imagen gráfica del entorno *QGIS* con la tabla de atributos de la capa al accionar el botón *Zoom 2x d*onde se puede apreciar todo lo comentado.

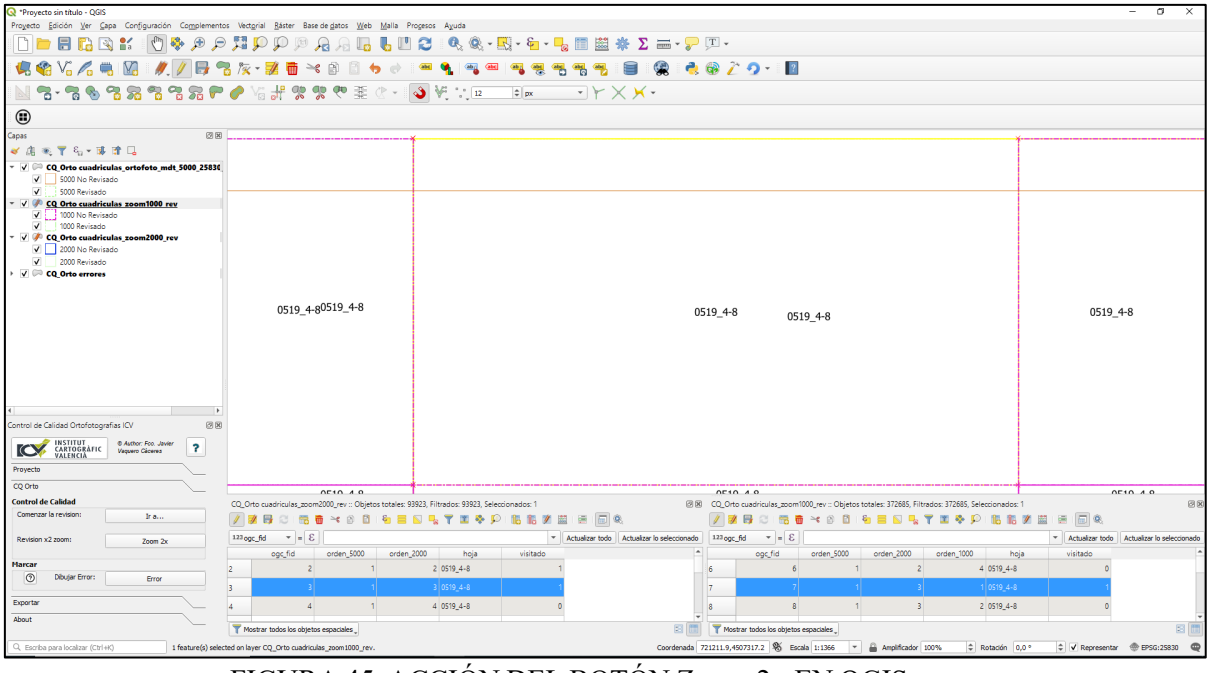


FIGURA 45. ACCIÓN DEL BOTÓN ZOOM 2X EN QGIS

Para marcar un error encontrado, se acciona el botón *error*, que nos permite automáticamente dibujar un polígono en la zona del error y rellenar los campos que tiene la capa "errores" que son los siguientes:

Id: se generar automáticamente

Tipo: se despliega un listado con los tipos de errores que se pueden encontrar. Tabla de errores que se contemplan en el ICV con su codificación adjunto en Anexo II.

Hoja: Manualmente se escribe la hoja con la ayuda de las label

Corregido: Desplegable con Si o No.

Lo vemos en el código en la clase error definida:

```
def error(self):
    layer = QgsProject.instance().mapLayersByName("CQ_Orto errores")[0]
    iface.setActiveLayer(layer)

    fieldIndex = layer.fields().indexFromName('tipo')
    editor_widget_setup = QgsEditorWidgetSetup('ValueMap', {
        'map': {'11_Desdoblamiento_puente': '11_Desdoblamiento_puente',
                '12_Corrimiento_Puente': '12_Corrimiento_Puente',
                '13_Aserrado_Puente': '13_Aserrado_Puente',
                '14_Otro_puente': '14_Otro_puente',
                '15_Error_presa': '15_Error_presa',
                '16_Error_Cantera': '16_Error_Cantera',
                '17_Casa_Deformada': '17_Casa_Deformada',
                '18_Via_deformada': '18_Via_deformada',
                '21_Relieve_corrimiento': '21_Relieve_corrimiento',
                '22_Relieve_Otro': '22_Relieve_Otro',
                '31_Seamline_Sombra': '31_Seamline_Sombra',
                '32_Seamline_Diferencia': '32_Seamline_Diferencia',
                '33_Seamline_Edificio': '33_Seamline_Edificio',
                '34_Seamline_Otro': '34_Seamline_Otro',
                '41_Reflejos_Embalse': '41_Reflejos_Embalse',
                '42_Reflejos_Piscina': '42_Reflejos_Piscina',
                '43_Reflejos_Techo': '43_Reflejos_Techo',
                '44_Reflejos_Via': '44_Reflejos_Via',
                '45_Reflejos_Otro': '45_Reflejos_Otro',
                '50_Radiometria_Reflejos': '50_Radiometria_Reflejos',
                '51_Radiometria_Seamline': '51_Radiometria_Seamline',
                '60_Humo': '60_Humo',
                '61_Nube': '61_Nube',
                '80_Continuidad_Solape': '80_Continuidad_Solape',
                 '90_Otro': '90_Otro',

                }
    }
                                )


    layer.setEditorWidgetSetup(fieldIndex, editor_widget_setup)

    fieldIndex = layer.fields().indexFromName('corregido')
    editor_widget_setup = QgsEditorWidgetSetup('ValueMap', {
        'map': {'Si': 'Si',
                'No': 'No',
                }
```

```
}
                    )
layer.setEditorWidgetSetup(fieldIndex, editor_widget_setup)
layer.startEditing()
iface.actionAddFeature().trigger()
```

Se convierte la capa errores activa y editable con los comandos *setActiveLayer* y *startEditing*(), al ser de tipo polígono con el comando *action.addFeature().trigger()* permite directamente dibujar un polígono.

Con el comando *QgsEditorWidgetSetup* se activa el editor del widget donde se indica '*ValueMap*' para crear campos predefinidos. En el caso del campo *tipo* se han puesto todos los tipos de errores codificados y contemplados en el ICV y en el campo corregido, dos opciones: *'Si' y 'No'*.

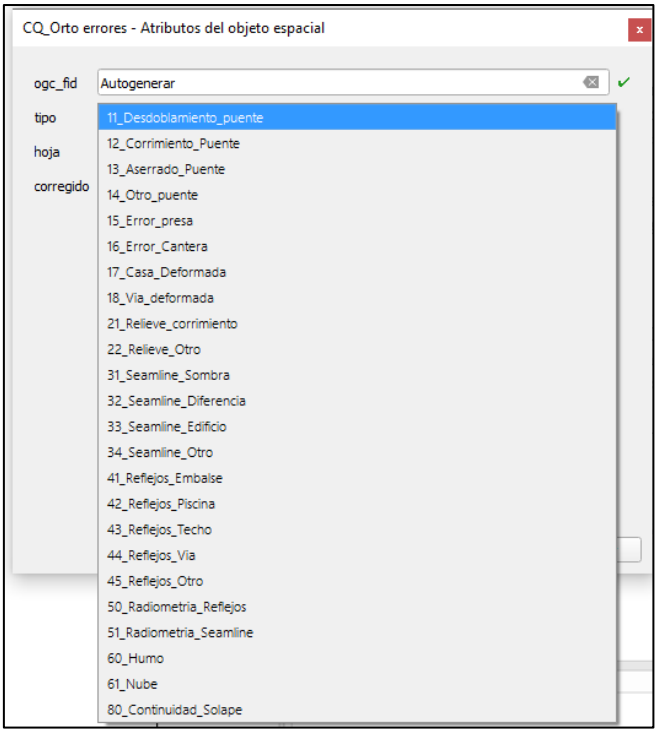


FIGURA 46. DESPLEGABLE DEL CAMPO TIPO

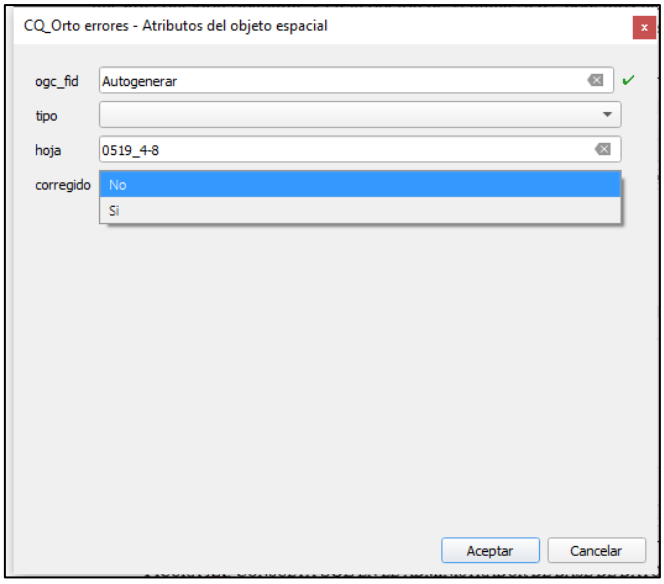


FIGURA 47. DESPLEGABLE DEL CAMPO CORREGIDO

El ultimo botón *"Exportar"* en la pestaña Exportar del flujo de trabaja consiste en la exportación de esos errores encontrados en un archivo Shapefile. Accionando el botón aparece una ventana de dialogo donde tiene que indicar el nombre del usuario y después la carpeta vacía donde se quiere guardar. Lo vemos en el código siguiente:

```
def aceptar(self):
    nombre = QInputDialog.getText(None, 'NOMBRE', 'Introduce el nombre del operador')
    Dir = QFileDialog.getExistingDirectory(self, "Seleccionar directorio de salida vacia",
"C:/temp")

    layer = QgsProject.instance().mapLayersByName("CQ_Orto errores")[0]
    iface.setActiveLayer(layer)

    _writer = QgsVectorFileWriter.writeAsVectorFormat(layer, Dir + '_' + nombre[0], 'utf-8',
                                        driverName='ESRI Shapefile')

    msg = "Capa guardada correctamente"
    QMessageBox.information(iface.mainWindow(), "Guardar ", msg)
```

Se define el objeto *aceptar,* con la variable *nombre* y el comando *QInputDialog.getText*, guarda el nombre introducido y lo inserta en el nombre del archivo. El comando *QFileDialog.getExistingDirectory*, permite al usuario navegar por las carpetas de Windows a través de una ventana y seleccionar un directorio vacío para su exportación, se guarda en la variable *Dir*.

Como la capa a exportar es la capa llamada "errores", se activa la misma y con el comando *QgsVectorFileWriter.writeAsVectorFormat* definimos el formato 'Esri Shapefile' en el *driverName* como formato final del fichero.

6.3.6. Código para la continuación del proyecto

Cuando el usuario cierra el proyecto de *QGIS* guardando todos los cambios hechos, se guardan automáticamente en el mismo proyecto generado. Surgió la necesidad de indicar en código la última situación o la última cuadrícula visitada para que cuando lo vuelva a abrir siga en donde se había quedado.

Esto se consigue escribiendo en el objeto "capas" (botón para abrir el proyecto) lo siguiente:

```
        # A continuación, busca en la capa cuadriculas_zoom2000 el primer valor con el
campo visitado=0
global posicion
global posicion2000
layer = QgsProject.instance().mapLayersByName("CQ_Orto cuadriculas_zoom2000_rev")[0]
if layer != None:
    layer.selectByExpression("\"visitado\"=" + str("0"))
    if layer.selectedFeatureCount() > 0:
        features2000 = layer.selectedFeatures()
        if layer.selectedFeatureCount() > 0:
```

```
feature = features2000[0]
posicion = feature.attribute("orden_5000")
posicion2000 = feature.attribute("orden_2000")

layer.selectByExpression("\"visitado\"=" + str("0"), QgsVectorLayer.RemoveFromSelection)
```

Hace activa la capa *cuadriculas_2000_rev* que será la principal que sigue el flujo de trabajo, y por una *selectbyexpression* se indica que seleccione el primer feature con el campo 'visitado' igual a 0. Esa feature la guarda en la variable *feature* y con el comando *feature.attribute* asignamos la posición actual a la variable global *posición* para el campo *Orden_5000* y la variable global *posicion2000* para el campo *Orden_2000.*

Así queda resuelto el problema y nos aseguramos que el usuario comenzará por donde se ha quedado la situación anterior y poder seguir con su trabajo.

6.3.6. Creación de Triggers

Para que se lleve un control exhaustivo y correcto, surgió la necesidad de relacionar las tablas automáticamente según los atributos que tuvieran en común.

En el proceso de revisión, cada vez que se revisa una cuadricula con escala zoom 2000, en el campo "*visitado*" pasa de 0 a 1. Por lo tanto, por lógica, las cuadriculas con escala zoom 1000 que estén dentro de esa cuadricula deberían tener el campo "visitado" igual a 1.

Este proceso no se contempla en el código porque se han creado dos trigger para ellos. Un trigger o disparador es un script que se usa en lenguaje de programación SQL, en especial en bases de datos como MySQL o PostgreSQL. Consiste en una serie de reglas predefinidas que se asocian a una tabla.

En el entorno de trabajo, en la pestaña Base de Datos (con la conexión realizada de la base de datos del proyecto) podemos acceder al administrador de BBDD. Buscamos nuestra conexión Spatialite y en la opción de ventana SQL, podemos hacer cualquier consulta SQL, crear disparadores o trigger… etc. Cualquier acción que se pueda hacer con SQL.

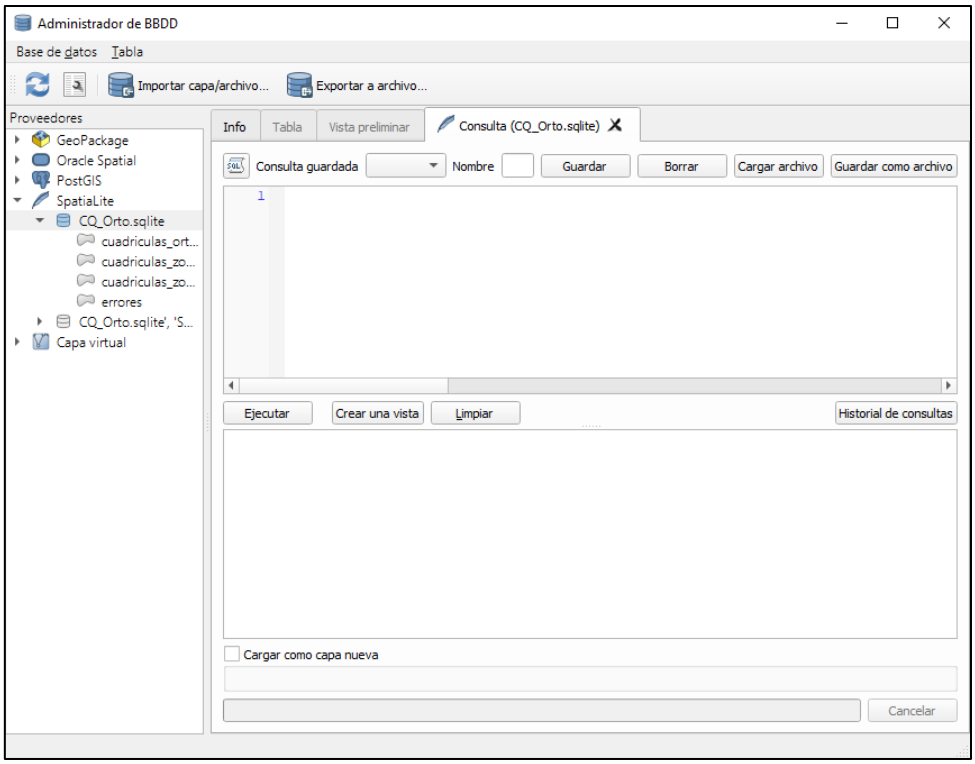


FIGURA 48. CONSULTA SQL EN EL ADMINISTRADOR DE BASE DE DATOS DE QGIS

Entonces, para crear ese registro en la capa zoom 1000 a partir del atributo visitado de la capa zoom 2000, escribimos el siguiente disparador:

"CREATE TRIGGER update_c1000_visitado AFTER UPDATE ON cuadriculas_zoom2000_rev

BEGIN

UPDATE cuadriculas_zoom1000_rev SET visitado = 1 WHERE orden_2000 = NEW.orden_2000 AND orden_5000 = NEW.orden_5000 AND NEW.visitado = 1;

END"

Significa, créame un disparador llamado *update_c1000_visitado* después de que se haya actualizado la capa *cuadriculas_zoom2000_rev.* Donde en la capa *cuadriculas_zoom1000_rev* actualiza el campo *visitado* igual a 1 donde coincidan los valores *orden_2000* y *orden_5000*. Este disparador se queda guardado en la base de datos "semilla" y siempre hará esa actualización.

Pasa lo mismo para la cuadricula 5000, se ha creado otro disparador para cuando se revisen todas las hojas de la cuadricula zoom 2000 con el mismo campo de *orden_5000*, se actualice en la cuadricula 5000 el valor de *visitado* igual a 1. Se actúa de forma análoga al anterior, se le pone el nombre de *update_c5000_visitado*:

"CREATE TRIGGER update_c5000_visitado AFTER UPDATE ON cuadriculas_zoom2000_rev

BEGIN

UPDATE cuadriculas_ortofoto_mdt_5000_25830_icv_round_10m SET visitado = 1 WHERE orden_5000 = NEW.orden_5000 AND NEW.visitado = 1;

END"

Así quedaría resuelto el problema. Si no se hacia este control, la revisión del usuario quedaría incompleta y sin ninguna relación, por lo que no se llegaría a un control exhaustivo.

Los disparadores se han guardado en un archivo .sql dentro de la carpeta del plugin/Triggers. Se puede abrir con un editor de texto y queda constancia de que en el presente plugin se utilizan disparadores para el correcto funcionamiento.

6. Creación de un repositorio en GitHub

Se ha creado una cuenta en GitHub para guardar el complemento creado para que la comunidad de desarrolladores lo puedan ver y descargar. La dirección es la siguiente:

https://github.com/FranVaqCac

Hay que crear un repositorio dentro de la cuenta creada, añadiendo el nombre del repositorio, una pequeña descripción y le damos a "Add a README file" para ayudar a otros desarrolladores u otras personas saber de qué trata el plugin subido.

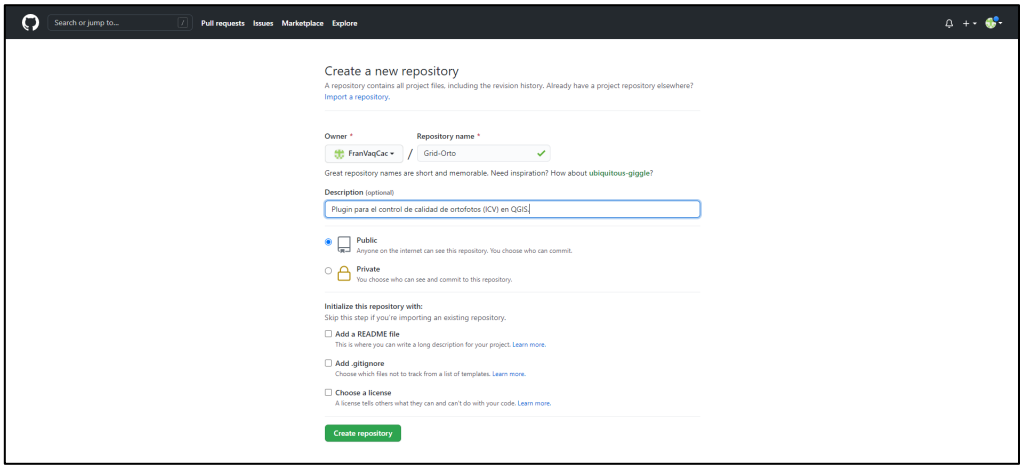


FIGURA 49. CREACIÓN DEL REPOSITORIO EN LA CUENTA DE GITHUB

Una vez creado el repositorio dentro de la cuenta, la dirección web para poder descargarte el plugin es la siguiente:

https://github.com/FranVaqCac/Grid-Orto

Puedes editar ese fichero y dar instrucciones de lo que creas conveniente:

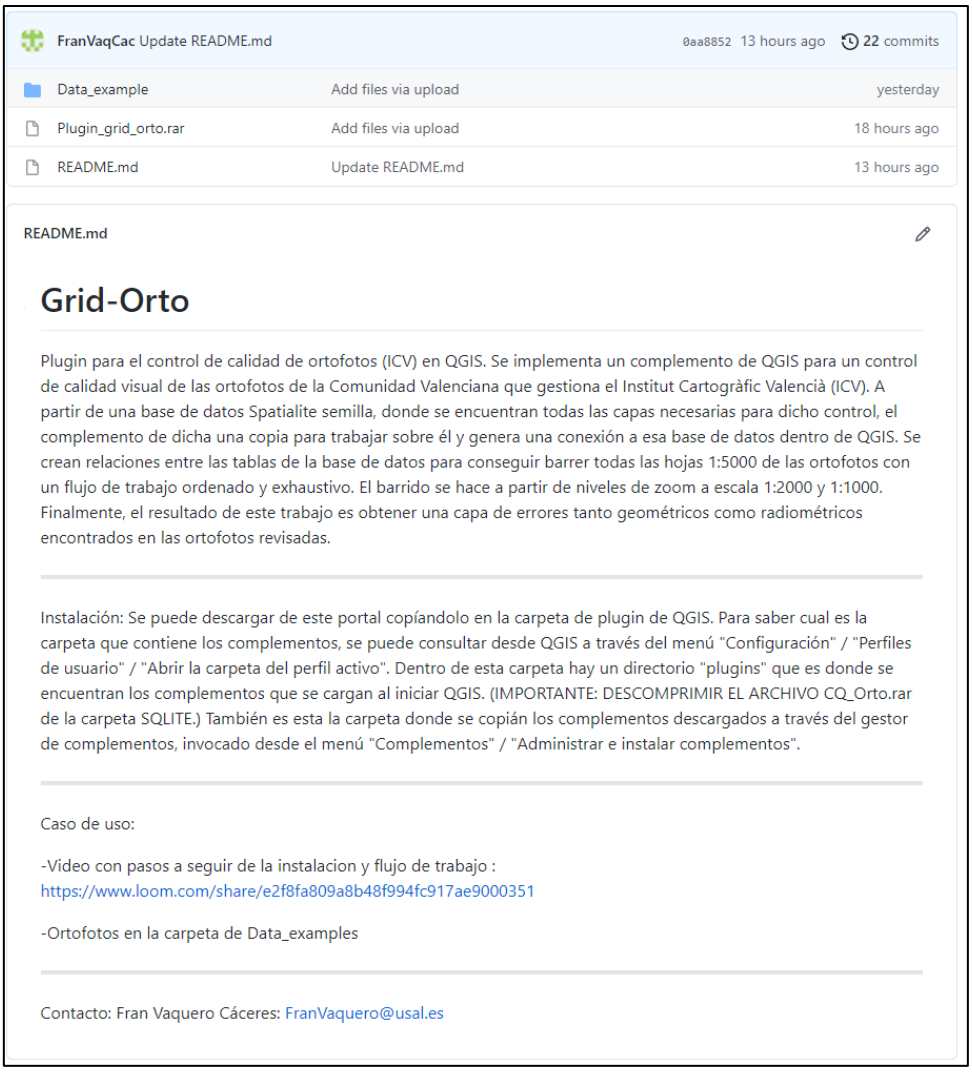


FIGURA 50. ARCHIVO README DEL REPOSITORIO (GITHUB)

Es una forma rápida y sencilla de dar visibilidad al trabajo hecho y que otros desarrolladores puedan verlo y poder interactuar con ellos para posibles consultas o errores que puedan surgir.

7. Publicación del complemento en el repositorio oficial de QGIS

Se ha informado al Instituto Cartográfico Valenciano del complemento poniendo a disposición del mismo el código fuente por si se quisiera adoptar como un producto oficial del mismo, para poder publicitarlo desde su plataforma, añadirle sus logos, lo pudiera extender y darle carácter de oficialidad.

Está pendiente de evaluación a partir de su uso y depuración, de manera que cuando se estabilice se solicitará su publicación, con la traducción al inglés... Se subiría al repositorio oficial de QGIS mediante una cuenta corporativa en vez de la de un usuario particular. Esto aumentaría las funcionalidades y el número de usuarios.

8. Resultados obtenidos

El resultado como se ha visto en los anteriores apartados es el desarrollo de un complemento de QGIS que está disponible en un repositorio de GitHub donde se puede descargar de forma libre y ayude al usuario a que la revisión del control de calidad visual de la ortofoto de la comunidad valencia gestionada por el Instituto Cartográfico Valenciano sea más optimizada y fiable.

Con este complemento se pretende obtener un mayor rendimiento en el trabajo y facilitar la información para ofrecer un seguimiento más completo del proyecto con:

-Modelo de datos completo

-Atributos clave para el barrido "*Orden_5000, Orden_2000, Orden_1000"*

-Atributo "visitado", dejando constancia de qué se ha revisado.

-Estilos de capas que ayudan que ayudan visualmente con el seguimiento

-Capacidad de exportar la capa de errores marcados en un archivo shapefile.

Se ha conseguido desarrollar a parte de un código fuente, un flujo de trabajo específico para la tarea, sencillo y a la vez exhaustivo, enfocado para un usuario no experto. También se ha creado unos ejemplos de errores que aparecen en las ortofotos a modo de ayuda. Se puede acceder a ambos documentos a través del plugin.

Se ha trabajado con una base de datos Spatialite y se ha realizado conexiones en *QGIS* de la misma para hacer más cómodo el trabajo.

Elaboración de una memoria que describe paso a paso la elaboración del complemento, desde la creación de su esqueleto, su programación y subida a un repositorio público para que pueda ser reutilizado por los usuarios.

También añadimos al trabajo dos documentos de ayuda para que el usuario sea totalmente independiente en el trabajo. El documento de ayuda para que el usuario a simple vista conozco el funcionamiento de los botones y del modelo de datos y otro documento para ayudar al operador en cuanto a los errores que se puede encontrar.

Todo este complemento se ha desarrollado con el lenguaje de programación *Python* en el software *Pycharm*

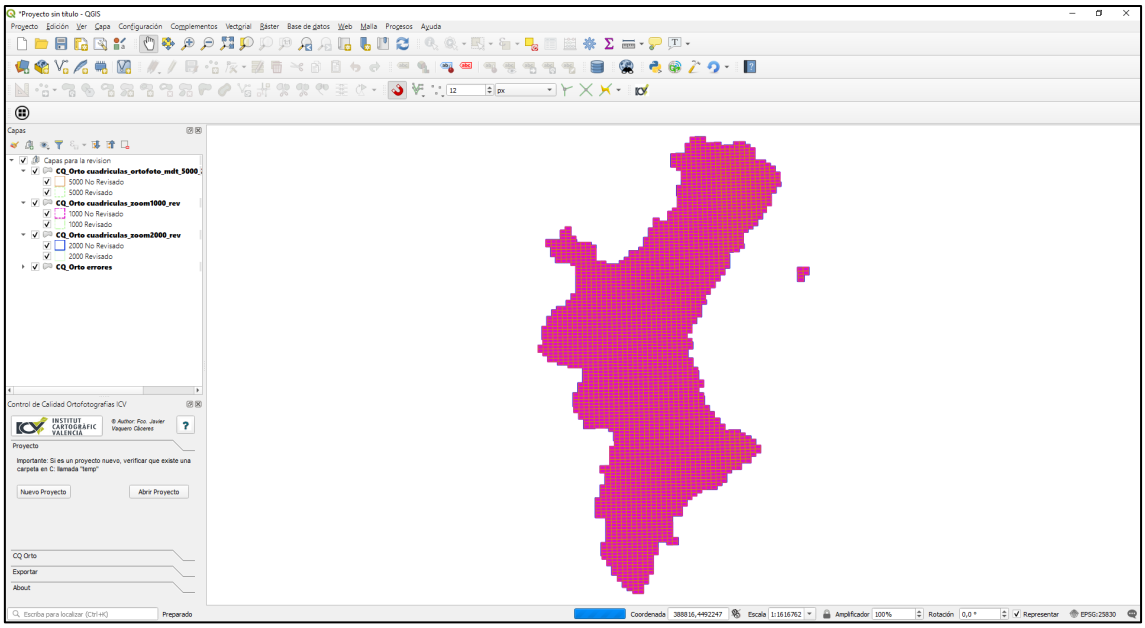


FIGURA 50. INTERFAZ DE QGIS CON EL COMPLEMENTO CARGADO

9. Conclusiones

A la vista de los resultados obtenidos en el presente trabajo final de master podemos concluir lo siguiente:

- Se ha conseguido integrar dentro de un software de SIG un flujo de trabajo exhaustivo del ICV, creando un complemento sencillo y enfocado a un usuario no experto.
- Poder disponer de un flujo de trabajo que se contempla en el Instituto Cartográfico Valenciano, en un software libre GIS. Facilita la forma de trabajo en el Servicio de Observación del Territorio ya que dispone de toda la información para realizarlo.
- Tener un mayor control de qué zonas se revisan y qué operador lo revisa.
- Generación de un archivo de errores para los reportes a la empresa adjudicataria y su posterior corrección.

Caben destacar posibles mejoras del plugin en futuras versiones, ya que la idea de este complemento es emplearlo en el ICV y mejorarlo:

- A la hora de crear el proyecto, darle posibilidad al usuario que elija donde quiere guardarlo
- Análogamente cuando se abre el proyecto, darle la oportunidad al usuario de elegir, entre las carpetas de Windows, la base de datos si hubiera más de una.
- Investigar en como generar automáticamente una cuadricula en *QGIS* virtual, sin tener que tenerla fija en un modelo de datos, que nos permita seguir con el flujo de trabajo definido
- Investigación de como cargar las ortofotos con algún "Tileindex" dentro de QGIS sin que el rendimiento del complemento afecte. Actualmente se carga cada hoja 5000 manualmente ya que se ha probado trabajando con llamadas WMS y no llega a tener un rendimiento asequible para este tipo de trabajo.

10.Bibliografía

Ackerman y Krauss: Presente y futuro de los MDT. Recuperado en junio de 2021. https://www.gtbi.net/wp-content/uploads/2020/12/16-DPes_ScopDtms.pdf

Apuntes de fotogrametría digital de F.J. Hermosilla y S. López Cuervo. Recuperado en junio de 2021. http://www.ign.es/web/resources/docs/IGNCnig/QSM-CNIG-Becas-Temario-A2.pdf

Clavo, Domingo. Fotogrametría II, Escuela Universitaria de Ingeniería Técnica de Topografía (Universidad Politécnica de Madrid)

Geographic Information System. Explore Our Questions. https://gis.stackexchange.com/

QGIS Python API documentation Project. V.3.10. https://qgis.org/pyqgis/3.10/

Martí, Carles. ¿Por qué saber FME Desktop es imprescindible para un profesional SIG? Por Asociación Geoinnova. Recuperado en junio de 2021.

Mehad Haggag, Mohammed Zahran, Mahmoud Salah, Towards Automated Generation of True Orthoimages for Urban Areas, American Journal of Geographic Information System, Vol. 7 No. 2, 2018, pp. 67-74. doi: 10.5923/j.ajgis.20180702.03

Modern Photogrammetry por : Edward M. Mikhail, James S. Bethel y J. Chris McGlone.De la editorial Jhon Wiley & Sons, Inc.

Moya Fuero, Alfonso. TFM: Desarrollo de un complemento para QGIS para cargar los servicios del Instituto Cartográfico Valenciano (ICV), Septiembre 2020.

NASCA- Expertyise en Systèmes d'information géographique. Python Methods for QGis: How to access vector data (postgis, spatiality). Recuperado en Junio de 2021

OGC Open Geospatial Consortium (2021). Recuperado en junio de 2021 de https://www.ogc.org

Perez Graterol, Luis Eduardo. Introducción al uso de Spatialite y Geopackage en QGIS 3. Recuperado en junio de 2021

Pérez Álvarez, Juan Antonio, Apuntes de Fotogrametría III. 2001. Centro Universitario de Mérida, Ingeniería Técnica Topografía (Universidad de Extremadura)

PyQt5 5.15.4. "Python bindings for the Qt cross platform application toolkit". Ultima actualización marzo de 2021. https://pypi.org/project/PyQt5/

REAL DECRETO 1071/2007, de 27 de julio, por el que se regula el sistema geodésico de referencia oficial en España. Recuperado en Junio de 2021.

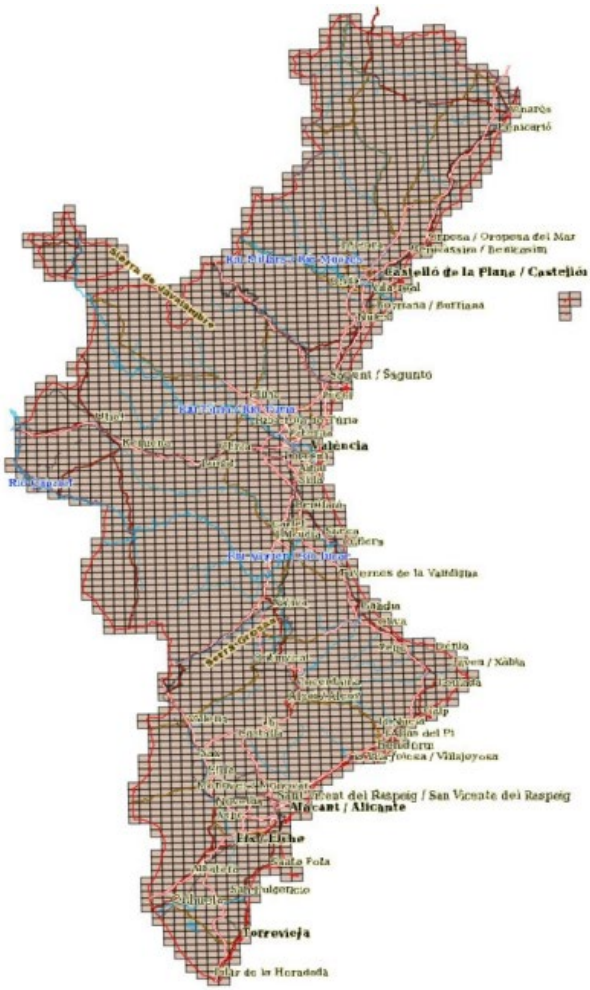
SAFE Software. FME Desktop. https://www.safe.com/fme/fme-desktop/

11. Anexo I. Ayuda del Complemento

12.1 Flujo de Trabajo

-FLUJO DE TRABAJO-

Control de Calidad Ortofotos de la Comunidad Valenciana (ICV)



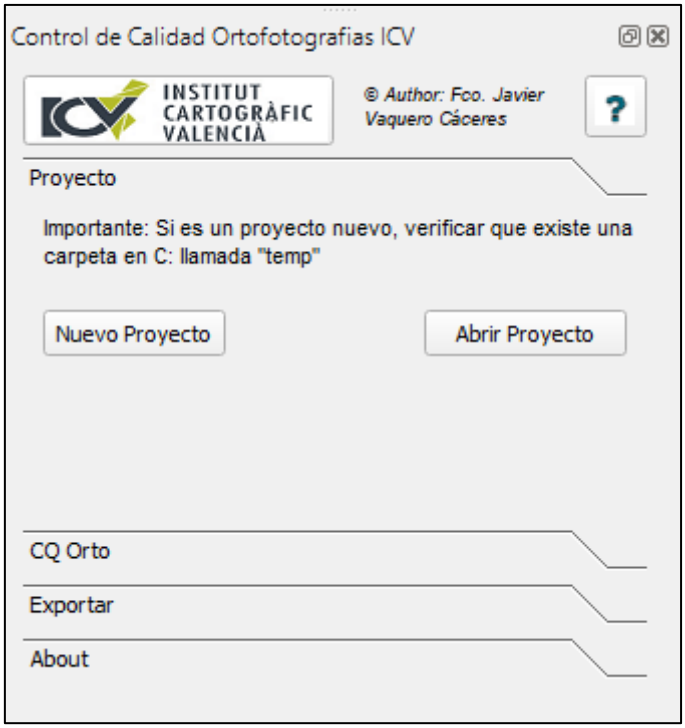
Versión	Fecha	Editor/Modificado por	Comentarios
1.0	Julio 2021	Fran Vaquero Cáceres franvaquero@usal.es	*Complemento QGIS Plugin*

Contenido

1. PLUGIN

En este documento se describe el flujo de trabajo que hay que llevar para el correcto funcionamiento del complemento de QGIS "Grid_Orto". Este complemento se ha creado para llevar un control de calidad visual de las ortofotos de la comunidad valenciana.

La interfaz del complemento es el siguiente:



Constituido por 4 pestañas:

-Pestaña Proyecto: Para comenzar a trabajar se crea el proyecto a partir de una base de datos Spatialite y se abre.

-Pestaña CQ Orto: Comienza la revisión visual de la ortofoto con variaciones de zoom establecidas fijas

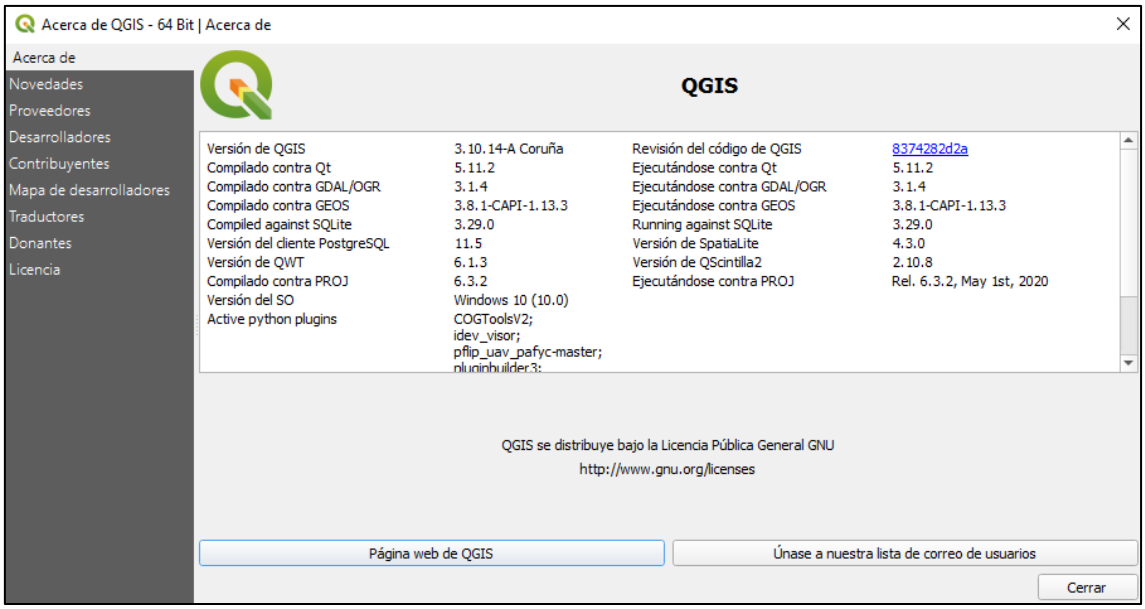
-Pestaña Exportar: Botón para poder exportar los errores encontrados

-Pestaña About: Información del desarrollador y contacto.

2. DEPENDENCIAS

Para el funcionamiento del complemento es indispensable disponer del software de Sistemas de Información Geográfica S.I.G QGIS. La versión que se ha utilizado para crear el complemento es la versión 3.10.14 -A Coruña, se puede descargar en la siguiente dirección.

Para verificar la versión de QGIS que tenemos instalada podemos ir a Ayuda/Acerca de



Para que el plugin se adapte a las cuadriculas con diferente zoom, se ha pensado en un equipo con un monitor de 24 pulgadas para crear esas capas con zoom alrededor de 1:2000 y 1:1000.

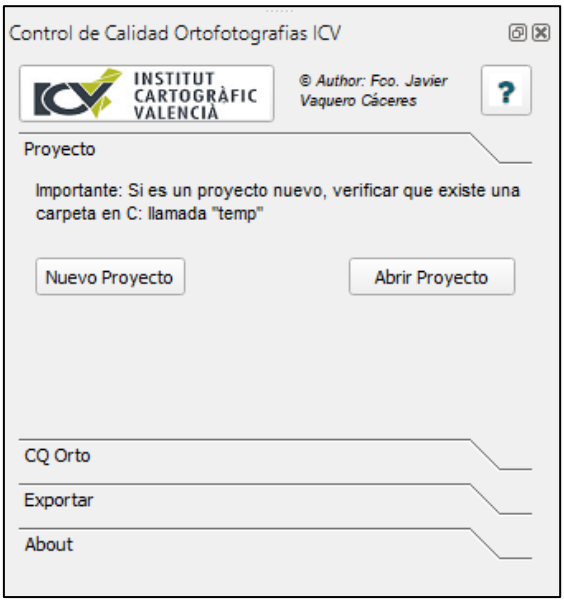
Por lo que, para monitores de 24 pulgadas o mayores, se adaptan los datos origen.

1. FLUJO DE TRABAJO

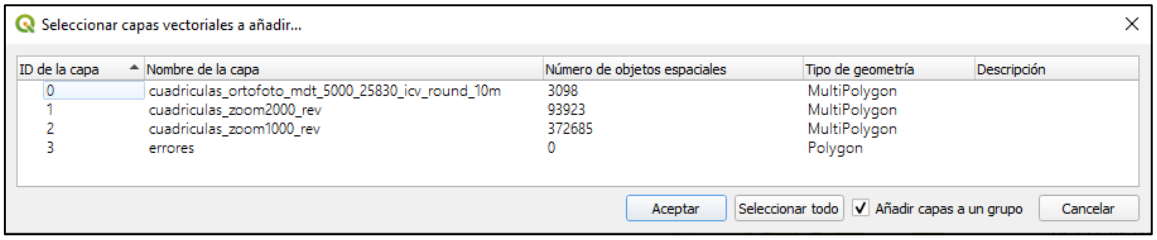
3.1. Flujo desde cero.

Como ya se ha indicado en el apartado de Plugin, existen 4 pestañas: Proyecto, CQ Orto, Exportar y About.

En primero lugar, y lo más importante, necesitamos crear una carpeta en C: que se llame "temp" donde se va a almacenar el proyecto .spatialite en el cual vamos a trabajar. Por lo tanto, en la pestaña Proyecto, accionamos el botón *Nuevo Proyecto*.



A continuación, se abre el proyecto creado en el siguiente botón de la pestaña *Abrir Proyecto*. Aparecen todas las capas que vamos a utilizar para el control de calidad visual:



Nos encontramos las siguientes capas:

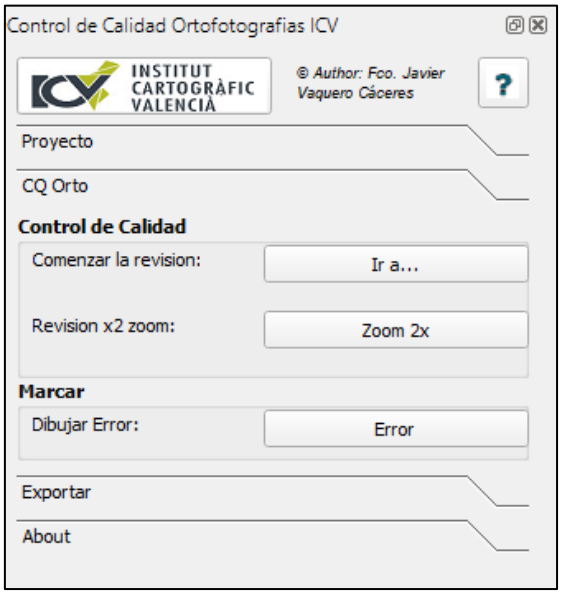
-Cuadriculas_ortofoto_mdt_5000_25830_icv_round_10m: Se corresponde a la cuadricula oficial del Instituto Cartográfico Valenciano, con escala 1:5000 de las ortofotos de la Comunidad Valenciana.

- Cuadriculas_zoom2000_rev: Capa de tipo polígono creada para que en un monitor de 24" se corresponda con un zoom a una cuadricula equivalente en QGIS a 1:2000

- Cuadriculas_zoom1000_rev: Capa de tipo polígono creada para que en un monitor de 24" se corresponda con un zoom a una cuadricula equivalente en QGIS a 1:1000. Por cada cuadricula 1:2000 corresponden 4 cuadriculas de 1:1000

- errores: Capa de tipo polígono vacía creada para almacenar los distintos errores que nos podamos encontrar en la revisión de las ortofotos.

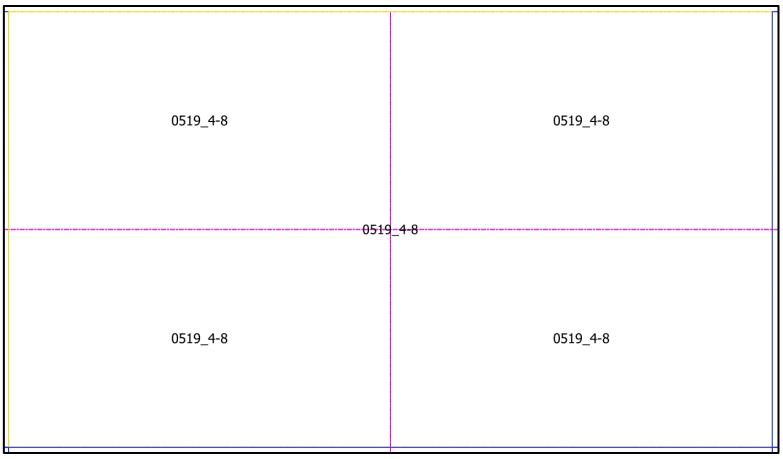
Una vez cargadas todas las capas, procedemos al barrido de las ortofotos por todas las hojas 5000. La primera hoja 5000 será en nuestro caso la 519_4-8 y la última 935_5-2. Abrimos la siguiente pestaña CQ Orto:



Con el botón *Ir a...* hace zoom a la primera hoja con zoom 1:2000 que también corresponde a la primera hoja 1:5000 ordenadas de forma ascendente. Las relaciones entre tablas se han generado por los campos en común que son Orden_5000, Orden_2000 y Orden_1000. Por lo tanto, irá a la cuadricula de la capa zoom_2000 donde el Orden_5000=1 y el Orden_2000=1.

Si se vuelve a accionar pasa a la siguiente cuadricula de zoom_2000 con Orden_5000=1 y Orden_2000=2 y así sucesivamente vamos barriendo hojas de ortofotos 1:5000.

En el caso de que necesitemos un zoom mayor en la vista anterior, el botón de *Zoom2x* permite hacer un zoom a la misma vista correspondiente a la capa zoom_1000.



La cuadricula con color amarillo es la vista actual con zoom 2000 y las cuadriculas magentas corresponden las cuadriculas con zoom 1000. Por lo tanto, si se acciona el botón *Zoom2*x, hará un barrido por la cuadricula zoom2000 con un zoom 1000.

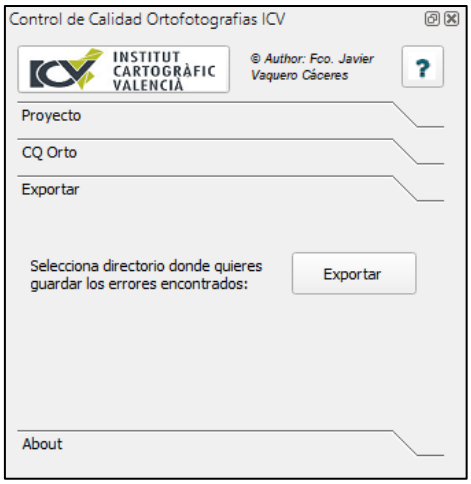
Cada vez que se acaba el barrido de zoom 1000, se tiene que accionar de nuevo el botón *ir a…* para que vaya a la siguiente vista. La relación entre tablas con la tabla cuadriculas_zoom1000_rev es con los campos Orden5000, Orden2000 y Orden1000.

Cuando nos encontremos un error, hay que accionar el botón *Error* dibujar manualmente un polígono en la zona en cuestión. Con el botón derecho se acepta el dibujo y rellenamos los campos de Tipo, Hoja y Corregido.

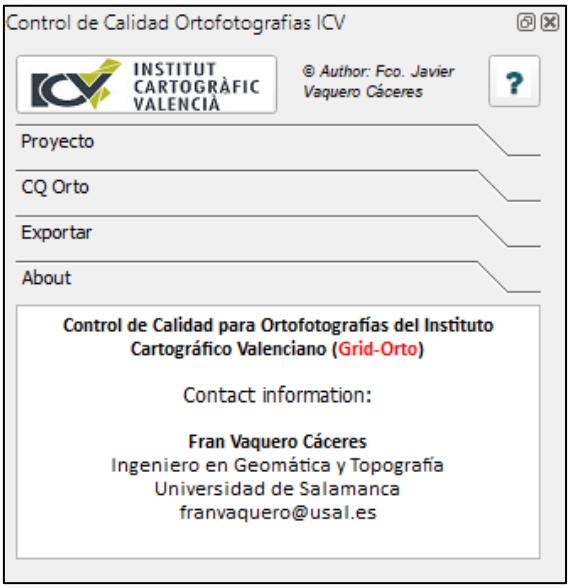
3.2 Flujo con proyecto creado.

Cuando queremos continuar con nuestro proyecto y barrido, se carga el plugin y accionamos el botón *Abrir Proyecto* y a continuación el botón *Ir a…* que nos hará zoom a la vista donde nos habíamos quedado antes de cerrar QGIS. Y seguimos con el flujo habitual.

En la siguiente pestaña Exportar, con el botón *Exportar* podemos guardar los errores marcados en un archivo Shapefile. El único requisito es que hay que indicar una carpeta vacía con el nombre que queremos poner al archivo.



Por último, este complemento no es una versión definitiva por lo que, en la pestaña About se indica el nombre y el correo de contacto del desarrollador del plugin para poder reportar algún error ocurrido durante el trabajo.



12. Anexo II. Ayuda del Complemento-Errores

AYUDA TIPOS DE ERRORES

TFM: "Complemento de QGIS para la optimización del control de calidad visual de las ortoimágenes del Instituto Cartográfico Valenciano (ICV)

AUTOR: FRAN VAQUERO CÁCERES

FECHA: JULIO, 2021

INDICE

13.1. Codificación de Errores (ICV)

CODIFICACION DE ERRORES (ICV)	
GEOMETRÍA	
11_DESDOBLAMIENTO_PUENTE	El puente se ha desdoblado
12_CORRIMIENTO_PUENTE	Se ha producido un corrimiento en los pixeles del puente
13_ASERRADO_PUENTE	Pixeles aserrados en los extremos del puente
14_OTRO_PUENTE	
15_ERROR_PRESA	Presa con geometría erronea
16_ERROR_CANTERA	Canteras con algún tipo de error, corrimiento, deformación, etc.
17_CASA DEFORMADA	Deformacion de casa
18_VIA_DEFORMADA	Via tren/carretera deformada
RELIEVE	
21_RELIEVE_CORRIMIENTO	Corrimiento de la Ortofoto
SEAMLINES	
31_SEAMLINE_SOMBRA	Sombras diferentes en una zona de la Seamline
32_SEAMLINE_DIFERENCIA	Diferencia brusca. No existe continuidad
33_SEAMLINE_EDIFICIO	Seamline que pasa por encima de un edificio y existe error
REFLEJOS	
41_REFLEJO_EMBALSE	
42_REFLEJO_PISCINA	
43_REFLEJO_TECHO	
44_REFLEJO_VIA	
RADIOMETRIA	
50_RADIOMETRIA_REFLEJO	
51_RADIOMETRIA_SEAMLINE	Diferencia de color entre los fotogramas por el corte de la Seamline
NUBES	
60_HUMO	
61_NUBE	
CONTINUIDAD	
80_CONTINUIDAD_SOLAPE	Diferencia de continuidad entre las zonas de solape
OTROS	
90_Otro	Otros errores que puedan aparecer

13.2. Ejemplos tipos de errores en ortofotos

Imagen 1. Ejemplo de desdoblamiento en un paso elevado.

Imagen 2. Ejemplo de borde aserrado en un puente.

Imagen 3. Error de pendiente

Imagen 4. Ejemplo de error provocado por la generación de seamlines en un edificio.

Imagen 5. Diferencia de color entre parcelas colindantes ubicadas a cada lado de la seamline

Imagen 6. Ejemplo de reflejo producido por la incidencia de los rayos del sol sobre una superficie metálica.

13. Anexo II. Código del Complemento

14.1. Código fuente del fichero *Grid_Orto.py*

```
        # -*- coding: utf-8 -*-
"""
/***************************************************************************
 Grid_Orto
                                 A QGIS plugin
 Plugin para el control de calidad de Ortofotos
 Generated by Plugin Builder: http://g-sherman.github.io/Qgis-Plugin-Builder/
                              -------------------
        begin                : 2021-07-03
        git sha              : $Format:%H$
        copyright            : (C) 2021 by Fran Vaquero Cáceres
        email                : franvaquero@usal.es
 ***************************************************************************/

/***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/
"""
from qgis.PyQt.QtCore import QSettings, QTranslator, QCoreApplication, Qt
from qgis.PyQt.QtGui import QIcon
from qgis.PyQt.QtWidgets import QAction
# Initialize Qt resources from file resources.py
from .resources import *

# Import the code for the DockWidget
from .Grid_Orto_dockwidget import Grid_OrtoDockWidget
import os.path
import sys
# sys.path.append("C:\Program Files\JetBrains\PyCharm 2018.3.3\debug-eggs\pycharm-debug.egg") # dhl
#sys.path.append("C:\Program Files\JetBrains\PyCharm 2020.3\debug-eggs\pydevd-pycharm.egg") # dhl
#sys.path.append("C:\Program Files\JetBrains\PyCharm 2020.1\debug-eggs\pydevd-pycharm.egg") # dhl
#import pydevd_pycharm


class Grid_Orto:
    """QGIS Plugin Implementation."""

    def __init__(self, iface):
        """Constructor.

        :param iface: An interface instance that will be passed to this class
```

```
            which provides the hook by which you can manipulate the QGIS
            application at run time.
        :type iface: QgsInterface
        """
        # pydevd.settrace('localhost',port=54100,stdoutToServer=True,stderrToServer=True)
        # pydevd_pycharm.settrace('localhost', port=54100, stdoutToServer=True,
stderrToServer=True)

        # Save reference to the QGIS interface
        self.iface = iface

        # initialize plugin directory
        self.plugin_dir = os.path.dirname(__file__)

        # initialize locale
        = QSettings().value('locale/userLocale')[0:2]
        locale_path = os.path.join(
            self.plugin_dir,
            'i18n',
            'Grid_Orto_{}.qm'.format(locale))

        if os.path.exists(locale_path):
            self.translator = QTranslator()
            self.translator.load(locale_path)
            QCoreApplication.installTranslator(self.translator)

        # Declare instance attributes
        self.actions = []
        self.menu = self.tr(u'&Grid_Orto')
        # TODO: We are going to let the user set this up in a future iteration
        self.toolbar = self.iface.addToolBar(u'Grid_Orto')
        self.toolbar.setObjectName(u'Grid_Orto')

        # print "** INITIALIZING Grid_Orto"

        self.pluginIsActive = False
        self.dockwidget = None

    # noinspection PyMethodMayBeStatic
    def tr(self, message):
        """Get the translation for a string using Qt translation API.

        We implement this ourselves since we do not inherit QObject.

        :param message: String for translation.
        :type message: str, QString

        :returns: Translated version of message.
        :rtype: QString
        """
        # noinspection PyTypeChecker,PyArgumentList,PyCallByClass
        return QCoreApplication.translate('Grid_Orto', message)

    def add_action(
            self,
```

```
    icon_path,
    text,
    callback,
    enabled_flag=True,
    add_to_menu=True,
    add_to_toolbar=True,
    status_tip=None,
    whats_this=None,
    parent=None):
"""Add a toolbar icon to the toolbar.

:param icon_path: Path to the icon for this action. Can be a resource
    path (e.g. ':/plugins/foo/bar.png') or a normal file system path.
:type icon_path: str

:param text: Text that should be shown in menu items for this action.
:type text: str

:param callback: Function to be called when the action is triggered.
:type callback: function

:param enabled_flag: A flag indicating if the action should be enabled
    by default. Defaults to True.
:type enabled_flag: bool

:param add_to_menu: Flag indicating whether the action should also
    be added to the menu. Defaults to True.
:type add_to_menu: bool

:param add_to_toolbar: Flag indicating whether the action should also
    be added to the toolbar. Defaults to True.
:type add_to_toolbar: bool

:param status_tip: Optional text to show in a popup when mouse pointer
    hovers over the action.
:type status_tip: str

:param parent: Parent widget for the new action. Defaults None.
:type parent: QWidget

:param whats_this: Optional text to show in the status bar when the
    mouse pointer hovers over the action.

:returns: The action that was created. Note that the action is also
    added to self.actions list.
:rtype: QAction
"""

icon = QIcon(icon_path)
action = QAction(icon, text, parent)
action.triggered.connect(callback)
action.setEnabled(enabled_flag)

if status_tip is not None:
    action.setStatusTip(status_tip)
```

```
        if whats_this is not None:
            action.setWhatsThis(whats_this)

        if add_to_toolbar:
            self.toolbar.addAction(action)

        if add_to_menu:
            self.iface.addPluginToMenu(
                self.menu,
                action)

        self.actions.append(action)

        return action

    def initGui(self):
        """Create the menu entries and toolbar icons inside the QGIS GUI."""

        icon_path = ':/plugins/grid_orto/icon.png'
        self.add_action(
            icon_path,
            text=self.tr(u'Grid_Orto'),
            callback=self.run,
            parent=self.iface.mainWindow())

    # --------------------------------------------------------------------------

    def onClosePlugin(self):
        """Cleanup necessary items here when plugin dockwidget is closed"""

        # print "** CLOSING Grid_Orto"

        # disconnects
        self.dockwidget.closingPlugin.disconnect(self.onClosePlugin)

        # remove this statement if dockwidget is to remain
        # for reuse if plugin is reopened
        # Commented next statement since it causes QGIS crashe
        # when closing the docked window:
        # self.dockwidget = None

        self.pluginIsActive = False

    def unload(self):
        """Removes the plugin menu item and icon from QGIS GUI."""

        # print "** UNLOAD Grid_Orto"

        for action in self.actions:
            self.iface.removePluginMenu(
                self.tr(u'&Grid_Orto'),
                action)
            self.iface.removeToolBarIcon(action)
        # remove the toolbar
```

```
        del self.toolbar

    # --------------------------------------------------------------------------

    def run(self):
        """Run method that loads and starts the plugin"""

        if not self.pluginIsActive:
            self.pluginIsActive = True

            # print "** STARTING Grid_Orto"

            # dockwidget may not exist if:
            #    first run of plugin
            #    removed on close (see self.onClosePlugin method)
            if self.dockwidget == None:
                # Create the dockwidget (after translation) and keep reference
                self.dockwidget = Grid_OrtoDockWidget(self.iface)

            # connect to provide cleanup on closing of dockwidget
            self.dockwidget.closingPlugin.connect(self.onClosePlugin)

            # show the dockwidget
            # TODO: fix to allow choice of dock location
            self.iface.addDockWidget(Qt.LeftDockWidgetArea, self.dockwidget)
            self.dockwidget.show()
```

14.2. Código fuente del fichero Grid_OrtoDockWidget.py

```
# -*- coding: utf-8 -*-
"""
/***************************************************************************
 Grid_OrtoDockWidget
                                 A QGIS plugin
 Plugin para el control de calidad de Ortofotos
 Generated by Plugin Builder: http://g-sherman.github.io/Qgis-Plugin-Builder/
                             -------------------
        begin                : 2021-07-03
        git sha              : $Format:%H$
        copyright            : (C) 2021 by Fran Vaquero Cáceres
        email                : franvaquero@usal.es
 ***************************************************************************/

/***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/
"""
```

```
import os
import webbrowser
from qgis.core import *
from qgis.utils import *
from qgis.PyQt import QtGui, QtWidgets, uic
from PyQt5.QtCore import *
from PyQt5.QtWidgets import QFileDialog, QMessageBox, QInputDialog



pluginsPath = QFileInfo(QgsApplication.qgisUserDatabaseFilePath()).path()
pluginPath = os.path.dirname(os.path.realpath(__file__))
pluginPath = os.path.join(pluginsPath, pluginPath)

FORM_CLASS, _ = uic.loadUiType(os.path.join(
    os.path.dirname(__file__), 'Grid_Orto_dockwidget_base.ui'))
posicion = 1
posicion2000 = 1
posicion1000 = 1


class Grid_OrtoDockWidget(QtWidgets.QDockWidget, FORM_CLASS):

    closingPlugin = pyqtSignal()

    def __init__(self,
                 iface,
                 parent=None):
        """Constructor."""
        super(Grid_OrtoDockWidget, self).__init__(parent)
        # Set up the user interface from Designer.
        # After setupUI you can access any designer object by doing
        # self.<objectname>, and you can use autoconnect slots - see
        # http://doc.qt.io/qt-5/designer-using-a-ui-file.html
        # #widgets-and-dialogs-with-auto-connect
        self.setupUi(self)
        self.setupPlugin()  # Appearance initialization
        self.iface = iface
        self.windowTitle = "Grid_Orto"

        self.getSpatialiteConnections()
        # qs = QSettings()
        # spatialiteConnections = qs.value("SpatiaLite/connections")

        # self.ruta_export= "defaultDirOutputImages"

        # self.qfw_abrirdb.setFilter("Base de datos (*.sqlite)")
        # self.btn_cuad.clicked.connect(self.cuadricula)
        # self.btn_revisado.clicked.connect(self.addrevisar)
        # self.btn_zoomselect.clicked.connect(self.zoomselect)
        # self.btn_siguiente_zoom.clicked.connect(self.nextzoom)

        """ Dashboard widget event connectors """
        self.btn_crearproyecto.clicked.connect(self.crear_proyecto)
        self.btn_agregarcapas.clicked.connect(self.capas)
```

```
        self.btn_zoom2000.clicked.connect(self.zoom2000)
        self.btn_zoom1000.clicked.connect(self.zoom1000)
        self.btn_error.clicked.connect(self.error)
        self.btn_help.clicked.connect(self.openHelp)
        self.btn_aceptar.clicked.connect(self.aceptar)
        self.btn_icv_logo.clicked.connect(self.icvweb)
        self.btn_ayudaerror.clicked.connect(self.helperror)

        """self.toolBox.currentChanged.connect(self.onToolBoxChanged)"""

    def setupPlugin(self):
        baseDirectory = os.path.dirname(os.path.realpath(__file__))  # Plugin absolute path
        self.btn_icv_logo.setIcon(QtGui.QIcon(os.path.dirname(os.path.realpath(__file__)) +
'/logo_icv.png'))
        self.btn_help.setIcon(QtGui.QIcon(os.path.dirname(os.path.realpath(__file__)) +
'/help.png'))
        self.btn_ayudaerror.setIcon(QtGui.QIcon(os.path.dirname(os.path.realpath(__file__)) +
'/errores.png'))

    def icvweb(self):
        webbrowser.open('http://www.icv.gva.es/es/inicio')

    def openHelp(self):
        pathInfoDoc = os.path.dirname(__file__) + '/help/Flujo_Trabajo.pdf'
        os.startfile(pathInfoDoc)

    def helperror(self):
        pathInfoDoc = os.path.dirname(__file__) + '/help/Ayuda_tipos_errores.pdf'
        os.startfile(pathInfoDoc)

    def crear_proyecto(self, event):
        ruta_sqlite = 'SQLITE/CQ_Orto.sqlite'
        plugin_dir = pluginPath
        ruta_guardar_sqlite = 'C:/temp/CQ_Orto.sqlite'
        path_TemplateDb = os.path.normcase(os.path.join(plugin_dir, ruta_sqlite))

        # db_template.open()
        if QFile.exists(ruta_guardar_sqlite):
            if not QFile.remove(ruta_guardar_sqlite):
                strError = "Existe el archivo en:\n" + ruta_guardar_sqlite
                msgBox = QMessageBox(self)
                msgBox.setIcon(QMessageBox.Information)
                msgBox.setWindowTitle(self.windowTitle)
                msgBox.setText("Error:\n" + strError)
                msgBox.exec_()
                return
        QFile.copy(path_TemplateDb, ruta_guardar_sqlite)

        if not QFile.exists(ruta_guardar_sqlite):
            strError = "No existe el directorio:\n" + ruta_guardar_sqlite
            msgBox = QMessageBox(self)
            msgBox.setIcon(QMessageBox.Information)
            msgBox.setWindowTitle(self.windowTitle)
            msgBox.setText("Error:\n" + strError)
            msgBox.exec_()
```

```
            return

        connectionName = QFileInfo(ruta_guardar_sqlite).fileName()
        con = [connectionName, ruta_guardar_sqlite]
        QSettings().setValue("SpatiaLite/connections/%s/sqlitepath" % (con[0]), con[1])
        self.iface.reloadConnections()
        self.getSpatialiteConnections()
        msgBox = QMessageBox(self)
        msgBox.setIcon(QMessageBox.Information)
        msgBox.setWindowTitle(self.windowTitle)
        msgBox.setText("Process completed successfully")
        msgBox.exec_()

    def getSpatialiteConnections(self):
        self.connections = {}
        settings = QSettings()
        settings.beginGroup('/SpatiaLite/connections')
        list_str_keys = settings.allKeys()
        paths = []
        for key in list_str_keys:
            if key != 'selected':
                paths.append(settings.value(key))
        connectionNames = settings.childGroups()
        cont = 0
        for connectionName in connectionNames:
            path = paths[cont]
            self.connections[connectionName] = paths[cont]
            cont = cont + 1

        yo = 1

    def capas(self, event):
        layer = iface.addVectorLayer("/temp/CQ_Orto.sqlite", "Capas para la revision",
"ogr")
        if not layer or not layer.isValid():
            print("Layer failed to load!")

        # Asignación de estilos a las cuadriculas

        ruta_qml5000 = 'qml/cuadriculas_ortofoto_mdt_5000_25830_icv_round_10m.qml'
        ruta_qml2000 = 'qml/cuadriculas_zoom2000_rev.qml'
        ruta_qml1000 = 'qml/cuadriculas_zoom1000_rev.qml'
        ruta_qmlerrores = 'qml/errores.qml'

        plugin_dir = pluginPath
        qml_path5000 = os.path.normcase(os.path.join(plugin_dir, ruta_qml5000))
        qml_path2000 = os.path.normcase(os.path.join(plugin_dir, ruta_qml2000))
        qml_path1000 = os.path.normcase(os.path.join(plugin_dir, ruta_qml1000))
        qml_patherrores = os.path.normcase(os.path.join(plugin_dir, ruta_qmlerrores))

        for layer in QgsProject.instance().mapLayersByName("CQ_Orto
cuadriculas_ortofoto_mdt_5000_25830_icv_round_10m"):
            layer.loadNamedStyle(qml_path5000)

        for layer in QgsProject.instance().mapLayersByName("CQ_Orto
```

```
cuadriculas_zoom2000_rev"):
        layer.loadNamedStyle(qml_path2000)

        # Labels
        layer_settings = QgsPalLayerSettings()
        text_format = QgsTextFormat()
        text_format.setSize(8)

        layer_settings.fieldName = "hoja"
        layer_settings.enabled = True
        layer_settings = QgsVectorLayerSimpleLabeling(layer_settings)
        layer.setLabelsEnabled(True)
        layer.setLabeling(layer_settings)
        layer.triggerRepaint()

    for layer in QgsProject.instance().mapLayersByName("CQ_Orto
cuadriculas_zoom1000_rev"):
        layer.loadNamedStyle(qml_path1000)

        # Labels
        layer_settings = QgsPalLayerSettings()
        text_format = QgsTextFormat()
        text_format.setSize(8)

        layer_settings.fieldName = "hoja"
        layer_settings.enabled = True
        layer_settings = QgsVectorLayerSimpleLabeling(layer_settings)
        layer.setLabelsEnabled(True)
        layer.setLabeling(layer_settings)
        layer.triggerRepaint()

    for layer in QgsProject.instance().mapLayersByName("CQ_Orto errores"):
        layer.loadNamedStyle(qml_patherrores)

        # Labels
        layer_settings = QgsPalLayerSettings()
        text_format = QgsTextFormat()
        text_format.setSize(8)

        layer_settings.fieldName = "hoja"
        layer_settings.enabled = True
        layer_settings = QgsVectorLayerSimpleLabeling(layer_settings)
        layer.setLabelsEnabled(True)
        layer.setLabeling(layer_settings)
        layer.triggerRepaint()

    # A continuación, busca en la capa cuadriculas_zoom2000 el primer valor con el campo
visitado=0
    global posicion
    global posicion2000
    layer = QgsProject.instance().mapLayersByName("CQ_Orto
cuadriculas_zoom2000_rev")[0]
    # layer = self.iface.activeLayer()
    # iface.setActiveLayer(layer)
```

```
        if layer != None:
            layer.selectByExpression("\"visitado\"=" + str("0"))
            if layer.selectedFeatureCount() > 0:
                features2000 = layer.selectedFeatures()
                if layer.selectedFeatureCount() > 0:
                    feature = features2000[0]
                    posicion = feature.attribute("orden_5000")
                    posicion2000 = feature.attribute("orden_2000")

        layer.selectByExpression("\"visitado\"=" + str("0"),
QgsVectorLayer.RemoveFromSelection)


    def zoom2000(self):
        global posicion
        global posicion2000
        layer = QgsProject.instance().mapLayersByName("CQ_Orto
cuadriculas_zoom2000_rev")[0]
        iface.setActiveLayer(layer)

        if layer != None:
            layer.startEditing()
            layer.selectByExpression("\"Orden_5000\"=" + str(posicion))

            if layer.selectedFeatureCount() > 0:
                layer.selectByExpression("\"Orden_5000\"=" + str(posicion) + " and
\"Orden_2000\"=" + str(
                    posicion2000) + "and \"visitado\"=" + str("0"),
QgsVectorLayer.IntersectSelection)
                posicion2000 = posicion2000 + 1
                features2000 = layer.selectedFeatures()
                iface.mapCanvas().zoomToSelected(layer)

                if layer.selectedFeatureCount() == 0:
                    msg = "Hoja 5000 terminada"  # Podria poner que hoja es la que se ha
terminado
                    QMessageBox.information(iface.mainWindow(), "Selección ", msg)
                    posicion = posicion + 1
                    posicion2000 = 1

                if layer.selectedFeatureCount() > 0:
                    feature = features2000[0]
                    feature["visitado"] = "1"
                    layer.updateFeature(feature)
                    # layer.commitChanges()
                    # Para guardar automaticamente los cambios es: layer.commitChanges()

                # else:
                #     msg = "No hay nada seleccionado"
                #     QMessageBox.information(iface.mainWindow(), "Selección ", msg)

            else:
                posicion2000 = posicion2000 + 1
                msg = "No hay nada seleccionado"
                QMessageBox.information(iface.mainWindow(), "Selección ", msg)
```

```
        else:
            str_msg = "No se ha encontrado capa activa"
            iface.messageBar().pushMessage("Info", str_msg, level=Qgis.Warning, duration=15)

    def zoom1000(self):
        global posicion
        layer = QgsProject.instance().mapLayersByName("CQ_Orto
cuadriculas_zoom1000_rev")[0]
        iface.setActiveLayer(layer)

        if layer != None:
            layer.startEditing()
            global posicion2000
            layer.selectByExpression("\"Orden_5000\"=" + str(posicion))

            if layer.selectedFeatureCount() > 0:
                global posicion1000
                layer.selectByExpression("\"Orden_2000\"=" + str(posicion2000 - 1),
QgsVectorLayer.IntersectSelection)
                layer.selectByExpression("\"Orden_1000\"=" + str(posicion1000),
QgsVectorLayer.IntersectSelection)
                iface.mapCanvas().zoomToSelected(layer)
                posicion1000 = posicion1000 + 1
                features1000 = layer.selectedFeatures()

                if layer.selectedFeatureCount() == 0:
                    msg = "Hoja 2000 terminada"
                    QMessageBox.information(iface.mainWindow(), "Selección ", msg)
                    layer.selectByExpression("\"Orden_2000\"=" + str(posicion2000 - 1),
                                             QgsVectorLayer.IntersectSelection)
                    posicion2000 = posicion2000
                    posicion1000 = 1

                if layer.selectedFeatureCount() > 0:
                    feature = features1000[0]
                    feature.setAttribute("visitado", "1")
                    feature["visitado"] = "1"
                    layer.updateFeature(feature)
                    # layer.commitChanges()

                # else:
                #     msg = "No hay nada seleccionado"
                #     QMessageBox.information(iface.mainWindow(), "Selección ", msg)



            else:
                msg = "No hay nada seleccionado"
                QMessageBox.information(iface.mainWindow(), "Selección ", msg)


        else:
            str_msg = "No se ha encontrado capa activa"
            # iface.messageBar().pushMessage("Info", str_msg, level=Qgis.Warning, duration=15)
```

```
    def error(self):
        layer = QgsProject.instance().mapLayersByName("CQ_Orto errores")[0]
        iface.setActiveLayer(layer)

        fieldIndex = layer.fields().indexFromName('tipo')
        editor_widget_setup = QgsEditorWidgetSetup('ValueMap', {
            'map': {'11_Desdoblamiento_puente': '11_Desdoblamiento_puente',
                    '12_Corrimiento_Puente': '12_Corrimiento_Puente',
                    '13_Aserrado_Puente': '13_Aserrado_Puente',
                    '14_Otro_puente': '14_Otro_puente',
                    '15_Error_presa': '15_Error_presa',
                    '16_Error_Cantera': '16_Error_Cantera',
                    '17_Casa_Deformada': '17_Casa_Deformada',
                    '18_Via_deformada': '18_Via_deformada',
                    '21_Relieve_corrimiento': '21_Relieve_corrimiento',
                    '22_Relieve_Otro': '22_Relieve_Otro',
                    '31_Seamline_Sombra': '31_Seamline_Sombra',
                    '32_Seamline_Diferencia': '32_Seamline_Diferencia',
                    '33_Seamline_Edificio': '33_Seamline_Edificio',
                    '34_Seamline_Otro': '34_Seamline_Otro',
                    '41_Reflejos_Embalse': '41_Reflejos_Embalse',
                    '42_Reflejos_Piscina': '42_Reflejos_Piscina',
                    '43_Reflejos_Techo': '43_Reflejos_Techo',
                    '44_Reflejos_Via': '44_Reflejos_Via',
                    '45_Reflejos_Otro': '45_Reflejos_Otro',
                    '50_Radiometria_Reflejos': '50_Radiometria_Reflejos',
                    '51_Radiometria_Seamline': '51_Radiometria_Seamline',
                    '60_Humo': '60_Humo',
                    '61_Nube': '61_Nube',
                    '80_Continuidad_Solape': '80_Continuidad_Solape',
                    '90_Otro': '90_Otro',

                    }
        }
                                    )
        layer.setEditorWidgetSetup(fieldIndex, editor_widget_setup)

        fieldIndex = layer.fields().indexFromName('corregido')
        editor_widget_setup = QgsEditorWidgetSetup('ValueMap', {
            'map': {'Si': 'Si',
                    'No': 'No',
                    }
        }
                                    )
        layer.setEditorWidgetSetup(fieldIndex, editor_widget_setup)
        layer.startEditing()
        iface.actionAddFeature().trigger()

    def aceptar(self):
        nombre = QInputDialog.getText(None, 'NOMBRE', 'Introduce el nombre del
operador')
        Dir = QFileDialog.getExistingDirectory(self, "Seleccionar directorio de salida vacia",
"C:/temp")

        layer = QgsProject.instance().mapLayersByName("CQ_Orto errores")[0]
```

```
        iface.setActiveLayer(layer)

        _writer = QgsVectorFileWriter.writeAsVectorFormat(layer, Dir + '_' + nombre[0], 'utf-8',
                                                          driverName='ESRI Shapefile')

        msg = "Capa guardada correctamente"
        QMessageBox.information(iface.mainWindow(), "Guardar ", msg)

    def closeEvent(self, event):
        self.closingPlugin.emit()
        event.accept()
```

14. Anexo III. Enlace a video de caso de uso

Como hemos visto en el apartado 7 la creación de un repositorio en GitHub, se ha creado un video con la plataforma *"Loom"* y se adjunta en el README del repositorio donde aparecen los pasos a seguir desde la instalación hasta el flujo de trabajo que tiene que seguir el usuario.

A continuación, dejo el enlace al video:

https://www.loom.com/share/e2f8fa809a8b48f994fc917ae9000351