



POLITÉCNICA

UNIVERSIDAD POLITÉCNICA DE MADRID

**ESCUELA TÉCNICA SUPERIOR
DE INGENIERÍA INDUSTRIAL**

**TRABAJO DE FIN DE GRADO EN INGENIERÍA DE
LA ENERGÍA**

E18089

**CREACIÓN DE DATOS SEMÁNTICOS A
PARTIR DE STREAM DE DATOS DE
SEGUIMIENTO DE VEHICULOS.**



Autor/a:

ADRIÁN TESO CAMPILLO

Tutor/a:

**JOAQUÍN BIENVENIDO ORDIERES
MERE**



Febrero 2022, MADRID.

AGRADECIMIENTOS

En primer lugar, encabezando el documento me gustaría agradecer a mi tutor, Joaquín Bienvenido Ordieres Mere, el privilegio de poder realizar un proyecto en el campo de la programación.

Joaquín me ha sabido guiar con paciencia y amabilidad ante la incertidumbre y el desafío que genera enfrentarse a una hoja en blanco. Desde que me admitió como su tutelado ha respondido a mis demandas con total atención y consideración, me ha dado los ánimos tan necesarios en ciertas situaciones en las que me costaba seguir a delante con el proyecto. Le agradezco mucho la fe en mí, en que voy a encontrar la manera de crear el proceso.

Me gustaría agradecer a mis padres el amor y el apoyo, este proyecto no hubiese sido posible sin ellos ni mi hermano. No ha habido día en el que mi familia no se sintiera orgullosa de mí y de mis logros. Gracias por que no me haya faltado nada en mi vida y gracias por aportarme lo necesario para llegar donde estoy. Gracias Papá, gracias Mamá, gracias Nico.

Agradecer a mi pareja, Marina, llenarme el corazón día tras día, año tras año. Has estado a mi lado conociendo mi sudor con este proyecto, sintiéndote orgullosa de mí y recordándome por qué lo hago. Eres lo más maravilloso que existe. Gracias por tanto amor y tanto apoyo, y por hacer legibles mis textos. ...las funcionalidades funcionan a base de las funciones que... gracias. Te mereces que te dedique todos mis proyectos.

Estoy muy agradecido a Rami, por tanto amor y tanto apoyo. Hemos estado juntos en la carrera desde prácticamente el primer día. Hemos crecido y hemos estudiado hasta la perder facultades juntos. No podía haber tenido un mejor compañero, en los años de carrera en los que he estado contigo no te he oído quejarte ni una sola vez, y aunque tengas motivos para ello, tú me has enseñado que la vida es como es, a veces injusta y no se puede hacer más que dar lo mejor de ti para superarla. Eres una persona admirable en todos los sentidos, amigo. Ni los itinerarios han logrado separarnos. Gracias.

Gracias a Ricar, Claudia y Javi, sin vosotros no podría haber programado ni el “*hola mundo*” del proyecto. Gracias por vuestra devoción para ayudarme de manera tan desinteresada. Siempre me habéis ofrecido la ayuda que tanto he necesitado de vosotros.

Quiero mencionar a Musta y a Frankie, gracias chicos por haberme acompañado en el transcurso de mi carrera, habéis sido un apoyo tan grande como vuestros corazones.

Quiero mencionar a Antonio, gracias por todo lo que has hecho por mí, mucha suerte aventurero.

Gracias a mis tutores profesionales, Ernesto y Juanma los cuales me han guiado en mis prácticas, de los cuales he aprendido una infinidad de cosas, gracias.

Gracias a Adri, Andrea, Jimbo, Hernán, Isa, Lucy, Cabrero, Lucas, Azul, Rebe, Anto, Bego, Irene, Fabri, Alba, Raúl, Abril, Valla, Jorge, Julián, Chuso, Julia, Lola, Cape, Villa, Carlos, gracias a una lista interminable de personas que quiero mencionar que me han alegrado el corazón en mi transcurso por la carrera, gracias.

Me quiero acordar de Jose, Clara y Edi, Laia y Carla, gracias por recibirme con los brazos abiertos y alegrarme el corazón.

Me quiero acordar de Sergio, te conozco desde hace poco, pero me has aportado mucho, tanto para realizar este proyecto como en mi vida personal.

Gracias a todos por todo el apoyo y amor que me habéis dado.

A Nana, a Lalo, a Lucky...

ÍNCICE GENERAL

| | |
|---------------------------------------|-----|
| AGRADECIMIENTOS..... | 3 |
| ÍNCICE GENERAL..... | 7 |
| ÍNDICE DE ILUSTRACIONES..... | 9 |
| ABREVIATURAS..... | 13 |
| RESUMEN | 14 |
| ABSTRACT..... | 16 |
| PARTE 1 | 18 |
| CAPÍTULO 1 | 19 |
| 1.1. OBJETIVOS Y ALCANCE | 20 |
| 1.2. MOTIVACION..... | 21 |
| 1.3. ANTECEDENTES | 21 |
| 1.4. VIABILIDAD | 22 |
| 1.5. ESTRUCTURA | 22 |
| CAPÍTULO 2 | 23 |
| 2.1. Main. | 24 |
| 2.2. Database..... | 38 |
| CAPÍTULO 3 | 66 |
| 3.1. Introducción. | 67 |
| 3.2. clust.py. | 69 |
| 3.3. SP20.py. | 81 |
| 3.4. add_toSQL.py. | 87 |
| CAPÍTULO 4 | 94 |
| 4.1. Introducción. | 95 |
| 4.2. segments.py. | 97 |
| 4.3. seg_toSQL.py. | 109 |
| 4.4. route_toSQL.py. | 121 |
| CAPÍTULO 5 | 132 |
| 5.1. Introducción. | 133 |
| 5.2. Map all SP. | 134 |
| 5.2. Map route/seg selection. | 138 |
| 5.3. Scatter Search..... | 155 |
| 5.4. Search in Database..... | 159 |
| 5.5. Reset DDBB..... | 161 |

| | |
|---------------------------------|-----|
| PARTE 2 | 163 |
| PROGRAMACIÓN DEL PROYECTO | 164 |
| PRESUPUESTACIÓN | 166 |
| LÍNEAS FUTURAS | 168 |
| BIBLIOGRAFÍA..... | 169 |

ÍNDICE DE ILUSTRACIONES

| | |
|--|----|
| <i>Ilustración 1 main_imports.</i> | 24 |
| <i>Ilustración 2 sc_1.</i> | 25 |
| <i>Ilustración 3 sc_2.</i> | 26 |
| <i>Ilustración 4 sc_3.</i> | 27 |
| <i>Ilustración 5 sc_4.</i> | 27 |
| <i>Ilustración 6 sc_5.</i> | 28 |
| <i>Ilustración 7 sc_6.</i> | 28 |
| <i>Ilustración 8 sc_7.</i> | 29 |
| <i>Ilustración 9 sc_8.</i> | 29 |
| <i>Ilustración 10 main_loop.</i> | 30 |
| <i>Ilustración 11 functionalities_menu_loop.</i> | 31 |
| <i>Ilustración 12 Routing_menu_loop.</i> | 32 |
| <i>Ilustración 13 Clustering_SP_loop.</i> | 33 |
| <i>Ilustración 14 Mapping_loop.</i> | 34 |
| <i>Ilustración 15 Scatter_Search_loop.</i> | 35 |
| <i>Ilustración 16 Search_in_Database.</i> | 36 |
| <i>Ilustración 17 Reset_DDBB_loop.</i> | 36 |
| <i>Ilustración 18 Importaciones_database.py</i> | 38 |
| <i>Ilustración 19 temporalvar_database.</i> | 39 |
| <i>Ilustración 20 strvar_database.</i> | 39 |
| <i>Ilustración 21 init_database.</i> | 40 |
| <i>Ilustración 22 select_unique_user_func.</i> | 41 |
| <i>Ilustración 23 call_example_database.</i> | 41 |
| <i>Ilustración 24 select_all_func.</i> | 43 |
| <i>Ilustración 25 examp_f_func.</i> | 44 |
| <i>Ilustración 26 examp_+_func.</i> | 44 |
| <i>Ilustración 27 examp_sintax_insertora_database.</i> | 44 |
| <i>Ilustración 28 examp_delete_func.</i> | 45 |
| <i>Ilustración 29 movements_table.</i> | 46 |
| <i>Ilustración 30 specialsites_table.</i> | 48 |
| <i>Ilustración 31 segments_table.</i> | 49 |
| <i>Ilustración 32 routes_table.</i> | 51 |
| <i>Ilustración 33 select_last_month_func.</i> | 52 |
| <i>Ilustración 34 select_dayselect_func.</i> | 53 |
| <i>Ilustración 35 select_users_20/30_func.</i> | 53 |
| <i>Ilustración 36 select_users_compr_func.</i> | 54 |
| <i>Ilustración 37 select_users_past_func.</i> | 54 |
| <i>Ilustración 38 remove_allSP_func.</i> | 55 |
| <i>Ilustración 39 addSp_func.</i> | 55 |
| <i>Ilustración 40 select_sp_func.</i> | 56 |
| <i>Ilustración 41 select_sp_name_func.</i> | 56 |
| <i>Ilustración 42 select_mat_0_func.</i> | 57 |
| <i>Ilustración 43 select_seg_func.</i> | 57 |
| <i>Ilustración 44 select_data_forseg_func.</i> | 58 |

| | |
|---|-----|
| <i>Ilustración 45 select_unique_sp_func.</i> | 58 |
| <i>Ilustración 46 add_seg_toSQL_func.</i> | 59 |
| <i>Ilustración 47 select_data_from_route_func.</i> | 59 |
| <i>Ilustración 48 add_route_toSQL_func.</i> | 60 |
| <i>Ilustración 49 select_mat_fromroutes_func.</i> | 60 |
| <i>Ilustración 50 select_data_fromsegments.</i> | 61 |
| <i>Ilustración 51 select_data_frommovements_func.</i> | 61 |
| <i>Ilustración 52 remove_route_func.</i> | 63 |
| <i>Ilustración 53 remove_seg_func.</i> | 63 |
| <i>Ilustración 54 remove_func.</i> | 64 |
| <i>Ilustración 55 connection_close_func.</i> | 65 |
| <i>Ilustración 56 cluster_definition.</i> | 67 |
| <i>Ilustración 57 users_list.</i> | 69 |
| <i>Ilustración 58 eps_correction.</i> | 69 |
| <i>Ilustración 59 DBSCAN.</i> | 70 |
| <i>Ilustración 60 db_cores.</i> | 70 |
| <i>Ilustración 61 clust_selection.</i> | 71 |
| <i>Ilustración 62 clust_imports.</i> | 73 |
| <i>Ilustración 63 clustering_20.</i> | 74 |
| <i>Ilustración 64 clusteo_1.</i> | 75 |
| <i>Ilustración 65 clusteo_1_zoom_1.</i> | 75 |
| <i>Ilustración 66 progresión_zoom_clust_1.</i> | 76 |
| <i>Ilustración 67 sec_depur_func.</i> | 77 |
| <i>Ilustración 68 graf_sec_depur.</i> | 77 |
| <i>Ilustración 69 sec_depur_grafic.</i> | 78 |
| <i>Ilustración 70 clustering_past_func.</i> | 79 |
| <i>Ilustración 71 clustering_func.</i> | 80 |
| <i>Ilustración 72 SP20_imports.</i> | 81 |
| <i>Ilustración 73 date_selection.</i> | 82 |
| <i>Ilustración 74 manual_func.</i> | 83 |
| <i>Ilustración 75 manual_func_process.</i> | 84 |
| <i>Ilustración 76 auto_func.</i> | 85 |
| <i>Ilustración 77 twoyears_input_func.</i> | 87 |
| <i>Ilustración 78 ty_input_func_str.</i> | 88 |
| <i>Ilustración 79 ty_input_progress.</i> | 89 |
| <i>Ilustración 80 month_input_func_1.</i> | 90 |
| <i>Ilustración 81 month_input_func_2.</i> | 91 |
| <i>Ilustración 82 month_input_func_3.</i> | 91 |
| <i>Ilustración 83 month_input_process.</i> | 92 |
| <i>Ilustración 84 segment_speed.</i> | 95 |
| <i>Ilustración 85 segment.py_imports.</i> | 97 |
| <i>Ilustración 86 segments_func_1.</i> | 98 |
| <i>Ilustración 87 secments_func_2.</i> | 98 |
| <i>Ilustración 88 segments_func_3.</i> | 99 |
| <i>Ilustración 89 segments_func_4.</i> | 100 |
| <i>Ilustración 90 segments_func_5.</i> | 101 |
| <i>Ilustración 91 segments_func_6.</i> | 102 |

| | |
|--|-----|
| <i>Ilustración 92 segments_func_7.</i> | 103 |
| <i>Ilustración 93 segment_func_8.</i> | 104 |
| <i>Ilustración 94 segments_func_9.</i> | 104 |
| <i>Ilustración 95 segments_func_10.</i> | 105 |
| <i>Ilustración 96 segments_process_1.</i> | 106 |
| <i>Ilustración 97 segments_process_2.</i> | 107 |
| <i>Ilustración 98_map_1.</i> | 108 |
| <i>Ilustración 99 seg_toSQL_imports.</i> | 110 |
| <i>Ilustración 100 seg_toSQL_func_1.</i> | 110 |
| <i>Ilustración 101 seg_toSQL_func_2.</i> | 112 |
| <i>Ilustración 102 index_out_of_range_error.</i> | 112 |
| <i>Ilustración 103 seg_toSQL_func_3.</i> | 113 |
| <i>Ilustración 104 seg_toSQL_func_4.</i> | 114 |
| <i>Ilustración 105 seg_toSQL_func_5.</i> | 114 |
| <i>Ilustración 106 seg_toSQL_func_6.</i> | 114 |
| <i>Ilustración 107108 seg_toSQL_func_7.</i> | 115 |
| <i>Ilustración 109 seg_toSQL_func_8.</i> | 115 |
| <i>Ilustración 110 seg_toSQL_func_9.</i> | 117 |
| <i>Ilustración 111 seg_toSQL_func_10.</i> | 117 |
| <i>Ilustración 112 seg_toSQL_func_11.</i> | 118 |
| <i>Ilustración 113 seg_toSQL_process.</i> | 119 |
| <i>Ilustración 114 seg_toSQL_process_2.</i> | 119 |
| <i>Ilustración 115 segment_map.</i> | 120 |
| <i>Ilustración 116 route_toSQL_imports.</i> | 122 |
| <i>Ilustración 117 route_toSQL_func_1.</i> | 122 |
| <i>Ilustración 118 Debug.</i> | 123 |
| <i>Ilustración 119 Interruption.</i> | 124 |
| <i>Ilustración 120 data.</i> | 124 |
| <i>Ilustración 121 data_shorted.</i> | 124 |
| <i>Ilustración 122 data_array.</i> | 124 |
| <i>Ilustración 123 route_toSQL_func_2.</i> | 124 |
| <i>Ilustración 124 route_toSQL_func_3.</i> | 125 |
| <i>Ilustración 125 route_toSQL_func_4.</i> | 125 |
| <i>Ilustración 126 route_toSQL_func_5.</i> | 126 |
| <i>Ilustración 127 route_toSQL_func_6.</i> | 127 |
| <i>Ilustración 128 route_1.</i> | 128 |
| <i>Ilustración 129 route_1_vis.</i> | 128 |
| <i>Ilustración 130 route_2.</i> | 129 |
| <i>Ilustración 131 route_2_vis.</i> | 129 |
| <i>Ilustración 132 route_3.</i> | 130 |
| <i>Ilustración 133 route_3_vis.</i> | 130 |
| <i>Ilustración 134 route_3_vis_back.</i> | 131 |
| <i>Ilustración 135 map_imports.</i> | 134 |
| <i>Ilustración 136 mapping_func.</i> | 135 |
| <i>Ilustración 137 SPs_Spain.</i> | 136 |
| <i>Ilustración 138 SPs_Hungary.</i> | 136 |
| <i>Ilustración 139 map.html.</i> | 137 |

| | |
|---|-----|
| <i>Ilustración 140 map.html_markers.</i> | 137 |
| <i>Ilustración 141 visualization_imports.</i> | 138 |
| <i>Ilustración 142 visualization_func_1.</i> | 139 |
| <i>Ilustración 143 visualization_func_2.</i> | 140 |
| <i>Ilustración 144 visualizarion_func_3.</i> | 141 |
| <i>Ilustración 145 visualization_func_4.</i> | 142 |
| <i>Ilustración 146 visualization_func_5.</i> | 143 |
| <i>Ilustración 147 visualization_func_6.</i> | 143 |
| <i>Ilustración 148 visualization_func_7.</i> | 145 |
| <i>Ilustración 149 visualization_func_8.</i> | 146 |
| <i>Ilustración 150 visualization_func_8.</i> | 147 |
| <i>Ilustración 151 visualization_func_9.</i> | 149 |
| <i>Ilustración 152 visualization_func_10.</i> | 149 |
| <i>Ilustración 153 visualization_process_1.</i> | 150 |
| <i>Ilustración 154 visualization_process_2.</i> | 150 |
| <i>Ilustración 155 visualization_process_3.</i> | 151 |
| <i>Ilustración 156 visualization_process_4.</i> | 151 |
| <i>Ilustración 157 visualizacion_process_5.</i> | 152 |
| <i>Ilustración 158 visualization_process_6.</i> | 153 |
| <i>Ilustración 159 visualization_process_7.</i> | 154 |
| <i>Ilustración 160 scattering_imports.</i> | 155 |
| <i>Ilustración 161 sactter_last_month_func.</i> | 156 |
| <i>Ilustración 162 scatter_dayselect_func_1.</i> | 157 |
| <i>Ilustración 163 scatter_dayselect_2.</i> | 157 |
| <i>Ilustración 164 scatter_1.</i> | 158 |
| <i>Ilustración 165 scatter_2.</i> | 158 |
| <i>Ilustración 166 search_func.</i> | 159 |
| <i>Ilustración 167 show_func.</i> | 159 |
| <i>Ilustración 168 search_1.</i> | 160 |
| <i>Ilustración 169 search_2.</i> | 160 |
| <i>Ilustración 170 resets_file.</i> | 161 |
| <i>Ilustración 171 Programación del proyecto.</i> | 164 |
| <i>Ilustración 172 Diagrama de Gantt.</i> | 165 |

ABREVIATURAS

- SP: Special Point.
- SPs: Special Points.
- BBDD: Base de Datos.
- eps: Épsilon.
- mpts: Mínimum Points.
- DDBB: Data Base.
- SC: Screen.

RESUMEN

El documento presente comprende de la construcción de la herramienta Python de control de flotas, la cual cuenta con dos funcionalidades principales: el minado de SPs, caracterizados por la formación de clústeres por parte de los vehículos de la flota; y la detección y trazado de rutas y segmentos de los vehículos para su control y análisis.

Se definen las funcionalidades como el resultado de las acciones finales realizadas por los procesos de la herramienta.

Todas las funcionalidades de la herramienta vienen provistas por la BBDD de los registros provenientes del sistema de control de flotas, a proveer por la empresa logística, y todas las funcionalidades vuelcan sus cálculos y análisis a la misma BBDD. Las funciones encargadas de realizar estas acciones son recopiladas en el mismo archivo, en la clase de DataBase.

La metodología que se usa pues en el documento para mostrar el trabajo realizado es explicar el algoritmo creado para la herramienta, así como su funcionamiento, propósitos y sinergia con otras partes de la herramienta, mediante la ilustración del código y su explicación. La parte *cursiva* del documento refiere a la parte del código que se plasma en el texto.

La herramienta presenta una interfaz desde la cual se permite acceder de forma clara a todas las funcionalidades, por la cual se navega mediante la introducción de caracteres numéricos y permite la introducción de los parámetros pertinentes.

La herramienta saca el máximo partido a las funciones que comprende, salva de los errores esperados y protege al usuario de un cierre de aplicación producido ante una mala entrada de los parámetros de elección.

Como primera funcionalidad principal se presenta el minado de SPs a partir de la formación de clústeres, los cuales constan de áreas con alta densidad de registros.

Estos clústeres vienen dados por las posiciones de los vehículos, evaluando su proximidad y cantidad de registros en un periodo de tiempo definido.

El SP se definirá como un punto representativo del clúster formado, el cual se somete a un análisis exhaustivo mediante depuraciones para su agregación a la BBDD.

Como segunda funcionalidad principal se presenta la detección y el trazado de rutas de vehículos en un periodo definido.

Estas rutas vienen dadas por la suma de segmentos definidos por cada vehículo. Los segmentos son el trazado de las posiciones de un vehículo a lo largo del tiempo. En los segmentos de ruta se recopila la información pertinente para el trazado de la ruta, junto con la información pertinente para su estudio.

Además de las funcionalidades principales, la herramienta incorpora funcionalidades de visualizado, mediante un API de Google Maps, que se abre de forma automática como una nueva pestaña en el navegador predeterminado.

La herramienta también incorpora la funcionalidad de manejo de la BBDD, mediante una introducción directa de consultas SQL. El manejo de esta funcionalidad debe realizarse con el conocimiento de la sintaxis, puesto que se expone a la capacidad de modificar las tablas para las cuales se posee derecho de escritura.

Este documento presenta la construcción de la herramienta y sus funcionalidades, explicando de la forma más detallada posible en qué consiste cada una de ellas, cómo funciona el proceso de cada archivo y la sinergia que tiene con otras funcionalidades o archivos.

Para el funcionamiento adecuado de la aplicación será necesario tener instalado en el PC Python 3.9, con su interpretador pertinente, además de las librerías importadas de los archivos.

ABSTRACT

This document presents the realization of a fleet control Python tool which owns two main functionalities: SP mining, characterised by cluster formations of the set of vehicles; and route and segment tracing of the vehicles with its analysis.

Its defined functionalities as the result of the final actions performed by the processes of the Python tool.

All functionalities of this tool come provided by the DDBB of the fleet control provided by the logistics company, and all functionalities deposit their data into the same DDBB. All these functions which perform data base actions are collected in the same file, into the Database Class.

The methodology used in this document for showing the project is to explain the algorithm, as its operation, purposes and synergy with other parts of the tool, by the illustration of the code and its explain. The *cursive* part of the document refers to the code part that shows the document text.

The tool presents an interface from which the user can access to all functionalities. The user browses through the interface by the introduction of numeric characters and here, at the interface, is where are introduced the relevant parameters.

The tool offers the maximum value of its own functions, and it saves the user from expected errors, and the app closing because of a wrong selection parameter entry.

As first main functionality, it is introduced the SPs mining from clusters formation.

The SP will be defined as a representative point of the cluster.

This clusters are come from vehicle position data, valuing their proximity and records density into a defined time.

The SP will be submitted to an exhaustive analysis through debugs for its insertion into de DDBB.

As second main functionality, it is introduced the vehicle route detection and outlining into a defined time period.

These routes are given by the sum of segments defined by each vehicle. Segments are the outline of the vehicle positions over time. In the segments it is collected relevant information for the tracing of the route, furthermore the relevant data for its study.

Beyond the main functionalities, the tool incorporates visual functionalities, through a Google Maps API, that shows up into a new tab of the default browser.

The tool also incorporates DDBB management functionalities, through the introduction of a direct query. The managements of this functionality must be performed with the correct known of the syntax, since it is exposed to the capacity of modify the boards for which owns writing rights.

This document presents the making of the tool and its functionalities, explaining in the most detailed way possible, what are those functionalities, how does them work on each file process and the synergy that shows with other functionalities.

For the correct tool performance, it is necessary to have installed Python 3.9, with the pertinent interpreter and the imported libraries that are contained in the python files.

PARTE 1

MEMORIA

**CREACIÓN DE DATOS SEMÁNTICOS A
PARTIR DE STREAM DE DATOS DE
SEGUIMIENTO DE VEHICULOS.**

CAPÍTULO 1

INTRODUCCIÓN

1.1. OBJETIVOS Y ALCANCE

El objeto de este proyecto es el diseño y construcción de una herramienta Python que desempeñe las funciones de extracción de puntos concurridos por vehículos a estudiar denominados SPs mediante algoritmos de clusterizado, y el trazado de rutas y segmentos de dichos vehículos para su estudio.

Para llevar a cabo este proyecto se cuenta con una base de datos MySQL proporcionada por la empresa en un servidor del tipo MariaDB Server. En la BBDD se encuentra, entre otras cosas, la tabla de ‘movements’, de donde se extraen los datos pertinentes de los vehículos para realizar el clusteo y, por consiguiente, los demás procesos. Se encuentra, además, las tablas ‘specialsites’, ‘routes’ y ‘segments’, donde son volcados los datos, cálculos y extracciones que realiza la herramienta.

El proyecto se ha llevado a cabo en el entorno de PyCharm a donde corresponden las ilustraciones. Con el sistema operativo de Ubuntu, Linux, para un mejor manejo de la aplicación desde la terminal del sistema, pero, como se indicará más a delante, la aplicación es creada de manera tal que resulta funcional en los sistemas operativos de Windows y Linux.

Los archivos finales de la herramienta son archivos “.py”.

El alcance de este proyecto de fin de grado es:

- I. Utilización y manejo del lenguaje Python para el diseño completo y construcción de una herramienta funcional.
- II. Utilización y manejo de bases de datos y demás recursos de almacenamiento.
- III. Utilización y manejo de módulos de importación, bibliotecas y demás recursos de programación adecuada.
- IV. Utilización y manejo de la terminal.
- V. Búsqueda de recursos de programación en la documentación de las librerías y módulos importados.
- VI. Utilización de métodos para una programación eficaz, evitar la redundancia y limitar la extensión de código.
- VII. Programación, manejo y realización de archivos “.html”

Este proyecto es sometido al marco lógico de trabajo, el cual comprende la identificación de ideas de proyecto, definición de objetivos, diseño, análisis y aprobación, ejecución y evaluación posterior a la ejecución y resultados obtenidos. El proyecto cuenta con la comunicación entre las partes pertinentes con objeto de realizar el proyecto lo más ajustado posible a la idea y los requerimientos iniciales.

1.2. MOTIVACION

Actualmente el transporte de productos y mercancías es una de las bases de la economía, y continúa siendo la opción más elegida entre las empresas en cuanto a la distribución. Hoy en día se pueden realizar envíos de cualquier substancia, material, producto o mercancía, se trate de materiales peligrosos, delicados, pesados o voluminosos a casi cualquier parte del mundo, y la distribución, en total o última instancia, es llevada a cabo mediante transporte terrestre.

Existe una reciente implantación en los sistemas de transporte terrestre denominada ‘tracking’, la cual proporciona de manera cada vez más detallada información de localización, ruta y tiempos, lo que aporta un gran grado de fiabilidad y confianza.

La necesidad de un sistema de control de flota terrestre es implícita.

El control en la flota aporta mayor seguridad al llevar a cabo un seguimiento exhaustivo en tiempo real del vehículo; así mismo, conlleva mayor eficiencia y, por tanto, ahorro al contar con la comunicación con, y entre, los conductores y choferes, conociendo su estilo de conducción, desviaciones y tiempos.

El transporte terrestre es un ámbito más que toma el rumbo de la digitalización. La capacidad de incorporar la programación en los ámbitos de la ingeniería es vital para el desarrollo de esta.

En los últimos años, la programación ha tenido su auge al ser accesible a toda persona que posea un ordenador, siendo fundamental en cualquier idea tecnológica, la base para el futuro y encargada de que la tecnología se siga desarrollando.

1.3. ANTECEDENTES

Este trabajo forma parte de un conjunto de proyectos dirigidos por el profesor Joaquín Bienvenido Ordieres Mere de la Escuela Técnica Superior de Ingenieros Industriales (ETSII) de la Universidad Politécnica de Madrid. El proyecto comprende la primera instancia de la herramienta, la cual pretende ser continuada.

El presente proyecto realiza la extracción de datos a partir de una BBDD de posición de vehículos industriales aportada por una empresa internacional de logística inversa del metal, donde se almacenan en forma tablas las cadenas de texto de la información de los vehículos. Esta BBDD incorpora a demás las tablas a las cuales se vuelcan en forma de cadenas de texto la información de los SPs, los segmentos de ruta y las rutas.

El algoritmo de clusterizado de la herramienta cuenta con la importación de DBScan, la cual es la biblioteca encargada de realizar el clusteo de los datos.

1.4. VIABILIDAD

Este trabajo se encuadra en el proyecto dirigido por Joaquín Bienvenido Ordieres Mere, como ya se ha mencionado en el apartado anterior. El proyecto pretende ser continuado mediante la incorporación a la herramienta de funciones para el control y medición de variables biométricas. Esto se consigue mediante la introducción en la cabina del vehículo de un dispositivo capaz de recopilar datos del conductor, o chofer, y entorno de cabina y exteriores.

El proyecto ofrece una herramienta para control sobre la flota de vehículos no lentes de la empresa, a través de la cual se pueden gestionar estructuras superiores a los puntos de geoposición, pudiendo extraer conclusiones sobre segmentos o rutas a las que vincular KPIs de gestión (por ejemplo, de consumo energético).

La idea principal del proyecto es tratar de dotar sistemas inteligentes que eviten cargar al conductor con indicar parámetros de su ruta de forma manual, como por ejemplo cuando comienza o termina un segmento.

Este hecho, además de poco eficiente, implica que es proclive a errores.

La identificación automática es compleja pero más robusta y no depende del operador en sí mismo, además de ahorrar tiempos en los procesos. Es un paso más en la secuencia de digitalización de los procesos, que no será el único ni el último.

En adición ayuda a la seguridad, ya que la automatización del proceso y despreocupación por parte del conductor evita distracciones como recordar súbitamente que no ha marcado la ruta y manipula el móvil para hacerlo mientras conduce.

1.5. ESTRUCTURA

Cada capítulo del documento se divide en secciones caracterizadas por los distintos archivos o funciones de cada uno, explicando las funciones que poseen, las importaciones que requieren y los archivos donde son importados.

El presente trabajo se divide en seis capítulos. En el primero se realiza una breve justificación a la realización del trabajo, se exponen los objetivos del mismo y se presentan los antecedentes, así como la viabilidad del proyecto.

En el capítulo dos se muestran las funciones, construcción, estructura y funcionamiento de los archivos Main y Database.

En el capítulo tres se explican las funciones, construcción, estructura y funcionamiento de los procesos que se encargan de la extracción o minado de SPs.

En el capítulo cuatro se explican las funciones, construcción, estructura y funcionamiento de los procesos que se encargan del trazado de rutas y segmentos.

En el capítulo cinco se explica la construcción, estructura y funcionamiento de las funciones que se encargan de funcionalidades secundarias de la herramienta.

CAPÍTULO 2

MAIN Y DATABASE

2.1. Main.

2.1.1. Introducción.

El archivo main.py es el archivo que comprende la estructura principal de la herramienta, recopila todas las funcionalidades y las condensa en una interfaz por la que se navega mediante la introducción de caracteres numéricos y parámetros a introducir (fecha, matrícula, etc.).

El archivo main.py es desde donde se opera la herramienta, por lo que la construcción de la interfaz debe contar con un manejo sencillo, una visualización clara y minimalista, evitar la acumulación de datos en la pantalla, y mostrar los datos relevantes a una velocidad de percepción humana.

El resto de archivos están construidos de manera tal que sean fácilmente importables para main.py, es decir, constan en su mayoría por el conjunto de funciones, o una clase, en el caso de database.py. Sería imposible la construcción de la herramienta si no se tomara esta medida en el resto de archivos, puesto que se ejecutarían al ser importados.

2.1.2. Cuerpo.

2.1.2.1. IMPORTACIONES.

Como primer objeto se obtienen las importaciones del archivo. El archivo main.py comprende la estructura principal de la herramienta, por lo que cuenta de forma mayormente directa con la importación de los demás archivos, a parte de las importaciones correspondientes para su correcto funcionamiento.

```
from addto_SQL import twoyears_input, month_input
import SP20
from os import system, name
import map
import segments
import search
import scattering
import visualizado
from time import sleep
import database
import resets
```

Ilustración 1 main_imports.

Las importaciones que se dan son las siguientes:

- Del archivo *addto_SQL* (capítulo 3), se importan las funciones *twoyears_input* y *month_input*, correspondientes a la extracción o minado de SPs a lo largo de tiempos.
- Se importa el archivo *SP20*, contenido de funciones de extracción o minado de SPs, de manera manual y automática.
- Se importan las funciones *system* y *name* de la librería de *os*, para el funcionamiento de la interfaz.
- Se importa el archivo *map*, continente de las funciones de visualizado.
- Se importa al archivo *segments* (capítulo 4), continente de las funciones de trazado de rutas y segmentos de ruta.
- Se importa el archivo *scattering*, continente de las funciones de visualizado de dispersión.
- Se importa el archivo *visualizado*, continente de las funciones de visualizado en Google Maps.
- De la librería de *time* se importa la función *sleep*, para el funcionamiento de la interfaz.
- Se importa el archivo de *database*, continente de las funciones de conexión de BBDD.
- Se importa el archivo de *resets*, continente de las funciones de reseteo.

2.1.2.2. PANTALLAS.

Puesto que este es el archivo que comprende el algoritmo de la interfaz, cuenta con las pantallas correspondientes a las funciones para su elección mediante la introducción de caracteres numéricos.

La primera de las pantallas comprende el menú de la herramienta, donde se muestran las siguientes opciones:

```
sc_1 = """
    MENU

    [0] Initiate.
    [1] Settings.
    [2] Credits.
    [3] Exit.

"""

```

Ilustración 2 sc_1.

- *Initiate*, con la cual se inicia la aplicación.
- *Settings*, opción no implementada la cual se pretende incluir en la continuación de la herramienta, con la que ofrecer la posibilidad de modificar ajustes de la aplicación como el idioma o el color en pantalla.

- *Credits*, con la cual se muestran los créditos de la aplicación.
- *Exit*, con la cual se cierra la conexión con la BBDD y la herramienta.

La segunda de las pantallas comprende la iniciación de la aplicación y por ende el menú de funcionalidades de la herramienta:

```
sc_2 = """
    FUNCTIONS

[0] Routing.
[1] Clustering SP.
[2] Mapping.
[3] Scatter Search.
[4] Search in Database.
[5] Reset BBDD.
[6] Back.

"""

```

Ilustración 3 sc_2.

- *Routing*, ofrece las diferentes opciones para el trazado de rutas y segmentos de ruta de los vehículos.
- *Clustering SP*, ofrece las diferentes opciones para la extracción o minado de SPs.
- *Mapping*, ofrece las diferentes opciones para la visualización de los distintos elementos, extraídos o trazados.
- *Scatter Search*, ofrece una de las funcionalidades secundarias de la herramienta, ofrece la visualización de las ubicaciones sin clusterizar de los vehículos en un cierto periodo de tiempo.
- *Search in Database*, ofrece otra de las funcionalidades secundarias de la herramienta, comprende un buscador, mediante lenguaje de SQL, de la BBDD, pudiendo comprobar la importación y exportación de datos de manera manual.
- *Reset DDBB*, ofrece las opciones de reseteo de la BBDD.
- *Back*, ofrece la posibilidad de volver al menú de la herramienta.

Clustering SP es una de las funcionalidades principales de la herramienta, con la cual, mediante algoritmos de clusteo (ver capítulo 3), se realiza una extracción o minado de SPs y su volcado hacia la BBDD. Las opciones que ofrece son las siguientes:

```

sc_3 = """
        CLUSTERING SP

[0] Month input.
[1] Day selection input.
[2] Reset all SP.
[3] Search for clusters.
[4] Back.

"""

```

Ilustración 4 sc_3.

- *Month input*, opción por la cual se realiza un minado de SPs de los últimos 30 días en dos fases solapadas para una mejor extracción.
- *Day selection input*, opción que ofrece el minado de SPs a partir de cierta fecha, la cual se introduce mediante el formato de fecha o la introducción de una cantidad de días previos.
- *Reset all SP*, permite resetear los SP ya extraídos y volcados en la BBDD.
- *Search for clusters*, permite el visualizado de los distintos clústeres que se forman en un cierto periodo de tiempo limitado.
- *Back*, permite regresar al menú de funcionalidades.

Scatter Search es una de las funcionalidades secundarias de la herramienta, con la cual se pretende ofrecer una visualización en cuanto a forma, densidad y elongación de las posiciones conjuntas de los vehículos. Las opciones que ofrece son las siguientes:

```

sc_4 = """
        SCATTER SEARCH

[0] Scatter 30days with day selection.
[1] Scatter last month.
[2] Back.

"""

```

Ilustración 5 sc_4.

- *Scatter 30days with day selection*, ofrece mostrar las posiciones conjuntas de los vehículos a lo largo de 30 días seguidos a elegir.
- *Scatter last month*, ofrece mostrar las posiciones conjuntas de los vehículos en el último mes.
- *Back*, permite regresar al menú de funcionalidades.

Mapping comprende una herramienta de visualización de los datos obtenidos, permite visualizar, mediante la creación de un archivo html y un API de Google Maps, los SPs de la BBDD, las rutas y los segmentos de ruta. Las opciones que ofrece son las siguientes:

```
sc_5 = """
        MAPPING

[0] Map route/seg selection.
[1] Map all sp.
[2] Back.

"""
```

Ilustración 6 sc_5.

- *Map route/seg selection*, ofrece la visualización de segmentos, o ruta completa de
- *Map all SPs*, ofrece la visualización de los SPs de la BBDD junto con la selección de en qué lugar mostrarlos.
- *Back*, permite regresar al menú de funcionalidades.

Mapping All SP, muestra la pantalla de selección de lugar para la generación de un mapa junto con la visualización de SPs, los lugares a mostrar son los países en los que se forman los clústeres que dan lugar a los SPs. La modificación de este menú y la incorporación de un nuevo mapa ha de hacerse de manera manual. Las opciones que ofrece son las siguientes:

```
sc_6 = """
        MAPPING ALL SP

[0] Spain.
[1] Hungary.
[2] Back.

"""
```

Ilustración 7 sc_6.

- *Spain*, procede a la creación de un mapa que muestra España junto con los SPs de la BBDD.
- *Hungary*, procede a la creación de un mapa que muestra Hungría junto con los SPs de la BBDD.
- *Back*, permite regresar al menú de selección de Mapping.

Routing, comprende una de las funcionalidades principales de la herramienta, con la cual, se realiza el trazado de rutas y segmentos de rutas, se vuelca a la BBDD los datos obtenidos y se imprime en pantalla la obtención de datos en vivo (ver capítulo 4). Las opciones que ofrece son las siguientes:

```
sc_7 = """
    Routing

[0] Daily routing.
[1] Selection routing.
[2] Back.

"""
```

Ilustración 8 sc_7.

- *Daily Routing*, comprende la funcionalidad de trazado de rutas y segmentos de ruta, su realización será diaria. Puesto que es un trazado en vivo, el algoritmo deberá ser ejecutado para el día anterior y así abarcar las rutas del día entero o bien una ejecución a una hora concreta en la que se hayan finalizado los trayectos.
- *Selection Routing*, comprende la funcionalidad del trazado de rutas para una fecha concreta. La introducción de dicha fecha se realiza mediante el formato de fecha o bien mediante la introducción de una cantidad de días previos.
- *Back*, ofrece volver al menú de funcionalidades.

Reset BBDD, comprende una de las funcionalidades secundarias de la herramienta, permite el reseteo, o vaciado, de la tabla de la BBDD a indicar en las opciones que ofrece:

```
sc_8 = """
    Reset BBDD.

[0] Reset segments.
[1] Reset routes
[2] Reset specialsites.
[3] Back.

"""
```

Ilustración 9 sc_8.

- *Reset segments*, permite resetear o vaciar la tabla de segments.
- *Reset routes*, permite resetear o vaciar la tabla de routes.
- *Reset specialsites*, permite resetear o vaciar la tabla de specialsites.

2.1.2.3. CÓDIGO.

El archivo main.py consta de un bucle principal con la estructura típica de cualquier aplicación. Dicho bucle se realiza con la intención de facilitar la navegación por la herramienta, de manera que, para las acciones y decisiones a tomar en la interfaz, ofrezca siempre la posibilidad de retroceder o volver a la pantalla anterior sin tener que iniciar la aplicación de nuevo. El bucle se repite siempre y cuando se cumpla la condición de no pulsar exit en la primera pantalla, con la cual cerrará la conexión con la BBDD y la herramienta.

Se utilizará el bucle principal como ejemplo para explicar distintos aspectos de la sintaxis en el archivo.

Como bucle principal obtenemos:

```
exit = False
while not exit:
    system('cls' if name == 'nt' else 'clear')
    print(sc_1)

    op = input("Select an Option: ")

    if op == "0":...
    if op == "1":...
    if op == "2":...
    if op == "3":
        exit = True
        database.DataBase().conection_close()
```

Ilustración 10 main_loop.

Podemos apreciar en la ilustración 10 la estructura del bucle más externo del archivo, donde se define la variable *exit* como booleana *False*, donde la condición es que mientras *no exit*, es decir, *True*, el bucle se repite.

Cada vez que el bucle se repite, imprime la pantalla *sc_1*, por lo que será necesario limpiar la pantalla antes de imprimir algo, para ello se hace una llamada a *system*, de la librería *os*, la cual son las siglas de Operator System. Aquí debemos hacer una distinción sobre desde qué sistema operativo se ejecuta la herramienta, es por ello que se comprueba el nombre del sistema operativo. Posteriormente se ejecutar el comando *cls* para Windows, y *clear* para Linux. Es importante hacer esta distinción, puesto que, si ejecutamos el comando incorrecto, el sistema no lo reconocerá y cerrará la herramienta emergiendo un error en la sintaxis.

Se define la variable *op* (como abreviatura de *option*), como una entrada *input* en forma de *str*, la cual es una abreviatura de *string*, que significa cadena de caracteres en programación, para elegir la opción a ejecutar.

Se crean los bucles condicionales *if*, o *elif*, según el caso para la ejecución de las opciones ofrecidas en cada pantalla. Es importante marcar los condicionales con la misma forma que las variables, es decir, debemos convertir a *str* las posibles opciones, ya que, por ejemplo: “*if op == 0 :*” reconocerá la variable *op* como *int*, abreviatura de *integer*, que significa carácter con valor numérico. Explicado de otra forma, un “0” de texto es distinto que un “0” numérico. Por

lo que se debe convertir la opción en *str* o la variable en *int*, aunque la segunda opción no es recomendable, puesto que, ante la introducción de un carácter alfabético, el algoritmo intentará convertirlo en *int*, emergiendo un error. Es por ello que la mejor opción es, por ejemplo: “*if op==str(0):*” o “*if op=='0':*”.

Las opciones propuestas son las enunciadas en la pantalla *sc_1*, estas son el iniciado de la herramienta, con la cual se accede al menú de funcionalidades; los ajustes, para una personalización; los créditos y la opción para salir de la herramienta.

En esta última opción se aplica el cambio en la variable booleana por la cual ya no se cumple la condición para seguir repitiendo el bucle, *exit = True*, por lo que *While not exit* es una condición falsa. La otra tarea desempeñada por la herramienta en esta opción es terminar la conexión con la BBDD, para ello se hace una llamada a la función *connection.close()* del archivo *database.py*.

Mediante la introducción de un 0 como iniciación de la herramienta ante el primer input se cumple la condición *if op == "0"* y se entra en el bucle del menú de funcionalidades, la cual se muestra a continuación.

```
if op == "0":
    back = False
    while not back:
        system('cls' if name == 'nt' else 'clear')
        print(sc_2)

        op_0 = input("Select an Option: ")

        if op_0 == "0":...
        if op_0 == "1":...
        if op_0 == "2":...
        if op_0 == "3":...
        if op_0 == "4":...
        if op_0 == "5":...
        if op_0 == "6":
            back = True
```

Ilustración 11 functionalities_menu_loop.

En calidad de definición de los bucles *while* utilizados en la herramienta se asumen como comunes, para su funcionamiento, los elementos repetidos, así como la variable booleana, definida en falsa, para regresar a la pantalla anterior; el *while en not False*; la variable definida como input, para la selección de funcionalidad; la llamada a *system*, para limpiar la pantalla cada vez que se inicia el bucle; los *if* condicionales y la última de las opciones de los *es* para convertir en *True* la variable booleana.

La nomenclatura de las variables *input* de cada bucle vendrá dada por las opciones elegidas hasta el punto de la elección actual, es decir, que para este ejemplo *op_0* es el resultado de ofrecer una selección de opciones después de haber escogido la opción 0 en la variable *op*, dejando así *op = "0"*. Si para *op_0* se escoge la opción 1, *op_0 = "1"* y la siguiente variable *input*, si se da el caso para la opción seleccionada, será *op_01*.

Las opciones a elegir en el menú de funcionalidades son las mostradas en la pantalla *sc_2*: *Routing, Clustering SP, Mapping, Scatter Search, Search in Database, Reset DDBB, Back*.

- “if *op_0* == “0”, *Routing*.

Mediante la introducción de “0” en *op_0*, entramos en el bucle de la funcionalidad de *Routing*, cuyo algoritmo se muestra a continuación.

```
if op_0 == "0":  
    back_to_func_00 = False  
    while not back_to_func_00:  
        system('cls' if name == 'nt' else 'clear')  
        print(sc_7)  
        op_00 = input("Select an Option: ")  
  
        if op_00 == "0" or op_00 == "1":  
            segments.segments(op_00)  
            sleep(3)  
  
        if op_00 == "2":  
            system('cls' if name == 'nt' else 'clear')  
            back_to_func_00 = True
```

Ilustración 12 *Routing_menu_loop*.

Las opciones que ofrece el menú de *Routing* son las que se muestran en la pantalla *sc_7*: *Routing, Selection routing y Back*.

A partir de esta profundidad en el algoritmo se encuentran las llamadas a funciones que desempeñan las acciones principales de la herramienta.

En adelante, se utilizará la denominación de matrícula para referirse a los vehículos, puesto que cada vehículo viene definido por su matrícula.

La llamada a la función *segments* se hace con la introducción de la variable *op_00* como argumento ya que la misma función realiza ambas tareas a elegir (capítulo 3). La función de *segments* es la encargada de realizar el trazado de los segmentos y rutas, la variación entre las opciones es la fecha para la cual se realiza el trazado. Mediante la elección *op_00* = “0”, se fija la realización del trazado para un periodo que comprende desde las 00:00 de la misma fecha, hasta la hora que se realiza el trazado, seleccionando además todas las matrículas disponibles en la fecha para hacer su trazado. Mediante la elección *op_00* = “1”, se toma un camino alternativo en el algoritmo el cual permite la introducción de una fecha, en formato fecha o como cantidad de días previos, para la realización del trazado, posteriormente se ofrece la selección de matrícula a trazar, o todas ellas. Las dos posibles iniciaciones de *segments* convergen en el mismo algoritmo una vez definido el día y la/las matrículas a trazar.

Puesto que la función de *segments* incorpora impresiones en pantalla y, por tanto, limpian esta, no es necesario incorporar esta acción en el archivo main.

La llamada a *sleep* se hace para dejar los datos en la pantalla el número de segundos que se introducen para que dé tiempo a leer a velocidad humana la pantalla antes de volver al menú de la funcionalidad.

- “if $op_0 == "1"$ ”, Clustering SP.

Mediante la introducción de “1” en op_0 , entramos en el bucle de la funcionalidad de *Clustering SP*, cuyo algoritmo se muestra a continuación.

```

if op_0 == "1":
    system('cls' if name == 'nt' else 'clear')
    back_to_func = False
    while not back_to_func:
        system('cls' if name == 'nt' else 'clear')
        print(sc_3)

        op_01 = input("Select an Option: ")

        if op_01 == "0" or op_01 == "1":
            system('cls' if name == 'nt' else 'clear')
            month_input(op_01)
            sleep(3.5)

        if op_01 == "2":
            system('cls' if name == 'nt' else 'clear')
            twoyears_input()
            sleep(3.5)

        if op_01 == "3":
            system('cls' if name == 'nt' else 'clear')
            SP20.manual()
            sleep(1.5)

        if op_01 == "4":
            system('cls' if name == 'nt' else 'clear')
            back_to_func = True

```

Ilustración 13 Clustering_SP_loop.

Las opciones que ofrece *Clustering SP* son las que se muestran en la pantalla sc_3 , estas son: *Month input*, *Day selection input*, *Reset all SP*, *Search for clusters*, *Back*.

Puesto que las funciones no incorporan la limpieza de pantalla en ellas es necesario hacerlo en el archivo main.py.

Al igual que en *segments*, *month input* también cuenta con la introducción de la variable de elección como su argumento para la distinción entre la primera y la segunda opción ofrecida dentro de la función. La función *month input* tiene dos caminos del algoritmo que bifurcan para la fecha por defecto o la introducción de fechas, respectivamente, y posteriormente convergen en el mismo algoritmo cuando se inicia el clusteo con los datos ya recogidos.

La función *Reset all SP* no es una función que únicamente elimina de la base de datos los SPs, sino que se encarga de eliminar los SPs obsoletos desde hace aproximadamente dos años. Es decir, los SPs que no son concurredos desde hace dos años no constarán más en la

base de datos. Este es un proceso que ha de ser ejecutado con cautela, puesto que las rutas ya guardadas en la BBDD constarán de los SPs existentes, si se eliminan SPs de la base de datos, la nueva lista de SPs seguirá una enumeración distinta a partir del SP eliminado. Esto trae como consecuencia que las rutas previas a la renovación de los SPs no contarán con el nuevo orden y, por tanto, si se comparan con los SPs actuales, es posible que carezca de sentido. Para solucionar este problema, la introducción en la BBDD de los SPs de los segmentos cuenta, a parte de la numeración, con las coordenadas del SP.

La función de *Search for clusters*, muestra los clústeres formados en una fecha a introducir, con un intervalo de treinta días posteriores. Esta función carece de la capacidad de introducir sus datos en la BBDD.

- “if op_0 == “2””, *Mapping*.

Mediante la introducción de “2” en *op_0*, entramos en el bucle de la funcionalidad de *Mapping*, cuyo algoritmo se muestra a continuación.

```
if op_0 == "2":
    back_to_func_2 = False
    while not back_to_func_2:
        system('cls' if name == 'nt' else 'clear')
        print(sc_5)
        op_02 = input("Select an Option: ")

        if op_02 == "0":
            system('cls' if name == 'nt' else 'clear')
            visualizado.visualization()
            sleep(1.5)

        if op_02 == "1":
            back_to_mapping = False
            while not back_to_mapping:
                system('cls' if name == 'nt' else 'clear')
                print(sc_6)
                place = input("Select place to MapSP: ")
                if place == '0' or place == '1':
                    map.mapping(place)
                if place == '2':
                    back_to_mapping = True

        if op_02 == "2":
            system('cls' if name == 'nt' else 'clear')
            back_to_func_2 = True
```

Ilustración 14 Mapping_loop.

Las opciones que ofrece *Mapping* son las que se muestran en la pantalla sc_5, estas son: *Map route/seg selection*, *Map all SP Back*.

Esta vez, puesto que hay numerosas elecciones, la función de *visualizado* se encarga de ellas, elecciones como: fecha, matrícula, segmento/segmentos/ruta completa, etc. La función cuenta con su propio bucle, para poder realizar una visualización de los segmentos de ruta seguidos sin tener que volver a iniciar la funcionalidad.

Map all SP cuenta con su propio bucle, puesto que la elección a tomar es sobre qué lugar se desea realizar la visualización, es por ello que se aprovecha la variable *place*, tanto para elegir la opción como para introducirla como argumento en la función y esta bifurque el algoritmo. En el momento de la creación de la herramienta, se cuenta únicamente con los lugares de *Spain* y *Hungary*, expuestos en la pantalla *sc_6*. Ante la aparición de clústeres en otras regiones ha de ampliarse esta sección a los nuevos lugares para realizar el visualizado.

- “if *op_0* == “3”, *Scatter Search*.

Mediante la introducción de “3” en *op_0*, entramos en el bucle de funcionalidad de *Scatter Search*, cuyo algoritmo se muestra a continuación.

```
if op_0 == "3":  
    back_to_func_1 = False  
    while not back_to_func_1:  
        system('cls' if name == 'nt' else 'clear')  
        print(sc_4)  
        op_03 = input("Select an Option: ")  
  
        if op_03 == "0":  
            system('cls' if name == 'nt' else 'clear')  
            scattering.scatter_dayselect()  
            sleep(1.5)  
  
        if op_03 == "1":  
            system('cls' if name == 'nt' else 'clear')  
            scattering.scatter_last_month()  
  
        if op_03 == "2":  
            system('cls' if name == 'nt' else 'clear')  
            back_to_func_1 = True
```

Ilustración 15 *Scatter_SeachLoop*.

Las opciones que ofrece *Scatter Search* son las que se muestran en la pantalla *sc_4*, estas son: *Scatter 30days with day selection*, *Scatter last month* y *Back*.

Se realizan las pertinentes llamadas al archivo de *scattering* para importar las funciones correspondientes a la opción a elegir. Las funciones realizan la misma acción para fechas diferentes, corresponden a una de las funcionalidades secundarias de la herramienta.

- “if *op_0* == “4”, *Search in Database*.

Mediante la introducción de “4” en *op_0*, entramos en la funcionalidad de *Search in Database*, cuyo algoritmo se muestra a continuación.

```
if op_0 == "4":
    system('cls' if name == 'nt' else 'clear')
    search.search()
```

Ilustración 16 Search_in_Database.

Search in Database comprende una de las funcionalidades secundarias de la herramienta, consta de un buscador directo a través de la interfaz actuando como terminal para la DDBB, desde la cual se pueden realizar todas las opciones que se ofrece utilizando la sintaxis debida.

- “if *op_0* == “5”, *Reset DDBB*.

Mediante la introducción de “5” en *op_0*, entramos en la funcionalidad de *Search in Database*, cuyo algoritmo se muestra a continuación.

```
if op_0 == "5":
    back_to_func_reset = False
    while not back_to_func_reset:
        system('cls' if name == 'nt' else 'clear')
        print(sc_8)
        op_05 = input("Select an option:")
        if op_05 == "0":
            sure = input("Are you sure to reset segments?[Y/n]: ")
            if sure == "Y":
                resets.reset_segments()
                sleep(1.5)
        if op_05 == "1":
            sure = input("Are you sure to reset routes?[Y/n]: ")
            if sure == "Y":
                resets.reset_routes()
                sleep(1.5)
        if op_05 == "2":
            sure = input("Are you sure to reset specialsites?[Y/n]: ")
            if sure == "Y":
                resets.reset_specialsites()
                sleep(1.5)
        if op_05 == "3":
            back_to_func_reset = True
```

Ilustración 17 Reset_DDBB_loop.

Las opciones que ofrece *Reset DDBB* son las que se muestran en la pantalla sc_8, estas son: *Reset Segments*, *Reset Routes*, *Reset specialsites* y *Back*.

Estas funcionalidades comprenden la parte de la herramienta que borra los datos de manera absoluta y rápida de la BBDD, es por ello que se introduce una variable de confirmación de la

acción. La eliminación de los datos recopilados, en cualquier aplicación o aspecto es necesaria tanto como su certidumbre a la hora de llevarlo a cabo. La herramienta se cerciora de que es una acción consciente mediante la variable *sure*, la cual nos ofrece un input con las recomendaciones *[Y/n]*, como iniciales de “Yes” y “no”. El algoritmo está diseñado de manera que ante la introducción de un carácter erróneo en la interfaz se interprete como negativa o cancelación. Para llevar a cabo la acción de eliminación de datos, es necesaria la introducción de “Y”, en mayúscula para una que la acción se realice de manera deliberada.

Reset BBDD forma parte de las funcionalidades secundarias de la herramienta.

Las funciones de reseteo son idénticas en cuanto a construcción, exceptuando la diferencia de la tabla de datos a la que se refieren. Ya que los datos se vuelcan en tres tablas distintas, hay tres opciones de reseteo en la herramienta.

Las funciones se incorporan desde *resets.py* un archivo diferente a *database.py*, aunque conste de funciones en relación a la BBDD. Esto es así para evitar aglomeración, separando las funciones de reseteo y manejándolas de forma clara.

2.2. Database.

2.2.1. Introducción.

El archivo database.py comprende de la clase que recoge las funciones encargadas del manejo de la BBDD. Las funciones se distribuyen a lo largo de todas las funcionalidades de la herramienta y la importación de la clase *Database()* se encuentra en todos los archivos de la herramienta.

La importancia de este archivo es vital para el funcionamiento de la herramienta, puesto que, sin ella, las funciones carecerían de conexión con la BBDD a tiempo real. Esto dejaría una herramienta de uso local con una incorporación de datos manual, lo que resultaría algo ciertamente imposible si se pretende obtener la dimensión de datos que dispone.

Las funciones de *Database()* poseen la sintaxis propia del lenguaje Python, pero cuentan con la introducción de datos *str* en la BBDD que poseen la sintaxis propia del lenguaje de la BBDD con la que trabajan, en este caso se trata de una base de MariaDB, en SQL.

La dinámica entonces en las funciones será la de nombrar una variable *sql* en forma de *str* que contenga las instrucciones que se le dan a la BBDD, a esta variable la denominaremos consulta. Se hará un bucle *try* con la ejecución de la consulta junto a las acciones pertinentes, ya sea crear una variable que almacene los datos recogidos como carácter general, o en ciertas ocasiones la confirmación y el guardado de los cambios realizados. Como contraposición a *try* se obtiene la aparición de un error esperado, en caso de que la función no sea capaz de acometer la tarea. Mas a delante, usaremos la primera función de la clase ‘*select_unique_user*’ para explicar de forma detallada los elementos descritos.

Las variables, dentro de la clase, que comparten las funciones estarán definidas por el constructor, explicado más a delante, y tendrán el prefijo *self*, el cual es el primer y/o único argumento de las funciones. De esta manera se reconocen como elementos de la clase y son capaces de realizar acciones definidas por el constructor.

2.2.2. Cuerpo.

2.2.2.1. IMPORTACIONES.

Como primer objeto en el archivo se obtienen las funciones. El archivo no comprende más funcionalidades que la de conexión con la BBDD, por lo que va a necesitar pocas importaciones de módulos y bibliotecas.

```
import pymysql
from datetime import timedelta, date, datetime
```

Ilustración 18 Importaciones_database.py

Las importaciones que se dan son las siguientes:

- Se importa *pymysql*, la biblioteca que contiene las funciones encargadas de realizar la conexión con la BBDD.
- De la biblioteca *datetime* se importan los archivos *timedelta*, *date*, *datetime*, funciones para la definición de las fechas a utilizar en el archivo, explicadas más adelante.

2.2.2.2. CLASE.

Antes de entrar en la definición de la clase y su contenido, se definen las variables comunes a numerosas funciones junto a su variable *sql* de interacción a la BBDD, como inciso. Se trata de variables temporales que corresponden a las fechas pertinentes para la realización de funciones de clusterizado, expuestas a continuación:

```
f_0 = date.today()
f_10 = (date.today() + timedelta(days=-20)).strftime('%Y-%m-%d %H:%M:%S')
f_20 = (date.today() + timedelta(days=-25)).strftime('%Y-%m-%d %H:%M:%S')
f_30 = (date.today() + timedelta(days=-45)).strftime('%Y-%m-%d %H:%M:%S')
```

Ilustración 19 temporalvar_database.

A continuación, en el archivo se encuentra la consulta común para varias funciones que corresponde con las fechas definidas, expuesta a continuación:

```
str = """SELECT id, matricula, X(ubicacion) as lon, Y(ubicacion) as lat, fecha_upos as fecha FROM movements """
      """WHERE velocidad = '0.0' and fecha_upos >= """
str_1 = str + "" + f_20 + """
str_2 = str + "" + f_30 + " and fecha_upos <= " + f_10 + """
```

Ilustración 20 strvar_database.

Este inciso se hace con la intención de una optimización de código, con la particularidad de definirlo fuera de la clase. La clase *Database()* se importa en los distintos archivos de manera distinta a las demás importaciones, se importa únicamente la clase y no el archivo, por lo que no se importan las variables definidas fuera de la clase, pero si la función lo requiere, sí.

A continuación, se define la clase y su contenido.

En primera instancia de la clase obtenemos el constructor de esta, definido mediante la función de `__init__`. El concepto de constructor se otorga a una subrutina en POO (Programación Orientada a Objetos) que define la inicialización de un objeto. Dentro del constructor se definen las propiedades del objeto y se adjuntan a las demás funciones mediante `self`.

```
class DataBase:  
    def __init__(self):  
        self.connection = pymysql.connect(  
            host='138.100.8*.*',  
            user='lectura',  
            password='*****',  
            db='gescrap'  
        )  
  
        self.cursor = self.connection.cursor()
```

Ilustración 21 `init_database`.

Para la creación del constructor, se incorpora como argumento `self` en referencia al objeto. El objeto se define como la conexión a la BBDD mediante la introducción de la variable `connection`, esta cuenta con la importación de la función `connect` de la librería de `pymysql` ya importada. Los argumentos de la función son los que caracterizan la BBDD, para definir una a la que conectarse, mediante: un `host` que comprende la IP (Internet Protocol) de la BBDD; un `user` como usuario de navegación como solo `lectura`; la introducción de contraseña, y la base de datos que comprenden las tablas a usar `db`, la cual es `gescrap`.

Para la protección de la BBDD, en este documento se han sustituido los caracteres por asteriscos, tanto en los cuatro últimos dígitos de la IP como en la contraseña, pero el archivo cuenta con los caracteres completos para la conexión.

Como segunda particularidad del objeto a definir, se le dota del cursor para navegar, recorrer y procesar los datos recogidos en cada consulta.

El cursor es un elemento crucial en la construcción del objeto, define una estructura de control para el procedimiento individual de las filas devueltas por la BBDD, es decir, un iterador en las filas. A demás de ello incorpora las sentencias para las consultas, que se refiere a las instrucciones a introducir en las variables `sql`.

Una vez definida la clase mediante el constructor se definen las funciones que comprenden la clase:

- *select_unique_user*:

```
def select_unique_user(self, id):
    sql = '''SELECT matricula, Y(ubicacion) as lat, X(ubicacion) as lon , fecha_upos as fecha FROM movements ''' \
          f''' WHERE id = {id} '''
    try:
        self.cursor.execute(sql)
        users = self.cursor.fetchall()

        return users

    except Exception as e:
        raise
```

Ilustración 22 *select_unique_user_func.*

Esta función la utilizaremos para explicar los elementos descritos anteriormente en la introducción.

La función en si se encarga de la extracción de: *matrícula*, *ubicación* y *fecha* la tabla de *movements* a partir de la introducción de una variable de identificación *id*. La ubicación se selecciona de manera cartesiana con sus respectivas coordenadas *X* e *Y*, guardadas en las variables *lon* y *lat*, en referencia a *longitud* y *latitud*. Con lo que, mediante la selección de un *id* determinado, se extraen y guardan los datos de la BBDD.

La consecución de los eventos es definir, como primer elemento, la consulta para cada función, siendo esta su principal particularidad. En caso de que la consulta requiera de una variable, como es este caso, ha de introducirse en el argumento de la función. Cuando la función es llamada, la secuencia de argumentos que tenga definidos serán las variables que después se introducen en la consulta, siguiendo el mismo orden. Al hacer la llamada a la función es necesario guardarla en una variable, a modo de ejemplo:

```
from database import DataBase

database = DataBase()
users = database.select_unique_user(3)
```

Ilustración 23 *call_example_database.*

Como se puede apreciar, para llamar a las funciones es necesario importar la clase y no el archivo entero. Una vez hecho se lo nombra de la manera deseada y se procede a la llamada de la función, se cita la clase y posteriormente la función con un punto. El argumento de la función es 3 y corresponde al *id* antes mencionado.

Se realiza un bucle *try* para realizar el cumplimiento de la tarea, es necesaria esta manera de hacerlo puesto que, ante un posible error, como el de que no se encuentren los resultados, la herramienta no se detenga. Para ello la estructura de la función será de la forma expuesta en la

ilustración, un bucle *try/except* con un error esperado. Ante un error, se genera *raise*, un uso especial de Python con el que se obtiene la excepción y se vuelve arriba en el algoritmo.

El corazón de la función reside en la ejecución de la consulta, para ello se llama al cursor del objeto de conexión y se le pide que ejecute el *str* ya definido anteriormente. Mediante la función *execute* y como argumento la consulta, la función se encargará de realizar la extracción de datos de la base.

Una vez ejecutada la consulta con sus correspondientes sentencias, han de seleccionarse, de los datos extraídos, cuales guardar en una variable. Para ello se define la variable *users* como la encargada de recoger los datos extraídos por el cursor, mediante *fetchall()*.

Se entiende entonces que, cada usuario *user* (variable hipotética), viene definido por un *id*, una *matrícula*, una *posición*, y una *fecha*, recogidos en forma de *tupla*. Una *tupla* es un conjunto de datos ordenados por una secuencia. Por tanto, *users* es la variable que consta de una *lista* de *tuplas* que almacena todos los datos recogidos. En este caso, únicamente habrá un registro, como máximo, puesto que la BBDD trabaja de tal manera que guarda sus datos con el *id* en *clave primaria*, lo que quiere decir que en la tabla se muestran registros con *ids* únicos. La estructura de las tablas viene explicada más adelante, tras la explicación de la estructura de las funciones.

Como último elemento común en las funciones, obtenemos *return*, un uso especial de Python con el que se devuelve la/las variables que se escogen al terminar la tarea de la función. En este caso, obtenemos *return users*, es decir, la función nos devuelve *users*. Las funciones que devuelven una o varias variables son necesitadas de otra que se defina como la variable de regreso, externa a la función. Es por ello que las funciones que realicen acciones sin devolver nada al algoritmo del que han sido llamadas no necesitan la variable que almacene lo que devuelvan, como es en el caso de las funciones encontradas en el archivo main.py.

La función *select_unique_user*, descrita a modo de ejemplo, no consta de una función funcional, puesto que no es incorporada en el algoritmo de la herramienta.

Posteriormente a la explicación de los tipos de funciones de la clase se procede a la descripción de las tablas de la BBDD. Necesaria para entender la sintaxis a realizar en las consultas.

2.2.2.3. Tipos de funciones.

A continuación, se describen los tipos de funciones que se encuentran en la clase, puesto que es necesario describir con carácter general, las diferentes acciones que realizan además de las diferencias fundamentales en su sintaxis.

- Funciones selectoras: tales como *select_unique_user*, son funciones encargadas de seleccionar los datos pertinentes de la BBDD con los que trabajar posteriormente, incluyendo cada ejecución en una tupla y devolviendo una lista de tuplas. Este tipo de funciones comprenden la misma estructura anteriormente descrita con los bucles *try* y

except, puesto que ante un resultado negativo en la búsqueda en la que no pueda seleccionar nada, necesitan de la excepción de error esperado para que la herramienta continúe en funcionamiento.

La sintaxis de consulta de las funciones selectoras es de la forma de: la introducción de la sentencia *SELECT*, por la cual, el cursor, al ejecutar la consulta, hará la acción de recopilar de la BBDD los datos correspondientes a los parámetros que se introducen a continuación de la sentencia. Posteriormente a los parámetros a recopilar, en la consulta, se encuentra la sentencia *FROM*, cuyo propósito es especificar la tabla de la BBDD de la que se va a seleccionar. Como última sentencia en la consulta, obtenemos *WHERE*, cuyo propósito es discriminar las selecciones, es decir, recoger en específico determinadas selecciones que cumplan la condición impuesta por *WHERE*.

Las sentencias *SELECT* y *FROM* son necesarias para realizar las consultas, en cambio *WHERE* no lo es. Ante la ausencia de alguna de las dos primeras, emerge un fallo en la sintaxis.

Para la introducción de los parámetros a seleccionar e incluir en las tuplas los datos seleccionados es necesario conocer de antemano los parámetros que guarda la tabla de la BBDD en la que se produce la selección.

A modo de ejemplo, se expone la función *select_all*, cuya función es la selección de la ubicación de todos los registros de la tabla *movements*.

```
def select_all(self):
    sql = '''SELECT X(ubicacion) as lon, Y(ubicacion) as lat  FROM movements '''
    try:
        self.cursor.execute(sql)
        users = self.cursor.fetchall()
    return users

    except Exception as e:
        raise
```

Ilustración 24 *select_all_func*.

Como podemos apreciar en la ilustración, se trata de una función de tipo selector con las sentencias y estructura propias. *select_all* no es una función funcional.

Ante la introducción de una variable en la sentencia *WHERE* es necesaria su introducción en el argumento de la función, para después poder utilizarla. La sintaxis propia en este tipo de casos es la siguiente: tal y como podemos ver en el ejemplo de *select_unique_user*, se puede realizar la introducción de la variable en la consulta mediante la introducción de *f* al inicio del *str* de la consulta o parte de la consulta que incorpore la variable.

```
f''' WHERE id = {id} '''
```

Ilustración 25 examp_f_func.

Otra manera de realizar la introducción de una variable en la consulta es mediante la finalización del *str*, a continuación, la introducción de la variable y posteriormente la continuación del *str*. La concatenación de la consulta se realiza con “+” entre las variables y los *strs*, tal y como se muestra en la siguiente ilustración.

```
def select_users_compr(self, lon, lat):
    str_compr = '''SELECT * FROM movements '''
    str_compr += '''WHERE Y(ubicacion) = ''' + "" + lat + "" + " and X(ubicacion) = " + "" + lon + ""
    sql = str_compr
```

Ilustración 26 examp_+_func.

Cabe destacar la necesidad de finalizar la consulta con *str* y no con la variable, como se muestra en la ilustración, tras la variable *lon*, es necesario “+ “””, esto se hace con la intención de que cuando el cursor ejecute la consulta, las variables se conviertan en *str*, por lo que se entenderá como conversión en potencia de las variables a *str*.

- Funciones insertoras: funciones cuyo propósito es la adición de registros a la BBDD. Estas funciones no comprenden la estructura de las funciones selectoras de bucles *try* y *except*, puesto que el algoritmo de la herramienta, ante la ausencia de registros a introducir en las tablas no realiza la introducción, por lo que no es necesario salvarse del error. Esto se consigue con un condicional de la existencia de datos *no null* (*if variable:*) que veremos en capítulos posteriores.

En estas funciones se introducen las variables a introducir en las tablas como argumento de la función, para después usarlas en la consulta. La primera de las sentencias es *INSERT INTO* con lo que se da la instrucción de insertar, en la tabla especificada a continuación de la sentencia, los registros. Posteriormente a la especificación de la tabla, se indican los parámetros de la tabla donde se van a introducir los registros. Por último, se da la sentencia de *VALUES*, con la que se introducen las variables del argumento en los puestos marcados por los parámetros anteriormente especificados en la consulta.

La introducción de registros en la tabla a partir de las variables se realiza según el orden en que se indican, es indiferente que coincida el nombre de la variable y el parámetro de la tabla, si no están en el mismo orden, el registro no es introducido como es debido.

A continuación, se expone un ejemplo de sintaxis de función instertora.

```
def addSP(self, lat, lon, cluster, desc):
    sql = """INSERT INTO specialsites(latitud, longitud, Cluster, Description) VALUES ('"""+lat+"', '"+lon+"",
    "+cluster+", '"+desc+"')"
```

Ilustración 27 examp_sintax_insertora_database

Cabe destacar la conversión en potencia a *str* de las variables mediante las comillas “”” al igual que en el anterior ejemplo.

Al igual que en muchos casos, cuando la frase es demasiado larga, mediante “\ ” se divide la frase, perteneciendo a la misma variable.

- Funciones eliminadoras: funciones cuyo propósito es la eliminación de registros de las tablas de la BBDD. Este tipo de funciones sí que comprenden la estructura de *try* y *except*, puesto que ante el intento de eliminar registros no existentes emergirá un error del que es necesario salvarse. Estas funciones son idénticas a las funciones de reseteo en cuanto a la acción que desempeñan, pero se encuentran en el archivo de database.py porque poseen un propósito muy distinto. Las funciones eliminadoras son llamadas en los procesos en los que posteriormente se da la introducción de registros, esto se realiza con el fin de evitar introducir registros que ya se encuentran en la tabla, mediante la eliminación de estos antes de su introducción actualizada. Es decir, si para un día se introducen los registros de los segmentos de ruta a una determinada hora, si se vuelve a realizar el proceso en el mismo día, carece de sentido obtener los registros repetidos hasta la hora de la primera realización del proceso, por ello, primero se eliminan los registros del día y posteriormente se incorporan los siguientes.

La sintaxis propia de estas funciones consta en la sentencia *DELETE*, con la que se da la instrucción de eliminar registros. Posteriormente, al igual que las funciones selectoras, se da la sentencia *FROM*, para la especificación de la tabla donde se van a eliminar los registros. Por último, se encuentra la sentencia *WHEN* con la que especificamos la particularidad del parámetro, o los parámetros, que queramos eliminar. La sentencia de especificación es necesaria siempre y cuando no se pretenda eliminar todos y cada uno de los registros de la tabla.

Para una mejor comprensión, se expone la función *remove_seg*:

```
def remove_seg(self, date_):
    sql = f"""DELETE FROM segments WHERE Dateseg = '{date_}'"""

    try:
        self.cursor.execute(sql)
        self.connection.commit()

        print(self.cursor.rowcount, "record(s) were deleted in segments.")

    except Exception as e:
        raise
```

Ilustración 28 examp_delete_func.

A modo de particularidad, existen en el archivo dos funciones que difieren de las anteriormente descritas, estas son la función para cerrar la conexión, *connection_close* y

la función de introducción de consultas de forma manual en la BBDD, *show*. Esta última se crea para la comprobación de los parámetros de las tablas donde se introducen los registros, entre otras cosas, mediante la introducción de *desc* y a continuación la tabla a analizar en la terminal creada en la interfaz.

2.2.2.4. Tablas de la BBDD.

A continuación, se describen las distintas tablas de la BBDD donde se trabaja, para ello se utiliza la función *show*, creada en el archivo database.py, copiada y usada desde el archivo search.py para una utilización independiente a la BBDD. La función obtendrá su explicación en el capítulo dedicado a la funcionalidad secundaria de *search in DDBB*.

- *movements*: comprende la tabla de extracción de los registros de los datos de los vehículos en forma de tupla, la herramienta no vuelca nada a esta tabla. En la próxima ilustración se muestran los parámetros que contiene con el tipo de dato que incorpora, este factor es el que hay que tener en cuenta a la hora de redactar la consulta.

```
Type 'exit' to Exit
Introduce consulta: desc movements
(id', 'int(11)', 'NO', 'PRI', None, 'auto_increment')
('matricula', 'varchar(25)', 'NO', 'MUL', None, '')
('estado', 'text', 'YES', '', None, '')
('localidad', 'text', 'YES', '', None, '')
('ubicacion', 'point', 'NO', 'MUL', None, '')
('fecha_upos', 'datetime', 'NO', 'MUL', None, '')
('velocidad', 'double', 'YES', '', None, '')
('kilometros', 'double', 'YES', '', None, '')
('tiempo_estado', 'varchar(20)', 'YES', '', None, '')
('s_motor', 'text', 'YES', '', None, '')
('s_grua', 'text', 'YES', '', None, '')
('o_sensores', 'text', 'YES', '', None, '')
('observaciones', 'text', 'YES', '', None, '')
('ts', 'timestamp', 'NO', 'MUL', 'CURRENT_TIMESTAMP', 'on update CURRENT_TIMESTAMP')
Introduce consulta: █
```

Ilustración 29 movements_table.

Como podemos observar en la ilustración, mediante la introducción de *desc movements*, se nos muestra en la interfaz los resultados de los parámetros de la tabla junto con sus datos. “*desc*” es el comando para “descripción” y a continuación la tabla a describir, en este caso *movements*.

La tabla consta de numerosos parámetros, entre los cuales, se extraen: *id*, *matrícula*, *localidad*, *ubicación*, *fecha_upos*, *velocidad* y *s_motor*.

Seguido al nombre del parámetro viene dado el tipo de dato que consta, este puede ser:

- *int()*: dato con valor numérico, entre los paréntesis se indica la longitud máxima del parámetro.
- *varchar()*: *variable character*, lo que quiere decir, un campo de longitud variable que almacena caracteres alfanuméricos y símbolos.
- *text*: cadena de texto.
- *point*: es una expresión *int* que referencia una posición espacial.
- *datetime*: expresión de fecha y hora.
- *double*: es una expresión numérica *float*, es decir, valor numérico decimal con 15 dígitos de precisión.
- *timestamp*: guarda la fecha y hora.
- *float* y *decimal*: caracteres decimales.

El siguiente campo consta de la permisión de una entrada nula al parámetro mediante “YES” o “NO”.

El siguiente campo consta de la clave que se le aporta al parámetro en la tabla, esta puede ser:

- *PRI*: *PRIMARY KEY* es una columna que identifica de forma exclusiva cada fila de la tabla.
- *MUL*: *MULTIPLE KEY* es una columna que permite la aparición de datos no únicos, es decir, que contengan el mismo valor.
- *UNI*: *UNIQUE KEY* hace que la columna no posea datos iguales.

El siguiente campo consta de la introducción de una función en el parámetro, es por ello que la mayoría obtiene el valor de *None* indicando un campo vacío. Como excepción se obtiene la función *CURRENT_TIMESTAMP* que devuelve el valor de la fecha y hora actuales.

El ultimo campo consta de las funciones automáticas, así como *auto_increment* se encarga de que el registro de *id* incremente con cada introducción y *on update CURRENT_TIMESTAMP* se encarga de que ante una actualización sin marca de tiempo explícita da el valor de la marca de tiempo actual.

Una vez aclarado lo que se muestra en las ilustraciones a cerca de los parámetros que existen en las tablas, se procede a describir la utilidad de cada parámetro a extraer:

- *id*: comprende la abreviatura de *identificación*, consta de un valor numérico, único para cada registro, gracias a la clave primaria y al autoincremento que posee, en esta y las tablas restantes.
 - *matricula*: comprende de la identificación de cada vehículo.
 - *localidad*: sirve para las comprobaciones de los clústeres formados.
 - *ubicación*: comprende de la posición geolocalizada mediante coordenadas.
 - *fecha_upos*: comprende de la fecha en la que fue realizado el registro.
 - *velocidad*: comprende de la velocidad que lleva el vehículo en el registro.
 - *s_motor*: comprende del estado binario del motor, *parado* o *en marcha*.
- *specialsites*: comprende la tabla donde se vuelcan los SPs minados. La herramienta extrae datos e inserta datos en esta tabla, por lo que se cuenta con funciones que se encargan de estas tareas.

```
Type 'exit' to Exit
Introduce consulta: desc specialsites
('id', 'int(11)', 'NO', 'PRI', None, 'auto_increment')
('latitud', 'decimal(12,5)', 'NO', '', None, '')
('longitud', 'decimal(12,5)', 'NO', 'MUL', None, '')
('Cluster', 'varchar(25)', 'NO', 'UNI', None, '')
('Description', 'varchar(200)', 'NO', '', None, '')
('tstamp', 'timestamp', 'YES', '', 'CURRENT_TIMESTAMP', '')
Introduce consulta: █
```

Ilustración 30 *specialsites_table*.

Los parámetros de esta tabla son los necesarios para describir el clúster seleccionado para guardarla como SP. Por lo que cuenta con el *id* característico para la identificación del registro. Además de:

- *latitud*: coordenada Y de la posición del SP.
- *longitud*: coordenada X de la posición del SP.
- *Cluster*: denominación, nombre del SP. Se realiza mediante la sucesión de autoincremento “C – *valor incremental*”.

- *Description*: ubicación en forma de dirección del SP.
- *tstamp*: campo automático, al igual que el *id*, comprende la fecha y hora de dada de alta del registro.

La tabla de *specialsites* es utilizada en todas las funcionalidades de la herramienta tanto principales como secundarias, comprende la tabla principal, puesto que registra los SPs que son utilizados para el trazado de segmentos y rutas. Las funcionalidades de clusteo y minado de SPs vuelcan los registros en la tabla y el resto de funcionalidades, incluidas la de clusteo, extraen los registros de *specialsites*.

Es necesaria la especial atención en la extracción de los registros referentes a la *ubicación*, es decir *latitud* y *longitud*. Tanto en la extracción de la tabla *movements* como en la tabla *specialsites*. La necesidad de concordancia en la extracción y el manejo de las coordenadas, puesto que dos errores en el orden entre dos variables equivalen a un acierto, pero el error existe y ante el manejo de una variable errónea resulta caótico recuperar el acierto. Es por ello que en las funciones selectoras que posean *ubicacion* en sus parámetros se trabaja siempre con la primera coordenada como *longitud* y la segunda como *latitud*. A excepción de las funciones selectoras que seleccionen todos los parámetros mediante la sentencia *SELECT **, ya que en la tabla de *specialsites* viene dada la *latitud* como primera coordenada, por lo que será necesario alternar el orden en su manejo.

- *segments*: comprende la tabla donde se vuelcan los registros de los segmentos de ruta.

```
Type 'exit' to Exit
Introduce consulta: desc segments
('id', 'int(11)', 'NO', 'PRI', None, 'auto_increment')
('Dateseg', 'date', 'NO', 'MUL', None, '')
('desde', 'time', 'NO', '', None, '')
('hasta', 'time', 'NO', '', None, '')
('Num', 'int(11)', 'NO', '', None, '')
('idorg', 'int(11)', 'NO', 'MUL', None, '')
('idend', 'int(11)', 'NO', 'MUL', None, '')
('matricula', 'varchar(25)', 'NO', 'MUL', None, '')
('PointA', 'varchar(100)', 'YES', '', None, '')
('PointB', 'varchar(100)', 'YES', '', None, '')
('Kms', 'double', 'YES', '', None, '')
('Vmax', 'double', 'YES', '', None, '')
('Vavg', 'double', 'YES', '', None, '')
('Obs', 'text', 'YES', '', None, '')
('ts', 'timestamp', 'NO', '', 'CURRENT_TIMESTAMP', 'on update CURRENT_TIMESTAMP')
Introduce consulta: |
```

Ilustración 31 *segments_table*.

La tabla *segments* es la tabla en la que más registros se introducen desde la herramienta, por lo que, al ejercer la funcionalidad del trazado, esta tabla está en continuo sometimiento a extracción e inserción por parte de las funciones de trazado.

La tabla cuenta con numerosos parámetros:

- *Dateseg*: comprende la fecha del segmento trazado.
- *desde*: comprende la hora en la que fue iniciado el segmento.
- *hasta*: comprende la hora de finalización del segmento.
- *Num*: comprende el número de segmento en ruta.
- *idorg*: trata del *id* perteneciente a la tabla *movements* con el que da comienzo el segmento.
- *idend*: trata del *id* perteneciente a la tabla *movements* con el termina el segmento.
- *matricula*: comprende la identificación del vehículo.
- *PointA*: comprende el SP del cual se inicia el segmento.
- *PointB*: comprende el SP donde se finaliza el segmento.
- *Kms*: comprende la cuantía de kilómetros recorridos en el segmento.
- *Vmax*: comprende la velocidad máxima alcanzada en el segmento.
- *Vavg*: comprende la velocidad media del segmento, realizada a partir de una media de los registros de velocidad totales del segmento.
- *Obs*: como abreviación de *Observaciones*, comprende el campo donde introducir particularidades del segmento, que no inicie o finalice en un SP, además de indicar los SPs por los que pasa, aunque no finalice el segmento en ellos.
- *ts*: comprende la fecha y hora de inserción en la BBDD.

- *routes*: es la tabla donde se guardan y extraen los registros generados de las rutas completas.

```
Type 'exit' to Exit
Introduce consulta: desc routes
('id', 'int(11)', 'NO', 'PRI', None, 'auto_increment')
('Date', 'date', 'NO', 'MUL', None, '')
('desde', 'time', 'NO', '', None, '')
('hasta', 'time', 'NO', '', None, '')
('matricula', 'varchar(20)', 'NO', 'MUL', None, '')
('NumSeg', 'int(11)', 'YES', '', None, '')
('Idorg', 'int(11)', 'NO', '', None, '')
('Idend', 'int(11)', 'NO', '', None, '')
('NumParadas', 'int(11)', 'NO', '', None, '')
('DurViajes', 'time', 'YES', '', None, '')
('DurParadas', 'time', 'YES', '', None, '')
('seqParadas', 'varchar(300)', 'NO', '', None, '')
('Kms', 'float', 'NO', '', None, '')
('Vmax', 'float', 'NO', '', None, '')
('Vavg', 'float', 'NO', '', None, '')
('DiaSem', 'int(11)', 'NO', '', None, '')
('ts', 'timestamp', 'NO', 'MUL', 'CURRENT_TIMESTAMP', 'on update CURRENT_TIMESTAMP')
Introduce consulta: █
```

Ilustración 32 *routes_table*.

La tabla *routes* es la tabla que más información aporta acerca del trayecto del vehículo, no solo porque abarque más tiempo y espacio al ser el conjunto de segmentos, sino porque además posee campos con mayor información útil.

- *Date*: comprende la fecha de la ruta trazada.
- *desde*: comprende la hora de iniciación de la ruta, la hora de iniciación del primer segmento.
- *hasta*: comprende la hora de finalización de la ruta, la hora de finalización del último segmento.
- *matricula*: comprende la identificación del vehículo.
- *NumSeg*: indica el número de segmentos que posee la ruta.
- *Idorg*: trata del *id* perteneciente a la tabla de *segments* que da lugar al primer segmento.
- *Idend*: trata del *id* perteneciente a la tabla de *segments* que da lugar al último segmento.
- *NumParadas*: indica el número de paradas que se han dado en la ruta.
- *DurViajes*: comprende la duración total de todos los viajes.
- *DurParadas*: comprende la duración total de todas las paradas.

- *seqParadas*: comprende la secuencia de los SPs en los que realiza la finalización e inicialización del segmento.
- *Kms*: indica los kilómetros totales de la ruta completa.
- *Vmax*: indica la velocidad máxima de la ruta.
- *Vavg*: indica la velocidad media de la ruta, realizada mediante la media ponderada entre las velocidades medias de los segmentos.
- *DiaSem*: indica el día de la semana en el que fue realizada la ruta.
- *ts*: indica la fecha y hora en la que fue dado de alta el registro.

2.2.2.4. FUNCIONES.

Una vez explicada la sintaxis de las consultas y las tablas junto a sus parámetros procederemos a entrar de lleno en las funciones con las que cuenta la herramienta dentro de la clase de *Database()*.

Las funciones que se encuentran en la clase son llamadas por todas las funcionalidades de la herramienta, a excepción de aquellas que cuenten de por si con su propia conexión con la BBDD, como son *Search in DDBB* y *Reset DDBB*.

El método para describir las funciones constará en explicar la funcionalidad que desempeña la función, los parámetros que recoge de la tabla correspondiente y el archivo donde es importada.

A continuación se muestran las funciones del archivo de database.py.

- *select_last_month*: el propósito de la función es seleccionar la ubicación de los registros de la tabla *movements*.

```
def select_last_month(self):
    sql = '''SELECT X(ubicacion) as lon, Y(ubicacion) as lat  FROM movements WHERE fecha_upos >= ''' + "" + \
          f_20 + """
    try:
        self.cursor.execute(sql)
        users = self.cursor.fetchall()

    return users

except Exception as e:
    raise
```

Ilustración 33 select_last_month_func.

Como se puede apreciar en la ilustración, la función hace uso de la variable *f_20* descrita en el inicio del capítulo. Selecciona la *ubicacion* con la *longitud* como primera coordenada y la *latitud* como la segunda.

Mediante *fetchall()* hace la selección de todos los registros que coinciden con *fecha_upos >= f_20*.

La función es llamada en la funcionalidad de *Satter search*.

- *select_dayselect*: el propósito de la función es seleccionar la posición de los registros de la tabla *movements* en un cierto periodo de tiempo definido por dos fechas a elegir.

```
def select_dayselect(self, pastdate, pastdate30):
    sql = '''SELECT X(ubicacion) as lon, Y(ubicacion) as lat  FROM movements WHERE fecha_upos >= ''' + "''"+ \
          pastdate + "'' and fecha_upos <= " + pastdate30 + "''"

    try:
        self.cursor.execute(sql)
        users = self.cursor.fetchall()

    return users

except Exception as e:
    raise
```

Ilustración 34 *select_dayselect_func.*

Como se puede apreciar en la ilustración, la función posee dos argumentos, los cuales corresponden a las fechas a introducir, de manera libre, en la función.

Mediante *fetchall()* hace la selección de las ubicaciones de los vehículos en los días comprendidos entre las fechas introducidas.

La función es llamada para la funcionalidad de *Scatter Search*.

- *select_users_20* y *select_users_30_10*: el propósito de las funciones es el de seleccionar *id*, *matricula*, *coordenadas de ubicación* y *fecha_upos* de la tabla *movements*, pero solo aquellos registros que cumplan que *velocidad* sea “0.0”, a partir de la fecha definida por la variable *f_20*, y entre las fechas definidas por *f_30* y *f_10*.

| | |
|--|---|
| <pre>def select_users_20(self): sql = str_1 try: self.cursor.execute(sql) users = self.cursor.fetchall() return users except Exception as e: raise</pre> | <pre>def select_users_30_10(self): sql = str_2 try: self.cursor.execute(sql) users = self.cursor.fetchall() return users except Exception as e: raise</pre> |
|--|---|

Ilustración 35 *select_users_20/30_func.*

Como se puede apreciar, son funciones semejantes, con la particularidad de que seleccionan los datos para fechas distintas. Las variables que se utilizan, *str_1* y *str_2* son las variables comunes descritas al principio del capítulo.

Ambas funciones se incorporan en el archivo de clust.py y pertenecen a la funcionalidad de *Clustering SP*.

- *select_users_compr*: el propósito de la función es la selección completa de todos los parámetros de un registro que coincide con la ubicación definida en la tabla *movements*.

```
def select_users_compr(self, lon, lat):
    str_compr = '''SELECT * FROM movements ''' \
        '''WHERE Y(ubicacion) = ''' + "" + lat + "' and X(ubicacion) = " + "" + lon + ""
    sql = str_compr
    try:
        self.cursor.execute(sql)
        users = self.cursor.fetchone()

        return users

    except Exception as e:
        raise
```

Ilustración 36 select_users_compr_func.

Comprende de una función selectora cuya particularidad, es que carece de la función *fetchall()*, en su lugar se encuentra *fetchone()* con lo que selecciona un único registro dentro de la lista de coincidencias con la sentencia WHERE.

La función pertenece a la funcionalidad de *Clustering SP*, en la parte del algoritmo encargada de la definición de los SPs.

- *select_users_past*: el propósito de la función es la selección de *id*, *matrícula*, *ubicación* y *fecha_upos*, de los registros de la tabla *movements* que posean *velocidad* con valor 0.0 y *fecha_upos* comprendida entre dos fechas: una definida en el argumento y otra como variable agregada.

```
def select_users_past(self, pastdate):
    pastdate_40 = (pastdate + timedelta(days=+25)).strftime('%Y-%m-%d %H:%M:%S')

    str_past = '''SELECT id, matricula, X(ubicacion) as lon, Y(ubicacion) as lat, fecha_upos as fecha FROM ''' \
        '''movements WHERE velocidad = '0.0' and fecha_upos >= '''

    str_past_2 = str_past + "" + pastdate.strftime(
        '%Y-%m-%d %H:%M:%S') + " and fecha_upos <= " + pastdate_40 + ""

    sql = str_past_2
    try:
        self.cursor.execute(sql)
        users = self.cursor.fetchall()

        return users

    except Exception as e:
        raise
```

Ilustración 37 select_users_past_func

Como se puede observar, el argumento de la función contiene la variable a introducir en la consulta, `pastdate`. Además de ello, se define `pastdate_40` a partir de la primera variable. Ambas variables constan de fechas, `pastdate` a elegir por el usuario, y `pastdate_40` definida como la fecha correspondiente a 25 días posteriores mediante la función `timedelta` de la librería `datetime`.

Esta función se incorpora a la funcionalidad de Clustering SP.

- `remove_allSP`: el propósito de la función es eliminar todos los registros de la tabla de `specialsites`.

```
def remove_allSP(self):
    sql = """DELETE from specialsites"""

    try:
        self.cursor.execute(sql)
        self.connection.commit()

    except Exception as e:
        raise
```

Ilustración 38 remove_allSP_func.

La sintaxis de la consulta no especifica ningún parámetro de la tabla a seleccionar para su eliminación, es por ello que procede a la eliminación de todos los registros de la tabla.

La función de `commit()` es análoga a guardar los cambios, es necesaria en las funciones insertoras y las eliminadoras puesto que sin ella no se realizan cambios en las tablas.

Esta función pertenece a la funcionalidad de Clustering SP.

- `addSP`: el propósito de la función es la inserción de los parámetros descriptivos de los SPs en la tabla de `specialsites`.

```
def addSP(self, lat, lon, cluster, desc):
    sql = """INSERT INTO specialsites(latitud, longitud, Cluster, Description) VALUES ('" + lat + "', '" + lon + \
          "', '" + cluster + "', '" + desc + "')"

    self.cursor.execute(sql)
    self.connection.commit()

    print(self.cursor.rowcount, "record was inserted.")
```

Ilustración 39 addSp_func.

La particularidad de esta función es que imprime en pantalla los registros que se van insertando en la tabla mediante `rowcount`, el cual es el encargado de contar los registros que ingresan en la tabla.

La función pertenece a la funcionalidad de Clustering SP.

- *select_sp*: el propósito de la función es seleccionar únicamente la ubicación de los SPs de la tabla *specialsites*.

```
def select_sp(self):
    sql = """SELECT longitud as lon, latitud as lat FROM specialsites"""

    try:
        self.cursor.execute(sql)
        sp = self.cursor.fetchall()

    return sp

    except Exception as e:
        raise
```

Ilustración 40 *select_sp_func.*

La variable sp que devuelve la función consta de una lista de ubicaciones correspondientes a las coordenadas de los SPs registrados.

La función se incorpora en la mayoría de funcionalidades de la herramienta.

- *select_sp_name*: el propósito de la función es el mismo que la función anterior sumado a la selección de la denominación del SP junto a la ubicación.

```
def select_sp_name(self):
    sql = """SELECT longitud as lon, latitud as lat, Cluster FROM specialsites"""

    try:
        self.cursor.execute(sql)
        sp = self.cursor.fetchall()

    return sp

    except Exception as e:
        raise
```

Ilustración 41 *select_sp_name_func.*

El motivo por el cual las dos funciones anteriores no comprenden la misma es la simplicidad a la hora de separarlas para los propósitos que cumplen. Esta función pertenece únicamente a la funcionalidad de visualizado, mientras que *select_sp* pertenece a la mayoría de funcionalidades, por lo que se debería particularizar todas las llamadas a *select_sp* para la selección de los dos primeros datos que devuelve. Es por ello que se opta por directamente hacer otra función que cumpla para el único propósito al que sirve.

- *select_mat_0*: el propósito de la función es de seleccionar la matrícula de los registros de un día concreto de la tabla de *movements*.

```
def select_mat_0(self, date_):
    date_1 = datetime.strptime(date_, '%Y-%m-%d %H:%M:%S') + timedelta(days=1)

    sql = f"""SELECT matricula FROM movements WHERE fecha_upos >= '{date_}' and fecha_upos < {date_1}"""

    try:
        self.cursor.execute(sql)
        mat = self.cursor.fetchall()

        return mat

    except Exception as e:
        raise
```

Ilustración 42 *select_mat_0* func.

Al pretenderse extraer los registros de un día concreto a elegir, ha de realizarse en forma de intervalo temporal, puesto que *fecha_upos* posee formato “Y-m-d H:M:S” no es posible en la sentencia WHERE igualar una fecha a *fecha_upos*. El proceso se da de manera que *fecha_upos* se encuentre en el intervalo de las 00:00:00 de una fecha a seleccionar mediante la variable del argumento y las 00:00:00 del día posterior.

La función pertenece a la funcionalidad del trazado de segmentos en *Routing*.

- *select_seg*: el propósito de la función es la selección de los parámetros necesarios para el trazado de un segmento de ruta desde la tabla *movements*.

```
def select_seg(self, mat, time, time_1):
    sql = f"""SELECT X(ubicacion), Y(ubicacion), s_motor, fecha_upos, ID, velocidad FROM movements WHERE \
matricula = {mat} and fecha_upos >= '"""+ time + """" and fecha_upos <= '"""+ time_1 + """"

    try:
        self.cursor.execute(sql)
        seg = self.cursor.fetchall()

        return seg

    except Exception as e:
        raise
```

Ilustración 43 *select_seg* func.

La función es realizada de manera que selecciona la *ubicación*, *s_motor*, *fecha_upos*, *ID* y *velocidad* para un vehículo concreto identificado por *mat* en un intervalo de tiempo comprendido entre las variables *time* y *time_1*.

La función forma parte de la funcionalidad de trazado de segmentos de ruta de *Routing*.

- *select_data_forseg*: el propósito de la función es recoger los parámetros necesarios para el trazado del segmento de ruta a partir del *id* y no de la matrícula y el intervalo como la función anterior.

```
def select_data_forseg(self, id):
    sql = f'''SELECT s_motor, matricula, velocidad, X(ubicacion), Y(ubicacion), fecha_upos, localidad FROM \
    movements WHERE id = {id} '''

    try:
        self.cursor.execute(sql)
        users = self.cursor.fetchone()

        return users

    except Exception as e:
        raise
```

Ilustración 44 *select_data_forseg_func.*

Como se puede apreciar, la función cuenta con *fetchone()* en vez de *fetchall()*, esto es debido a que mediante la selección a partir del *id*, carece de sentido seleccionar varios, ya que existe solo un único registro por cada *id*.

- *select_unique_sp*: el propósito de la función es el de seleccionar el SP correspondiente a las coordenadas que se suministran en el argumento.

```
def select_unique_sp(self, lon, lat):
    sql = '''SELECT * FROM ''' \
        f'''specialsites WHERE latitud = {lat} and longitud = {lon}'''

    try:
        self.cursor.execute(sql)
        users = self.cursor.fetchone()

        return users

    except Exception as e:
        raise
```

Ilustración 45 *select_unique_sp_func.*

Al igual que la anterior función, esta cuenta con *fetchone()*, ya que no pueden existir dos SPs con las mismas coordenadas. En el caso de que la herramienta falle y posea en la tabla de *specialsites* dos registros del mismo SP, seleccionando uno se salva del error.

La función pertenece a funcionalidades múltiples, tanto *Routing* como *Clustering SP*.

- *add_seg_toSQL*: el propósito de la función es realizar el ingreso del segmento de ruta en la tabla de *segments*.

```
def add_seg_toSQL(self, Dateseg, desde, hasta, Num, idorg, idend, mat, PointA, PointB, Kms, Vmax, Vavg, Obs):
    sql = """INSERT INTO segments(Dateseg, desde, hasta, Num, idorg, idend, matricula, PointA, PointB, Kms, Vmax,
    Vavg, Obs) VALUES ('" + Dateseg + "', '" + desde + "', '" + hasta + "', '" + Num + "', '" + idorg + "', '" +
    idend + "', '" + mat + "', '" + PointA + "', '" + PointB + "', '" + Kms + "', '" + Vmax + "', '" + Vavg +
    "', '" + Obs + "')"

    self.cursor.execute(sql)
    self.connection.commit()

    print(self.cursor.rowcount, "record was inserted in segments.")
```

Ilustración 46 *add_seg_toSQL_func*

Los parámetros obtenidos en la funcionalidad de *Routing* son incorporados a la BBDD mediante esta función. Los parámetros se introducen como argumento en la función y son utilizados en la incorporación mediante *VALUES*. Como ya se ha expresado en la definición de funciones insertoras, es de suma importancia prestar atención al orden en la sintaxis.

La función imprime en pantalla mediante *print* el recuento de registros que inserta en la tabla *segments*.

- *select_data_for_route*: el propósito de la función es seleccionar los datos pertinentes para el trazado de rutas completas a partir de segmentos desde la tabla de *segments*.

```
def select_data_for_route(self, mat, date_):
    sql = f"""SELECT desde, hasta, Num, Kms, PointA, PointB, Vmax, Vavg, idorg, idend, id FROM segments WHERE \
    Dateseg = '{date_}' and matricula = '{mat}' """

    try:
        self.cursor.execute(sql)
        users = self.cursor.fetchall()

    return users

except Exception as e:
    raise
```

Ilustración 47 *select_data_from_route_func*.

Mediante la introducción de una fecha con *date_* y una matrícula con *mat*, la función selecciona los segmentos que ha realizado un vehículo en una fecha concreta. Cabe destacar la distinción entre el parámetro *Dateseg* y el parámetro *fecha_upos*, la

diferencia es que *fecha_upos* comprende la fecha y la hora, mientras que *Dateseg* solo la fecha. Es por ello que *Dateseg* sí se puede igualar a una variable temporal, mientras que *fecha_upos* debe permanecer manejándose mediante intervalos.

La función pertenece a la funcionalidad de *Routing*.

- *add_route_toSQL*: el propósito de la función es la inserción de los registros de las rutas completas trazadas en la tabla *routes*.

```
def add_route_toSQL(self, Date, desde, hasta, matricula, NumSeg, Idorg, Idend, NumParadas, DurViajes, DurParadas,
                    seqParadas, Kms, Vmax, Vavg, DiaSem):
    sql = """INSERT INTO routes(Date, desde, hasta, matricula, NumSeg, Idorg, Idend, NumParadas, DurViajes,
                               DurParadas, seqParadas, Kms, Vmax, Vavg, DiaSem) VALUES ('{}', '{}', '{}', '{}', '{}', '{}', '{}', '{}', '{}',
                               '{}', '{}', '{}', '{}', '{}')"""
    sql = sql.format(Date, desde, hasta, matricula, NumSeg, Idorg, Idend, NumParadas, DurViajes, DurParadas,
                     seqParadas, Kms, Vmax, Vavg, DiaSem)
    self.cursor.execute(sql)
    self.connection.commit()

    print(self.cursor.rowcount, "record was inserted in routes.")
```

Ilustración 48 *add_route_toSQL_func*.

Se aplican al argumento de la función los parámetros a añadir a la BBDD y posteriormente se insertan en la sintaxis mediante la sentencia VALUES.

La función imprime en pantalla el recuento de registros insertados en la *routes* y pertenece a la funcionalidad de *Routing*.

- *select_mat_fromroutes*: el propósito de la función es la selección de matrículas para la identificación de los vehículos de la tabla *routes*.

```
def select_mat_fromroutes(self, date_):
    sql = f"""SELECT matricula, NumSeg FROM routes WHERE Date = '{date_}'"""

    try:
        self.cursor.execute(sql)
        users = self.cursor.fetchall()

    return users

except Exception as e:
    raise
```

Ilustración 49 *select_mat_fromroutes_func*.

La función selecciona la matrícula de todos los registros de una fecha concreta, introducida en el argumento y pertenece a la funcionalidad de *Mapping*.

- *select_data_fromsegments*: el propósito de la función es la selección de todos los parámetros de los registros de la tabla *segments* que coincidan con una fecha, una matrícula y un número de segmento en ruta.

```
def select_data_fromsegments(self, date_, mat, Num):
    sql = f"""SELECT * FROM segments WHERE Dateseg = '{date_}' and matricula = '{mat}' and Num = '{Num}'"""

    try:
        self.cursor.execute(sql)
        users = self.cursor.fetchall()

    return users

except Exception as e:
    raise
```

Ilustración 50 select_data_fromsegments.

La función selecciona un segmento concreto definido, además de por la fecha y el vehículo, por el número de segmento en ruta que consta.

Se devuelve como variable una tupla con los datos del segmento seleccionado.

La función pertenece a la funcionalidad de Mapping.

- *select_data_frommovements*: el propósito de la función es la selección de las ubicaciones de los registros de la tabla movements a partir de sus *ids* y su matrícula.

```
def select_data_frommovements(self, idorg, idend, matricula):
    sql = f"""SELECT X(ubicacion), Y(ubicacion) FROM movements WHERE id >= '{idorg}' and id <= '{idend}' and
    matricula = '{matricula}'"""

    try:
        self.cursor.execute(sql)
        users = self.cursor.fetchall()

    return users

except Exception as e:
    raise
```

Ilustración 51 select_data_frommovements_func.

Gracias a la clave primaria de los *ids* y su autoincremento, es posible manejarlos como los parámetros temporales, es decir, es conocido que su disposición en la tabla es ordenada. Por lo que ante la selección de las posiciones de un vehículo con registros definidos es posible mediante el intervalo en sus *ids*.

Otra manera de realizar esta acción es posible mediante la lista de *ids* que comprenden la ruta y una función que recoja la posición con el *id* como argumento de entrada. El problema es que la ejecución de la acción es necesariamente realizada por un bucle *for*, por lo que se opta por la primera opción.

La razón por la que no se opta por la opción del bucle es que se tardaría más en realizar la acción, se consume más memoria RAM, y el código se complica de manera innecesaria. Este es uno de los ejemplos de optimización de código.

La función pertenece a la funcionalidad de *Mapping*.

A continuación, se explican las funciones eliminatorias generales.

- *remove_route* // *remove_seg*: el propósito de las funciones es eliminar los registros de las tablas *routes* o *segments* los registros de una fecha concreta.

```
def remove_route(self, date_):
    sql = f"""DELETE FROM routes WHERE Date = '{date_}'"""

    try:
        self.cursor.execute(sql)
        self.connection.commit()

        print(self.cursor.rowcount, "record(s) were deleted in routes.")

    except Exception as e:
        raise
```

Ilustración 52 remove_route_func.

```
def remove_seg(self, date_):
    sql = f"""DELETE FROM segments WHERE Dateseg = '{date_}'"""

    try:
        self.cursor.execute(sql)
        self.connection.commit()

        print(self.cursor.rowcount, "record(s) were deleted in segments.")

    except Exception as e:
        raise
```

Ilustración 53 remove_seg_func.

Las funciones se distinguen en la tabla en la que realizan las acciones. Se utilizan, al igual que *remove_allSP* para evitar la repetición de los registros en las tablas donde se ingresan posteriormente, los registros eliminados actualizados, su propósito es el mismo.

La diferencia con *remove_allSP* es la introducción de una fecha para realizar la eliminación. Esto es debido a que las funciones insertoras de segmentos y rutas se ejecutan a diario, mientras que la función que se encarga de actualizar los SPs, tiene previsto realizarse cada más de dos años.

Sin estas funciones, al repetir el trazado de segmentos y rutas en una misma fecha, los registros se repetirían.

Las funciones pertenecen a la funcionalidad de *Routing*.

- *remove_allsp/ remove_allseg/ remove_allroutes*: el propósito de estas funciones es la eliminación completa de todos los registros de las tablas a las que hacen referencia.

```

def remove_allseg(self):
    sql = """DELETE FROM segments"""

    try:
        self.cursor.execute(sql)
        self.connection.commit()

        print(self.cursor.rowcount, "record(s) were deleted from segments.")

    except Exception as e:
        raise

def remove_allroutes(self):
    sql = """DELETE FROM routes"""

    try:
        self.cursor.execute(sql)
        self.connection.commit()

        print(self.cursor.rowcount, "record(s) were deleted from routes.")

    except Exception as e:
        raise

def remove_allsp(self):
    sql = """DELETE FROM specialsites"""

    try:
        self.cursor.execute(sql)
        self.connection.commit()

        print(self.cursor.rowcount, "record(s) were deleted from specialsites.")

    except Exception as e:
        raise

```

Ilustración 54 remove_func.

Estas funciones pertenecen a la funcionalidad de *Reset DDBB*. Comprenden la eliminación indiscriminada de registros en las tablas de la BBDD e imprimen en pantalla el conteo de los registros que han eliminado.

La función de reseteo consta en devolver a su estado inicial aquello a lo que se está reseteando, en este caso vaciar las tablas.

- *connection_close*: el propósito de la función es cerrar la conexión con la BBDD.

```
def connection_close(self):  
    self.connection.close()
```

Ilustración 55 connection_close_func.

Es importante enfatizar que la razón por la cual se debe cerrar la conexión con la BBDD. Mientras se mantenga abierta una conexión se consumen recursos en la BBDD, memoria. Las BBDD están configuradas de manera que son capaces de mantener un número máximo de conexiones, por lo si no se cierran las conexiones, aumenta la probabilidad de quedarse sin ellas.

Esta función no pertenece a ninguna funcionalidad, se encuentra en el primer bucle del algoritmo del archivo main.py, dentro de la opción de cerrar la herramienta.

Una vez se han explicado archivos main.py y database.py, se puede proceder con la explicación de las funcionalidades de la herramienta entrando a fondo en los algoritmos que las componen.

Damos paso así al siguiente capítulo mostrando y explicando la funcionalidad de Clustering SP.

CAPÍTULO 3

CLUSTERING SP

3.1. Introducción.

Clustering SP es la funcionalidad de la herramienta que se encarga de la extracción o minado de SPs a partir de la formación de clústeres de las diferentes ubicaciones que toma el conjunto de los vehículos de la BBDD. El proceso se realiza para un periodo de tiempo definido.

Como ya hemos visto en el capítulo anterior, los registros se obtienen desde la tabla *movements* a partir de funciones selectoras. Los registros se componen de diferentes parámetros (matrícula, velocidad a la que se encuentra el vehículo, ubicación, etc.) de los cuales se selecciona la ubicación en forma de coordenadas cartesianas para su posterior tratado.

Para entender el proceso por el cual se extraen los SPs se debe entender previamente en qué consiste el clusteo de los registros. A partir de ahora, en el capítulo, nos referiremos a los registros, al solo extraer las ubicaciones, como posiciones. Por lo tanto, debemos entender en qué consiste el clusteo de las posiciones.

La palabra clúster proviene del inglés y cuyo significado es el de “grupo”. Entendemos las posiciones como puntos definidos en un plano bidimensional, y los grupos que se forman dependerán de la densidad superficial de aquellos puntos. En otras palabras, dependen de la separación entre los puntos (distancia) y la cantidad de puntos que forman el grupo.

En la siguiente ilustración se puede apreciar los grupos que se forman en un conjunto de puntos, sobre los cuales se ha aplicado un algoritmo de clusteo. El resultado comprende de tres clústeres que abarcan gran cantidad de puntos.

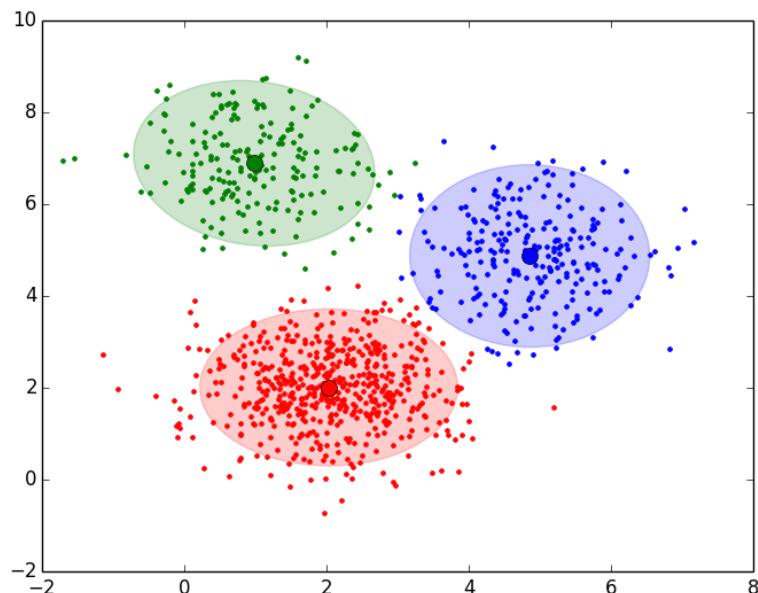


Ilustración 56 cluster_definition.

Como es de esperar, el clusteo es relativo, por ello se abre una discusión sobre los parámetros correctos en la aplicación del algoritmo que realiza el clusteo del conjunto. Usaremos la ilustración anterior para explicar la relatividad del clusteo. Ante una selección de la densidad de puntos diferente, el resultado puede abarcar desde la formación de un único clúster hasta la formación de clústeres de un solo punto. Por ejemplo: los registros de posiciones de los

vehículos en España, formarán un único clúster si realizamos el proceso con la superficie total de Europa, puesto que resultan puntos muy juntos en comparación.

La clave es dar con los parámetros de: distancia entre puntos y los puntos mínimos, para la formación de clústeres que resulten útiles.

Además de ello, la selección adecuada de los parámetros de clusteo evita la agregación de puntos que de forma natural no pertenecen al clúster, entendemos estos puntos sueltos como *puntos huérfanos*.

Es por ello que los parámetros de los algoritmos de clusteo creados para la herramienta poseen parámetros adecuados para cada propósito, es decir, que para los distintos procesos en los que se lleva a cabo el clusteo, se cuenta con un estudio para la selección adecuada de los parámetros.

Entendemos pues, que los clústeres resultantes que se extraigan de los registros comprenderán los SPs potenciales a volcar a la BBDD.

Los SPs son áreas que cuentan con una gran concurrencia por parte de los vehículos de la flota. El propósito de esta funcionalidad es encontrar aquellos puntos donde los vehículos realizan sus paradas, ya sean cargas, descargas, descansos, o finalizaciones de los trayectos. Es precisamente por esto por lo que se seleccionan los registros que posean *velocidad 0.0*, aunque se cuenta con los registros que obtengan esa velocidad sin estar realizando una parada, por ejemplo, un registro que conste una detención ante un semáforo, entendemos estos puntos como *puntos de ruido*.

Posteriormente, ante la creación de los clústeres, se selecciona el punto representativo del clúster. Ante la obtención de estos SPs, se realiza una depuración por la posibilidad de obtener SPs muy juntos. Posteriormente, se agregan los SPs de la tabla de *specialsites* y se comparan con los obtenidos hasta ahora, se vuelven a depurar y los SPs que no son los mismos que los que ya se encuentran en la BBDD, los *SPs de tercera generación* son los que se vuelcan a la tabla *specialsites*.

Todos los procesos descritos son recopilados en funciones para posteriormente importarlos al archivo main.py. Los archivos que recopilan estas funciones son: clust.py, SP20.py y add_toSQL.py.

3.2. clust.py.

3.2.1. Introducción.

El archivo clust.py es el archivo que comprende las funciones de clusterizado de los registros. Las funciones se rigen por la misma estructura y se distinguen por los propósitos que cumplen, realizan la misma acción con ciertas particularidades.

Las funciones tienen variables comunes para la realización del clústeo, estas son:

- *eps*: distancia máxima entre puntos pertenecientes al mismo clúster, es decir, la distancia máxima a la que tiene que encontrarse un punto cualquiera con otro perteneciente a un clúster para pertenecer a él.
- *mpts*: el número mínimo de puntos que tienen que darse para la formación de un clúster.
- *users*: lista de registros.

Se distinguen tres funciones principales: *clustering_20*, *clustering_past* y *sec_depur*. Todas ellas son funciones que realizan la extracción de clústeres. Como argumento, las funciones tienen las tres variables comunes. Su estructura principal se muestra de la siguiente forma:

- Como primer paso se listan las posiciones mediante las coordenadas de longitud y latitud. La lista de registros suministrados por la variable *users* contiene parámetros que no interesan en el clústeo, es por ello que se listan a partir de un bucle *for* que recorra la variable y seleccione los parámetros de cada tupla que interesan. Siendo estas la posición en coordenadas geográficas, colocándose en las posiciones 2 y 3 de cada tupla.

```
locationsForDBSCAN = [list([user[2], user[3]])  
                      for user in users_past]  
  
x = np.array(locationsForDBSCAN)
```

Ilustración 57 users_list.

Posteriormente se transforma en una estructura *array*(formación) a través de la librería *numpy*.

- Como segundo paso se corrige el parámetro *eps*.

```
AVERAGE_CIR = 6378.137 * 1000 * 2 * math.pi  
eps = eps * 360 / AVERAGE_CIR
```

Ilustración 58 eps_correction.

El problema con el manejo de los datos según se importan es que el parámetro *eps* se encuentra dado en metros, mientras que la lista de la variable a clusterizar se encuentra en coordenadas geográficas, las cuales comprenden los ángulos de longitud y latitud. Como se pretende graficar el resultado según la posición en coordenadas cartesianas, es mejor opción transformar el parámetro *eps* en su equivalencia angular geográfica. Para ello se calcula la circunferencia media de la Tierra asemejando el planeta a una esfera, posteriormente se hace una regla de tres transformando el parámetro dado en metros a un parámetro dado en grados.

- Ya se puede realizar el clusteo, mediante la función *DBSCAN* de la librería *sklearn*.

```
db = DBSCAN(eps=eps, min_samples=mpts,  
            metric='euclidean').fit(x)
```

Ilustración 59 *DBSCAN*.

A parte de los parámetros de *eps* y *mpts*, se especifica la métrica que ha de seguir para el cálculo, asemejando la superficie de los registros a un plano bidimensional para seguir la métrica euclídea.

Posteriormente se utiliza la función *fit* para aplicar el clusteo a la lista de posiciones.

Se escoge *fit* frente a *fit_predict*, la cual también es otra opción para el algoritmo, el cual posee características diferentes.

A esta altura en el algoritmo, tras aplicar la función de clusteo, poseemos una lista de puntos de los clústeres que se han formado.

- *DBSCAN* devuelve una matriz de 1 y 0 con la que no sirve para el proceso, por lo que es necesaria la extracción de las posiciones de la variable a través de *components_*.

```
# PARA SPECIAL POINTS  
cores = db.components_
```

Ilustración 60 *db_cores*.

Se almacena la lista de posiciones de los registros que forman clústeres en la variable *cores*.

- Por último, se realiza una selección de un punto representativo de los clústeres el cual será denominado SP de primera generación. Además de ello, las funciones contienen el graficado del resultado solapando las posiciones de las ubicaciones totales de los registros y sobre ello, las posiciones que forman parte de los clústeres formados, con lo que se aprecia en la grafía la formación de los clústeres.

```

clusters = []
for i in range(0, len(cores)):
    if len(clusters) > 0:
        counter = 0
        for clust in clusters:
            if abs(cores[i, 0] - clust[0]) <= eps and abs(cores[i, 1] - clust[1]) <= eps:
                counter = counter + 1

    if counter == 0:
        clusters.append(cores[i])

    else:
        clusters.append(cores[i])

```

Ilustración 61 clust_selection.

El proceso por el cual se realiza la selección es complejo, para almacenar los puntos representativos de cada clúster se crea la variable de lista vacía *clusters*.

Con un bucle for se hace que, mediante in range, la variable i adquiera los valores desde 0 hasta el valor correspondiente a la longitud de la lista cores con len(cores) a paso de 1.

Posteriormente se obtienen los condicionales que cuestionan si la lista clusters está vacía o no, mediante len(clusters) > 0, es decir, si la longitud de la lista es mayor que 0.

En caso contrario, else, se procede a agregar el elemento número i de cores mediante la función append que añade a la lista lo que posea en el argumento (el caso en el que la lista clusters se encuentra vacía es el caso de primera adición, por lo que i estará tomando el valor de 0 en ese instante). Esto se realiza con el propósito de evitar el error que aparecería si se procediese directamente a referirse a elementos de la lista, siendo esta vacía.

A continuación, se crea un contador y otro bucle for con el que se recorre la lista de clusters denominando cada elemento como clust.

El siguiente condicional es el elemento principal de la selección de puntos representativos. En caso de cumplirse el condicional, el contador deja de ser 0, y el siguiente condicional no se cumple para agregar la posición a la lista de puntos representativos.

Este condicional consta de una doble comparación, resulta del valor absoluto de la resta entre las coordenadas de la posición que se está valorando añadir a la lista de puntos representativos y los puntos representativos ya seleccionados. Si ambas restas resultan menores que el valor de eps (en coordenadas geográficas), se cumple el condicional y no se añade a la lista clusters.

El propósito de esta selección es escoger un punto para situar cada clúster que se ha formado, es por ello que se pretende descartar aquellos que disten menos de eps de puntos representativos escogidos.

Explicado de manera contraria, sólo se escogerán como puntos representativos, los puntos pertenecientes a clústeres que disten más que eps de otros puntos representativos, puesto que no se reconoce el clúster al que pertenecen.

Por lo que el procedimiento será el siguiente: clusters se encuentra vacía, no contiene ningún punto representativo, por lo que se le añade el primer elemento de la lista cores, cores[0]. A continuación, si el siguiente elemento de cores pertenece al mismo clúster, es decir, si dista menos que eps del primer elemento de clusters, no se añade a la lista, en caso contrario sí. Pongamos el caso de que cores[1] pertenece a otro clúster, entonces dista más que eps y se añade a clusters, entonces cores[2] tiene que compararse con cores[0] y cores[1] que son los elementos de clusters. En caso de que diste menos que eps de alguno de los dos elementos, el contador aumenta en 1 y no se añade cores[2] a clusters. Por lo que un elemento genérico cores[i], se compara con todos los elementos de clusters para considerar su adición como punto representativo.

La sintaxis debe ser precisa, en el caso del contador es necesario situarlo dentro del primer bucle, pero fuera del otro, de esta manera vuelve a ser 0 cuando se completa una vuelta del bucle externo.

El problema de este método es que, ante la formación de clústeres grandes, cabe la posibilidad de que se elija un punto representativo del clúster que posea una determinada posición, y posteriormente se analice la posición de otro punto que forma parte del mismo clúster, pero diste más que eps del punto representativo, y se tome este último punto como otro punto representativo.

Como solución a este problema, se realiza, mediante otra función, una depuración de los puntos representativos. Se vuelve a realizar una clusterización con el parámetro de mpts siendo 1, es decir, los clústeres se forman con una sola posición como mínimo, y un eps mayor, para abarcar los posibles errores de la selección de puntos representativos.

El resultado de la segunda depuración es la formación de clústeres de 1 solo registro, y si algún clúster se da con 2 registros o más, significa entonces que ha ocurrido el error en la selección. Se vuelve, pues, a realizar una selección en los puntos representativos, de esta manera, queda como resultado únicamente un punto por clúster formado, es decir, los SPs.

3.2.2. Cuerpo.

3.2.2.1. IMPORTACIONES.

Como importaciones obtenemos librerías comunes a otros archivos, como son las de graficado y formaciones numéricas, pero también se obtienen librerías únicas en la herramienta para el archivo de *clust.py*.

```
import math
from database import DataBase
import matplotlib.pyplot as plt
from sklearn.cluster import DBSCAN
import numpy as np

database = DataBase()
```

Ilustración 62 *clust_imports*.

La primera de las importaciones, *math*, consta de una librería con herramientas matemáticas, como funciones, tablas y constantes.

La segunda importación es la función que se encuentra en todos los archivos de la herramienta, se importa la clase *Database* del archivo *database* y posteriormente se denomina *database* a la clase.

La librería *matplotlib* es una librería de graficado, de la cual se importa *pyplot* como *plt* para realizar esta función.

Se importa *DBSCAN* de la librería *sklearn.cluster* para realizar el clusteo.

Se importa la librería *numpy* como *np* para configurar la formación de las variables.

3.2.2.2. FUNCIONES.

Como primera función obtenemos *clustering*, pero se explicará más a delante debido a que utiliza funciones que se dan en el algoritmo, por lo que, para su comprensión, es necesario comprender antes las funciones que utiliza.

Pasamos pues a explicar *clustering_20*. La función tiene el propósito de, ante la entrada de una lista de registros, *eps* y *mpts*, devolver una lista de puntos representativos de los clústeres.

```

def clustering_20(users_20, eps, mpts):
    locationsForDBSCAN = [list([user[2], user[3]])
                           for user in users_20]

    x = np.array(locationsForDBSCAN)
    AVERAGE_CIR = 6378.137 * 1000 * 2 * math.pi
    eps = eps * 360 / AVERAGE_CIR

    db = DBSCAN(eps=eps, min_samples=mpts,
                metric='euclidean').fit(x)

    # PARA SPECIAL POINTS
    cores = db.components_

    clusters = []
    for i in range(0, len(cores)):
        if len(clusters) > 0:
            counter = 0
            for clust in clusters:
                if abs(cores[i, 0] - clust[0]) <= eps and abs(cores[i, 1] - clust[1]) <= eps:
                    counter = counter + 1

            if counter == 0:
                clusters.append(cores[i])

        else:
            clusters.append(cores[i])

    # GRAFICADO

    plt.gca().set_facecolor('white')
    plt.scatter(x[:, 1], x[:, 0], c='black')
    plt.scatter(cores[:, 1], cores[:, 0], c='aqua')
    plt.show()

    return clusters

```

Ilustración 63 clustering_20.

La función posee los elementos anteriormente descritos, con la particularidad del graficado, mediante *plt*. El proceso para el graficado es el siguiente. Como primer paso se define el fondo, mediante *set_facecolor*, con *gca* se ajusta los ejes de la figura a los ejes de la instancia, son las siglas de “*get current axes*”. Mediante *scatter* se colocan en la figura los puntos a definir en el argumento, la sintaxis que posee comprende, la primera coordenada, la segunda coordenada y el color. Los parámetros de la forma *x[:;1]* son instancias que indican: el elemento 1 de todas las filas de la matriz *x* (ya que *x* pasó a ser formación anteriormente).

Es importante el orden en el cual se hacen, puesto que el graficado, cuando hay varios *scatter*, se hace uno sobre el otro, por lo que el último que se realiza es el último que agrega sobre los demás. Es por ello que interesa dejar *cores* como último elemento a graficar, puesto que así se observan los clústeres formados.

El último elemento es *show()* para mostrar aquello que se ha graficado.

A continuación, se muestran algunas imágenes de lo que resulta del proceso de la función de *clustering_20*.

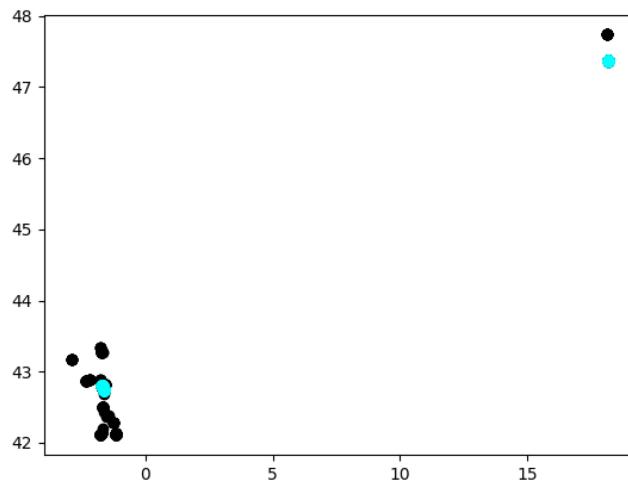


Ilustración 64 clusteo_1.

Podemos apreciar en la ilustración el resultado de la búsqueda de clústeres realizada para 25 días hasta el día Domingo 03 de enero de 2022. Resultan formaciones muy separadas puesto que se manejan los registros de España y Hungría. En la siguiente ilustración se hace zoom para mostrar únicamente los registros de España.

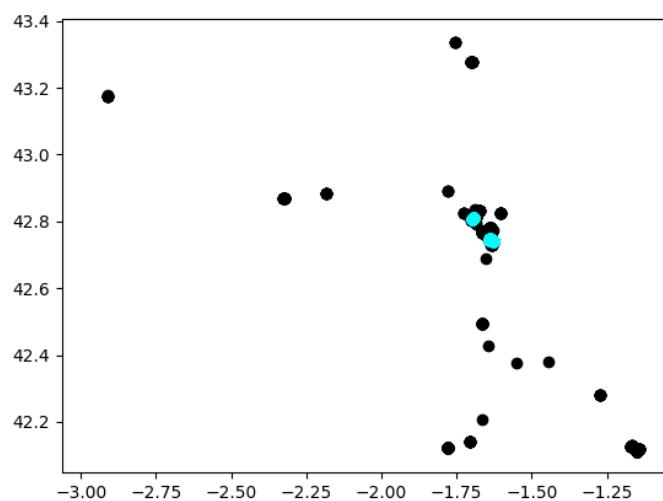


Ilustración 65 clusteo_1_zoom_1.

Podemos apreciar en la imagen los registros de los lugares donde los vehículos tienen velocidad 0.0 en España, los puntos que se precian en negro no son registros únicos, son grandes cúmulos que no llegan a ser clústeres, se aprecian como puntos debido a la falta de zoom. En las siguientes imágenes, se hace mayor zoom en el área donde se han dado la formación de los clústeres.

A continuación, se muestra una progresión de zoom en las áreas marcadas para la visualización de los elementos resultado de la función *clustering_20*.

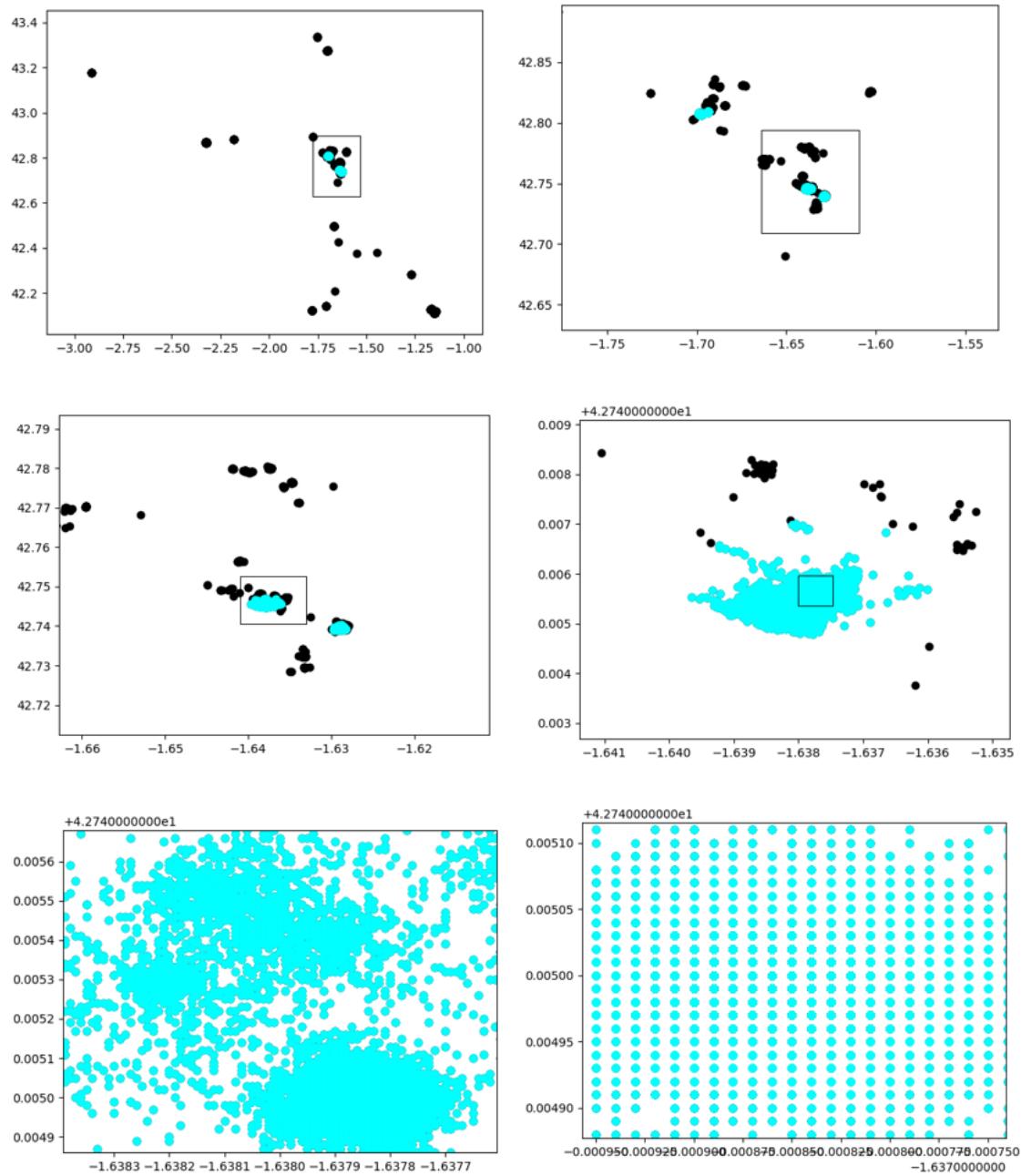


Ilustración 66 progresión_zoom_clust_1.

Observamos pues cómo se muestran los registros pertenecientes a los clústeres formados. La última imagen corresponde a la máxima precisión de los registros, puesto que las coordenadas geográficas se suministran con una determinada precisión, por lo que muchos de las posiciones que se muestran están repetidas en el mismo punto.

La siguiente función fundamental en el proceso de minado de SPs, es la depuración de clústeres detectados, para ello se utiliza *sec_depur*.

```
def sec_depur(clusters, eps):

    cls = np.array(clusters)

    AVERAGE_CIR = 6378.137 * 1000 * 2 * math.pi
    eps = eps * 3 * 360 / AVERAGE_CIR

    sp = DBSCAN(eps=eps, min_samples=1,
                metric='euclidean').fit(cls)

    sp_cores = sp.components_

    sp = []
    for i in range(0, len(sp_cores)):
        if len(sp) > 0:
            counter = 0
            for clust in sp:
                if abs(sp_cores[i, 0] - clust[0]) <= eps and abs(sp_cores[i, 1] - clust[1]) <= eps:
                    counter = counter + 1

            if counter == 0:
                sp.append(sp_cores[i])

        else:
            sp.append(sp_cores[i])

    return sp
```

Ilustración 67 *sec_depur_func.*

La función tiene el propósito de eliminar los puntos representativos de los clústeres falsos, para ello realiza un segundo proceso de cluterizado de los puntos suministrados para formar clústeres a partir de esos puntos y elegir un punto representativo de los clústeres nuevos.

Como particularidad, la función, posee como argumento la lista de puntos representativos *clusters* y *eps*, pero carece de *mpts* en su argumento, esto es porque en la depuración, el mínimo número de puntos para formar el clúster es de 1, y se fija en el argumento de *DBSCAN* directamente.

La función en sí, carece de graficado, pero para señalar el proceso que realiza se le dota del siguiente proceso:

```
sp_ = np.array(sp)
plt.gca().set_facecolor('white')
plt.scatter(cls[:, 1], cls[:, 0], c='red')
plt.scatter(sp_[:, 1], sp_[:, 0], c='aqua')
plt.show()
```

Ilustración 68 *graf_sec_depur.*

Consta de un graficado de manera que, para las variables del algoritmo de la función, se pinta en rojo las posiciones de puntos representativos suministrados por las demás funciones, y se pinta en azul las posiciones seleccionadas como puntos representativos del clúster real.

Es necesario transformar a *array* la lista de puntos representativos seleccionados por *sec_depur* para su graficado. Para ello se crea otra variable *sp_y* y no se realiza con la misma para evitar errores de formación más adelante en el algoritmo.

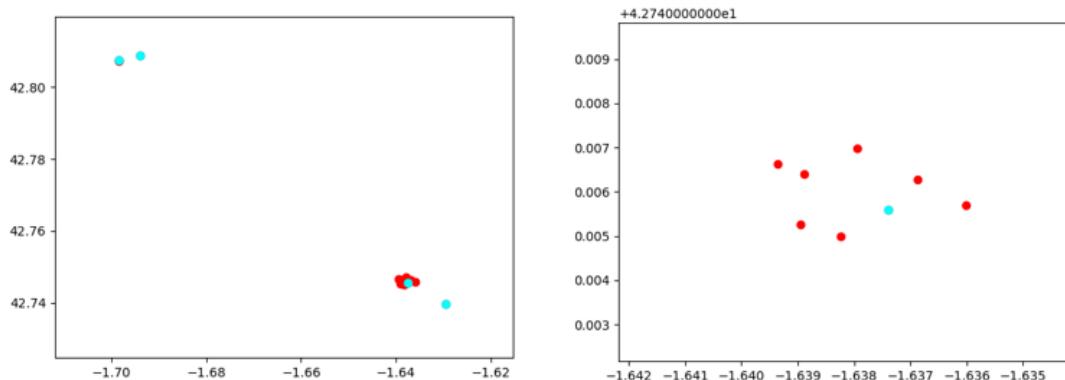


Ilustración 69 *sec_depur_grafic*.

En las ilustraciones se muestra una progresión de zoom en el mismo clúster en el que se ha realizado la progresión anteriormente, para el mismo periodo de tiempo. Se aprecian en rojo las posiciones suministradas como puntos representativos del clúster, y en rojo, el punto representativo real, suministrado por *sec_depur*.

El proceso realizado pues, ha sido el de formación de clústeres de los puntos representativos y posteriormente la elección de un punto representativo.

La función *sec_depur* es además aplicada en el proceso de detección de SPs formados que coinciden con los de la tabla de *specialsites*, mediante el mismo proceso.

Como alternativa a *clustering_20*, se obtiene *clustering_past*, la nomenclatura viene dada por el propósito al que sirve, es la misma función que *clustering_20* pero sin contar con el graficado.

```
def clustering_past(users_past, eps, mpts):
    locationsForDBSCAN = [list([user[2], user[3]])
                           for user in users_past]

    x = np.array(locationsForDBSCAN)
    AVERAGE_CIR = 6378.137 * 1000 * 2 * math.pi
    eps = eps * 360 / AVERAGE_CIR

    db = DBSCAN(eps=eps, min_samples=mpts,
                metric='euclidean').fit(x)

    # PARA SPECIAL POINTS
    cores = db.components_

    clusters = []
    for i in range(0, len(cores)):
        if len(clusters) > 0:
            counter = 0
            for clust in clusters:
                if abs(cores[i, 0] - clust[0]) <= eps and abs(cores[i, 1] - clust[1]) <= eps:
                    counter = counter + 1

            if counter == 0:
                clusters.append(cores[i])

        else:
            clusters.append(cores[i])

    return clusters
```

Ilustración 70 *clustering_past_func.*

La aplicación de esta función es para un proceso que requiere muchos clusterizados, el graficado, aparte de consumir memoria, ralentiza el proceso. La funcionalidad a la que sirve esta función es la de resetear los SPs, es por ello que es necesario que carezca de grafiado.

Ante la necesidad de que la función no tenga graficado, es preferible crear otra función que no tenga grafiado que crear otra variable que condicione el graficado y que sea necesario introducirla como argumento o variable por defecto cada vez que se tenga que llamar a la función. Puesto que de esta manera se manejan menos variable y se procesa menos código.

Una vez explicados *clustering_20* y *sec_depur* procedemos con la función encargada de realizar el minado de SPs del último mes.

Como primera función en el archivo obtenemos *clustering*, esta función es la encargada de realizar un clusteo completo para un tiempo definido por dos periodos solapados. Puesto que el algoritmo *DBSCAN* puede manejar una cantidad limitada de registros, para evitar aproximaciones a los límites de su capacidad, se realizan dos clusteos independientes.

Se pretende realizar un clusteo que abarque la mayor cantidad de registros posible, para detectar los SPs formados en último mes. Es necesario la ampliación del dominio del registro ya que el clúster se forma en base a la concurrencia del vehículo en un área determinada, por lo que, si realizamos el clusteo para el último mes, es necesario barrer los registros de tiempo antes. Por lo tanto, se estudian los límites de capacidad del algoritmo, siendo este entorno a los registros de 30 días.

Con el fin de evitar el error de exceso de memoria (*SIGKILL9*) en aquellos meses en los que existe una gran cantidad de registros a procesar, ya que se ha dado el caso, se realiza una partición con solapamiento de los períodos. El proceso consiste en generar dos ventanas temporales de 25 días (suficiente menos de 30) y solapar en si 15 días, de esta manera el proceso tarda más tiempo en cumplirse, pero es necesario para evitar el fallo de la aplicación.

```
def clustering():
    eps = 150
    mpts = 300

    users_20 = database.select_users_20()
    users_30_10 = database.select_users_30_10()

    clusters_20 = clustering_20(users_20, eps, mpts)
    clusters_30_10 = clustering_past(users_30_10, eps, mpts)

    sp = sec_depur(clusters_20 + clusters_30_10, eps)

    return sp
```

Ilustración 71 *clustering_func.*

La función engloba los procesos anteriormente descritos y retorna una variable que comprende una lista de puntos representativos de los clústeres formados para el último mes. La se aprovecha la depuración para fusionar en un único SP los puntos representativos que se repiten en ambos procesos de clusteo.

A continuación, se expone el archivo SP20.py que utiliza las funciones descritas de clust.py.

3.3. SP20.py.

3.3.1. Introducción.

SP20.py comprende de dos funciones que sirven a distintas funcionalidades de *Clustering SP*, las funcionalidades de búsqueda de clústeres para un periodo de tiempo definido de maneras manual y automática.

El archivo es creado con la intención de crear una función que barra la BBDD en busca de clústeres a lo largo de todos sus registros. Este proceso es necesario si se pretende obtener un conjunto de SPs de partida con los que comparar las incorporaciones mensuales y no realizar el minado mes a mes.

3.3.2. Cuerpo.

Como primer elemento, como en el resto de archivos, obtenemos las importaciones.

```
import clust
from datetime import timedelta, date, datetime
from database import DataBase
import numpy as np
import matplotlib.pyplot as plt

database = DataBase()
```

Ilustración 72 SP20_imports.

Se importa el archivo *clust* completo para los procesos de clusterización. De la librería *datetime* se importan las funciones *timedelta*, *date*, y *datetime* para el manejo de fechas y tiempos. Del archivo *database* se importa la clase *Database* y se la nombra *database*. Se importan las librerías *numpy* y *matplotlib.pyplot* como *np* y *plt*, respectivamente, para los procesos de graficado.

Como primera función obtenemos *manual* que comprende la más sencilla de las funciones de clusterización. La denominación resulta del manejo y la búsqueda de clústeres en los registros mediante la introducción manual de las fechas.

A continuación, se procede a explicar un elemento común en varias funciones, la elección de la fecha. Este elemento consta de la introducción de un parámetro temporal que se puede realizar de dos maneras distintas, y la herramienta interpreta de forma automática la forma en la que ha sido introducido.

```
date_ = str(input("""Insert date [dd/mm/yyyy]:      Or      Insert amount of backdays (from now):"""))
if date_:
    if len(date_) == len('dd/mm/yyyy'):
        date_ = datetime.strptime(date_, '%d/%m/%Y')
    elif len(date_) <= 7:
        date_ = date.today() + timedelta(days=-int(date_))
```

Ilustración 73 *date_selection*.

Se crea una variable *input* y se fuerza a que resulte en *str*, esto es debido a que, ante la entrada libre de caracteres, puede dar lugar a la interpretación del *input* como *int*, lo que emerge en error si posteriormente se intenta convertir en fecha. Ante esta situación se introduce en el *input* lo que se desea reproducir en pantalla.

Ante el manejo del *input*, en *len(date_)*, emergirá un error que interrumpe la herramienta si no existe la variable a tratar, para evitarlo, se introduce un condicional que cuestiona la existencia de la variable *if date_*:. Este error salva las malas interpretaciones de las fechas, por ejemplo: si se introduce [28/6/1999] en vez de [28/06/1999] la herramienta no podrá interpretar la fecha.

La elección de la fecha para realizar los procesos se puede introducir de dos formas: bien mediante el formato de fecha [dd/mm/yyyy] o bien mediante la cantidad de días previos desde la fecha actual.

Para la distinción entre los formatos, se crea un condicional que cuestiona la longitud del *input*, si coincide con la longitud del formato [dd/mm/yyyy] se convierte el *str* a formato *strftime* a través de la función *datetime*.

En caso contrario, se permiten un máximo de valor para el cálculo de días previos, pudiendo tener hasta 7 dígitos, es decir, el número máximo que se puede introducir es: 9.999.999 lo cual carece de sentido aplicar valores cercanos. Ante esta introducción en el *input*, se recoge la fecha actual mediante la función *date.today()* y se le restan los días correspondientes al valor introducido. Es necesario convertir en *int* la variable para este proceso, además es necesario introducir un signo negativo, ya sea restando el *timedelta* como sumando una cantidad negativa de días.

A continuación, se procede a realizar la llamada a la función correspondiente para la incorporación de los registros, con la fecha como argumento.

Por consiguiente, la función *manual()* resulta:

```
def manual():
    date_ = str(input("""Insert date [dd/mm/yyyy]:      Or      Insert amount of backdays (from now):
"""))
    if date_:
        if len(date_) == len('dd/mm/yyyy'):
            date_ = datetime.strptime(date_, '%d/%m/%Y')
        elif len(date_) <= 7:
            date_ = date.today() + timedelta(days=-int(date_))

        users_past = database.select_users_past(date_)
        cls = []

        all_sp = list(database.select_sp())

        if users_past:
            cls = clust.clustering_20(users_past, 150, 500)
        if cls:
            sp = clust.sec_depur(cls, 150)

            sp = np.array(sp)
            all_sp = np.array(all_sp)
            plt.gca().set_facecolor('white')
            plt.scatter(all_sp[:, 1], all_sp[:, 0], c='black')
            plt.scatter(sp[:, 1], sp[:, 0], c='aqua')
            plt.show()
            all_sp = np.array(all_sp).tolist()
            all_sp.extend(sp)
        else:
            print(f"No Data for {date_}")
        if all_sp:
            all_sp = clust.sec_depur(all_sp, 30)

    return all_sp
```

Ilustración 74 *manual_func.*

Como se aprecia en la ilustración, la función selectora de registros es *select_users_past*, con la que se recogen los registros pertinentes para la realización del clusteо, estos son, los registros que abarcan desde la fecha introducida hasta 25 días posteriores.

La función *manual()* comprende en sí, de forma directa, la funcionalidad de *Search for clusters* del menú de *Clustering SP*. en este punto es necesario marcar una distinción entre las funcionalidades de *Search for clusters* y la *Scatter with day selection*. Ambas funcionalidades carecen de la inserción de los registros a la BBDD, muestran un gráfico de los registros y los SPs de la BBDD, en cambio, en *Scatter with day selection*, el usuario es capaz de decidir libremente la ventana temporal de incorporación de registros. Esto es debido a que mientras que sólo realiza el graficado de las posiciones, *Search for clusters* a demás aplica el algoritmo de *DBSCAN* el cual posee una capacidad limitada, por ello se limita la acción de esta funcionalidad.

Como se aprecia en la ilustración, además de los registros, también se incorporan los SPs de *specialsites* y se guardan en la variable *all_sp*.

Para salvar de errores de variable vacía se crean los condicionales que cuestionan la existencia de la variable, en caso afirmativo se continúa con el proceso, como ya hemos visto otras veces.

Se realiza el clusteo de los registros mediante *clustering_20* y se guardan los puntos representativos en la variable *cls* como abreviación de *clusters*.

Posteriormente se realiza la depuración de los puntos representativos, se forman las listas a *arrays* para su graficado. Hasta aquí interviene la funcionalidad, esta función fue creada en un principio para ayudar en el desarrollo de la herramienta y estudiar los mejores parámetros de *eps* y *mpts*, por ello continúa con fusionar ambas listas, y devolver una variable con las dos listas fusionadas y depuradas, para comprobar la existencia de puntos nuevos con la variación de parámetros.

En caso contrario a no tener registros con los que trabajar, aparte de salvar el error, se informa de que no existen registros para la fecha introducida, mediante *else*.

Al ejecutar la funcionalidad se observa:

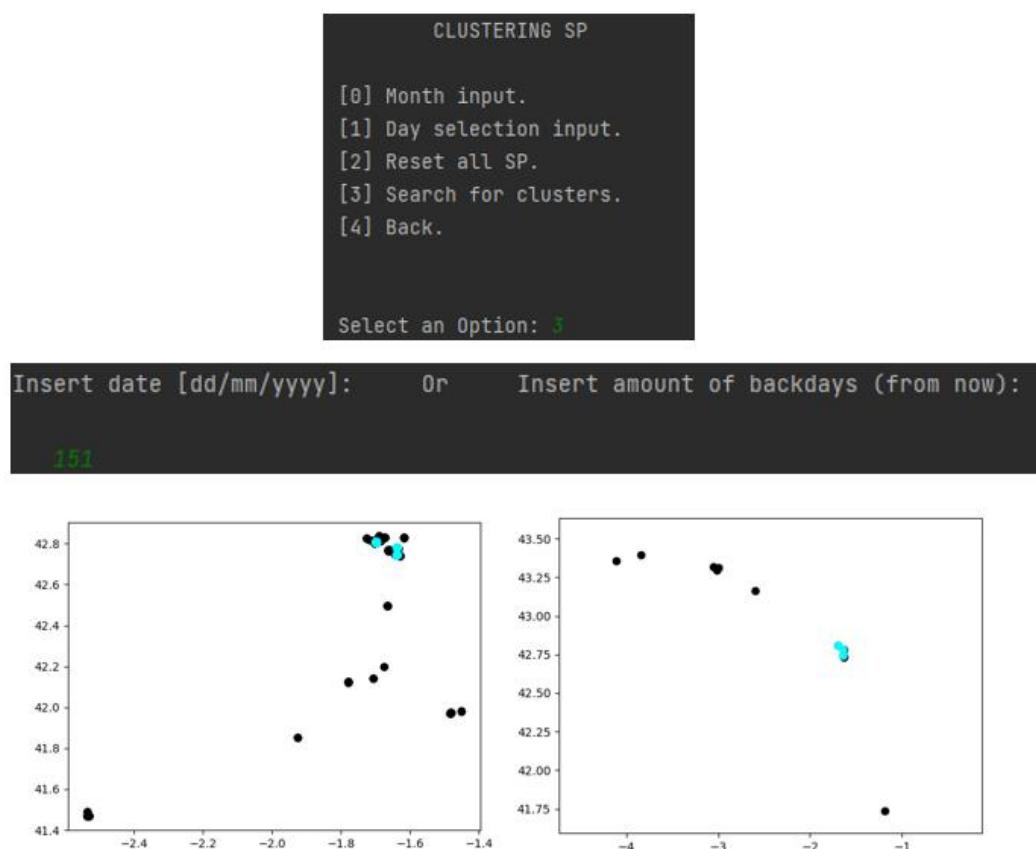


Ilustración 75 manual_func_process.

La primera ilustración comprende la selección de la funcionalidad en el menú de *Clustering SP*. La segunda ilustración comprende la selección de fecha para realizar la búsqueda de clústeres.

Las tercera y cuarta ilustraciones comprenden las grafías del proceso, la tercera la grafía realizada por *clustering_20* y la cuarta la grafía realizada del solapamiento entre SPs extraídos del proceso en azul y los SPs que se encuentran en la BBDD.

Como segunda función obtenemos *auto()*, la cual es utilizada para la funcionalidad de *Reset all SP*. Comprende de una función que realiza un proceso de clusterización y minado de SPs de 600 días en ventanas de 20 días. Esta función pretende realizar un barrido de la BBDD lo suficientemente amplio para obtener SPs de referencia ante la incorporación de nuevos. Supongamos la siguiente situación, se acuerda en realizar un cambio en los parámetros de *eps* y *mpts*, esto supone la inclusión o exclusión de SPs en la tabla de *specialsites*, para tener una referencia precisa y contar con SPs de referencia ante los nuevos parámetros, hace falta una función que realice esta acción.

```
def auto():
    all_sp = []
    for days in range(600, 0, -20):

        pastdate = date.today() + timedelta(days=-days)
        print(pastdate, days)
        users_past = database.select_users_past(pastdate)
        cls = []

        if users_past:
            cls = clust.clustering_past(users_past, 100, 500)
        if cls:
            sp = clust.sec_depur(cls, 150)
            all_sp.extend(sp)
        if all_sp:
            all_sp = clust.sec_depur(all_sp, 150)

    all_sp_ = all_sp
    all_sp = clust.sec_depur(all_sp, 250)
    all_sp = np.array(all_sp)
    all_sp_ = np.array(all_sp_)
    plt.gca().set_facecolor('white')
    plt.scatter(all_sp_[:, 1], all_sp_[:, 0], c='black')
    plt.scatter(all_sp[:, 1], all_sp[:, 0], c='aqua')
    plt.show()

    return all_sp
```

Ilustración 76 *auto_func.*

La función se basa en el bucle *for* que realiza los procesos de clusterizado, este bucle se define como un comienzo en 600, a paso de -20, hasta 0. Como es un proceso repetitivo, es necesario que el clusteo carezca de graficado, para ello se creó la función *clustering_past*.

Se define la variable *all_sp* como una lista vacía para poder extenderla dentro del bucle. La clave del bucle es la fecha que maneja, para ello, mediante *timedelta*, se resta a la fecha actual la

cantidad de días que marca la variable del bucle. Con esta fecha se procede a la selección de registros mediante *select_users_past*, que como ya hemos visto en el anterior capítulo, selecciona los registros en una ventana de 25 días. Al obtener registros en una ventana temporal de 25 días y la variación ser de 20, queda un solapamiento de 5 días adicionales para la formación de clústeres.

Se imprime en la interfaz las variables *pastdate* y *days* para observar el proceso, es decir, la fecha para la cual se realiza el clusteo y la cantidad de días previos que comprende esa fecha.

El proceso de clusteo consta del minado de puntos representativos, su depuración, agregación de los puntos resultantes a la lista de *all_sp* y otra depuración. La última depuración es necesaria puesto que se puede haber añadido un SP que pertenezca a un clúster que también se formó en la ventana previa.

Posteriormente se procede al graficado, se duplica la variable *all_sp* con *all_sp_* para formarla en *array* y graficarla, y poder devolverla en forma de lista sin realizar la formación inversa.

3.4. add_toSQL.py.

3.4.1. Introducción.

El siguiente archivo comprende las funcionalidades del menú de *Clustering SP, Reset all SP* y *Month Input*. Las funcionalidades contenidas en este archivo representan los principales procesos del menú.

Como primera función se continúa el proceso de la función *auto* descrita en el apartado anterior de *SP20*. La razón por la que las funciones son separadas, aunque sirvan a la misma funcionalidad, es la necesidad de distinguir los procesos que se dan por los archivos que los contienen, es decir, en un archivo las funciones de clusterizado, en otro las funciones que usan las funciones de clusterizado de manera compleja y por último las funciones que realizan acciones de inserción de registros en la BBDD.

3.4.2. Cuerpo.

La primera función, *twoyears_input*, comprende la inserción en la tabla de *specialsites* los SPs minados de los últimos 600 días. La necesidad de creación de esta función reside tanto en la posibilidad de que se acuerde un cambio en los parámetros de *eps* y *mpts*, como en la eliminación de SPs obsoletos, es decir, puntos representativos de clústeres que ya no se forman desde hace 600 días.

```
def twoyears_input():
    database.remove_allSP()
    sp = SP20.auto()
    i = 0

    for point in sp:
        lon = point[0]
        lon = str(lon)
        lat = point[1]
        lat = str(lat)

        i = i + 1
        cls = "C - " + str(i)

        special_site = database.select_users_compr(lon, lat)

        desc = (str(special_site[3]) + " ID " + str(special_site[0]) + " MAT " + str(special_site[1])).replace(".", "")
        desc = desc.replace(" ", "")

        database.addSP(lat, lon, cls, desc)
```

Ilustración 77 *twoyears_input_func.*

Como se aprecia en la ilustración, tal y como se explicó en el capítulo 1, la primera acción a tomar ante la inserción de registros sin compararse con los que ya hay, es su eliminación. Esto se realiza mediante la función *remove_allSP*.

A continuación, se llama a la función *auto* para continuar su proceso, el cual devuelve una lista de SPs a insertar en la BBDD.

La variable *i* es se da únicamente para la denominación de los SPs, cada vez que se agrega uno, aumenta de valor. Es necesario un contador ya que se recorren los puntos de la lista *sp*, no se da valores a una variable que comprenda la longitud de la lista. Es importante la distinción de los tipos de bucles *for*.

Se crea, entonces, un bucle *for* que recorre *sp* y almacena cada coordenada en variables *lon* y *lat*, acto seguido las transforma en *str*. Es necesaria la transformación en *str* de los parámetros a introducir en la BBDD puesto que, si no, no se reconocen en la sintaxis.

Se define la variable *cls* para la denominación del SP. Los clústeres se nombran como *C-* y a continuación el número que posea *i*.

Se introducen las coordenadas para extraer el resto de parámetros y se define la descripción de cada SP, con los datos de *ID*, *matricula* y *localización* con calle, número y localidad.

Como último paso se insertan en *specialsites* mediante *addSP*.

A continuación, se muestra el proceso que realiza ejecución de *twoyears_input*, pero se ha suprimido la parte en la que se eliminan los SPs de la tabla y la parte de insertarlos, para ello no hay más que convertir en *str* la parte del algoritmo que se ocupa de realizar esta acción.

Es decir:

```
def twoyears_input():

    """database.remove_allSP()
    sp = SP20.auto()
    i = 0

    for point in sp:
        lon = point[0]
        lon = str(lon)
        lat = point[1]
        lat = str(lat)

        i = i + 1
        cls = "C - " + str(i)

        special_site = database.select_users_compr(lon, lat)

        desc = (str(special_site[3]) + " ID " + str(special_site[0]) + " MAT " + str(special_site[1])).replace(".", "")

    """database.addSP(lat, lon, cls, desc)"""

    
```

Ilustración 78 *ty_input_func_str*.

El proceso queda de la siguiente forma:

| | | |
|--------------------------|----------------|----------------|
| CLUSTERING SP | 2020-06-03 580 | 2021-03-10 300 |
| [0] Month input. | 2020-06-23 560 | 2021-03-30 280 |
| [1] Day selection input. | 2020-07-13 540 | 2021-04-19 260 |
| [2] Reset all SP. | 2020-08-02 520 | 2021-05-09 240 |
| [3] Search for clusters. | 2020-08-22 500 | 2021-05-29 220 |
| [4] Back. | 2020-09-11 480 | 2021-06-18 200 |
| | 2020-10-01 460 | 2021-07-08 180 |
| | 2020-10-21 440 | 2021-07-28 160 |
| | 2020-11-10 420 | 2021-08-17 140 |
| | 2020-11-30 400 | 2021-09-06 120 |
| | 2020-12-20 380 | 2021-09-26 100 |
| | 2021-01-09 360 | 2021-10-16 80 |
| | 2021-01-29 340 | 2021-11-05 60 |
| | | 2021-11-25 40 |
| | | 2021-12-15 20 |

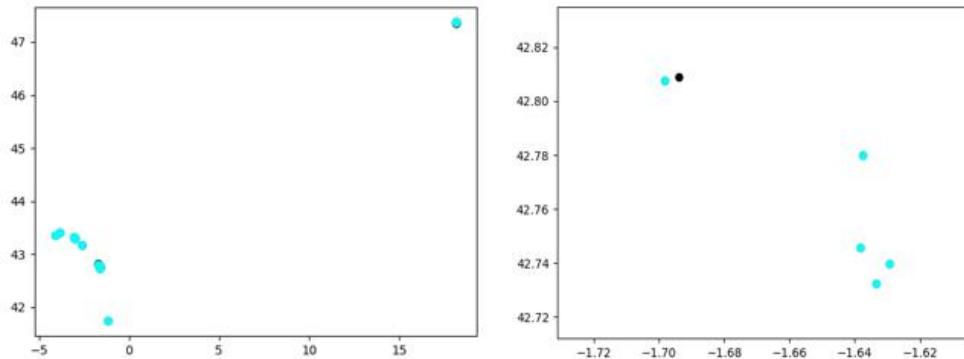


Ilustración 79 ty_input_progress.

Las ilustraciones muestran el progreso ordenado de la funcionalidad *Reset all SP*. La primera ilustración muestra la selección de la funcionalidad, las dos ilustraciones siguientes muestran la impresión en pantalla del barrido realizado y por la fecha que va. Cabe destacar que el proceso de barrido tarda aproximadamente cinco minutos con un procesador AMD Ryzen 2700 de 7^a generación, una tarjeta gráfica Geforce GTX y 32 Gb de memoria RAM, por lo que se debe tener precaución al ejecutar esta funcionalidad.

Las ilustraciones siguientes muestran los SPs minados, descartando uno de ellos. Esto es posible que ocurra por la necesidad de abarcar más registros a la hora de realizar el clusteo, puesto que el clusterizado normal, abarca 40 días, no 25. La posibilidad de que resulte un SP antiguo queda descartada por ya que la BBDD es más joven que 600 días, por lo que no hay registros tan antiguos.

Faltaría por añadir la impresión en pantalla de los registros agregándose a la BBDD, pero eso acarrearía la necesidad de eliminar lo que ya hay y no debe hacerse sin motivo aparente.

La siguiente función comprende la funcionalidad de *Month Input* y *Day Selection Input*, a partir de *month_input*. La función resulta de un clusteo preciso que abarca mayor rango que *twoyears_input*.

Al resultar de una función demasiado extensa, y, por tanto, no se puede mostrar completa, se muestra por partes explicando cada una de ellas.

```
def month_input(input_):

    global new_sp, cls
    if input_ == "1":
        date_ = str(input("""Insert date [dd/mm/yyyy]:      Or      Insert amount of backdays (from now):
                    """))
        if date_:
            if len(date_) == len('dd/mm/yyyy'):
                date_ = datetime.datetime.strptime(date_, '%d/%m/%Y')
            elif len(date_) <= 7:
                date_ = datetime.date.today() + datetime.timedelta(days=-int(date_))

            users_past = database.select_users_past(date_)
            if users_past:
                cls = clust.clustering_20(users_past, 150, 500)
            if cls:
                new_sp = clust.sec_depura(cls, 150)

    sp = database.select_sp()

    if input_ == "0":
        new_sp = clust.clustering()
```

Ilustración 80 *month_input_func_1*.

La función incorpora como argumento la variable de selección *input* del menú de *Clustering SP*, esta variable diferencia una funcionalidad de otra, por lo que ante la elección de *Day Selection Input*, el algoritmo se bifurca hacia la selección de la fecha de la misma manera anteriormente descrita.

Los parámetros globales son opcionales de definir, puesto que su incorporación resulta del cuestionamiento de su existencia, y estos son definidos mediante condicionales por los que el algoritmo no tiene por qué pasar. Pero como se puede observar entre esta y la siguiente ilustración las variables están definidas para todos los caminos que tome el algoritmo, este es error que salva por tener la variable únicamente definida dentro de condicionales.

Posteriormente se realiza una extracción de registros a partir de la función *select_users_past*, se realiza su clusteo mediante *clustering_20* y posteriormente su segunda depuración.

Ante el cumplimiento del condicional de la variable *input_ == 0* se da lugar a la funcionalidad *Month Input*.

Se guardan los SPs de *specialsites* en la variable *sp*.

Posteriormente se realiza el graficado de los SPs minados, la función cuenta además por el graficado que ofrece *clustering_20*.

```
plt.gca().set_facecolor('white')
plt.scatter(np.array(sp)[:, 1], np.array(sp)[:, 0], c='black')
plt.scatter(np.array(new_sp)[:, 1], np.array(new_sp)[:, 0], c='aqua')
plt.show()
```

Ilustración 81 month_input_func_2.

A continuación, se describe el proceso por el cual se cotejan las incorporaciones mensuales con los SPs de la BBDD, es un proceso un tanto complejo.

```
for point in new_sp:
    sp_act = clust.sec_depur(sp + (point,), 300)

    if len(sp_act) > len(sp):
        lon = str(point[0])
        lat = str(point[1])

        special_site = database.select_users_compr(lon, lat)

        cls = "C - " + str(len(database.select_sp()) + 1)

        desc = (str(special_site[3]) + " ID " + str(special_site[0]) + " MAT " + str(special_site[1])).replace(
            ",",
            "")

        print(f"Se ha dado de alta {point} en la BBDD.")
        database.addSP(lat, lon, cls, desc)

    else:
        print(f"El punto {point} pertenece a un clúster ya dado de alta.")
```

Ilustración 82 month_input_func_3.

El proceso consta de intentar comprobar si los SPs minados coinciden con los SPs de la BBDD. Para ello se crea un bucle que recorra cada punto de *new_sp* y lo primero que se hace es compararlos con los puntos de *sp* mediante una depuración.

Los posibles resultados que pueden darse de este proceso son los siguientes:

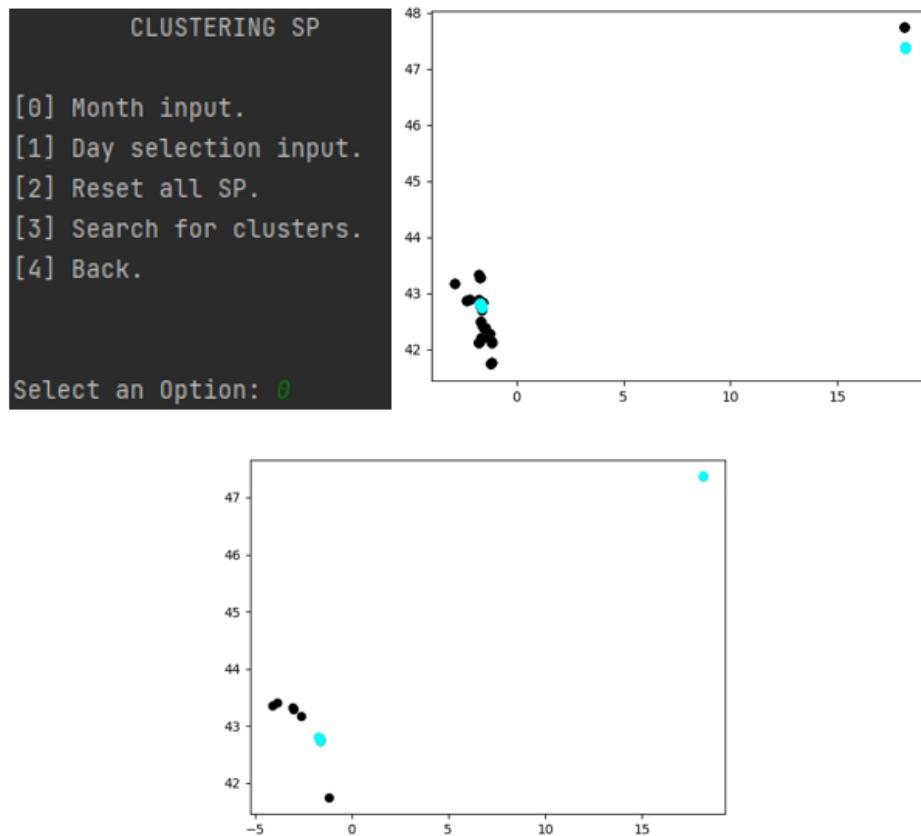
- El punto no coincide con ningún SP de la BBDD, por lo que, al depurarse, la lista *sp_act* cuenta con un registro más.
- El punto coincide con un SP de la BBDD, por lo que, al depurarse, la lista cuenta *sp_act* cuenta con el mismo número de registros que *sp*.
- El último caso es extraño que se dé, resulta que el punto se sitúa entre dos SPs de la BBDD y al depurarse, se forme un clúster que englobe a los tres (o más) puntos. Como resultado al depurarse, la lista *sp_act* cuenta con menos registros que *sp*.

Por lo que la forma perfecta de comprobar si el SP que se está analizando coincide con alguno que los que ya existen, es depurando la lista conjunta y analizando el número de registros que posee.

Por lo tanto, se abre el condicional que analiza la longitud de las listas, y en caso de que $\text{len}(\text{sp_act}) > \text{len}(\text{sp})$ se procede a la descripción del SP y su ingreso en *specialsites*.

En caso contrario se indica que el punto ya pertenece a un clúster dado de alta en la tabla.

A continuación, se muestra el proceso de *month_input* mediante la funcionalidad de *Month Input*.



El punto [42.74502 -1.63771] pertenece a un cùster ya dado de alta.
 El punto [47.37954 18.18307] pertenece a un cùster ya dado de alta.
 El punto [42.73955 -1.62956] pertenece a un cùster ya dado de alta.
 El punto [42.80878 -1.69403] pertenece a un cùster ya dado de alta.
 El punto [42.80766 -1.69846] pertenece a un cùster ya dado de alta.
 El punto [42.77996 -1.6374] pertenece a un cùster ya dado de alta.

Ilustración 83 month_input_process.

La primera ilustración comprende la selección de funcionalidad en el menú de *Clustering SP*. La segunda funcionalidad comprende la grafía suministrada por *clustering20*. La tercera ilustración comprende la grafía realizada para comparar los SPs minados en el último mes con

los que ya pertenecen a la BBDD. La última de las imágenes comprende la impresión en pantalla del ingreso, o no, de los SPs en la tabla *specialsites*.

La función *replace* se utiliza para eliminar los puntos, o, mejor dicho, sustituirlos por “nada”, mediante *replace(“.”, “”)*.

Cabe destacar que normalmente, los SPs que se forman mes a mes suelen ser los mismos que los que ya hay, puesto que los vehículos empiezan y finalizan el día en los lugares que frecuentan por lo tanto los SPs minados suelen no ingresar en *specialsites*.

Con esta función se cierra el capítulo de la funcionalidad de *Clustering SP*.

A continuación, da comienzo el capítulo de *Routing*.

CAPÍTULO 4

ROUTING

4.1. Introducción.

Routing comprende la funcionalidad de la herramienta que se encarga de realizar el trazado de los segmentos y rutas de los vehículos.

La herramienta realiza el trazado de segmentos mediante los registros de posición de los vehículos para un periodo de tiempo igual o menor a 24 horas. El trazado de rutas viene dado por la suma de los segmentos de ruta.

Los segmentos de ruta se definen como los puntos consecutivos de los registros de un vehículo que abarcan desde que el vehículo se pone en marcha hasta que el vehículo se para.

Cabe esperar que el inicio y finalización de cada segmento se realice en las proximidades de un SP. En caso contrario se indica como anomalía.

A su vez, los vehículos pasan por SPs sin parar el motor, por lo que no se registra una parada en la trayectoria, en cambio poseen velocidad nula. Es necesario definir el segmento a partir del estado del motor, puesto que, en caso contrario, no se podrán definir los segmentos con exactitud. Cada segmento cuenta con numerosos registros con velocidad nula entre otros registros con velocidad no nula.

En la siguiente ilustración se aprecia un fragmento de un segmento con velocidades variantes.

| | | | | | | |
|-------------|------------|----------|------------|----------|-----------|------|
| ('9848LKX') | 2022-01-05 | 12:25:00 | 2022-01-05 | 13:47:04 | En marcha | 77.0 |
| ('9848LKX') | 2022-01-05 | 12:30:00 | 2022-01-05 | 13:47:04 | En marcha | 59.0 |
| ('9848LKX') | 2022-01-05 | 12:35:00 | 2022-01-05 | 13:47:04 | En marcha | 0.0 |
| ('9848LKX') | 2022-01-05 | 12:40:00 | 2022-01-05 | 13:47:04 | En marcha | 0.0 |
| ('9848LKX') | 2022-01-05 | 12:45:00 | 2022-01-05 | 13:47:04 | En marcha | 10.0 |
| ('9848LKX') | 2022-01-05 | 12:50:00 | 2022-01-05 | 13:47:04 | En marcha | 0.0 |
| ('9848LKX') | 2022-01-05 | 12:55:00 | 2022-01-05 | 13:47:04 | En marcha | 6.0 |

Ilustración 84 segment_speed

La última columna contiene los valores de la velocidad del registro, como se puede apreciar, muchos de los registros cuentan con velocidad nula. Esto es debido a semáforos, stops, etc., donde el vehículo se para, pero el motor sigue en marcha.

Los registros son incorporados cada cinco minutos, suficientemente precisos para analizar el segmento y lo suficientemente dispersos para que el proceso no se demore al realizarse por un exceso de registros a manejar.

Por lo que el proceso principal de la funcionalidad es realizar los trazados de los segmentos a partir de los registros de los vehículos en base al estado del motor. El trazado de las rutas se realiza a partir de los segmentos extraídos en el momento.

La funcionalidad completa comprende de dos opciones para las cuales realizar el mismo proceso, uno para la fecha actual, analizar los registros que se llevan hasta ese momento en el día, mediante *Daily Routing*, y el otro es realizar el proceso para otra fecha a elegir, mediante *Selection Routing*.

La finalidad de la funcionalidad es alcanzar la posibilidad de realizar un estudio con los SPs minados de la funcionalidad de *Clustering SP*, por ello se procura proporcionar la mayor cantidad de información posible a cerca de las trayectorias realizadas por los vehículos. Los casos en los que un vehículo para en un SP, pero no para el motor, no se registra como una parada, por lo que al suministrar la información de los SPs de partida y finalización se pierde información.

Es por ello que, en los segmentos de ruta se incorpora, en el parámetro de observaciones, los SPs por los que pasa en el segmento sin hacer parada, y cuánto tiempo pasa en ellos. El proceso para realizar esta acción es complejo y ralentiza mucho la ejecución, pero merece la pena no perder la información, después se ve plasmada en la funcionalidad de *Mapping*.

Al igual que en el minado de SPs descartábamos los puntos de ruido, en *Routing* debemos descartar los movimientos de vehículos lentos como grúas, además de los segmentos que son demasiado cortos para analizarlos e insertarlos, estos son segmentos que no llegan al kilómetro de recorrido.

En caso de que se dé con un segmento que no llegue al kilómetro se indica como *No segment*. En el caso de que existan segmentos *No segment* que no comprendan una ruta se indica con *No route*. Para los casos en los que no existen segmentos debido a que el vehículo no se ha movido, no se indica la ausencia del segmento, se asume como que no se ha analizado ese vehículo.

El archivo principal de la funcionalidad es *segments.py*, engloba las dos funcionalidades de *Routing* al comprender una función con la variable de selección de opción como argumento. Las acciones que toma esta función son las de procesado de los registros recorriendo un bucle temporal cada cinco minutos, analizando cada uno de ellos y construyendo el segmento de ruta y las rutas.

La función en su desarrollo incorporaba el graficado del proceso, pero para una agilización de la ejecución se suprimió esta acción. En la explicación de la función se incorpora el graficado para una visualización minimalista del trazado.

Los otros dos archivos, *seg_toSQL.py* y *route_toSQL.py*, realizan las conexiones con la BBDD e insertan los datos de los parámetros trazados en las tablas correspondientes.

A continuación, se explica la función de *segments*.

4.2. segments.py.

4.2.1. Introducción.

El archivo segments.py comprende la función de *segments*, esta función es la más extensa y compleja de la herramienta, por la cantidad de procesos que realiza y la profundidad de bucles que posee.

La función se separa en dos partes principales, la de selección de funcionalidad, con la que se otorga la posibilidad de selección de fecha o se asigna la fecha del día actual, y posteriormente, la función converge en la segunda parte de la función, el proceso de trazado.

El proceso de trazado consiste, a grandes rasgos, en el barrido de los registros que pertenecen a un vehículo mediante una función selectora del archivo de database.py que comprenda la selección de registros para una matrícula y un intervalo de tiempo determinado. Posteriormente, la creación de segmentos a partir del parámetro *s_motor*, se agregan al segmento los registros que posean *s_motor == 'en marcha'* y se finalizará el segmento en el registro que contenga *s_motor == 'parado'*. El trazado de rutas se realiza como la sucesión de segmentos de una misma matrícula.

Con esta metodología básica se procede a la explicación de la función.

Al ser demasiado extensa la función, se explica por partes aisladas que realizan acciones diferentes, pero es necesario incidir en que la mayoría de las acciones tomadas por la función poseen sinergia entre sí.

4.2.2. Cuerpo.

Como primer elemento a describir del archivo obtenemos, al igual que en otros casos, las importaciones.

```
import itertools

from seg_toSQL import seg_toSQL
from route_toSQL import route_toSQL
from database import DataBase
import matplotlib.pyplot as plt
import numpy as np
from datetime import date, timedelta, datetime
from os import system, name

database = DataBase()
```

Ilustración 85 segment.py_imports.

Se importa *itertools* para realizar el proceso de ruta en cadena. Se importan las funciones *route_toSQL* y *seg_toSQL* de sus archivos para las acciones en relación a la BBDD. Se importa la clase *Database* y se nombra como *database*. Se importa *numpy* como *np* para las formaciones de los datos. La importación de *matplotlib.pyplot* permanece en gris, debido a que no se ha utilizado, puesto que se ha suprimido su acción para la agilización del proceso. Se importan las funciones de manejo temporal de *datetime* y las funciones de manejo de la interfaz de *os*.

Como siguiente elemento en la función encontramos:

```
def segments(input_):
    global unique_mats, date_, seg, parados, seg_id, i, dat
    system('cls' if name == 'nt' else 'clear')
    sp = np.array(database.select_sp())
```

Ilustración 86 segments_func_1

En la ilustración se puede apreciar el argumento de la función, consta de la variable de elección de funcionalidad del archivo main.py descrito en el capítulo 1.

A continuación, se definen las variables globales de manera opcional ante la posibilidad de error por definir las variables dentro de los condicionales, se limpia la pantalla y, con el fin de graficar el proceso, se extraen los SPs de la tabla de *specialsites* y se guardan en la variable *sp*.

La variable permanece oscurecida por haber eliminado la acción de graficado.

A continuación, el primer condicional de la variable *input_*.

```
if input_ == "1":
    date_ = str(input("""Insert date [dd/mm/yyyy]:      Or      Insert amount of backdays (from now):
"""))
    if date_:
        if len(date_) == len('dd/mm/yyyy'):
            date_ = datetime.strptime(date_, '%d/%m/%Y')
        elif len(date_) <= 7:
            date_ = date.today() + timedelta(days=-int(date_))

        date_ = date_.strftime('%Y-%m-%d %H:%M:%S')
        mats = database.select_mat_0(date_)
```

Ilustración 87 segments_func_2.

Tal y como se puede apreciar, se trata de la selección de fecha realizada de la misma forma ya vista en otras funciones. A continuación, mediante la función *select_mat_0* se selecciona la matrícula de todos los registros que comprendan la fecha *date_*.

Para el procesado de rutas, se debe obtener la lista de matrículas, las que se aportan en *mats* son las matrículas totales, de los cientos de registros diarios que hay, aproximadamente una consta de una decena de matrículas diferentes, pero repetidas en cada registro.

```

unique_mats_ = []

for mat in mats:

    if mat not in unique_mats_:
        if len(mat[0]) <= 7:
            unique_mats_.append(mat)

system('cls' if name == 'nt' else 'clear')
print("Mat Selection: [0 for all]")

for i in range(0, len(unique_mats_)):
    print(f"{i+1}. {unique_mats_[i][0]}")

mat = input("Select mat: ")
system('cls' if name == 'nt' else 'clear')

if int(mat) <= i + 1:
    if mat == "0":
        unique_mats = unique_mats_
    if mat != "0":
        unique_mats = [unique_mats_[int(mat)-1]]
else:
    unique_mats = []

```

Ilustración 88 segments_func_3.

Mediante el primer bucle expuesto en la ilustración se logra crear una lista de matrículas que únicas en la lista, que comprenden las matrículas que se dan en los registros. Esto se consigue mediante la creación de la lista que guarda matrículas únicas, *unique_mats*, el bucle que barre toda la lista de matrículas aportada por la función selectora, y, ante una matrícula que no se encuentre en la lista de *unique_mats* se le añade mediante *append*.

Cabe destacar que el condicional que limita la longitud de las matrículas es el responsable de excluir los vehículos de movimiento lento, ya que las matrículas de estos vehículos constan de nombres propios y números.

A continuación, se limpia la pantalla y se le ofrece al usuario la selección de una matrícula en particular para realizar el trazado, mediante la selección del número que ocupa, o mediante 0, realizar el trazado para todas las matrículas. Esto se realiza puesto que el algoritmo forma parte de la funcionalidad de *Selection Routing*, en cambio, *Daily Routing* realiza de manera automática el proceso para todas las matrículas, puesto que se supone que es un proceso diario.

Es necesario entender que, en programación, el primer elemento de cualquier variable, es el elemento *[0]*, no el *[1]* que es el segundo. Es por ello que se maneja la variable *i* sumando 1.

Posteriormente, se define la variable *unique_mats* en función de lo que se haya escogido. El primer condicional, se realiza para evitar que emerja el error de haber introducido un valor mayor que el de la longitud de la lista, con el que no se podría trabajar. El segundo condicional define la variable, ante la elección de 0 se define como el conjunto de matrículas, ante la

elección de algo diferente se define como la elección realizada, pero se le debe restar 1 para que el valor introducido coincida con la posición de la matrícula.

Como primera funcionalidad obtenemos *Daily Routing*, cuyo inicio se expone a continuación.

```
if input_ == "0":
    date_ = date.today().strftime('%Y-%m-%d %H:%M:%S')

    mats = database.select_mat_0(date_)
    unique_mats = []

    for mat in mats:

        if mat not in unique_mats:
            if len(mat[0]) <= 7:
                unique_mats.append(mat)
```

Ilustración 89 segments_func_4.

Resulta del mismo proceso que el descrito anteriormente con la particularidad de que no se deja opción a la selección de la fecha ni la matrícula. En esta funcionalidad se procesan todas las matrículas de la lista *unique_mats* en la fecha actual definida por *date_*.

El proceso de trazado se realiza en primera instancia por cada matrícula, por lo todo el proceso se da dentro de un bucle que recorre la lista de matrículas únicas.

El primer paso a realizar antes de la incorporación de registros a una tabla es eliminar los que ya existen para el mismo momento en el cual se va a realizar el proceso.

El proceso se basa en la recolección de registros de cada matrícula en intervalos de tiempo definidos, por lo que deben existir variables temporales que acoten el procedimiento. Las variables temporales definen tanto el intervalo de tiempo para el cual se realiza el proceso como el intervalo de tiempo en el que se realiza el muestreo, por lo que deben existir: una variable que defina el inicio, la cual comprende las 00:00 de la fecha para la que se realiza el proceso; una variable que marque el final del intervalo, definida como cinco minutos posteriores a la primera variable; y una última que marque la finalización del proceso.

La variable temporal de finalización del proceso, es diferente para cada funcionalidad, puesto que ante *Daily Routing* la variable de finalización es definida por el momento actual, puesto que no tiene sentido realizarla para las 23:59 de la fecha actual, ya que no hay registros a la hora actual. Mientras que en *Selection Routing* la variable final se define para 23 horas y 59 minutos posteriores a la variable de inicio. No se da para las 00:00 del día posterior por no solapar registros.

Cabe la posibilidad de elegir *Selection Routing* e introducir 0 en la elección de fecha, es decir, 0 días previos, es decir, la fecha actual. La función entonces barrerá hasta las 23:59 del día sin haber registros, no es preocupante puesto que las funciones selectoras salvan del error, pero sí resulta una pérdida de tiempo para el proceso.

El proceso pues, resulta de ir cambiando el intervalo temporal para la selección de los registros.

```
if unique_mats:

    database.remove_seg(date_)
    database.remove_route(date_)

    print(date_)

    for mat in unique_mats:
        mat = str(mat)
        mat = mat.replace(", ", "")

        time = datetime.combine(datetime.strptime(date_, '%Y-%m-%d %H:%M:%S'), datetime.min.time())
        time_1 = time + timedelta(minutes=5)

        time = time.strftime('%Y-%m-%d %H:%M:%S')
        time_1 = time_1.strftime('%Y-%m-%d %H:%M:%S')

        if input_ == "1":
            actual = (datetime.strptime(date_, '%Y-%m-%d %H:%M:%S') +
                      timedelta(hours=23, minutes=59)).strftime('%Y-%m-%d %H:%M:%S')
        else:
            actual = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
```

Ilustración 90 segments_func_5.

Podemos observar en la ilustración los procesos anteriormente descritos, sumado al formato de las variables, es decir, la conversión a *str*, la eliminación de comas y la conversión a formato temporal.

A continuación, se crean los bucles *while* que recorren los intervalos de tiempo. El proceso es un poco complicado y se pretende describir de fuera a dentro.

El proceso de trazado consta de una búsqueda del segmento y el trazado del segmento, por lo que existen dos bucles *while*, uno para el recorrido temporal y otro para el recorrido temporal perteneciente al segmento.

El primer bucle contiene al segundo, el segundo bucle, el cual denominaremos *while not parado*, consta de agregaciones de posiciones al segmento, mientras que el primer bucle, el cual denominaremos *while notnow*, consta de agregaciones de segmentos a las rutas.

Cada vez que se recorre el *not parado*, se cambia el intervalo temporal, por lo que ante cada nueva posición se define si está parado o no. Ante un registro en marcha, el *not parado* agrega la posición al segmento y se repite hasta encontrar un registro *parado*. En ese instante, la condición no se cumple y se repite el *notnow* agregando el segmento a la ruta, agregando el segmento a la tabla *segments* y vaciando el segmento. Este proceso se repite hasta que *notnow* se vuelve falso, es decir, que la variable temporal de recorrido que define el intervalo alcanza la variable temporal de finalización. En ese momento, se añade la ruta completa a la tabla de *routes* y al no cumplirse la condición para que el bucle se repita, se procede con la siguiente matrícula. El proceso se repite con todas las matrículas hasta que finalice.

A continuación, se muestra el código abreviado de ambos *whiles*. Posteriormente se explica los procesos que se dan en cada uno de los apartados del algoritmo ocultados para observar bien la estructura.

```
route = []
route_id = []
num_seg = len(route)
Num = 1

notnow = True
while notnow:

    if len(route) > num_seg:...

        if datetime.strptime(time, '%Y-%m-%d %H:%M:%S') >= datetime.strptime(actual, '%Y-%m-%d %H:%M:%S'):...

            seg = []
            seg_id = []
            parados = []

            parado = False

            while not parado:

                dat = database.select_seg(mat, time, time_1)

                if dat:
                    print(mat, time, datetime.now().strftime('%Y-%m-%d %H:%M:%S'), dat[2], dat[5])

                    if dat[2] != 'Parado':...

                        if dat[2] == 'Parado':...

                            if datetime.strptime(time, '%Y-%m-%d %H:%M:%S') >= datetime.strptime(actual, '%Y-%m-%d %H:%M:%S'):...

                                time = (datetime.strptime(time, '%Y-%m-%d %H:%M:%S') + timedelta(minutes=5)).strftime(
                                    '%Y-%m-%d %H:%M:%S')
                                time_1 = (datetime.strptime(time_1, '%Y-%m-%d %H:%M:%S') + timedelta(minutes=5)).strftime(
                                    '%Y-%m-%d %H:%M:%S')


```

Ilustración 91 segments_func_6.

Como se puede apreciar, se muestran los dos *whiles* en la ilustración, se muestra la selección de los registros en el intervalo temporal para la matrícula mediante *select_seg* y el cambio en las variables temporales.

Las variables que se definen fuera de los bucles son las variables que se resetean para el siguiente proceso, por ejemplo, las listas de *seg* se deben vaciar para trazar el siguiente segmento.

Para cada trazado existe una lista de posiciones y otra lista de *ids*, las cuales una se usa para el graficado y la otra se utiliza para las funciones de inserción en la BBDD, pero las incorporaciones a las listas se realizan de forma simultánea.

La variable *num_seg* como abreviación de número de segmentos, se define como la longitud de la lista de *routes* esto es debido a que *routes* consta de una lista de listas, al agregarse a la variable la lista del segmento, que consta una lista de posiciones. Por lo tanto, si se desea tratar a *routes* como una lista completa se debe transformar la lista de listas de ubicaciones en una lista de ubicaciones.

En la recogida del registro, el tercer elemento de la tupla, *dat[2]* comprende el parámetro *s_motor*, y el sexto elemento de la tupla *dat[5]* comprende el parámetro de *velocidad*. Con estos parámetros podemos determinar si el vehículo se encuentra en marcha o parado. A continuación, se muestran los condicionales tales a este dato.

```
dat = database.select_seg(mat, time, time_1)

if dat:
    print(mat, time, datetime.now().strftime('%Y-%m-%d %H:%M:%S'), dat[2], dat[5])

    if dat[2] != 'Parado':
        pos = [dat[0], dat[1]]
        seg.append(pos)
        seg_id.append(dat[4])

    if dat[2] == 'Parado':
        parado = True

        if seg and datetime.strptime(time, '%Y-%m-%d %H:%M:%S') <= datetime.strptime(actual,
            '%Y-%m-%d %H:%M:%S'):

            pos = [dat[0], dat[1]]
            seg.append(pos)
            seg_id.append(dat[4])
            parados.append(pos)
            route.append(seg)
            route_id.append(seg_id)
```

Ilustración 92 segments_func_7.

Como se puede apreciar en la ilustración, ante el cumplimiento de que el vehículo se encuentre en marcha, se define la posición que tiene mediante longitud y latitud, y posteriormente se agrega a la lista que define el segmento, tanto la tupla de posición como el *id* que representa ese registro.

Si el segmento no se encuentra en marcha, se cambia la variable booleana de *parado* y se define el punto final del segmento, siempre y cuando todavía no se haya alcanzado el límite temporal puesto que la ruta puede continuar y no se debe definir como un punto parado. Esto se debe a que se cumple que *dat[2]* no es *Parado* porque no existe el registro, pero tampoco existe *dat*, por lo que no se debería entrar en el condicional, pero ante un posible error, esta condición adicional lo salvaría.

Se agrega la posición a las listas de *seg* y de *parados* para el graficado, se agregan los segmentos a las listas de ruta.

Con el fin de observar el proceso se pinta en pantalla el registro que se está procesando en el momento, con los datos de matrícula, hora para la que se procesa, hora actual, *s_motor* y *velocidad*.

La visualización de este elemento se otorgará en la ejecución de la funcionalidad, más a delante.

El segundo condicional del bucle *not parado* comprende la finalización del segmento por alcance de la variable temporal de finalización.

```

if datetime.strptime(time, '%Y-%m-%d %H:%M:%S') >= datetime.strptime(actual, '%Y-%m-%d %H:%M:%S'):
    parado = True
    if dat:
        if dat[2] != 'Parado' and dat[5] != '0.0':
            pos = [dat[0], dat[1]]
            seg.append(pos)
            seg_id.append(dat[4])
            parados.append(pos)
            route.append(seg)
            route_id.append(seg_id)

```

Ilustración 93 segment_func_8.

La primera acción a tomar es el cambio en la booleana *parado* para que no se cumpla la condición del bucle y no se repita.

Se agregan los datos de posición e *id* a las listas pertinentes y se procede con la repetición del bucle externo entrando en el condicional descrito a continuación.

Este condicional es necesario puesto que, si se alcanza la variable temporal de finalización en marcha, no existe motivo por el cual salir del bucle *not parado* por lo que sigue realizando registros para la misma matrícula eternamente, aunque no existan registros, puesto que no hay error que salte al no encontrarlo.

El segundo condicional del bucle *notnow* comprende la agregación de la ruta a la tabla de *routes*. La condición a cumplir es que se alcance de la variable temporal de finalización para la cual indica que el trazado ha finalizado y la ruta queda completa, o no hay más registros a añadir todavía si se trata de *Daily Routing*.

A continuación, se muestra el condicional de finalización de bucle *notnow*.

```

if datetime.strptime(time, '%Y-%m-%d %H:%M:%S') >= datetime.strptime(actual, '%Y-%m-%d %H:%M:%S'):
    notnow = False

    route = list(itertools.chain(*route))

    if not route:
        if dat:
            pos = [dat[0], dat[1]]
            id = dat[4]
            route.append(pos)
            route_id.append(id)

    route_toSQL(route_id, date_)

    """if route:
        route = np.array(route)
        plt.scatter(sp[:, 1], sp[:, 0], c='black')
        plt.scatter(route[:, 1], route[:, 0], c='lightblue')
        plt.plot(route[:, 1], route[:, 0], c='lightblue')
        plt.show()"""

    route = np.array(route).tolist()
    seg = np.array(seg).tolist()
    parados = np.array(parados).tolist()

```

Ilustración 94 segments_func_9.

La primera acción a tomar es el cambio en la booleana *notnow*, por la cual no se cumplirá la condición para repetir el bucle y se procesará la siguiente matrícula, redefiniendo las variables

como listas vacías. El siguiente proceso es el de transformar *routes* de lista de listas a simplemente lista de ubicaciones.

Ante la no generación de ruta, pero existencia de registros de matrícula, se procede a agregar la posición inmóvil en el proceso como ruta de punto único, pero no se inserta en la tabla de *routes* por los condicionales internos de la función *route_toSQL*.

Se inserta la ruta en la tabla *routes*.

Se procede con el graficado, eliminado para la agilización de proceso, se muestra entrecamillado para que no conste.

Ante el graficado, las listas se toman en formación *array*, es por ello que se deben reconvertir en lista una vez terminado el proceso.

A continuación, se explica el primer condicional del bucle *notnow*, el cual es el que realiza la tarea de incorporar los segmentos a la tabla *segments*.

La condición de que se debe cumplir es la de que la longitud de la lista *routes* sea mayor que lo que era antes de agregar el último segmento, es decir, que cambie de longitud. Se define la variable *num_seg* como la longitud de *routes* debido a que, la longitud de la lista representa la cantidad de listas que porta, en su caso, por ende, el número de segmentos. Al ser *num_seg* una variable definida a partir de *routes* si, se pueden comparar para cada parte del proceso.

Para los casos en los que no haya segmentos que agregar, la longitud de *routes* no cambia, pero en el momento en el que se agregue un segmento de ruta, se cumple el condicional.

```
if len(route) > num_seg:  
    seg = np.array(seg)  
    parados = np.array(parados)  
  
    Num = seg_toSQL(seg_id, Num, date_)  
  
    """plt.scatter(seg[:, 1], seg[:, 0], c='orange')  
    plt.plot(seg[:, 1], seg[:, 0], c='orange')  
    plt.scatter(sp[:, 1], sp[:, 0], c='black')  
    plt.scatter(parados[:, 1], parados[:, 0], c='red')  
    plt.show()"""  
  
    num_seg = len(route)
```

Ilustración 95 segments_func_10.

Se transforman las listas en *arrays* para su graficado.

Se aprovecha la llamada a la función *seg_toSQL* tanto para la inserción del segmento en *segments* como para definir la variable *Num*, ya que esta depende de si se añade el segmento o no a la ruta.

Por último, se redefine la variable *num_seg* con la nueva longitud de *route*.

Quedando la función definida se procede a mostrar su ejecución.

Para mostrar detalladamente el proceso, se han eliminado los entrecomillados de los graficados, por lo que las variables y las importaciones que sólo se daban en el graficado, han dejado de oscurecerse.

El graficado fue realizado en primera instancia para comprender el proceso que se estaba creando, ayuda a mejorar el código y corregir las fallas que este pudiera tener.

El graficado de segmentos comprende en naranja los registros en marcha y en rojo los registros de finalización del segmento, los registros parados.

El graficado de rutas se hace para la ruta entera en azul.

Ambos graficados se muestran en primera instancia para todos los SPs, por lo que la primera imagen abarca los SPs de España y Hungría, por lo que es preciso hacer zoom, en la grafía. Se mostrará en las ilustraciones correspondientes las gráficas con zoom ya realizado.

```
Routing

[0] Daily routing.
[1] Selection routing.
[2] Back.

05 record(s) were deleted in segments.
1 record(s) were deleted in routes.

Select an Option: 0 2022-01-05 00:00:00

('9848LKX') 2022-01-05 07:25:00 2022-01-06 00:30:43 En marcha 0.0
('9848LKX') 2022-01-05 07:30:00 2022-01-06 00:30:43 En marcha 45.0
('9848LKX') 2022-01-05 07:35:00 2022-01-06 00:30:43 En marcha 86.0
('9848LKX') 2022-01-05 07:40:00 2022-01-06 00:30:43 En marcha 87.0
('9848LKX') 2022-01-05 07:45:00 2022-01-06 00:30:43 En marcha 90.0
('9848LKX') 2022-01-05 07:50:00 2022-01-06 00:30:43 En marcha 91.0
('9848LKX') 2022-01-05 07:55:00 2022-01-06 00:30:43 En marcha 90.0
('9848LKX') 2022-01-05 08:00:00 2022-01-06 00:30:43 En marcha 72.0
('9848LKX') 2022-01-05 08:05:00 2022-01-06 00:30:43 En marcha 83.0
('9848LKX') 2022-01-05 08:10:00 2022-01-06 00:30:43 En marcha 88.0
('9848LKX') 2022-01-05 08:15:00 2022-01-06 00:30:43 En marcha 81.0
('9848LKX') 2022-01-05 08:25:00 2022-01-06 00:30:43 En marcha 79.0
('9848LKX') 2022-01-05 08:35:00 2022-01-06 00:30:43 En marcha 86.0
```

Ilustración 96 segments_process_1.

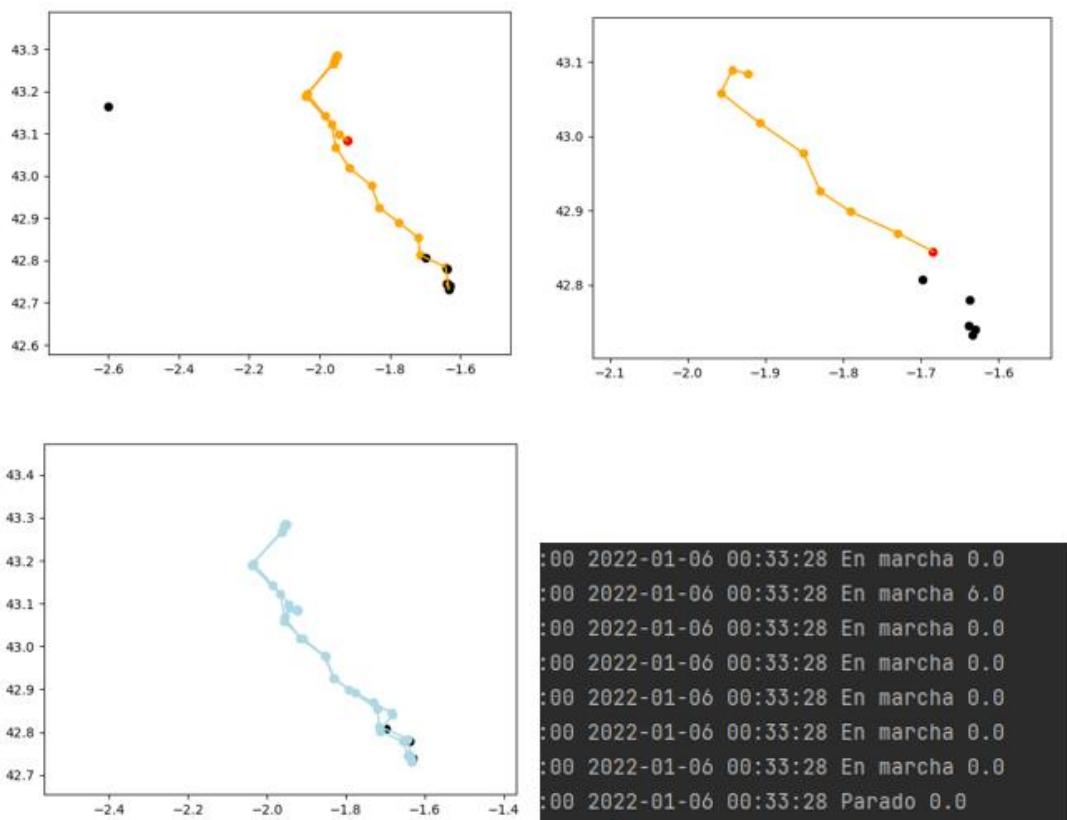


Ilustración 97 segments_process_2.

En las ilustraciones podemos apreciar el proceso que se da para la ejecución de segments. Se observa la primera acción a tomar que es eliminar los registros que hay para esa fecha. El proceso, como se puede comprobar en la interfaz, se da el día 06 de enero de 2022 para un día anterior. Se muestra en la ilustración una captura del momento en el que aparece en pantalla el vehículo en marcha. En adelante se muestran dos segmentos de ruta, no comprenden la ruta completa. Por último, se muestra la ruta completa junto con el registro en el que acaba la ruta, el registro *parado*.

Los puntos negros representan los SPs d la tabla de *specialsites*, los puntos en naranja representan los registros en marcha y los puntos en rojo los parados de finalización del segmento.

La visualización de la ruta completa se realiza de manera que no se distingan los segmentos, para comprobar una aportación completa de todos los segmentos.

El graficado de los segmentos y las rutas se realiza de forma minimalista para el desarrollo de la función en su creación. La visualización real de los segmentos de ruta se realiza mediante la creación de un mapa en lenguaje html.

A modo de demostración se procede a mostrar los segmentos y la ruta completa graficada anteriormente sin el formato minimalista, donde se pueden comprobar las regiones que comprenden la ruta.

Esta comprende la funcionalidad de *Mapping* que será descrita en el próximo capítulo.

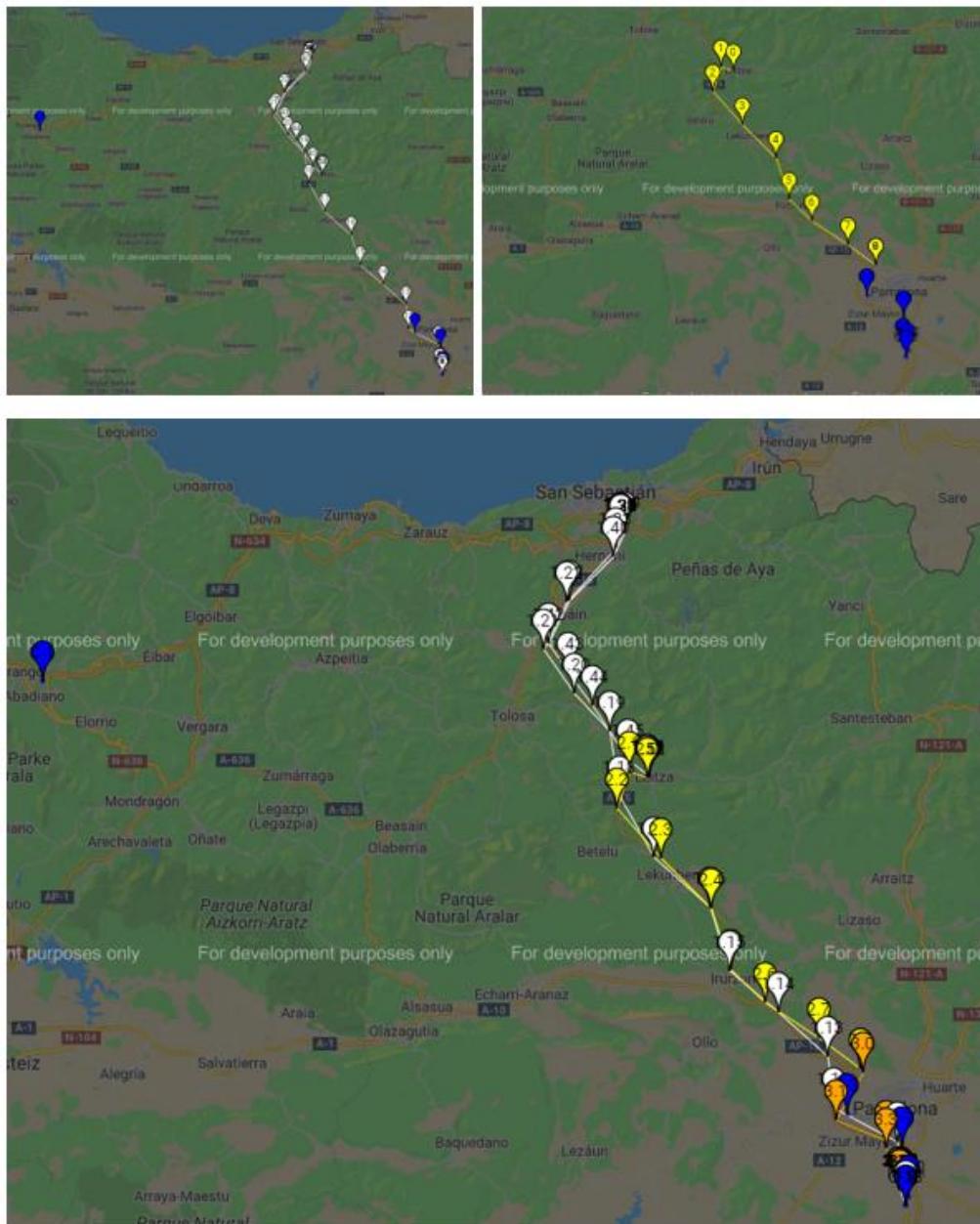


Ilustración 98_map_1

Como se puede apreciar de manera más detallada, se tratan de los segmentos de la ruta trazada anteriormente. Se puede observar que la ruta consiste en ida y vuelta desde Pamplona hasta San Sebastián.

A continuación, se explican los procesos de inserción en la BBDD de los segmentos trazados.

4.3. seg_toSQL.py.

4.3.1. Introducción.

El archivo *seg_toSQL.py* es el archivo continente de la función *seg_toSQL*, cuyo propósito principal es la inserción del segmento en la tabla *segments*. La función se encarga además para numerosos propósitos, entre ellos se aprovecha para devolver la variable *Num* a *segments.py*.

La variable se define en *segments* y se redefine en *seg_toSQL*, este proceso necesita ser de esta manera ya que la definición del segmento en ruta que ocupa necesita ser restablecido cada vez que se inicia un segmento nuevo, y necesita ser redefinido en función de si se incorpora el segmento a la tabla o no.

En la función de *seg_toSQL* se realiza el proceso de detección de los SPs por los que pasa el vehículo, pero no realiza parada, este proceso ralentiza mucho la ejecución, pero merece la pena realizarlo puesto que de esta manera no se pierde información muy valiosa.

Gracias a la acción de detectar los SPs en los que se realiza la parada sin parar el motor, se puede realizar un proceso de cara a futuro que recopile esta información y realice otro proceso de extracción de SPs a partir de estos registros.

El proceso realizado dentro de la función comprende de numerosas acciones: el tratado del segmento se realiza mediante el suministro de una lista de *ids* que la función posee como argumento. Mediante la lista de *ids*, se realiza un proceso de extracción de datos para cada registro con el que se construye cada parámetro del segmento.

Los parámetros a insertar en la tabla son los siguientes: *Dateseg, desde, hasta, Num, idorg, idend, mat, PointA, PointB, Kms, Vmax, Vavg, Obs*.

- *Dateseg* comprende la fecha en que se realizó el segmento, no el trazado del segmento.
- *desde* comprende la hora de inicio del segmento.
- *hasta* comprende la hora de finalización del segmento.
- *Num* comprende el número de segmento en ruta.
- *idorg* e *idend* comprenden los *ids* del registro que inicia y finaliza el segmento, respectivamente.
- *mat* comprende la identificación del vehículo para al cual se está realizando el trazado.
- *PointA* comprende el SP del que inicia el segmento.
- *PointB* comprende el SP en el que acaba el segmento.
- *Kms* comprende la distancia recorrida en kilómetros.
- *Vmax* comprende la velocidad máxima que se alcanza en el segmento.
- *Vavg*, como abreviatura de *average* comprende la velocidad media que porta el segmento.

- *Obs* es el parámetro con el que se informa de las anomalías sucedidas en el segmento, estas pueden ser inicio o finalización del segmento en lugares que no comprenden un SP, o la información de la estancia en un SP sin realizar parada.

4.3.2. Cuerpo.

Como primer elemento en el cuerpo del archivo, al igual que en el resto, obtenemos las importaciones, estas son:

```
from database import DataBase
import math
from datetime import timedelta

database = DataBase()

sp = database.select_sp()
```

Ilustración 99 seg_toSQL_imports.

Se importa la clase *Database()* y se la denomina *database* para tener acceso a las funciones del archivo. Se importa la librería *math* para la realización de cálculo matemáticos y acceso a las constantes numéricas. Se importa la función *timedelta* de la librería de *datetime* para el manejo de fechas y expresiones temporales.

Se aprovecha para fijar la lista de SPs de *specialsites* en la variable *sp*.

Ante la explicación de la función, al resultar esta demasiado extensa, se realiza por partes sin olvidar la sinergia que posee con el resto de acciones descritas posteriormente.

El argumento de la función posee tres variables: la variable de lista de *ids* del segmento, *seg_id*; la variable de número de segmento en ruta, *Num*; y la variable de fecha del segmento, *date_*.

A continuación, se definen las variables globales, cuya definición real se encuentra dentro de los condicionales.

```
def seq_toSQL(seg_id, Num, date_):
    global wav, Obs_or, Obs_end

    dat_org = database.select_data_forseg(seg_id[0])
    dat_end = database.select_data_forseg(seg_id[-1])
    SPdata_A = "    "
    SPdata_B = "    "
    PointA = "NotSP"
    PointB = "NotSP"

    Dateseg = date_
    desde = dat_org[5]
    hasta = dat_end[5]
    idorg = seg_id[0]
    idend = seg_id[-1]
    mat = dat_org[1]
```

Ilustración 100 seg_toSQL_func_1.

En primer lugar, se hace una extracción de los registros inicial y final del segmento, los parámetros a extraer, mediante la función *selec_data_forseg* son: *s_motor*, *matricula*, *velocidad*, *X(ubicacion)*, *Y(ubicacion)*, *fecha_upos* y *localidad*. Constando así de siete parámetros enumerados del 0 al 6.

La extracción de datos se hace para el primer y último elemento de la lista que define el segmento, estos son los elementos números 0 y -1, respectivamente. El término referido al elemento -1 indica, con carácter general, el último elemento de la lista, resultando -2 el penúltimo y así consecutivamente.

En una lista de un elemento, el término 0 y -1 constan del mismo elemento, pero el término 1 no existe.

Mediante este proceso, se obtienen los datos representativos de los registros inicial y final del segmento.

Para lo cual, *dat_org* representa los datos del registro inicial del segmento, el origen, y de la misma manera, *dat_end*, representa los datos del registro final del segmento.

Se definen las variables por defecto *SPdata* y *Point* para los registros iniciales *A* y los registros finales *B*, esa será la nomenclatura. Se definen como *str* vacíos o “*NotSP*” para su posterior redefinición en caso de que exista el parámetro, lo cual resulta lo que se espera.

Cada parámetro a incorporar a la tabla se realiza de forma particular, los parámetros que se realizan de forma directa se describen en la ilustración anterior. Para los cuales, *Dateseg* se define con la variable del argumento que comprende la fecha en la que se realiza el segmento, *date_*. El parámetro *desde* se define por el sexto elemento de la tupla de *dat_org*, en el orden de la extracción consta de *fecha_upos*, al igual que su homólogo, hasta, mediante el sexto elemento de *dat_end*. Los parámetros de *idorg* e *idend* constan del primer y último elemento de la lista de *ids* del segmento. El parámetro *mat* consta del segundo registro de cualquiera de las dos tuplas, resultando el mismo dato en ambas.

A continuación, se procede a explicar el proceso de cálculo para el parámetro *Kms*.

Puesto que no es un parámetro inmediato, es necesario realizar un proceso iterativo mediante un bucle *for* que calcule los kilómetros recorridos en el segmento.

El proceso se da para cada pareja formada por puntos consecutivos que hay en el segmento, es decir, cada punto y el siguiente.

Se calcula la distancia que hay entre los puntos que se están tratando y posteriormente se guarda en una variable que va sumando la distancia cada vez que se repite el bucle.

Como dificultades al proceso, se deben transformar la diferencia entre las coordenadas geográficas de latitud y longitud a metros. Para ello, primero, se resta cada coordenada a la del punto posterior y de esta manera se crea un vector que define el desplazamiento del vehículo entre los dos registros. A continuación, se calcula el módulo de dicho vector y mediante una regla de tres, con la circunferencia de 360° y la circunferencia media de la Tierra, en kilómetros, se obtiene el valor de los kilómetros recorridos por el vehículo entre los dos registros.

En el bucle, se crea una variable que se redefina con ella misma sumando la distancia y cuando termine el bucle, se consigue la distancia total del segmento.

```
AVERAGE_RADIUS = 6378.137
AVERAGE_CIR = 2 * math.pi * AVERAGE_RADIUS

Kms = 0
for i in range(0, len(seg_id) - 1):
    point = database.select_data_forseg(seg_id[i])
    post = database.select_data_forseg(seg_id[i + 1])
    p_1 = point[3], point[4]
    p_2 = post[3], post[4]

    dist = p_2[0] - p_1[0], p_2[1] - p_1[1]
    Kms = Kms + math.sqrt(dist[0] ** 2 + dist[1] ** 2) * AVERAGE_CIR / 360
```

Ilustración 101 seg_toSQL_func_2.

En primera instancia, se debe crear la variable de tipo *int* vacía *Kms* para poder referirse a ella dentro del bucle.

El bucle debe definirse respetando los límites de los elementos de las listas, puesto que hay elementos *[i+1]* refiriéndose al siguiente término de la lista (siguiente punto), el límite del bucle se debe establecer para un valor antes de la longitud de la lista. De esta manera se salva el error *list index out of range*.

El error *list index out of range* es el error del que se salva cuando se pone un condicional que cuestiona la existencia de una variable. Ante el manejo de esta, si estamos manejando un elemento de la variable que no existe, emergirá el error.

Con el fin explicativo, se elimina de la función el *-1* de la limitación del bucle, de esta manera emerge el error:

```
post = database.select_data_forseg(seg_id[i + 1])
IndexError: list index out of range

Process finished with exit code 1
```

Ilustración 102 index_out_of_range_error.

Volviendo a la función, las ubicaciones de los puntos se realizan mediante la misma función *select_data_forseg*, con la cual, las coordenadas de la ubicación comprenden los elementos cuarto y quinto de la tupla.

La acción de elevar a una potencia se realiza mediante ****.

Con el fin de optimizar el código, el proceso se condensa en pocas líneas para ahorrar la definición de variables que consuman memoria. Por ello el cálculo, la conversión a kilómetros, y la redefinición de la variable se realizan en la misma línea.

El siguiente parámetro a definir es *PointA*, posteriormente, de la misma manera *PointB*.

El proceso consta de recoger el punto de origen y cotejarlo con la lista de *sp*, mediante un proceso iterativo en un bucle *for*.

El proceso será, entonces, la definición de un contador que recorra los SPs de la lista, en el momento en el que encuentre el que coincide con el punto de origen, se le sumará una unidad al contador. Si el contador permanece en 0 tras haber comparado el punto con la lista, se dará por iniciado el segmento en un punto no coincidente con un SP.

Es importante la distinción entre el anterior bucle y este, el anterior bucle manejaba una variable de iteración porque se necesitaba definir un elemento y el siguiente, mientras que, en este bucle, se itera con los elementos de la lista directamente, sin variable de iteración.

```
eps = 150 * 360 / (2 * math.pi * AVERAGE_RADIUS * 1000)

# PointA = [dat_org[3], dat_org[4]]

counter = 0
for point in sp:
    if abs(float(point[0]) - float(dat_org[3])) <= 3 * eps and abs(float(point[1]) - float(dat_org[4])) <= 3 * eps:
        counter = counter + 1
        SPdata_A = database.select_unique_sp(point[0], point[1])
        if SPdata_A:
            PointA = f"{SPdata_A[3]} {[dat_org[3], dat_org[4]}"
            Obs_or = ""
        else:
            PointA = "NotSP. Special Point not defined {[dat_org[3], dat_org[4]}"
            Obs_or = "Origin of segment does not match with Special Point. "

    if counter == 0:
        PointA = f"NotSP. Special Point not defined {[dat_org[3], dat_org[4]}"
        Obs_or = "Origin of segment does not match with Special Point. "
```

Ilustración 103 seg_toSQL_func_3.

Puesto que los parámetros del registro vienen dados en *str*, es necesario convertirlos en *float* para compararlos con *eps*.

La variable *eps* se define como 150 metros de radio, convertido a un parámetro de coordenadas geográficas.

La comparación se basa en que se debe cumplir que ambas coordenadas del punto deben distar menos, en valor absoluto, que tres veces *eps*.

En caso de cumplirse la condición, se extraen los datos del SP con el que ha coincidido mediante *select_unique_sp*, y los guarda en la variable *SPdata_A*. La tupla que extrae consta de los parámetros: *id*, *latitud*, *longitud*, *Cluster*, *Description*, etc.

Posteriormente se define *PointA* como el nombre del SP con *SPdata_A[3]* y las coordenadas del punto. El parámetro *Obs_or* permanece vacío.

Ante la carencia de coincidencia con un SP, es necesario indicar que *PointA* no coincide con ningún SP redefiniendo la variable, además de ello se indica en observaciones del segmento redefiniendo la variable *Obs_or* que más tarde se utilizará.

El proceso para el punto final del segmento es análogo al del punto inicial.

```

counter = 0
for point in sp:
    if abs(float(point[0]) - float(dat_end[3])) <= 3 * eps and abs(float(point[1]) - float(dat_end[4])) <= 3 * eps:
        counter = counter + 1
    SPdata_B = database.select_unique_sp(point[0], point[1])
    if SPdata_B:
        PointB = f"{SPdata_B[3]} {[dat_end[3], dat_end[4]}"
    Obs_end = ""

if counter == 0:
    PointB = f"NotSP. Special Point not defined {[dat_end[3], dat_end[4]}"
    Obs_end = "End of segment does not match with Special Point."

```

Ilustración 104 seg_toSQL_func_4.

Tal y como se observa en la ilustración, el proceso es idéntico para definir las variables *PointB* y *Obs_end*, lo único que cambia es que se maneja *dat_end* en vez de *dat_org*.

Posteriormente se calculan los parámetros de velocidad.

Para calcular el parámetro de velocidad máxima, mediante un bucle *for* que recorre la lista de *ids* del segmento, se extrae el parámetro de velocidad de cada registro y se añade a la lista *v* definida como lista vacía.

```

v = []

for id in seg_id:
    dat = database.select_data_forseg(id)
    v.append(dat[2])

Vmax = max(v)

```

Ilustración 105 seg_toSQL_func_5.

Posteriormente se define la variable *Vmax* como el valor máximo de la lista.

En cuanto al cálculo de la velocidad media, se crea un bucle *for* con una variable de iteración con la longitud de la lista de velocidades. No se crea el bucle que recorre la lista directamente porque se va a usar el valor de la variable para más acciones que la de definir la posición del elemento.

```

sumv = 0
for i in range(0, len(v)):
    sumv += v
    vav = sumv / (i + 1)

Vavg = vav

```

Ilustración 106 seg_toSQL_func_6.

El proceso consta de crear una variable que realice la suma de todos los elementos y una variable que divida la suma de los elementos por la cantidad de elementos que consta la suma. Puesto que *i* comienza en 0 como primer elemento, es necesario sumarle una unidad a la variable. Posteriormente se redefine la variable por separado para un control más claro, aunque es una acción opcional.

Para la introducción al argumento de la función que inserta los parámetros en la tabla, es necesario transformarlos en *str*. Debido a que, si no se realiza esta acción, emergerá un error en la sintaxis de la BBDD.

```
Dateseg = str(Dateseg)
desde = str(desde)
hasta = str(hasta)
Num = str(Num)
idorg = str(idorg)
idend = str(idend)
Kms = str(Kms)
Vmax = str(Vmax)
Vavg = str(Vavg)
```

Ilustración 107108 seg_toSQL_func_7.

A continuación, se describe el proceso por el cual se informa de los SPs por los que pasa el segmento sin realizar la parada.

Se tomarán los SPs que visita, sin contar con el de partida ni con el de finalización como secuencia, aunque no consta como tal, pues se informa de los SPs que visita y cuánto tiempo permanece en ellos, pero no el orden, puesto que, para el mismo segmento, se suman los tiempos de estancia sin realizar parada si pasa dos veces por el mismo SP.

```
seq_ = []
for point in sp:
    seq = []
    for id in seg_id:
        lon, lat, time = database.select_data_forseq(id)
        if abs(float(point[0]) - lon) <= 3 * eps and abs(float(point[1]) - lat) <= 3 * eps:
            SPdata = database.select_unique_sp(point[0], point[1])
            if SPdata:
                if not seq:
                    seq.append((time, SPdata[3], (lon, lat), (float(SPdata[2]), float(SPdata[1]), eps)))
                if (seq[-1])[1] == SPdata[3]:
                    seq.append((time, SPdata[3], (lon, lat), (float(SPdata[2]), float(SPdata[1]), eps)))

            if seq:
                seq_.append(seq)

seq_.sort()
```

Ilustración 109 seg_toSQL_func_8.

El algoritmo consta pues, en una doble iteración, una para los SPs de la tabla de *specialsites* y otra para los puntos definidos del segmento, a base de sus *ids*.

Se crean dos listas, *seq* y *seq_*, para que la segunda comprenda un alista de listas, siendo cada lista, una lista de los puntos que coinciden con las proximidades de un SP.

Una vez definidos los bucles, se guarda en las variables los registros extraídos de *select_data_forseq* para cada *id*.

Ante la comparación de las coordenadas de cada punto SP y la longitud y latitud de cada punto del segmento, se condiciona a que sean ambas menores que *eps*, por lo que el punto

estaría en la proximidad de un SP y hay que agregar el punto a la lista *seq*, pero no de cualquier manera.

Se guarda en *SPdata* los datos pertinentes al SP.

Si *seq* es aún una lista vacía y se cumple el condicional, se agregan los datos pertinentes a *seq*. *time* es la hora a la que el punto se encuentra en el segmento; *SPdata[3]*, consta al igual que anteriormente, del nombre del SP; el resto de datos son las coordenadas tanto del SP como del punto, por último, se aporta *eps*, para que, en el desarrollo de la construcción de la función, al imprimir en pantalla esta lista, se comprobase que funcionase el algoritmo. Los datos de utilidad, pues son los dos primeros.

En caso de que *seq* no comprenda una lista vacía, sólo se añaden las tuplas si del último elemento de la lista *seq[-1]*, el segundo elemento *[1]*, es el mismo SP que *SPdata[3]*, es decir, el nombre del SP. Esto se realiza para que se incluyan las tuplas en la lista que pertenezcan únicamente al mismo SP. De esta manera, aunque el vehículo se vaya a otro SP y regrese al mismo, *seq* solo guardará los registros de un único SP por cada vez que se repita el *for* externo.

Una vez se termina el bucle *for* de los puntos del segmento, si existe *seq*, se añade a *seq_*. El condicional de la existencia de *seq* resulta de evitar la agregación de listas vacías. Por último, se ordena la lista de listas.

A estas alturas se posee una lista que contiene las listas de los puntos en marcha, normalmente con velocidad nula, que están por las proximidades de los SPs.

Explicando el proceso de forma inversa: la posición de cada punto del segmento se compara con un SP concreto cada vez, ante una distancia menor a tres veces *eps*, el punto se agrega a la lista *seq* junto con la hora del registro, se repite el proceso para todos los puntos del segmento. Posteriormente se agrega la lista a la lista *seq_* y se cambia el SP, por lo que cada lista de *seq_* corresponde a los puntos en las proximidades de un SP concreto, uno por cada lista.

A continuación, se evalúan los puntos en las listas, cuanto tiempo permanecen en cada SP. El siguiente proceso distingue si el vehículo permanece en un SP o se va y regresa. Es importante la distinción que realiza este proceso puesto que no es lo mismo que un vehículo permanezca una hora en un SP, a que pase por un SP, se registre y vuelva a pasar en una hora por el mismo SP, aunque los intervalos de tiempo sean los mismos.

El proceso que se realiza a continuación, consta otra vez de dos iteraciones, una que recorre la lista *seq_* y otra que recorre la lista *seq*. Se define la variable *obs* como una lista vacía y la variable *time_* como una variable temporal diferencial de valor nulo.

El proceso consta de que se comparan los tiempos de cada tupla de la lista *seq*, es decir, de cada registro. En el archivo *segments.py*, observamos que los registros se realizaban para cada cinco minutos, por lo que dos registros consecutivos, deben tener una diferencia temporal de cinco minutos.

Se compara cada registro con el siguiente mediante un bucle *for* que recorra cada lista de registros, ante la condición de que se cumpla que la diferencia temporal entre los dos es menor que cinco minutos, se redefine la variable *time_* con la suma consigo misma y la diferencia

temporal que resulta entre los dos registros. De esta manera se guarda en una variable el tiempo total que permanece en el SP sin irse.

En el caso de que el vehículo regrese después de un tiempo, no se cumple la condición de que la diferencia temporal sea menor a cinco minutos y por lo tanto no se va a guardar en la variable. La variable temporal se redefinirá nuevamente si el próximo registro se da en menos de cinco minutos.

```
obs = []
if seq_:
    for seq in seq_:
        time_ = timedelta()
        for i in range(0, len(seq) - 1):
            if (seq[i + 1][0] - (seq[i])[0]) <= timedelta(minutes=5):
                time_ = time_ + (seq[i + 1])[0] - (seq[i])[0]
        obs.append((seq[0][1], time_))
```

Ilustración 110 seg_toSQL_func_9.

Una vez termina el bucle interno, se agrega a la lista *obs* una tupla con el nombre del SP y el tiempo total que pasa en él.

Cabe destacar que cuando se manejan variables temporales en el proceso se debe hacer mediante la aplicación de intervalos, es por ello que, aunque la diferencia temporal de muestreo en los registros sea de cinco minutos, la diferencia temporal de agregación de los registros a la tabla de *movements* no tiene por qué ser igual, ya sea debido a un error o un proceso realizado deliberadamente, no se encuentra en el control de la herramienta, por lo que se deben asumir posibles errores que salvar.

Los primeros parámetros de cada registro en la lista *seq* constan de la variable temporal, por lo que la selección de estos es de la forma de *(seq[i])[0]*.

A continuación, se describe el proceso de realización de la variable de *obs*.

```
Obs_seq = ""
for seq in obs:
    if seq[0] != SPdata_A[3] and seq[0] != SPdata_B[3]:
        Obs_seq += str(seq[0]) + "(" + str(seq[1]) + ") / "
    Obs_seq = Obs_seq[:-2]

Obs = Obs_seq + Obs_or + Obs_end
```

Ilustración 111 seg_toSQL_func_10.

La variable completa comprende de la suma de *strs* de las variables *Obs_seq*, *Obs_or* y *Obs_end*, de las cuales las dos últimas quedan definidas en el apartado de definiciones de SPs de origen y destino.

Se realiza un proceso iterativo mediante un bucle *for* que recorre la lista *obs*. Para evitar agregar la estancia en los SPs origen y destino se impone la condición de que el primer elemento de cada tupla, el nombre del SP, sea diferente al nombre de los SPs de origen y destino, *SPdataX[3]*.

A continuación, ante el cumplimiento del condicional, se redefine la variable *Obs_seq* con la agregación del nombre del SP y el tiempo total de la estancia entre paréntesis, sumando al final una / para marcar la separación entre SPs de estancia sin parada. Es necesaria la conversión a *str* de las variables que se agregan para la concatenación correcta, en caso contrario emergirá un error de concatenación.

Posteriormente se redefine *Obs_seq* como la misma variable hasta su penúltimo lugar, para así eliminar la barra / final que quedaría como separación, resultando innecesario.

A continuación, se describe el proceso por el cual ingresan los parámetros definidos a la tabla de *segments* y se imprime en pantalla el resultado.

```
if Vmax != "0.0" and Vavg != "0.0" and float(Kms) > 0.5:
    seg_data = (Dateseg, desde, hasta, Num, idorg, idend, mat, PointA, PointB, Kms, Vmax, Vavg, Obs)
    database.add_seg_toSQL(Dateseg, desde, hasta, Num, idorg, idend, mat, PointA, PointB, Kms, Vmax, Vavg, Obs)
    Num = int(Num) + 1

    print(
        f"""
        Desde: {seg_data[1]}
        Hasta: {seg_data[2]}
        Número de Segmento en Ruta: {seg_data[3]}
        ID origen: {seg_data[4]}
        ID final: {seg_data[5]}
        Matrícula: {seg_data[6]}
        Special Point Origen: {seg_data[7]}
        Special Point Final: {seg_data[8]}
        Kilómetros: {seg_data[9]}
        Vmax: {seg_data[10]}
        Vavg: {seg_data[11]}
        Observaciones: {seg_data[12]}
        """
    )

    return Num

else:
    print("No segment.")
    return int(Num)
```

Ilustración 112 *seg_toSQL_func_11*.

Para evitar la inserción en la tabla de registros “ruido”, es decir, registros que no comprenden de valor útil, se impone el condicional de que las velocidades máximas y medias sean distintas a 0 y que el segmento en si comprenda más de medio kilómetro.

Se definen los datos a introducir en el segmento para la impresión en pantalla y posteriormente ingresan en la tabla de *segments* a través de *add_seg_toSQL* del archivo *database.py*. Es necesario respetar el orden de los parámetros en el argumento.

Se redefine la variable *Num* como ella misma sumando una unidad para el siguiente segmento si se da el caso. Siendo necesario transformar la variable en *int* para poder sumarle la unidad. Posteriormente se retorna a *segments*.

En caso de que el segmento no sea suficientemente relevante para ingresar en la BBDD la variable no se modifica y se retorna como *int*.

La impresión en pantalla se realiza de la forma que se muestra en la ilustración, deshaciendo las tabulaciones, para que cuando se muestre en la interfaz, se muestren todos los parámetros al mismo nivel.

Por último, se muestra la ejecución de la función.

```
('9848LKX') 2022-01-05 13:20:00 2022-01-07 01:56:18 En marcha 0.0
('9848LKX') 2022-01-05 13:25:00 2022-01-07 01:56:18 Parado 0.0
1 record was inserted in segments.
Dateseg: 2022-01-05 00:00:00
Desde: 2022-01-05 12:20:01
Hasta: 2022-01-05 13:27:42
Número de Segmento en Ruta: 3
ID origen: 988631
ID final: 988738
Matrícula: 9848LKX
Special Point Origen: NotSP. Special Point not defined [42.842, -1.68117]
Special Point Final: C - 2 [42.74561, -1.63726]
Kilómetros: 17.018349223047863
Vmax: 77.0
Vavg: 13.571428571428571
Observaciones: Origin of segment does not match with Special Point.
```

Ilustración 113 seg_toSQL_process.

Como se aprecia en la ilustración, la línea de “*I record was inserted in segments*” es tarea de la función insertora mediante el *rowcount*.

Este segmento posee como anomalía que empieza en un punto no perteneciente a un SP.

```
('4317JCD') 2021-10-19 10:20:00 2022-01-07 02:00:37 Parado 0.0
1 record was inserted in segments.
Dateseg: 2021-10-19 00:00:00
Desde: 2021-10-19 09:05:34
Hasta: 2021-10-19 10:22:38
Número de Segmento en Ruta: 3
ID origen: 845492
ID final: 845630
Matrícula: 4317JCD
Special Point Origen: C - 3 [42.80736, -1.69832]
Special Point Final: C - 3 [42.8074, -1.69833]
Kilómetros: 24.272944354251543
Vmax: 89.0
Vavg: 25.5625
Observaciones: C - 7 (0:00:00) / C - 2 (0:04:00)
```

Ilustración 114 seg_toSQL_process_2.

El segmento mostrado resulta interesante, puesto que comprende de un segmento que inicia y acaba en el mismo SP, pasa por otro SP distinto y permanece cuatro minutos en un tercer SP.

Con el fin de mostrar lo interesante del segmento, se procede a la visualización detallada de este.

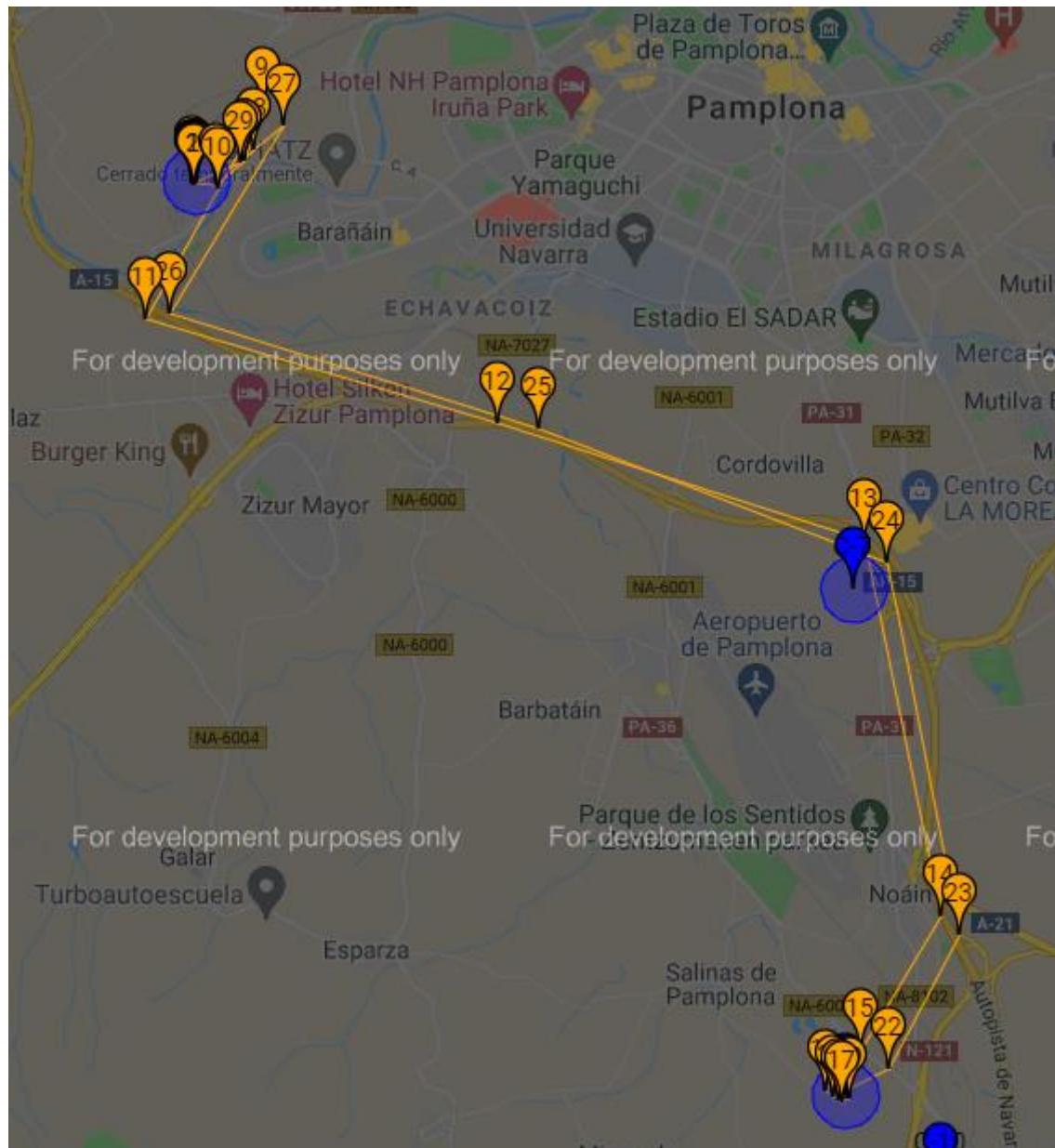


Ilustración 115 segment_map.

Se aprecia en la ilustración como el segmento comprende de una ruta cerrada, sin parar, pero en cambio permanece cuatro minutos en el SP “C-2”, el situado más al sur, y posteriormente vuelve. Este fenómeno se repite para este vehículo numerosas veces, por lo que el proceso permite sacar conclusiones que no serían posibles sin la incorporación del proceso de estancia en SPs sin parada.

4.4. route_toSQL.py.

4.4.1. Introducción.

El archivo *route_toSQL.py* es el archivo continente de la función *route_toSQL*, que comprende la función que realiza la acción de insertar la ruta completa, en cuestión de los parámetros que calcula, en la tabla *routes*.

Las rutas se definen como la sucesión de segmentos para una misma fecha y una misma matrícula, de la cual se distinguen los segmentos dentro de ella como partes únicas no solapadas.

Puesto que existe una discriminación de segmentos demasiado cortos para ser estudiados, es posible que, en la ruta, el primer registro de un segmento no coincida con el último registro del anterior segmento. El segmento consecutivo comienza en un punto cercano al último registro del segmento anterior, nunca a más de medio kilómetro.

El proceso realizado dentro de la función comprende la definición de los siguientes parámetros: *Date*, *desde*, *hasta*, *matricula*, *NumSeg*, *Idorg*, *Idend*, *NumParadas*, *DurViajes*, *DurParadas*, *seqParadas*, *Kms*, *Vmax*, *Vavg*, *DiaSem*.

- *Date* comprende la fecha para la que se realiza el trazado de la ruta.
- *desde* comprende la hora a la que da comienzo la ruta, la hora que marca el primer registro *en marcha* del primer segmento.
- *hasta* comprende la hora de finalización de la ruta, la hora que marca el último registro del último segmento.
- *matricula* comprende la identificación del vehículo para el que se está realizando el trazado.
- *NumSeg* comprende el número de segmentos en los que se traza la ruta.
- *Idorg* e *Idend* comprenden los *ids* de la tabla de *segments* correspondientes al segmento de inicio y al segmento final de la ruta.
- *NumParadas* comprende el número de paradas que existen en la ruta, (1 menos que el número de segmentos)
- *DurViajes* comprende el tiempo total *en marcha* de la ruta completa.
- *DurParadas* comprende el tiempo total, desde que inicia la ruta hasta que finaliza, en el que el vehículo se encuentra *parado*.
- *seqParadas* comprende la secuencia de SPs en los que se realizan las paradas en la ruta.
- *Kms* comprende los kilómetros totales de la ruta.
- *Vmax* comprende la velocidad máxima alcanzada en la ruta.

- *Vavg* comprende la velocidad media de la ruta, calculada a partir de la media de las velocidades media de los segmentos.
- *DiaSem* comprende el día de la semana que corresponde a la fecha para la cual se realiza el trazado de la ruta.

4.4.2. Cuerpo.

Como primer objeto en el cuerpo del archivo, al igual que en el resto, obtenemos las importaciones.

```
from database import DataBase
import statistics
import datetime
import numpy as np

database = DataBase()

sp = database.select_sp()
```

Ilustración 116 route_toSQL_imports.

Se importa la clase *Database* y se la nombra *database* para el acceso a las funciones del archivo database.py. Se importa la librería *statistics* para realizar operaciones algebraicas. Se importa *datetime* para el manejo de variables temporales. Se importa *numpy* para la formación de *arrays*.

Se aprovecha para definir la variable *sp* como la lista del conjunto de SPs de la tabla de *specialsites*.

A continuación, se define la función, al ser muy extensa se define por partes que realizan acciones particulares sin olvidar la sinergia que posee cada parte con el resto de la función.

La definición de la función se realiza para dos variables en su argumento: la lista de listas de *ids* que pertenecen a los registros de cada segmento, y la fecha para la que se realiza el trazado.

```
def route_toSQL(route, date_):
    if len(route) > 1:

        seg_org = route[0]
        dat_org = database.select_data_forseg(seg_org[0])

        data = list(database.select_data_for_route(dat_org[1], date_))
        data.sort(key=lambda s: s[2])
        data = np.array(data)
```

Ilustración 117 route_toSQL_func_1.

El primer paso es cuestionar la existencia de la ruta, se cuestiona si la longitud de la ruta es mayor que uno. El cuestionamiento de la ruta se realiza de este modo debido a que, si se cuestiona su existencia de la manera habitual, cumplirá la condición gran parte de las veces que no debería.

En la función de *segments* se describió uno de los procesos por el cual ante la ausencia de segmento y de ruta, al llegar a la variable temporal de finalización, se añadía la posición al segmento y el segmento a la ruta. Por lo que la ruta comprende de un segmento de una posición inmutable. Es por ello, que mediante el método de cuestionamiento de longitud mayor que 1, se excluyen las rutas de punto único y las que conforman una lista vacía.

Se define el segmento inicial y se realiza la extracción de los parámetros de: *s_motor*, *matricula*, *velocidad*, *X(ubicacion)*, *Y(ubicacion)*, *fecha_upos*, *localidad*, para el primer *id* del primer segmento a través de *select_data_forseg*. La tupla se guarda en la variable *dat_org*.

A continuación, con el segundo dato de la tupla, la matrícula, y la fecha, se realiza una extracción de los parámetros de: *desde*, *hasta*, *Num*, *Kms*, *PointA*, *PointB*, *Vmax*, *Vavg*, *idorg*, *idend*, *id* de la tabla *segments*. Se guardan los parámetros en la variable *data* en forma de lista en vez de tupla.

Es decir, se realiza una extracción de *segments* de los segmentos que existen para una matrícula y una fecha.

El proceso sucede de la forma que: la función *segments* recoge registros de *movements*, traza el segmento, llama a *seg_toSQL*, la cual ingresa el segmento en la tabla *segments*. En cuanto termina el proceso de trazado de segmento para la fecha se traza la ruta, la función *segments* entonces llama a *route_toSQL*, la cual, mediante lo el proceso descrito en el párrafo anterior, recoge los registros de *segments* recién ingresados. Es decir, en el mismo proceso de la función *segments*, se ingresan los segmentos en la tabla *segments* y se extraen de *segments* para *routes*.

Por lo que *data* comprende una lista con los segmentos insertados en *segments* “poco antes” en el proceso de trazado.

Para trazar la ruta es necesario ordenar la lista, para ello se utiliza la función *short()* con la clave *lambda* para el tercer término de cada tupla (*s[2]*) que comprende el número del segmento.

Para comprender mejor el proceso que se acaba de describir, se proporcionan las siguientes ilustraciones de una variable *data* de una ruta cualquiera antes y después de pasar por *short()*.

Acceder a una variable, a medio proceso, para observar su contenido sin tener que imprimirla en pantalla, solo es posible accediendo al *Debug* del sistema. El *Debug* nos permite realizar el proceso controlando nosotros los pasos para observar detenidamente el comportamiento de cada variable.



Ilustración 118 Debug.

Para el *Debug* de PyCharm, es necesario interrumpir el proceso en una línea de código y posteriormente ejecutarlo con el *Debug*, este ejecutará el archivo de la misma manera que el *run* común con la particularidad de que se detendrá en la línea de código elegida para observar el comportamiento del algoritmo. En adelante el usuario elige proceder accionando de forma manual cada paso.

Por lo que para observar la acción de la función *short()* interrumpimos el código en la línea que queremos analizar:

```

● | data = list(database.select_data_for_route(dat_org[1], date_))   data: [(datetime.timedelta(se
|     data.sort(key=lambda s: s[2])
|     data = np.array(data)

```

Ilustración 119 Interruption.

La consecución de la variable en las tres líneas de código mostradas en la ilustración anterior queda de la forma:

```

▼ └─ data = {list: 3} [(datetime.timedelta(seconds=40816), datetime.timedelta(seconds=43205), 2, 45.82412068444224, 'NotSP. Special Point not defined [43.0
  > └─ 0 = {tuple: 11} (datetime.timedelta(seconds=40816), datetime.timedelta(seconds=43205), 2, 45.82412068444224, 'NotSP. Special Point not defined [4
  > └─ 1 = {tuple: 11} (datetime.timedelta(seconds=25342), datetime.timedelta(seconds=36603), 1, 121.24535717318216, 'C - 2 [42.74578, -1.63746]', 'NotSP.
  > └─ 2 = {tuple: 11} (datetime.timedelta(seconds=44401), datetime.timedelta(seconds=48462), 3, 17.018349223047863, 'NotSP. Special Point not defined [4
  └─ __len__ = {int} 3

```

Ilustración 120 data.

```

▼ └─ data = {list: 3} ... Loading Value
  > └─ 0 = {tuple: 11} (datetime.timedelta(seconds=25342), datetime.timedelta(seconds=36603), 1, 121.24535717318216, 'C - 2 [42.74578,
  > └─ 1 = {tuple: 11} (datetime.timedelta(seconds=40816), datetime.timedelta(seconds=43205), 2, 45.82412068444224, 'NotSP. Special P
  > └─ 2 = {tuple: 11} (datetime.timedelta(seconds=44401), datetime.timedelta(seconds=48462), 3, 17.018349223047863, 'NotSP. Special
  └─ __len__ = {int} 3

```

Ilustración 121 data_shorted.

| | 0 | 1 | 2 | 3 | 4 | |
|---|----------|----------|---|--------------------|--|-------|
| 0 | 7:02:22 | 10:10:03 | 1 | 121.24535717318216 | C - 2 [42.74578, -1.63746] | Not |
| 1 | 11:20:16 | 12:00:05 | 2 | 45.82412068444224 | NotSP. Special Point not defined [43.0837, -1... | Not |
| 2 | 12:20:01 | 13:27:42 | 3 | 17.018349223047863 | NotSP. Special Point not defined [42.842, -1.... | C - 2 |

Ilustración 122 data_array.

Las ilustraciones muestran el proceso, en *Debug*, para una ruta de travesías segmentos. La primera ilustración muestra la lista de los registros de la tabla segments, la segunda ilustración muestra la misma lista ordenada en base al segundo parámetro, la tercera ilustración muestra la transformación de la lista en *array*.

A continuación, se muestran las variables que necesitan guardar en listas la agregación de sus elementos en todos los segmentos.

```

if len(data) > 1:
    Km = []
    Vmaxs = []
    Vavgs = []
    seqs = []
    seqs_fin = []

    for seg in data:
        Km.append(seg[3])
        Vmaxs.append(seg[6])
        Vavgs.append(seg[7])
        seqs.append(seg[4])
        seqs_fin.append(seg[5])

    if seqs_fin:
        seqs.append(seqs_fin[-1])

    Kms = sum(Km)
    Vmax = max(Vmaxs)
    Vavg = statistics.mean(Vavgs)

```

Ilustración 123 route_toSQL_func_2.

Se definen las listas vacías para la posterior agregación de elementos mediante un bucle *for* que recorra las tuplas en la lista de segmentos y agregue en cada lista el elemento correspondiente.

Las listas *seqs* y *seqs_fin* son para la realización de la secuencia de SPs que comprenden la parada del vehículo. La secuencia comprende el punto inicial, posteriormente el punto final del segmento que a su vez es el punto inicial del siguiente segmento, así sucesivamente hasta llegar al punto final. El proceso entonces es guardar todos los puntos iniciales y finales en listas separadas, posteriormente se agrega el último punto final a la lista de puntos iniciales, de esta manera queda la ruta completa. El mismo resultado queda de agregar el primer elemento de la lista de puntos iniciales a la lista de puntos finales.

Se define *Kms* como la suma de todos los elementos de la lista de *Km*, se define *Vmax* como el valor máximo de la lista *Vmaxs*, se define *Vavg* como la media de los elementos de la lista *Vavgs*.

Se define la secuencia de paradas como un *str* mediante el método de:

```
seqParadas = ""
for point in seqs:
    seqParadas = seqParadas + "/" + point[0: 6]
seqParadas = seqParadas[1: ].replace(" ", "")
```

Ilustración 124 route_toSQL_func_3.

Con este bucle se separan los puntos con barras /, se elimina la primera barra y se eliminan los espacios para compactar el *str* final.

A continuación, se muestra el proceso por el cual se calculan los datos de los tiempos de viaje y parada.

```
seg_ini = []
seg_fin = []
for seg in data:
    seg_ini.append(seg[0])
    seg_fin.append(seg[1])

DurViajes = datetime.timedelta()
for i in range(0, len(seg_ini)):
    DurViajes = DurViajes + (seg_fin[i] - seg_ini[i])

DurParadas = datetime.timedelta()
for i in range(0, len(seg_ini) - 1):
    DurParadas = DurParadas + (seg_ini[i + 1] - seg_fin[i])
```

Ilustración 125 route_toSQL_func_4.

El primer paso del proceso es la creación de dos listas, una para las horas iniciales de los segmentos y otra para las horas de finalización de los segmentos.

Los tiempos de viaje se definen como la diferencia entre las horas de finalización y las horas de inicio de los segmentos. Los tiempos de parada se definen como la diferencia entre las horas de inicio del segmento siguiente y las horas de finalización del segmento.

Por lo que el proceso consta en la creación de dos bucles *for* con variables iterativas, y unas variables temporales que se autodefinen con la agregación de los tiempos definidos por cada vuelta del bucle.

La variable iterativa *i* del bucle de los tiempos de parada se debe definir para un valor anterior a la longitud total de la lista, en caso contrario emergirá el error *list index out of range* al referirnos al término sucesivo cuando alcance su valor máximo.

Las variables de duración se definen en primera instancia como variables temporales vacías, pero es importante que posean el formato, puesto que de lo contrario no se le podrían agregar variables temporales.

A continuación, se definen las variables *org* y *end* como los segmentos iniciales y finales de la ruta y se definen los parámetros restantes para la inserción de la ruta en la tabla.

```
org = data[0]
end = data[-1]

date_ = datetime.datetime.strptime(date_, '%Y-%m-%d %H:%M:%S')

Date = date_
DiaSem = date_.strftime('%A')
desde = org[0]
hasta = end[1]
Idorg = org[10]
Idend = end[10]
matricula = dat_org[1]
NumSeg = (data[-1])[2]
NumParadas = (data[-1])[2] - 1

route_data = (
    Date, desde, hasta, matricula, NumSeg, Idorg, Idend, NumParadas, DurViajes, DurParadas, seqParadas, Kms,
    Vmax, Vavg, DiaSem)
```

Ilustración 126 route_toSQL_func_5.

Se transforma la fecha en fecha y hora para que coincida con la sintaxis de los parámetros temporales de la BBDD.

Se definen los parámetros para su inserción:

- *DiaSem* comprende la variable temporal de la fecha y hora, expresada en el formato *%A*, el cual indica el día de la semana que resulta.
- *desde* comprende la hora de inicio del primer segmento *org*.
- *hasta* comprende la hora de finalización del último segmento *end*.
- *Idorg* comprende el *id* de *org*.
- *Idend* comprende el *id* de *end*.
- *matricula* comprende el segundo parámetro de la tupla de *dat_org*.
- *NumSeg* comprende el segundo parámetro del último segmento *end*.

- *NumParadas* comprende el valor de *Numseg - 1*.

Por último, se recogen todos los parámetros en una variable que englobe la tupla total, se procede a la inserción en la tabla *routes* mediante *add_route_toSQL* y se realiza la impresión en pantalla de los parámetros de ruta.

```

database.add_route_toSQL(str(Date), str(desde), str(hasta), str(matricula), str(NumSeg), str(Idorg), str(Idend),
                         str(NumParadas), str(DurViajes), str(DurParadas),
                         seqParadas, str(Kms), str(Vmax), str(Vavg), DiaSem)

        print(
            f"""
Date: {route_data[0]}
Desde: {route_data[1]}
Hasta: {route_data[2]}
Matrícula: {route_data[3]}
Número de Segmentos: {route_data[4]}
ID origen: {route_data[5]}
ID final: {route_data[6]}
Número de Paradas: {route_data[7]}
Duración Viajes: {route_data[8]}
Duración Paradas: {route_data[9]}
Secuencia Paradas: {route_data[10]}
Kilómetros: {route_data[11]}
Vmax: {route_data[12]}
Vavg: {route_data[13]}
Dia de la Semana: {route_data[14]}
        """
        )

    else:
        print("No route")

```

Ilustración 127 *route_toSQL_func_6*.

Es necesaria la conversión de los parámetros a *str* para su inserción en la BBDD.

Al igual que en *seg_toSQL*, se deshacen las tabulaciones para que la impresión de los parámetros quede todas al mismo nivel horizontal.

En última instancia, ante que la ruta no cumpla el condicional de existencia, *if len(route) > 1*, se imprime en pantalla *No route* en señal de que, para la matrícula analizada y la fecha introducida, no existe ruta para trazar.

A continuación, se procede a mostrar la ejecución de la función, aunque al igual que *seg_toSQL*, únicamente se muestra en pantalla ante la inserción de una ruta en la BBDD.

Se muestran registros de rutas con interés.

```
('4317JCD') 2021-11-25 00:00:00 2022-01-07 20:06:07 En marcha 0.0
No segment.
1 record was inserted in routes.
Date: 2021-11-24 00:00:00
Desde: 0:05:27
Hasta: 22:25:33
Matricula: 4317JCD
Número de Segmentos: 12
ID origen: 3895
ID final: 3906
Número de Paradas: 11
Duración Viajes: 15:51:05
Duración Paradas: 6:29:01
Secuencia Paradas: C-3/C-3/C-2/C-3/C-3/C-2/C-3/C-3/C-3/C-3/C-2/C-3
Kilómetros: 293.6895804899579
Vmax: 90.0
Vavg: 22.467103668390433
Día de la Semana: Wednesday
```

Ilustración 128 route_1.

Como primer ejemplo de ruta obtenemos una con 12 segmentos, los cuales han resultado entre dos SPs concretos. Observamos que a lo largo del día ha realizado casi 300 km, resultando en 16 horas de viaje y 6 horas y media de parada.

Se procede a mostrar de manera detallada la trayectoria de la ruta completa.

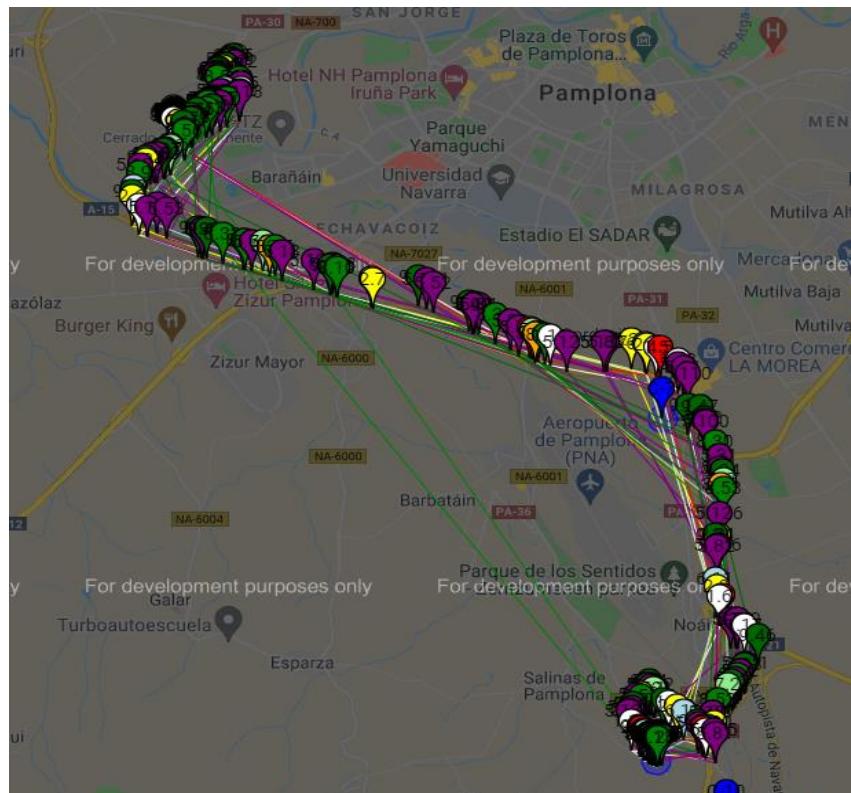


Ilustración 129 route_1_vis.

Como se puede apreciar, la ruta consiste en la ida y vuelta continuada entre dos SPs.

```
('4317JCD') 2021-08-09 23:55:00 2022-01-07 20:01:04 Parado 0.0
1 record was inserted in routes.
Date: 2021-08-09 00:00:00
Desde: 6:01:08
Hasta: 13:11:38
Matricula: 4317JCD
Número de Segmentos: 4
ID origen: 3879
ID final: 3882
Número de Paradas: 3
Duración Viajes: 6:10:43
Duración Paradas: 0:59:47
Secuencia Paradas: C-2/C-2/C-2/NotSP./C-2
Kilómetros: 121.62870948382036
Vmax: 90.0
Vavg: 24.858963585434175
Día de la Semana: Monday
```

Ilustración 130 route_2.

Como segundo ejemplo, obtenemos una ruta para la cual la secuencia resulta una sucesión de segmentos de trayectorias cerradas que inician y acaban siempre en *C-2*.

Se procede a mostrar de manera detallada la trayectoria de la ruta completa.

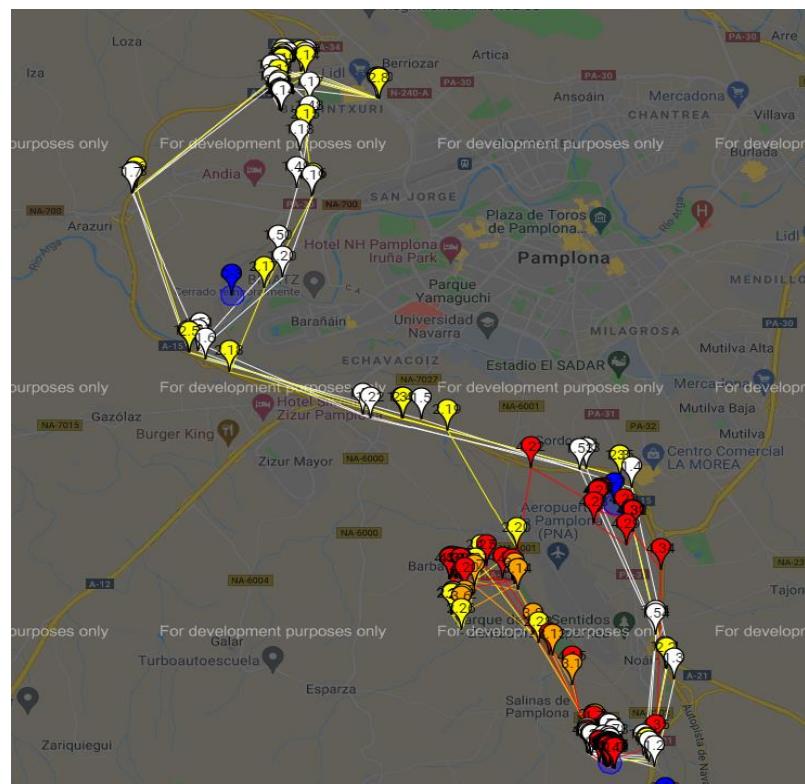


Ilustración 131 route_2_vis.

Como se puede apreciar, la ruta consta de una serie de trayectorias cerradas sobre el SP de *C-2*. *C-2* es el SP en azul situado más al sur, donde se sitúa la mayor concurrencia de la ruta.

```
('9848LKX') 2021-08-09 17:55:00 2022-01-07 20:01:16 En marcha 0.0
1 record was inserted in routes.
Date: 2021-08-09 00:00:00
Desde: 4:38:20
Hasta: 13:15:41
Matricula: 9848LKX
Número de Segmentos: 5
ID origen: 3883
ID final: 3887
Número de Paradas: 4
Duración Viajes: 6:22:38
Duración Paradas: 2:14:43
Secuencia Paradas: C-2/NotSP./NotSP./NotSP./NotSP./NotSP.
Kilómetros: 202.0545259059118
Vmax: 91.0
Vavg: 34.29981954887218
Día de la Semana: Monday
```

Ilustración 132 route_3.

Por último, un ejemplo de ruta que realiza paradas en puntos en los que no se encuentra un SP.

Se procede a mostrar de manera detallada la trayectoria de la ruta completa.

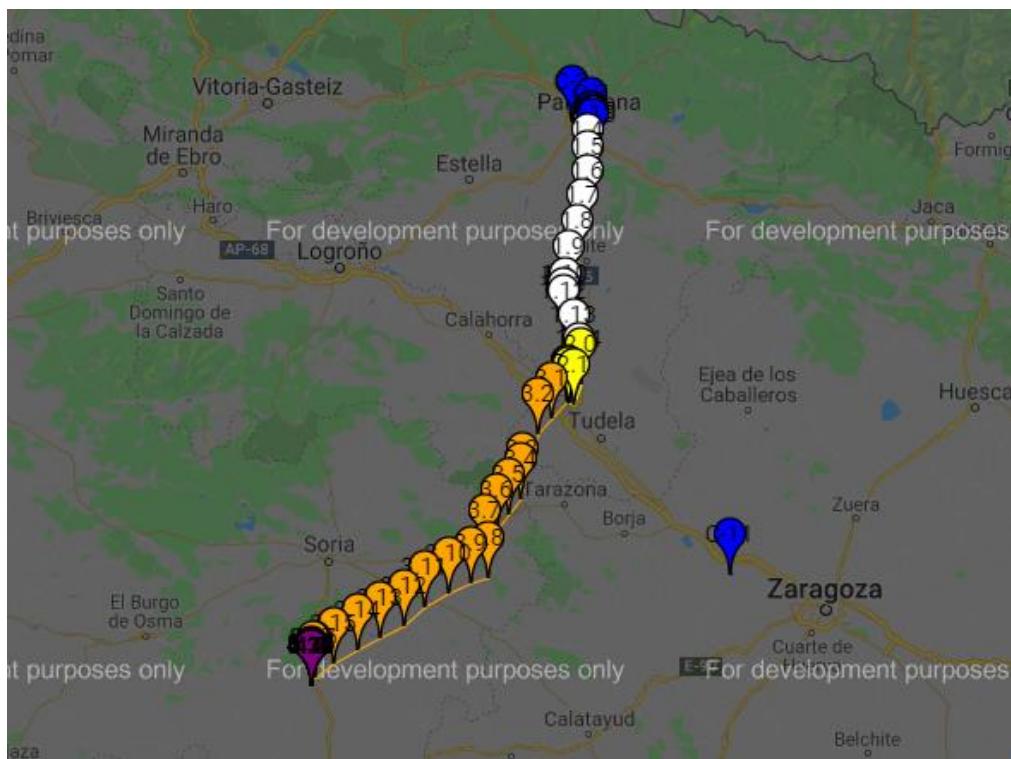


Ilustración 133 route_3_vis.

Como se puede observar, consta de una ruta de ida desde C-2, hasta un lugar al sur de Soria.

A modo de curiosidad, el vehículo, al día siguiente regresa a C-2.

```
1 record was inserted in routes.
Date: 2021-08-10 00:00:00
Desde: 6:45:46
Hasta: 13:02:46
Matricula: 9848LKX
Número de Segmentos: 3
ID origen: 3946
ID final: 3948
Número de Paradas: 2
Duración Viajes: 3:52:05
Duración Paradas: 2:24:55
Secuencia Paradas: NotSP./NotSP./NotSP./C-2
Kilómetros: 206.48142161604025
Vmax: 88.0
Vavg: 37.632097069597066
Dia de la Semana: Tuesday
```

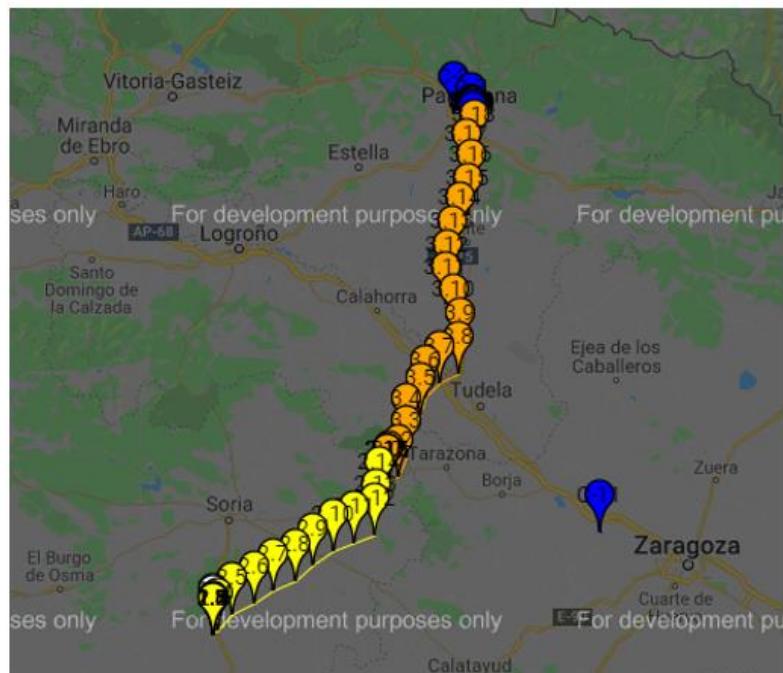


Ilustración 134 route_3_vis_back.

Con este capítulo quedan explicadas las funcionalidades de *Routing*.

En el siguiente capítulo se explicarán las funcionalidades secundarias de la herramienta, las cuales comprenden de un archivo.

CAPÍTULO 5

FUNCIONALIDADES SECUNDARIAS

5.1. Introducción.

Como introducción al capítulo se indican las funcionalidades secundarias de la herramienta.

La funcionalidad secundaria más relevante es *Mapping* que comprende de *Map route/seg selection* y *Map all sp*. La funcionalidad crea, mediante un archivo .html, un API de Google Maps que recoge, dependiendo de la funcionalidad seleccionada, las rutas y segmentos junto a los SPs, o únicamente los SPs.

La funcionalidad se *Scatter Search* comprende *Scatter with day selection* y *Scatter last month*. La funcionalidad recoge los registros de *movements* para un periodo de tiempo definido y los grafica junto a los SPs.

La funcionalidad de *Search in DDBB* comprende de una conexión directa entre la interfaz del usuario y la BBDD. En esta funcionalidad es necesario que el usuario conozca bien la sintaxis SQL. Junto a la explicación de la funcionalidad se otorgan algunos ejemplos que pueden resultar útiles.

La funcionalidad de *Reset DDBB* comprende de un conjunto de funciones eliminadoras que apuntan a la tabla de la que hacen referencia. Esta funcionalidad elimina todos los registros de una tabla a elegir entre: *specialsites*, *segments*, *routes*. Para tabla *movements*, no tiene sentido que exista una función de reseteo, puesto que la función no ingresa en la tabla, además, se carece del derecho de modificarla.

Las funcionalidades vienen explicadas en el archivo que las comprende.

Se darán ejemplos de ejecución en todas las funcionalidades excepto para la funcionalidad de reseteo, la cual se ejecutará hasta cierto punto donde todavía no ha realizado su propósito.

5.2. Map all SP.

5.2.1. Introducción.

Con el fin de explicar de manera más comprensiva las funcionalidades de mapeo de puntos, se explica previamente *Map all sp* antes que *Map route/seg selection*, puesto que la funcionalidad de mapeo de rutas y segmentos resulta notablemente más extensa que la primera.

La funcionalidad de *Map all sp* comprende del archivo map.py y consta del graficado en detalle de los SPs de la tabla de *specialsites* mediante un archivo .html que conlleva un API de Google Maps.

El archivo map.py comprende el continente de la función *mapping*, encargada de realizar el proceso.

La construcción del archivo .html se realiza mediante la librería de *gmplot* la cual es una abreviación de *Google Map Plot*.

Posteriormente a la construcción del archivo, la herramienta lo abre en el navegador predeterminado mostrando el mapa generado.

5.2.2. Cuerpo.

Como primer objeto en el cuerpo del archivo, se obtienen las importaciones:

```
from os import getcwd  
  
import gmplot  
from database import DataBase  
import numpy as np  
import webbrowser  
  
database = DataBase()  
  
sp = np.array(database.select_sp_name())  
  
path = getcwd()
```

Ilustración 135 map_imports.

De la librería de *os* se importa la función de *getcwd* para definir la ruta de acceso al archivo. Se importa la librería de *gmplot* para construir el archivo .html. Se importa la librería *numpy* como *np* para la formación de *arrays*. Se importa la librería *webbrowser* para abrir el archivo generado.

Se importa la clase *Database* y se renombra *database* para su manejo.

Se aprovecha el exterior de la función para definir la variable que recoge los SPs, en forma de *array* en vez de lista, *sp*, a partir de la función *select_sp_name*. La función recoge, en el

el mismo orden, *longitud*, *latitud* y *nombre del SP* de la tabla de *specialsites*. La función se diferencia con *select_sp* en que incorpora el nombre del SP.

Se define la variable *path* como la ruta de acceso al archivo mediante la función *getcwd*.

A continuación, se describe el proceso que se da dentro de la función.

```
def mapping(place):  
  
    global gmap  
  
    if place == '0':  
        gmap = gmplot.GoogleMapPlotter(39.9902498, -3.9255631, 7)  
  
    if place == '1':  
        gmap = gmplot.GoogleMapPlotter(47.1939666, 19.2374724, 7)  
  
    for i in sp:  
        gmap.scatter([i[0]], [i[1]], 'blue', label=i[2].replace(" ", ""), size=50)  
    gmap.draw(f'{path}/map.html')  
  
    webbrowser.open_new_tab(f'{path}/map.html')
```

Ilustración 136 *mapping_func*.

La función engloba ambas funcionalidades, mediante la variable del argumento *place* se decide el lugar (España o Hungría) que se va a mapear.

Se define como global la variable *gmap* y posteriormente se define en los condicionales que indican la selección de lugar.

Cada condicional consta de la creación de un mapa. Los mapas vienen dados por la coordenada central en la que se sitúa el dentro de la pantalla y el zoom con el que se muestra en un principio. Mediante la función de *GoogleMapPlotter* se define la variable *gmap* que consta del mapa creado para cada lugar.

A continuación, se insertan los puntos en el mapa mediante la función *scatter*.

Para ello se crea un bucle con una variable que toma los valores de las tuplas de la lista de *sp*, cada punto del mapa, cada elemento de *sp*, contiene *longitud*, *i[0]*; *latitud*, *i[1]*; y *name*, *i[2]*. Cada vuelta del bucle define un punto.

Posteriormente se crea el archivo .html en el mismo directorio donde se encuentra el archivo *map.py*. Esto se realiza de esta manera debido a que el archivo *map.py* se encuentra en el mismo directorio que *main.py*, por lo que, mediante este método, todos los archivos quedan en el mismo directorio. Esta acción se realiza mediante *draw*, como argumento de la función se encuentra la ruta, el nombre del archivo a generar y el tipo de archivo a generar.

Por último, se abre el archivo mediante la misma ruta que en su generación, esta acción viene dada por *open_new_tab* de *webbrowser*.

A continuación, se procede a mostrar la ubicación de los SPs mediante su mapeado a detalle.

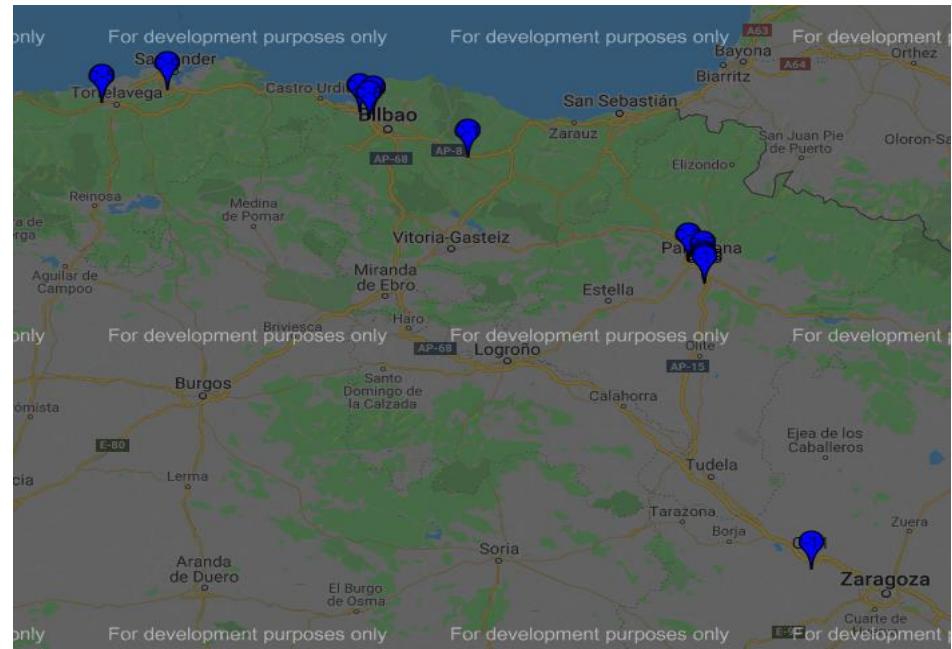


Ilustración 137 SPs_Spain.

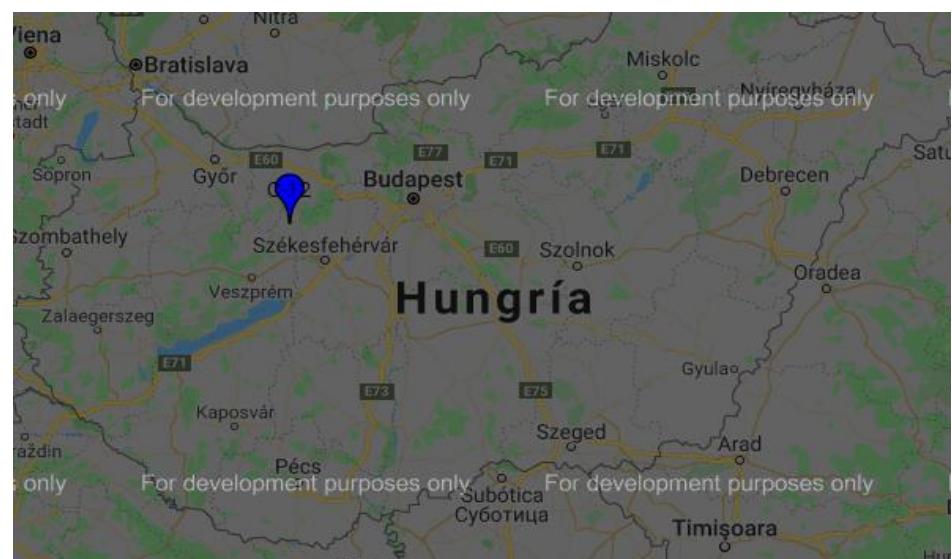


Ilustración 138 SPs_Hungary.

Esta funcionalidad se realiza con el objeto de crear un método de observación directa de los SPs sin pasar por un procesado de ruta.

A continuación, se procede a mostrar el archivo .html creado.

```

<html>
<head>
<meta name="viewport" content="initial-scale=1.0, user-scalable=no" />
<meta http-equiv="content-type" content="text/html; charset=UTF-8" />
<title>Google Maps - gmplot</title>
<script type="text/javascript" src="https://maps.googleapis.com/maps/api/js?libraries=visualization"></script>
<script type="text/javascript">...</script>
</head>
<body style="..." onload="initialize()">
    <div id="map_canvas" style="..." />
</body>
</html>

```

Ilustración 139 map.html.

La ilustración comprende el archivo que genera el mapa en el navegador, se ha comprimido con el fin de visibilizar de manera clara la estructura que consta.

Para abrir un componente e introducir elementos en él se llama al componente entre “<>”. Para cerrar el componente se le llama entre “</>”.

La introducción de los SPs se realiza en el script del archivo.

Se inicia una función que define una imagen, que consta del mapa, y posteriormente pinta *markers* en la imagen en ciertas coordenadas y con ciertos nombres.

```

<script type="text/javascript">
    function initialize() {
        var map = new google.maps.Map(document.getElementById("map_canvas"), {
            zoom: 7,
            center: new google.maps.LatLng(47.193967, 19.237472)
        });

        var marker_icon_0000FF = {
            url: "data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAABUAAAAAiCAYAACwaJKDAAAABmJLR0
            labelOrigin: new google.maps.Point(10, 11)
        };

        new google.maps.Marker({
            position: new google.maps.LatLng(43.163560, -2.601550),
            label: "C-1",
            icon: marker_icon_0000FF,
            map: map
        });

        new google.maps.Marker({
            position: new google.maps.LatLng(42.745610, -1.638370),
            label: "C-2",
            icon: marker_icon_0000FF,
            map: map
        });
    }

```

Ilustración 140 map.html_markers.

A continuación, se procede a explicar la función de visualizado de rutas y segmentos.

5.2. Map route/seg selection.

5.2.1. Introducción.

La funcionalidad *Map route/seg selection* comprende del archivo visualizado.py y conta del graficado en detalle de los segmentos de ruta de manera individual o en conjunto a elegir.

El graficado se realiza mediante la creación de un archivo .html que conlleva un API de Google Maps donde se plasman los puntos específicos de los segmentos y SPs.

El propósito de la función es crear un método de visualizado preciso de las rutas y segmentos, contando con una flexibilidad que permita la elección libre de fecha, matrícula y segmentos que mapear.

La elección de segmentos propone tres opciones: segmento único, secuencia concreta y ruta completa.

La funcionalidad comprenda, además del graficado, la impresión en pantalla de los registros de cada segmento para realizar la selección en base a ello.

El proceso que ejecuta el usuario consta de la introducción de una fecha, para la cual se le muestran las matrículas que dieron lugar a registros en esa fecha. Una vez elegida la matrícula se procede a extraer la ruta completa e imprimir en pantalla el registro al que pertenece, mostrando el número de segmento que posee. Por último, se eligen los segmentos a mapear y se imprimen en pantalla sus registros, de manera que se pueda comparar su análisis con el mapeado. Mediante *ENTER* en la terminal se abre el mapa en el navegador, si se pulsa nuevamente se vuelve a la selección de segmentos para mapear.

5.2.1. Cuerpo.

Como primer objeto en el cuerpo del archivo obtenemos las importaciones:

```
import gmplot
from database import DataBase
import numpy as np
import webbrowser
from datetime import datetime, date, timedelta
from os import system, name, getcwd

database = DataBase()
sp = np.array(database.select_sp_name())
path = getcwd()
```

Ilustración 141 visualization_imports.

Se importa la librería *gmplot* para la generación de mapas en Google Maps. Se importa la clase *Database* y se la denomina *database* para el acceso a las funciones de la BBDD. Se

importa la librería *numpy* como *np* para la formación de *arrays*. Se importa la librería *webbrowser* para abrir los archivos .html generados. De la librería *datetime* se importan las funciones *date*, *datetime* y *timedelta* para el manejo de variables temporales. Se importan las funciones *system*, *name* y *getcwd* de la librería de *os* para el manejo de la interfaz y establecer las rutas de los archivos.

Se aprovecha para extraer los SPs como *array* y guardarlos en la variable *sp*.

Se establece la variable *path* como la ruta del archivo continente de la función.

En primer lugar, en la función, se establece el código de colores para el orden de los segmentos. De manera que, para el mapeo de los segmentos, se asocie directamente el color con el número de segmento que constan, incluso en el mapeo de segmentos aislados debe conservarse el código de color.

El código de colores se define de la siguiente manera, constan de diez colores, desde el más claro al más oscuro sin ser negro, y se termina con el dorado para la distinción del segmento número 10. Por lo tanto, el primer segmento será siempre *blanco*, el segundo *amarillo*, el tercero *naranja*, el cuarto *rojo*, el quinto *morado*, el sexto *azul claro*, el séptimo *verde claro*, el octavo *verde*, el noveno *verde oscuro*, el décimo *dorado*, el undécimo *blanco* de nuevo, y así sucesivamente.

Se omite el color *azul oscuro* porque los SPs vendrán dados en este color.

```
def visualization():
    global i
    color = ['white', 'yellow', 'orange', 'red', 'purple', 'lightblue', 'lightgreen', 'green',
              'darkgreen', 'gold']
    date_ = str(input("""Insert date [dd/mm/yyyy]:      Or      Insert amount of backdays (from now):
"""))
    if date_:
        if len(date_) == len('dd/mm/yyyy'):
            date_ = datetime.strptime(date_, '%d/%m/%Y')
        elif len(date_) <= 7:
            date_ = date.today() + timedelta(days=-int(date_))

        date_ = str(date_)

        data = database.select_mat_fromroutes(date_)

        system('cls' if name == 'nt' else 'clear')
```

Ilustración 142 *visualization_func_1*.

Se define la variable *i* como variable global ante su definición sujeta a condicionales.

Se determina la fecha a elegir mediante el mismo método empleado en las demás funciones y se almacena en la variable *date_*, posteriormente se convierte en *str*.

Se seleccionan las matrículas de los registros de la fecha establecida mediante *select_mat_fromroutes*, guardándose en *data* y se limpia la pantalla.

Cabe destacar la diferencia entre el proceso de extracción de matrículas para un día concreto de la función de *segments* y este. El proceso de *segments* recogía cientos de matrículas y agregabas las diferentes a una lista *unique_mats*, mientras que *visualization* recoge las matrículas de forma inmediata a partir de una función. La diferencia reside en que *visualization* recoge las matrículas para un día concreto de la tabla de *routes*, en la cual, cada ruta pertenece a un vehículo, por lo tanto, solo existe un registro por matrícula.

A continuación, se procede a la selección de la matrícula a mapear.

```
print(f"""Day {date_}""")
if data:
    print("")
    for i in range(0, len(data)):
        print(i + 1, (data[i])[0])
    print("")
    mat = input("Select mat: ")
    system('cls' if name == 'nt' else 'clear')
```

Ilustración 143 visualization_func_2.

Se imprime la fecha y posteriormente, mediante un bucle con una variable iterativa *i* se imprimen las matrículas y su número de selección.

Se guarda la matrícula en la variable *mat*, que comprende de un valor numérico.

Las impresiones vacías son para una mejor presentación en la interfaz, posteriormente se limpia la pantalla.

En la ejecución del proceso se mostrará la interfaz resultante de la selección de matrículas.

Con el fin de evitar el error de *list index out of range* ante la introducción de un número fuera de rango de la lista, es necesaria la introducción de un condicional que cuestione esta premisa.

A continuación, se redefine *mat* como la matrícula real, no el número que la representa y se procede a extraer los datos de la ruta definida por una matrícula y una fecha. La acción se realiza mediante la función *select_data_fromroutes*. Posteriormente se pasa a formación *array* la ruta extraída.

En este punto en el algoritmo empieza el bucle *while* de visualizado.

Con el fin de evitar una tarea tediosa de selección de fecha, matrícula y segmento, para el análisis de una ruta o un vehículo, se realiza un bucle para únicamente ir insertando en la interfaz el segmento que se pretende mapear. De esta forma la herramienta se vuelve mucho más cómoda.

Este método de ejecución evita situaciones como: analizar un segmento, querer mapear el siguiente y tener que volver a iniciar la funcionalidad, introduciendo de nuevo los parámetros y realizando el proceso de nuevo; comparar dos registros, ir y volver de uno a otro y necesitar introducir la fecha de manera manual cada vez que se pretenda cambiar de segmento puede resultar tedioso.

A continuación, se muestra el proceso descrito.

```

if int(mat) <= i + 1:
    mat = (data[int(mat)] - 1)[0]

route_data = database.select_data_fromroutes(date_, mat)
route_data = np.array((route_data[0])[1:])

end = False
while not end:
    print(
        f"""Date: {route_data[0]}
Desde: {route_data[1]}
Hasta: {route_data[2]}
Matrícula: {route_data[3]}
Número de Segmentos: {route_data[4]}
ID origen: {route_data[5]}
ID final: {route_data[6]}
Número de Paradas: {route_data[7]}
Duración Viajes: {route_data[8]}
Duración Paradas: {route_data[9]}
Secuencia Paradas: {route_data[10]}
Kilómetros: {route_data[11]}
Vmax: {route_data[12]}
Vavg: {route_data[13]}
""")

    print("")
    print("Segments to draw: [x] [x-y] [0 = complete route] [exit to Exit]")
    print("\n")
    seg = input("Select segments to draw: ")
    print('-----')
    system('cls' if name == 'nt' else 'clear')

```

Ilustración 144 visualizarion_func_3.

Cada vez que se repite el bucle se imprime el registro de la ruta para recordar el número de segmentos, la secuencia, las horas de viaje, etc.

La selección de los segmentos viene dada de tres formas diferentes: número único, 0 o secuencia mediante dos números separados por un guion.

La introducción de la selección se guarda en la variable *seg*.

La variable booleana que condiciona la repetición del bucle se define para *end*.

Una vez marcada la selección de los segmentos a mapear se procede a limpiar la pantalla.

Cabe destacar por qué se selecciona el primer elemento de *route_data* para redefinirse, puesto que, en teoría consta de una tupla que comprende los parámetros de la ruta, lo cual es falso. La extracción que resulta de la tabla *routes* comprende una lista de un elemento que comprende una tupla de los parámetros de ruta, por lo que sí, directamente, seleccionamos algún elemento distinto del primero en *route_data* pretendiendo obtener un parámetro de ruta emergirá el error de *list index out of range*.

La anotación siguiente [1:] indica la selección desde el segundo elemento de la lista, hasta el final. Esta acción se realiza para suprimir de la tupla el parámetro de *id*.

Por lo que queda definida la variable de parámetros de ruta como *array*, *route_data*, para la cual se acude al imprimir en pantalla el registro de ruta.

A continuación, se procede a partir de la selección del segmento. La ilustración mostrada comprende de una abreviación del algoritmo para comprender de manera más clara su estructura.

```
if len(seg) == 1 or len(seg) == 2:  
    if int(seg) <= route_data[4]:  
        if seg == "0":...  
  
    else:...  
  
elif seg != 'exit':...  
  
elif seg == 'exit':  
    end = True
```

Ilustración 145 visualization_func_4.

El primer condicional cuestiona la longitud de la variable *seg*, si mide 1 o 2, se descarta la posibilidad de secuencia, por lo que se procede a cuestionar si es un segmento único o 0 para la ruta completa.

Previo a esa cuestión, se cuestiona si es una introducción válida, puesto que ante una introducción numérica más grande que la cantidad de segmentos, emerge el error de *list index out of range*.

Para el segundo condicional, cuya longitud es más de 2, se cuestiona si comprende de la introducción de *exit* para salir del bucle, con el cual, ante su cumplimiento, se cambia la variable booleana y no se cumple la condición para repetir el bucle.

Las tres abreviaciones que se muestran en la ilustración comprenden el proceso por el cual se realiza el mapeo de cada una de las opciones ofrecidas en la anterior pantalla.

A continuación, se describen los procesos que suceden en cada una de las opciones ofrecidas. Cada algoritmo se distingue por tres partes: extracción de segmentos, impresión en pantalla y mapeado.

Comenzando por el segmento único. Se cumple el condicional de longitud 1 o 2 y posteriormente no se cumple que sea igual a 0.

La extracción de segmento se realiza de manera simple:

La variable *seg* se convierte en *int* y actúa en nombre del parámetro *NumSeg*. Se procede a extraer, mediante *select_data_forsegments*, los registros del segmento. Siguiendo el mismo esquema realizado previamente para la extracción de la ruta y eliminación del primer elemento de la tupla, el *id*.

```

        else:
            seg = int(seg)
            seg_data = database.select_data_fromsegments(date_, mat, seg)
            seg_data = np.array((seg_data[0])[1:])
            print(
                f"""Dateseg: {seg_data[0]}
Desde: {seg_data[1]}
Hasta: {seg_data[2]}
Número de Segmento en Ruta: {seg_data[3]}
ID origen: {seg_data[4]}
ID final: {seg_data[5]}
Matrícula: {seg_data[6]}
Special Point Origen: {seg_data[7]}
Special Point Final: {seg_data[8]}
Kilómetros: {seg_data[9]}
Vmax: {seg_data[10]}
Vavg: {seg_data[11]}
Observaciones: {seg_data[12]}"""
            )
    """

```

Ilustración 146 visualization_func_5.

Esta parte del algoritmo comprende la extracción y la impresión en pantalla de los registros de los segmentos, a continuación, se procede con la parte del mapeado de cada registro.

```

idorg = seg_data[4]
idend = seg_data[5]

ids = database.select_data_frommovements(idorg, idend, mat)
ids = np.array(ids)

i = int(len(ids) / 2)

gmap = gmplot.GoogleMapPlotter((ids[i])[0], (ids[i])[1], 11)
for i in sp:
    gmap.scatter([i[0]], [i[1]], 'blue', label=i[2].replace(" ", ""), size=150)
    gmap.scatter([i[0]], [i[1]], 'blue', size=250, marker=False)
j = 0
for point in ids:
    gmap.scatter([point[0]], [point[1]], color[seg - 1], label=f'{j}', size=50)
    j += 1
gmap.plot(ids[:, 0], ids[:, 1], f'{color[seg - 1]}', label=f'{seg}', size=50)
gmap.draw(f'{path}/map_.html')

webbrowser.open_new_tab(f'{path}/map_.html')

```

Ilustración 147 visualization_func_6.

El primer paso es la definición de las variables que guardan los *ids* del punto inicial del segmento y el punto final.

Las variables se utilizan en el argumento de la función *select_data_frommovements*, el cual, a partir de un *id* inicial y un *id* final seleccionaba los registros, esta vez las posiciones únicamente, de todos los *ids* intermedios que coincidían con la matrícula en la tabla *movements*.

La extracción se guarda en la variable de *ids*.

La variable *i*, esta vez no es una variable iterativa, se define como el valor de la mitad de la lista, en número entero. El propósito de esta variable, es que el elemento número *i* de la lista sea un valor que se encuentre en la mitad. De esta manera, *ids[i]* el punto que comprende la mitad del segmento, este punto se usa para centrar el mapa generado.

Se genera un mapa mediante *gmplot.GoogleMapPlotter* centrado en *ids[i]* con un *zoom* de 11.

El primer bucle *for* se realiza para colocar los SPs en el mapa. *label* consta de la etiqueta del punto, consta del nombre del SP. la colocación se realiza en dos pasos debido a que con uno se coloca el “Pinpoint” y con el otro se coloca el área envolvente al SP. El área consta de un círculo de 250 metros de diámetro, lo que consta de algo menos que el *eps* comúnmente usado. Se eliminan los espacios para mayor compactación en el *label*.

A continuación, se colocan los puntos del segmento. La variable *j* se define para el *label* de cada punto. Cada punto porta una etiqueta que indica el número de registro que ocupa en el segmento.

Cabe destacar que la definición del color se realiza en función de la variable que indica el número de segmento que ocupa en la ruta.

Posteriormente, mediante *plot* se unen los puntos, se define el mismo color para la unión y como etiqueta, la variable *seg*, indicando el número de segmento.

Como último paso, se crea el archivo .html en la ruta especificada por *path* mediante *draw* y por último se abre el archivo mediante *open_new_tab*.

Acabamos de mostrar el caso más sencillo de las tres opciones de definición de segmentos, utilizado a modo de ejemplo para explicar los procesos básicos.

A continuación, se describe el proceso para mapear la ruta completa mediante la introducción de 0 en la selección de segmentos a dibujar.

La nomenclatura de elección 0 para mapear la ruta completa se hace debido a que no existe el segmento 0 y ahorraremos complejidad al código para detectar la introducción de una variable que indique la selección de ruta completa.

El método a realizar consta de un bucle *for* que recorra los segmentos que existen en la ruta y realice un proceso, con ciertas particularidades, semejante al de mapear un solo segmento.

```

if seg == "0":

    segs = []
    for i in range(1, route_data[4] + 1):
        seg_data = database.select_data_fromsegments(date_, mat, i)
        seg_data = np.array((seg_data[0])[1:])
        print(
            f"""Dateseg: {seg_data[0]}
Desde: {seg_data[1]}
Hasta: {seg_data[2]}
Número de Segmento en Ruta: {seg_data[3]}
ID origen: {seg_data[4]}
ID final: {seg_data[5]}
Matrícula: {seg_data[6]}
Special Point Origen: {seg_data[7]}
Special Point Final: {seg_data[8]}
Kilómetros: {seg_data[9]}
Vmax: {seg_data[10]}
Vavg: {seg_data[11]}
Observaciones: {seg_data[12]}"""

        )
        idorg = seg_data[4]
       idend = seg_data[5]

        ids = database.select_data_frommovements(idorg, idend, mat)
        segs.append(ids)

input("\nEnter to continue")

```

Ilustración 148 visualization_func_7.

Se recuerda que el parámetro *route_data[4]* consta del *NumSeg* que indica el número de segmento que ocupa en la ruta. Puesto que no existe el segmento 0, el bucle se define desde 1 hasta el número de segmentos + 1, para conservar el número de segmentos empezando por el primero como 1.

La variable iteradora se introduce como argumento de la función selectora, además de la fecha y la matrícula.

El resto del proceso es el mismo, se elimina el primer parámetro y se redefine como *array*.

Cabe destacar que es necesario crear la lista vacía fuera del bucle, donde se añaden posteriormente los segmentos.

En adición en el bucle, se realiza el proceso de extracción de ubicaciones del conjunto de segmento y se agrega a la lista.

En las opciones de mapeo de segmentos que constan de más de un segmento, previamente a la creación del archivo .html se imprimen en pantalla los registros de los segmentos para la observación por parte del usuario, y mediante una entrada *input*, se da permiso a la herramienta para continuar con el proceso, creando y mostrando el mapa.

La razón por la que esta acción no se da en el mapeo de un segmento concreto es que, al ser un segmento, no se necesita una observación detallada comparativa con los demás segmentos, igualmente se obtiene el segmento en pantalla de la misma forma.

A continuación, el método de mapeo.

```
input("\nEnter to continue")

i = int(len(ids) / 2)
gmap = gmplot.GoogleMapPlotter((ids[i])[0], (ids[i][1]), 11)

i = 0
for ids in segs:
    ids = np.array(ids)
    j = 0
    for point in ids:
        gmap.scatter([point[0]], [point[1]], color[i], label=f'{i + 1}.{j}', size=50)
        j += 1
    gmap.plot(ids[:, 0], ids[:, 1], color[i], size=50)
    i += 1
    if i == len(color):
        i = 0

for i in sp:
    gmap.scatter([i[0]], [i[1]], 'blue', label=i[2].replace(" ", ""), size=50)
    gmap.scatter([i[0]], [i[1]], 'blue', size=150, marker=False)

gmap.draw(f"{path}/map_.html")

webbrowser.open_new_tab(f'{path}/map_.html')
```

Ilustración 149 visualization_func_8.

La razón por la que no se incluye el mapeo dentro de los puntos de segmento dentro del primer bucle, puesto que comprende del mismo rango y variables, es la decisión deliberada de separar el algoritmo por partes que se dedican al mismo propósito.

Es necesario definir el mapa previamente a la agregación de puntos, por lo que, si se decidiera introducir en el bucle de impresión en pantalla la agregación de los puntos al mapa, ciertamente simplificaría el código al ahorrarse realizar un bucle, pero dificultaría su comprensión al estar separadas las acciones con el mismo, o semejante, propósito.

La definición del mapa y la agregación de los SPs se realizan de manera idéntica, la particularidad se encuentra en la agregación de puntos de segmento.

Puesto que la ruta completa consta de una lista de listas de puntos a agregar al mapa, es necesaria una doble iteración para recorrer la variable, además de ello, se definen los contadores *i* y *j* para los *label* de cada punto.

i comprende el segmento que se está mapeando, *j* comprende la posición que ocupa el registro en el segmento. De esta manera, cada “Pinpoint” poseerá dos parámetros indicando el segmento

y el número de registro, de esta manera, en el visionado se puede observar en qué dirección avanza el vehículo.

Puesto que el contador i define el número de segmento y el color del segmento viene dado en función del número de segmento, es necesario resetear su valor una vez llega al valor de la longitud de la lista de colores.

Se unen los puntos de cada segmento al terminar de mapearse y empezar con el siguiente. La siguiente vuelta del bucle externo, comprende un nuevo segmento, por lo que la variable j se resetea.

Por último, se crea el archivo .html mediante *draw*, y se abre mediante *open_new_tab*.

A continuación, se procede a explicar las particularidades de la última de las opciones a ofrecer para el mapeo de segmentos.

```
elif seg != 'exit':
    ch = ["1", "2", "3", "4", "5", "6", "7", "8", "9", "0", "-"]
    range_ = ""
    for i in seg:
        if i in ch:
            range_ += i

    range_ = range_.split("-")
    if len(range_) > 1:
        if int(range_[1]) <= route_data[4]:
            segs = []
            for i in range(int(range_[0]), int(range_[1]) + 1):
                seg_data = database.select_data_fromsegments(date_, mat, i)
                seg_data = np.array((seg_data[0])[1:])
                print(
                    f"""Dateseg: {seg_data[0]}

Desde: {seg_data[1]}
Hasta: {seg_data[2]}
Número de Segmento en Ruta: {seg_data[3]}
ID origen: {seg_data[4]}
ID final: {seg_data[5]}
Matricula: {seg_data[6]}
Special Point Origen: {seg_data[7]}
Special Point Final: {seg_data[8]}
Kilómetros: {seg_data[9]}
Vmáx: {seg_data[10]}
Vavg: {seg_data[11]}
Observaciones: {seg_data[12]}

""")
            idorg = seg_data[4]
            idend = seg_data[5]

            ids = database.select_data_frommovements(idorg, idend, mat)
            segs.append(ids)
```

Ilustración 150 visualization_func_8.

La ilustración muestra la parte extractora de segmentos, de parámetros de los segmentos y la que imprime en pantalla.

El proceso es idéntico salvo por la particularidad de la definición del bucle.

Ante la entrada del tipo $[x-y]$, siendo x e y valores numéricos, es necesario la conversión en *int* de los dos valores para acotar la selección de segmentos.

Puesto que únicamente los caracteres numéricos se pueden convertir a *integers*, es necesario salvar el error que emergería al intentar convertir a *int* un carácter alfabético.

Para ello se define la lista de caracteres permitidos en la introducción del *input*, *ch*, y se crea un bucle para descartar aquellos caracteres que no pertenecen a la lista. Se define la variable *range_* como *str* vacío y se redefine dentro del bucle con la agregación de caracteres permitidos.

Posteriormente la variable se parte en dos elementos, la partición se realiza para los elementos de antes y después del guion -, mediante *split*. El resultado ante una entrada correcta es la tupla *range_* con dos elementos, por lo que es necesario cuestionar si la entrada es correcta, además de que alguna de las entradas no comprenda un valor por encima del máximo de segmentos.

La definición del bucle se hace para el rango: desde la primera entrada hasta la segunda + 1, para conservar el número de segmentos.

El resto del proceso está descrito en el apartado anterior.

A continuación se describen las particularidades del proceso de mapeo para la opción a tratar.

```

i = int(len(ids) / 2)
gmap = gmplot.GoogleMapPlotter((ids[i])[0], (ids[i][1]), 11)

i = int(range_[0]) - 1
for ids in segs:
    ids = np.array(ids)
    j = 0
    for point in ids:
        gmap.scatter([point[0]], [point[1]], color[i], label=f'{i + 1}.{j}', size=50)
        j += 1
    gmap.plot(ids[:, 0], ids[:, 1], color[i], size=50)
    i += 1
    if i == len(color):
        i = int(range_[0]) - 1

for i in sp:
    gmap.scatter([i[0]], [i[1]], 'blue', label=i[2].replace(" ", ""), size=150)
    gmap.scatter([i[0]], [i[1]], 'blue', size=250, marker=False)

gmap.draw(f"{path}/map_.html")

webbrowser.open_new_tab(f'{path}/map_.html')

```

Ilustración 151 visualization_func_9.

La única particularidad del proceso es que el contador se define y se resetea como el valor de la primera entrada – 1, para la conservación de segmentos.

En última instancia, se obtiene el condicional ante el no cumplimiento de encontrar datos para el día seleccionado.

Este caso, es posible, bien para un día que no se hayan dado rutas, o bien para un día que no se haya realizado el proceso de trazado.

```

else:
    print(f"No data for {date_}")

```

Ilustración 152 visualization_func_10.

A continuación se dota con algunos ejemplos la ejecución de la funcionalidad.

```

Day 2021-11-12.

1 4317JCD
2 NXK840
3 4938LKX

Select mat: J

@Date: 2021-11-12
Desde: 0:47:17
Hasta: 12:05:23
Matricula: 4938LKX
Número de Segmentos: 4
ID origen: 3531
ID final: 3534
Número de Paradas: 3
Duración Viales: 9:02:54
Duración Paradas: 2:15:12
Secuencia Paradas: C-3/C-2/NotSP./NotSP./C-2
Kilómetros: 250.429
Vmax: 90.0
Vavg: 30.335

Segments to draw: [x] [x-y] [0 = complete route] [exit to Exit]

Select segments to draw: 0

```

Ilustración 153 visualization_process_1.

Se escoge a analizar la matrícula 4938LKX para el día 12/11/2021. La primera ilustración muestra la selección de matrícula, la segunda los parámetros de ruta y la selección de segmentos. Constando una ruta de cuatro segmentos. Se selecciona 0.

| | |
|--|---|
| <pre> @Dateseg: 2021-11-12 Desde: 0:47:17 Hasta: 5:45:12 Número de Segmento en Ruta: 1 ID origen: 893540 ID final: 893891 Matricula: 4938LKX Special Point Origen: C - 3 [42.80719, -1.69828] Special Point Final: C - 2 [42.74537, -1.63897] Kilómetros: 33.83742432498102 Vmax: 86.0 Vavg: 7.816666666666666 Observaciones: </pre> | <pre> Dateseg: 2021-11-12 Desde: 5:55:09 Hasta: 8:01:45 Número de Segmento en Ruta: 2 ID origen: 893910 ID final: 894086 Matricula: 4938LKX Special Point Origen: C - 2 [42.74504, -1.63784] Special Point Final: NotSP. Special Point not defined Kilómetros: 109.19630209567956 Vmax: 90.0 Vavg: 50.03846153846154 Observaciones: End of segment does not match with Spec </pre> |
| <pre> Dateseg: 2021-11-12 Desde: 8:56:46 Hasta: 9:07:07 Número de Segmento en Ruta: 3 ID origen: 894194 ID final: 894215 Matricula: 4938LKX Special Point Origen: NotSP. Special Point not defined Special Point Final: NotSP. Special Point not defined Kilómetros: 1.333568354446439 Vmax: 23.0 Vavg: 9.666666666666666 Observaciones: Origin of segment does not match with Sp </pre> | <pre> Dateseg: 2021-11-12 Desde: 10:17:21 Hasta: 12:05:23 Número de Segmento en Ruta: 4 ID origen: 894340 ID final: 894549 Matricula: 4938LKX Special Point Origen: NotSP. Special Point not defined [42.11064, -1.63875] Special Point Final: C - 2 [42.74548, -1.63875] Kilómetros: 106.06215581753258 Vmax: 90.0 Vavg: 53.81818181818182 Observaciones: C - 10 (0:00:00) Origin of segment does not match with Spec Enter to continue </pre> |

Ilustración 154 visualization_process_2.

Se imprimen en pantalla los registros de los cuatro segmentos al seleccionar *0* para ruta completa. Se da permiso para proceder pulsando *ENTER* y se abre el navegador mostrando la ruta completa.

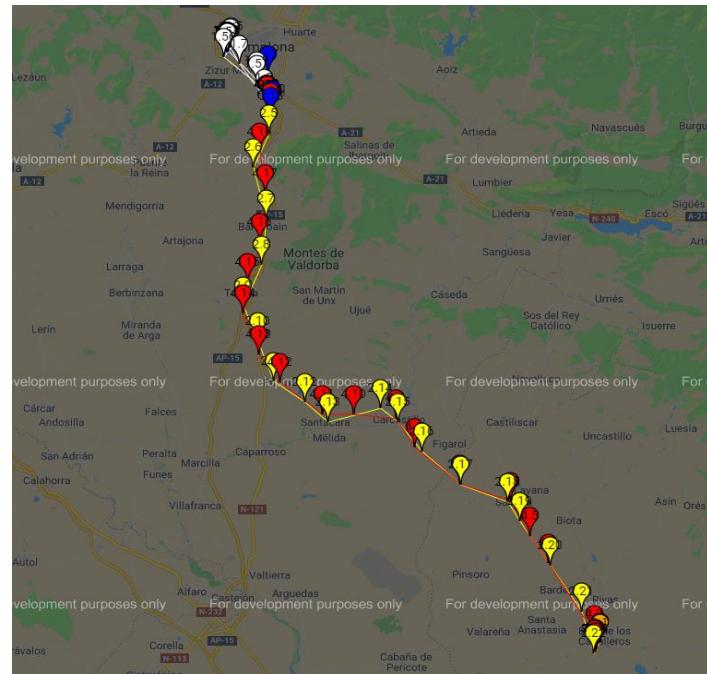


Ilustración 155 visualization_process_3.

En la interfaz, se vuelve a mostrar la selección de segmentos. Observamos que el segmento 4, posee una anomalía pasa cerca de *C-10* pero no se queda en él, por lo que se procede a elegir el segmento 4 como segmento único a representar.

```
Date: 2021-11-12
Desde: 0:47:17
Hasta: 12:05:23
Matrícula: 4938LKX
Número de Segmentos: 4
ID origen: 3531
ID final: 3534
Número de Paradas: 3
Duración Viajes: 9:02:54
Duración Paradas: 2:15:12
Secuencia Paradas: C-3/C-2/NotSP./NotSP./C-2
Kilómetros: 250.429
Vmax: 90.0
Vavg: 30.335

Segments to draw: [x] [x-y] [0 = complete route] [exit to Exit]

Select segments to draw: 4
```

Ilustración 156 visualization_process_4.

Se muestra el registro del segmento 4 en la interfaz, pero no se incluye otra vez puesto que se repetiría con la ilustración tercera del proceso.

Se muestra el segmento, con zoom en el SP *C-10*.

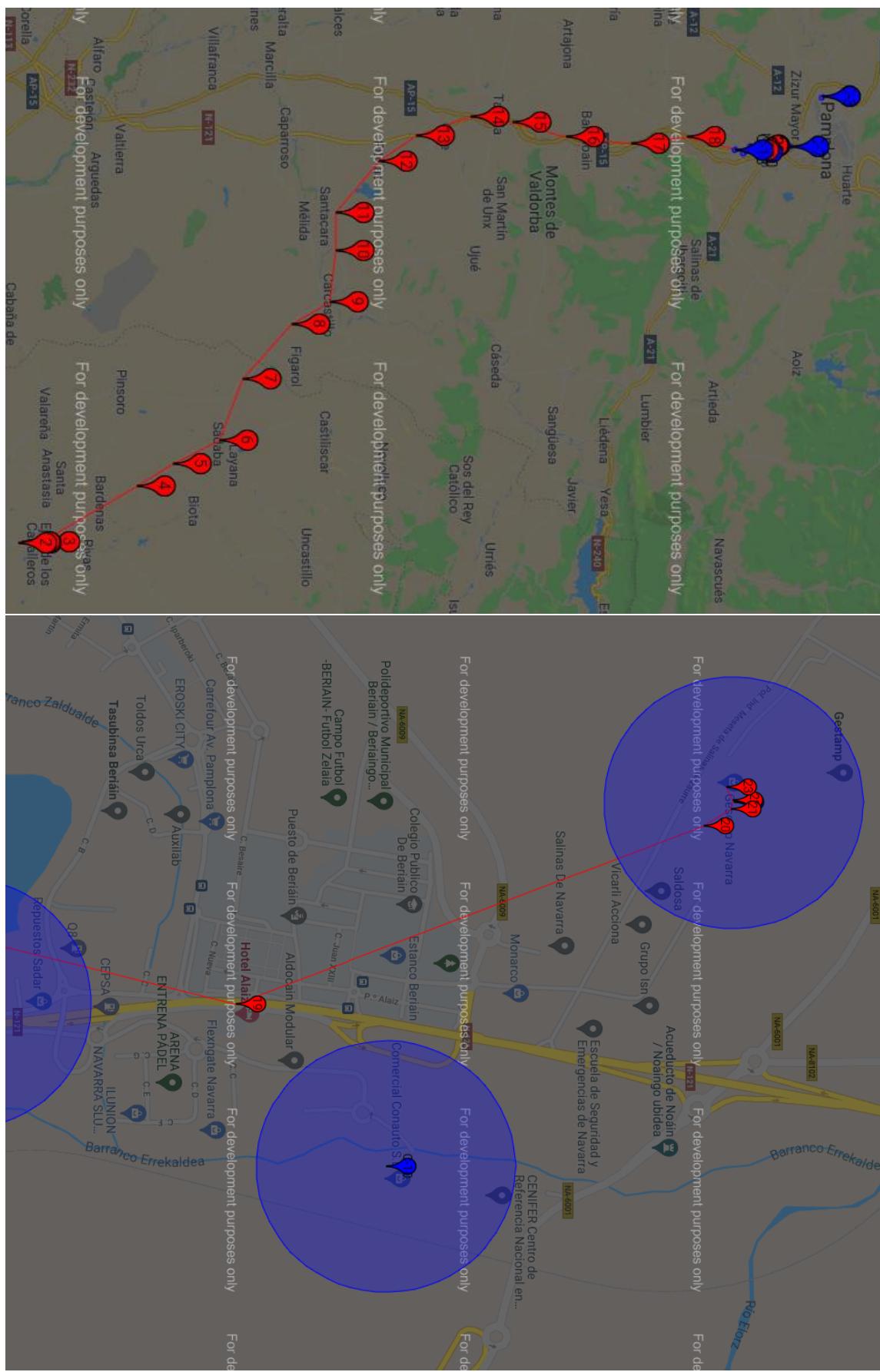


Ilustración 157 visualizacion_process_5.

Se decide analizar la parte de ida de la ruta, por lo que se deciden representar los segmentos 1, 2 y 3 de la ruta.

A continuación, se muestra el proceso realizado por el usuario, suprimiendo la observación de los registros de los segmentos ya que ya se ha realizado para la ruta completa.

Se realiza zoom en los segmentos 1 en la salida de C-3 y 3 en el destino.

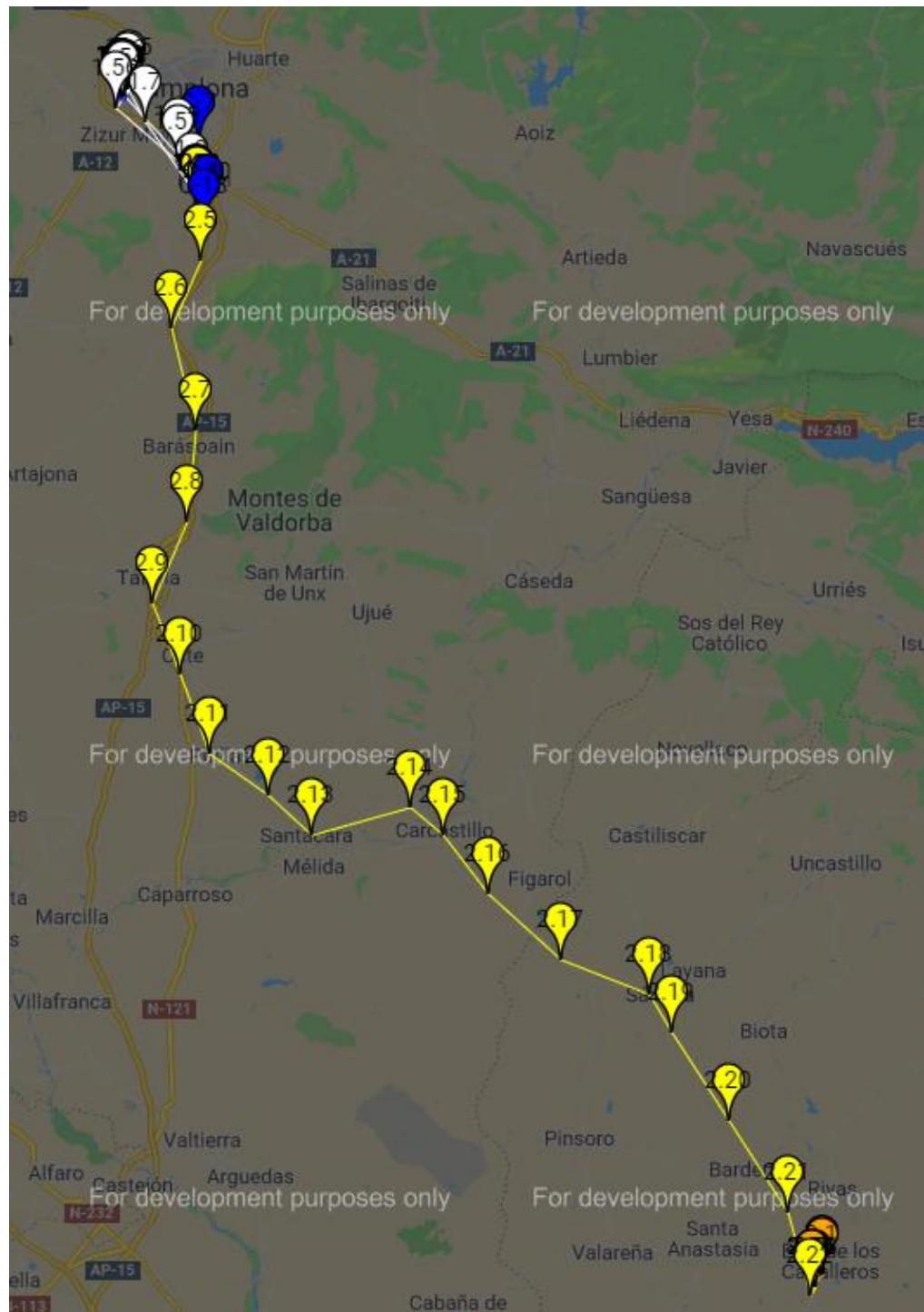


Ilustración 158 visualization_process_6.

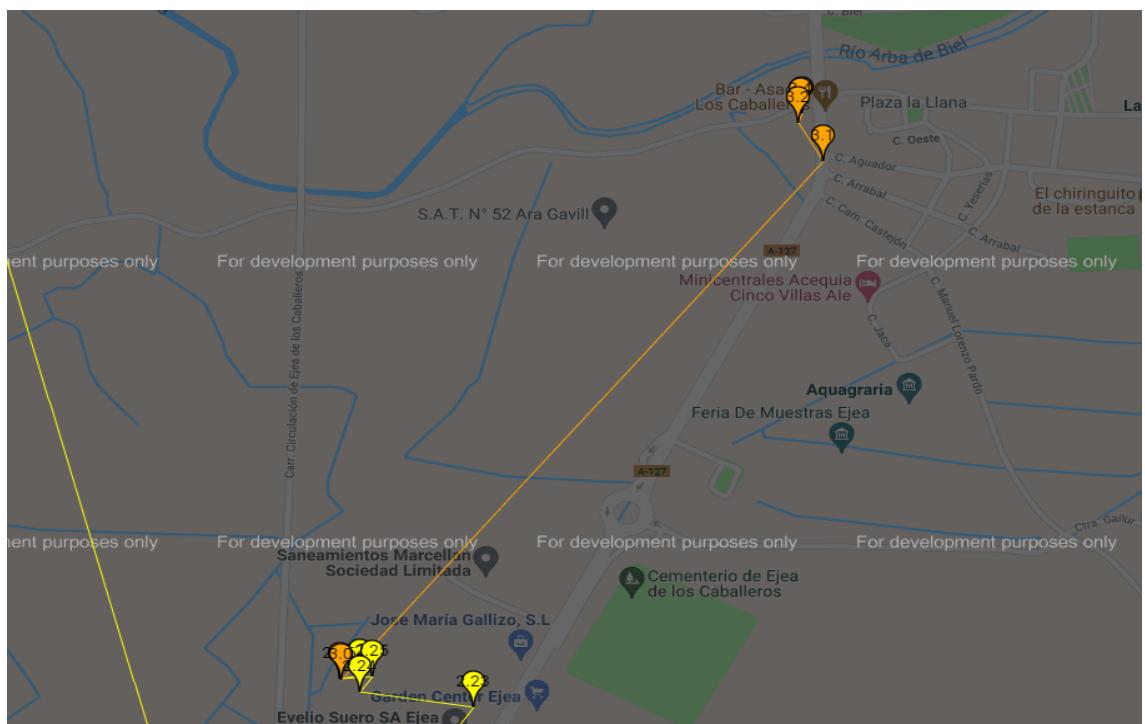


Ilustración 159 visualization_process_7.

Con esto se cierra el capítulo de la funcionalidad de *Mapping*.

A continuación, se procede con la funcionalidad de *Scatter Search*.

5.3. Scatter Search.

5.3.1. Introducción.

La funcionalidad completa de *Scatter Search* se encuentra contenida en el archivo de *scatter.py*.

La funcionalidad comprende el graficado minimalista de los puntos de los registros comprendidos entre dos fechas, junto a los SPs.

Las funcionalidades contenidas son: *Scatter with day selection* y *Scatter last month*; de las cuales comprenden las funciones *scatter_dayselect* y *scatter_last_month*.

5.3.2. Cuerpo.

Como primer objeto del cuerpo del archivo se obtienen las importaciones.

```
from database import DataBase
from matplotlib import pyplot as plt
import numpy as np
from datetime import date, timedelta, datetime
```

Ilustración 160 scattering_imports.

Se importa función *pyplot* de la librería *matplotlib* como *plt* para realizar el graficado. Se importa la clase *Database*, la librería *numpy* como *np* para la formación de *arrays*, se importan las funciones *datetime*, *date* y *timedelta* de la librería *datetime* para el manejo de funciones temporales.

La denominación de la clase se realiza dentro de las funciones.

Como primera función en el archivo, se obtiene *scatter_last_month*. El propósito de la función es graficar todos los registros dados en el último mes.

Para ello extrae los registros de la tabla *movements* a partir de la función *select_last_month*.

Mediante *select_sp* crea una lista con los SPs de la tabla *specialsites*.

Posteriormente realiza el graficado.

```

def scatter_last_month():
    database = DataBase()

    data = np.array(database.select_last_month())

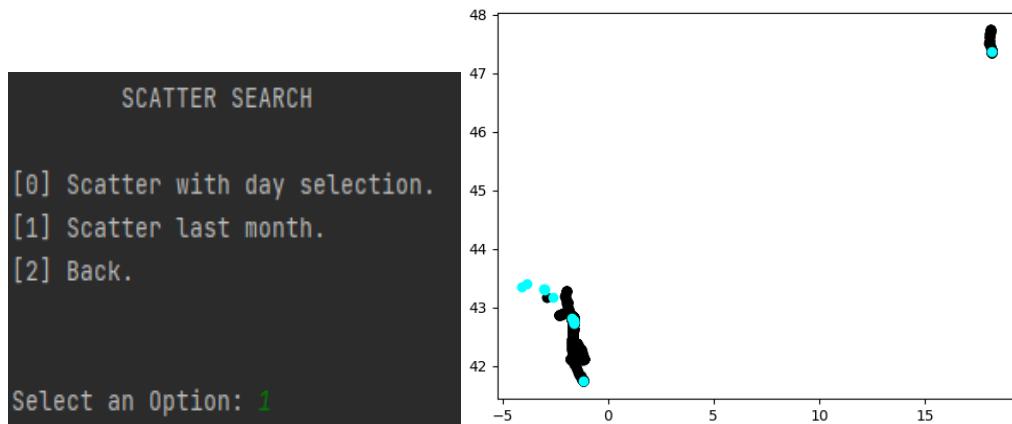
    sp = np.array(database.select_sp())

    plt.gca().set_facecolor('white')
    plt.scatter(data[:, 1], data[:, 0], c='black')
    plt.scatter(sp[:, 1], sp[:, 0], c='aqua')
    plt.show()

```

Ilustración 161 scatter_last_month_func.

A continuación, se procede a la ejecución de la funcionalidad *Scatter last month*.



Como se puede apreciar, se realiza un visionado minimalista con los registros en negro y los SPs en color “aqua”.

A continuación, se procede con la funcionalidad *Scatter with day selection*.

La función *scatter_dayselect* realiza el mismo proceso con la particularidad de selección de las fechas para definir el intervalo de muestra.

La introducción de la primera fecha se realiza de la misma manera que el resto de funciones. La segunda fecha se puede introducir mediante el parámetro de fecha: *[dd/mm/yyyy]*, o bien mediante la cantidad de días previos o posteriores a la primera fecha.

```

def scatter_dayselect():
    database = DataBase()

    date_ = str(input("""Insert initial date [dd/mm/yyyy]:      Or      Insert amount of backdays (from now):
"""))
    if date_:
        if len(date_) == len('dd/mm/yyyy'):
            date_ = datetime.strptime(date_, '%d/%m/%Y')
        elif len(date_) <= 7:
            date_ = date.today() + timedelta(days=-int(date_))

    date_end = str(input("""Insert end date [dd/mm/yyyy]:      Or      Insert amount of (+-) days (from initial date):
"""))
    if date_end:
        if len(date_end) == len('dd/mm/yyyy'):
            date_end = datetime.strptime(date_end, '%d/%m/%Y')
        elif len(date_end) <= 7:
            date_end = date_ + timedelta(days=int(date_end))

```

Ilustración 162 scatter_dayselect_func_1.

Se puede apreciar que las fechas se convierten en parámetros temporales mediante el mismo método que se emplea en el resto de funciones, la única particularidad es que la segunda fecha carece del “signo menos” en la conversión.

Esto es debido a que ya no se realiza para una cantidad de días previos, sino que el usuario decide si la introducción es para días previos o para días posteriores.

Este hecho genera el problema de que las fechas se deben ordenar de más antigua a menos antigua, ya que la selección de registros se hace a partir de un intervalo definido para la introducción de dos fechas, es decir: se seleccionan los registros que posean *fecha_upos* mayor que la primera fecha y menor que la segunda fecha.

Ante una introducción de una segunda fecha menor que la primera, será imposible que se dé ningún registro, por lo que es necesario ordenar las fechas una vez definidas por el usuario.

```

if date_end < date_:
    date_1 = date_end
    date_end = date_
    date_ = date_1

data = np.array(database.select_dayselect(str(date_), str(date_end)))

sp = np.array(database.select_sp())

if len(data) > 0:
    plt.gca().set_facecolor('white')
    plt.scatter(data[:, 1], data[:, 0], c='black')
    plt.scatter(sp[:, 1], sp[:, 0], c='aqua')
    plt.show()

else:
    print(f"No data for {date_}")

```

Ilustración 163 scatter_dayselect_2.

Como se puede apreciar en la ilustración, se abre un condicional para el cual, si la fecha final es menor que la fecha inicial, estas cambian de posición para su introducción.

La extracción de registros se realiza mediante *select_dayselect*.

El resto del algoritmo comprende del graficado, con la condición de que existan registros para el intervalo elegido, en caso contrario se indica que no existen registros para la primera fecha seleccionada.

A continuación, unos ejemplos de ejecución para esta funcionalidad.

Esta funcionalidad permite comprobar de manera rápida, sin hacer el procedimiento de *Routing*, si existen rutas para el día de hoy o cualquier día.

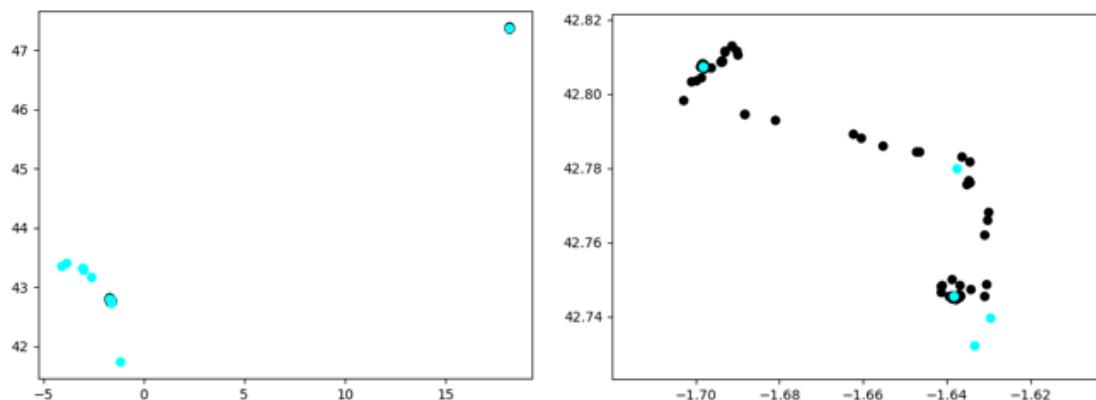


Ilustración 164 scatter_1.

Las imágenes comprenden los movimientos del día 08/01/2022.

Además de ello, se puede hacer una visualización de todos los movimientos de la tabla de *movements*.

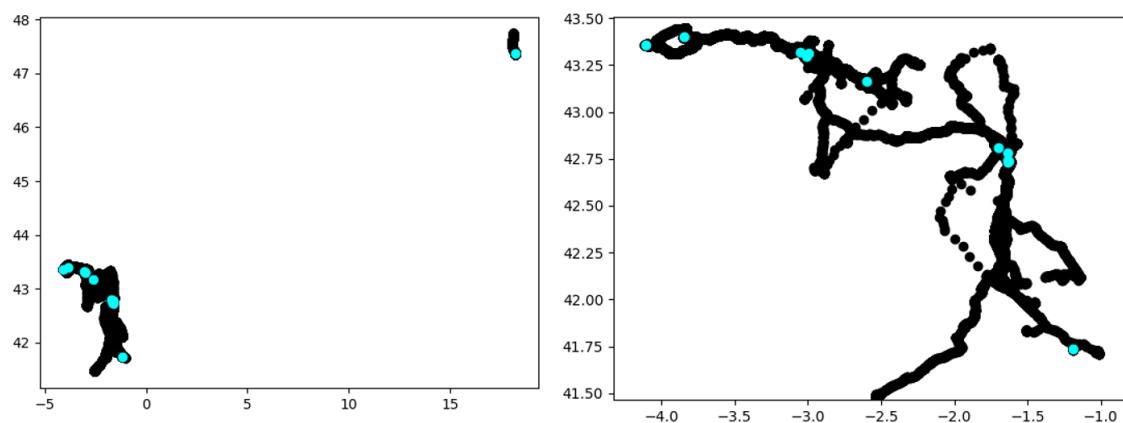


Ilustración 165 scatter_2.

La funcionalidad se utiliza para comprobar los movimientos de manera ágil para un intervalo de fechas definidas.

A continuación, se procede con la explicación de *Search in DDBB*.

5.4. Search in Database.

5.4.1. Introducción.

La funcionalidad *Search in DDBB* se contiene en el archivo de *search.py*.

La función *search* se basa en la función *show* del archivo de *database.py*.

La funcionalidad comprende un manejo de la BBDD mediante la introducción de la sintaxis de la base de manera manual.

5.4.2. Cuerpo.

Esta función es usada en el desarrollo de la herramienta para comprobar los registros de las tablas, así como analizar sus descripciones.

```
from database import DataBase

def search():
    print("Type 'exit' to Exit")
    end = True
    while end:
        end = DataBase().show_()
```

Ilustración 166 *search_func.*

El archivo consta de la función que se muestra en la ilustración. Comprende de un bucle *while* que únicamente no se cumple la condición si se introduce *exit* en la consulta. Para realizar consultas llama a la función *show*.

```
def show_(self):
    consulta = input("Introduce consulta: ")
    end = True
    if consulta != "":
        if consulta != "exit":
            self.cursor.execute(consulta)
            filas = self.cursor.fetchall()
            for fila in filas:
                print(fila)
            self.connection.commit()
        else:
            end = False

    return end
```

Ilustración 167 *show_func.*

Únicamente si la consulta es *exit*, se realiza el cambio en la booleana, en caso contrario, se ejecuta el cursor, se selecciona todo lo que realiza la consulta mediante *fetchall()* y se imprime en pantalla con un bucle *for* que recorre las filas de la variable donde se han guardado los resultados de las consultas.

A continuación, unos ejemplos en la ejecución de *Search in Database*.

```
Type 'exit' to Exit
Introduce consulta: select * from specialsites
(286, Decimal('-2.60155'), Decimal('43.16356'), 'C - 1', 'Lugar Barr
(287, Decimal('-1.63837'), Decimal('42.74561'), 'C - 2', 'Carr Salin
(288, Decimal('-1.69823'), Decimal('42.80752'), 'C - 3', 'Poligono L
(289, Decimal('-2.99310'), Decimal('43.31145'), 'C - 4', 'Plaza Urbi
(290, Decimal('-4.10954'), Decimal('43.35511'), 'C - 5', 'Barrio la
(291, Decimal('-3.04849'), Decimal('43.31990'), 'C - 6', 'Antonio Al
(292, Decimal('-1.63761'), Decimal('42.77988'), 'C - 7', 'NC ID 915
(293, Decimal('-3.01088'), Decimal('43.29566'), 'C - 8', 'Poligono I
(294, Decimal('-3.84134'), Decimal('43.39839'), 'C - 9', 'Poligono G
(295, Decimal('-1.62952'), Decimal('42.73959'), 'C - 10', 'Poligono
(296, Decimal('-1.18935'), Decimal('41.73552'), 'C - 11', 'Calle Gra
(297, Decimal('18.18719'), Decimal('47.36624'), 'C - 12', 'Mór Veleg
(298, Decimal('-1.63348'), Decimal('42.73224'), 'C - 13', 'Pol Ind M
Introduce consulta:
```

Ilustración 168 search_1.

Se aprecian en la ilustración los registros que comprenden SPs.

```
Introduce consulta: select * from routes
(383, datetime.date(2021, 10, 7), datetime.timedelta(seconds=106), datetime.timedelta(seconds=86230),
(384, datetime.date(2021, 10, 7), datetime.timedelta(seconds=944), datetime.timedelta(seconds=218), '
(385, datetime.date(2021, 10, 7), datetime.timedelta(seconds=30573), datetime.timedelta(seconds=65623
(386, datetime.date(2021, 10, 7), datetime.timedelta(seconds=21284), datetime.timedelta(seconds=49885
(391, datetime.date(2021, 8, 6), datetime.timedelta(seconds=29219), datetime.timedelta(seconds=41490)
(392, datetime.date(2021, 8, 6), datetime.timedelta(seconds=25186), datetime.timedelta(seconds=49541)
(393, datetime.date(2021, 9, 3), datetime.timedelta(seconds=3955), datetime.timedelta(seconds=80664),
(394, datetime.date(2021, 9, 3), datetime.timedelta(seconds=183), datetime.timedelta(seconds=45573),
(395, datetime.date(2021, 9, 3), datetime.timedelta(seconds=46308), datetime.timedelta(seconds=77473)
(396, datetime.date(2021, 9, 3), datetime.timedelta(seconds=25482), datetime.timedelta(seconds=68073)
(397, datetime.date(2021, 9, 3), datetime.timedelta(seconds=25135), datetime.timedelta(seconds=48101)
```

Ilustración 169 search_2.

Se aprecia en la ilustración los registros que comprenden las rutas.

La funcionalidad se utiliza en la búsqueda de registros específicos o la eliminación de registros específicos, pero la sintaxis ha de ser precisa, es sencillo equivocarse y que la herramienta se cierre por un error.

A continuación, se describe la funcionalidad de *Reset DDBB*.

5.5. Reset DDBB.

5.5.1. Introducción.

La funcionalidad de reseteo comprende el vaciado de la tabla a elegir, las funciones están contenidas en el archivo de database.py, pero se llaman desde reset.py.

La razón por la que se realiza este desvío en el algoritmo es para poseer una vía para comprobar de forma rápida si los reseteos están activos o, por el contrario, especialmente en el desarrollo de la aplicación, permanecen desactivados, convirtiendo la llamada a las funciones en str.

De esta manera, se podía generar introducciones aleatorias en la herramienta con todas las funcionalidades operativas para detectar posibles errores, y no había posibilidad de resetear las tablas accidentalmente.

El reseteo conlleva la pérdida de datos masiva, por ello hace falta estar seguro de lo que se está haciendo. El archivo main.py pregunta al usuario si está seguro de resetear una tabla antes de ejecutar la acción, únicamente mediante Y se procede al reseteo.

5.5.2. Cuerpo.

```
from database import DataBase

database = DataBase()

def reset_segments():
    database.remove_allseg()

def reset_routes():
    database.remove_allroutes()

def reset_specialsites():
    database.remove_allsp()
```

Ilustración 170 resets_file.

El archivo comprende las llamadas a las funciones de reseteo explicadas en el capítulo 1.

Con este capítulo se cierra la Parte 1 del Documento.

PARTE 2

**ANÁLISIS ECONÓMICO Y LINEAS
FUTURAS.**

PROGRAMACIÓN DEL PROYECTO

Este capítulo pretende mostrar la programación del proyecto, para ello se muestra un gráfico de Gantt, realizado con GanttProject, con el que se aprecia la consecución de cada una de las fases de la elaboración de la herramienta de manera más visual.

A continuación, se muestran en la ilustración las fases principales de realización del proyecto.

| Nombre | Fecha de inicio | Fecha de fin |
|--|-----------------|--------------|
| • PRIMERA REUNION DE CONTEXTUALIZACIÓN | 2/12/20 | 2/12/20 |
| • PUNTO DE PARTIDA Y CONTEXTO | 1/12/20 | 9/12/20 |
| • MEJORA DE EQUIPO INFORMÁTICO | 10/12/20 | 24/12/20 |
| • DESARROLLO OBJETIVO 1 | 25/12/20 | 11/2/21 |
| • REVISIONES Y DESARROLLO OBJ 1 | 12/2/21 | 2/8/21 |
| • META 1 | 3/8/21 | 3/8/21 |
| • SEGUNDA REUNION CONTEXT OBJ 2 | 7/6/21 | 7/6/21 |
| • DESARROLLO OBJETIVO 2 | 8/6/21 | 21/9/21 |
| • REVISIONES Y DESARROLLO OBJ 2 | 22/9/21 | 25/10/21 |
| • META 2 | 26/10/21 | 26/10/21 |
| • ENTREGAS Y CORRECCIONES | 26/10/21 | 3/12/21 |
| • ENTREGA DEFINITIVA DE CODIGO | 6/12/21 | 6/12/21 |
| • REDACCION MEMORIA | 1/12/21 | 10/1/22 |
| • ENTREGA MEMORIA | 11/1/22 | 11/1/22 |

Ilustración 171 Programación del proyecto.

El proyecto se divide principalmente en la consecución de dos objetivos: el minado de SPs y el trazado de rutas y segmentos, para los cuales es necesario contextualizar lo que se espera de la realización de cada tarea.

Ante las exigencias que requiere la realización de una herramienta que maneje una cantidad ingente de datos es necesario contar con un equipo que sea capaz de soportar tales exigencias, es por ello que se ha tenido que dedicar un periodo de tiempo a renovar el equipo con el que se pretende realizar la herramienta.

El desarrollo de cada uno de los objetivos viene dado por un periodo de desarrollo en solitario y cuando se consigue lo esperado de las tareas se procede a la revisión y corrección del proceso. Estas fases del proyecto se realizan en simultáneo con una continuación del desarrollo puesto que los métodos a corregir, cancelan o anulan otros métodos que quedan obsoletos o es necesario rehacerlos.

Se indican en la programación las fechas en las que se han aprobado los procesos que ejecutan las tareas esperadas, la consecución de las metas.

La última fase del proyecto ha sido la redacción del documento que comprende la memoria.

A continuación, se muestra el diagrama de Gantt para una visualización comparativa de las duraciones de cada fase.

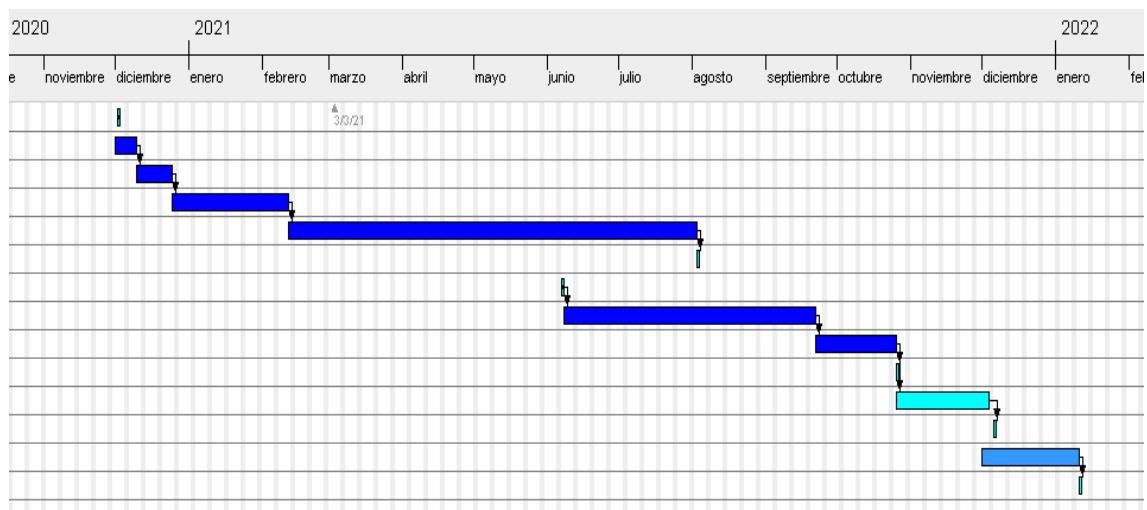


Ilustración 172 Diagrama de Gantt.

Como se puede apreciar, existe un solapamiento entre las fases de desarrollo de los objetivos 1 y 2. Esto es debido a que ante la implementación de nuevos procesos que necesitaban de la ejecución de otros existentes, a veces es necesario modificar los procesos ya creados para obtener la sinergia esperada.

PRESUPUESTACIÓN

Para la presupuestación se deben analizar los recursos que han sido necesarios para la realización del proyecto. Estos son: el análisis de los recursos, objetivos, revisiones de código, etc, ejecutado por el analista; el código de la herramienta, ejecutado por el programador y la documentación del proyecto.

| | Horas (h) | Presupuesto (€/h) | Total (€) | |
|----------------|-----------|-------------------|-----------|-----------|
| Analista | 30 | 55 | 1650 | |
| Programador | 250 | 40 | 10000 | |
| Documentalista | 45 | 30 | 1350 | |
| | | | 13000 | Total (€) |

A esta cantidad se le debe sumar el servicio web, compuesto por el servicio de control de flota y el servicio proporcionado por la página web para las herramientas de visualizado, a lo largo de la realización del proyecto.

| | Meses | Presupuesto (€/mes) | Total (€) |
|--------------|-------|---------------------|-----------|
| Servicio web | 12 | 100 | 1200 |

Lo cual suma una cantidad perteneciente al presupuesto de ejecución material de:

| | |
|--------------|-------|
| Servicio web | 1200 |
| Recursos | 13000 |
| Total (€) | 14200 |

Posteriormente se le debe sumar el Beneficio Industrial y los gastos generales para obtener el presupuesto de Ejecución por Contrata:

| | | | |
|----------------------|-------|----------------------|----------|
| Ejecución Material | 14200 | Ejecución Industrial | 15052 |
| Beneficio Industrial | 6% | Gastos Generales | 13% |
| Total (€) | 15052 | Total (€) | 17008,76 |

Por último, al total se le suma el IVA y nos queda el presupuesto total del proyecto:

| | |
|-----------------------|----------|
| Total Presupuesto (€) | 17008,76 |
| IVA | 0,21 |
| Total Proyecto (€) | 20580,6 |

El total del proyecto es por tanto 20580,6 €.

Para el cálculo del presupuesto no se han tenido en cuenta las mejoras de equipo para la realización de este puesto que cuentan como objeto personal del programador.

LÍNEAS FUTURAS

Como ya se ha mencionado en el capítulo introductorio, la herramienta pretende ser continuada con la implementación de nuevas funcionalidades. La principal aportación del trabajo reside en integrar la semántica de las rutas, con los parámetros del negocio para hacer posible su estudio y análisis.

Entre las posibles funcionalidades a implementar a la herramienta se encuentra la integración de KPIs que se implanten con datos ambientales del entorno, como la lluvia o la niebla, que permita agregar a la herramienta una utilidad para la valoración de seguridad.

Otra de las funcionalidades a implementar es la integración de sistemas automáticos de seguimiento que altere de los posibles desviamientos de los estándares aceptables, con el fin de que la gestión no tenga que supervisar todo y pueda centrarse en la gestión de excepciones.

Es así como la herramienta presenta la capacidad de mejorar la gestión de las flotas de vehículos, aporta la capacidad de estudio para la mejora y eficiencia de los sistemas de transporte y tiene el potencial de ampliarse con la implementación de nuevas funcionalidades.

BIBLIOGRAFÍA

A modo de bibliografía se deja el repositorio de GitHub donde se encuentra el código perteneciente a la herramienta, a demás de una copia del documento a modo de manual de usuario.

La conexión con la BBDD queda anónima con el fin de protegerla.

[AdriTesoUPM/TFG_CREACI-N-DE-DATOS-SEM-NTICOS-A-PARTIR-DE-STREAM-DE-DATOS-DE-SEGUIMIENTO-DE-VEHICULOS.](https://github.com/AdriTesoUPM/TFG_CREACI-N-DE-DATOS-SEM-NTICOS-A-PARTIR-DE-STREAM-DE-DATOS-DE-SEGUIMIENTO-DE-VEHICULOS) · GitHub