UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

Caracterización de la fiabilidad en transistores de potencia: Una aproximación integrada de hardware y software para la generación y el análisis de datos

TRABAJO FIN DE GRADO

Julio -2025

AUTOR: David Alcaraz Martínez

DIRECTORES: David Marroquí Sempere

Ausiàs Garrigós Sirvent

AGRADECIMIENTOS

A mis tutores, David y Ausiàs, por confiar en mí y darme el espacio para formarme en todos los sentidos como futuro ingeniero.

A Cristian, Pablo, Carlos y todos los integrantes del laboratorio, por fomentar un entorno positivo y hacerme sentir uno más.

A Yolanda Campello, por acogerme como parte de su familia.

A María, por ser mi faro en los momentos oscuros.

A mi familia, por hacer posible que llegara hasta aquí.

ÍNDICE DE CONTENIDOS

1. Introducción 18
1.1. Motivación 18
1.2. Fiabilidad de componentes 21
1.3. Definiciones y eventos de sobrecarga 24
1.4. Caracterización de transistores FET 27
1.5. Objetivos 29
2. Estado del arte 31
2.1. Revisión de entornos de trabajo similares 31
2.2. Revisión de técnicas de obtención de voltaje umbral de transistores FET 32
2.3. Revisión de técnicas de tratamiento de señal 33
2.4. Revisión de técnicas de almacenamiento y gestión de datos 36
3. Desarrollo del banco de trabajo para el enevejecimiento acelerado de transistores 39
3.1. Descripción del hardware para el banco de trabajo para la realización controlada de ensayos de estés en dispositivos de protección 39
3.2. Descripción del software para el banco de trabajo para la realización controlada de ensayos de estés en dispositivos de protección 45
3.2.1. Objetivos específicos y limitaciones 45
3.2.2. Protocolos de comunicación 46
3.2.3. Interfaz principal, modos de operación y funcionamiento 48
3.4.4.1. Módulo común de control 49
3.4.4.2. Módulo de Configuración inicial 49
3.4.4.3. Módulo de Fase de pruebas 51
3.4.4.3. Módulo de iniciar programa 52
3.4.4.4. Módulo de Tratamiento de errores 54
3.4.4.5. Estructura interna 56
4. Desarrollo de herramientas de análisis de datos 59
4.1. Análisis de eventos de sobrecarga 59
4.1.1. Introducción y objetivos 59
4.1.2. Ajuste del filtrado de ruido 61
4.1.3. Ajuste del error de *offset* 67
4.1.4. Funciones 70
4.1.4.1. Estructura de organización de los archivos 71
4.1.4.2. Descripción general del programa 72
4.1.4.3. Función de análisis de valor máximo 75
4.1.4.4. Función de análisis de intervalo de transición y desconexión 78
4.1.4.5. Función de análisis de energía disipada 90

4.1.4.6. Módulo de análisis de corriente media de limitación 92
4.1.5. Visualización de los datos procesados 94
4.2. Análisis de datos de caracterización 97
4.2.1. Introducción y objetivos 97
4.2.2. Funciones 99
4.2.2.1. Función de organización de datos 99
4.2.2.2. Representación gráfica de resultados, primer enfoque 101
4.2.2.3. Representación gráfica de resultados, segundo enfoque 105
4.2.2.4. Representación gráfica de transconductancia y obtención de voltaje umbral 107
4.3. Herramienta de cribado y visualización 114
4.3.1. Objetivos, Alcance y limitaciones 115
4.3.2. Interfaz principal, funcionalidades y módulos 116
4.3.3. Operaciones internas y control de errores 127
4.3.4. Ejemplos de uso y casos prácticos 138
5.Conclusiones y líneas futuras 147
ANEXO 1: Tablas de grupos de dispositivos 152
ANEXO 2: Shield para Arduino 155
ANEXO 3: Programa de secuencia de pulsos de Arduino 159
ANEXO 4: Diagrama de bloques del Programa de Labview en el banco de trabajo ... 160
ANEXO 5: Publicaciones relacionadas 173
ANEXO 6: Código para la herramienta de visualización de datos 183
ANEXO 7: Programas de análisis de datos de caracterización 197
ANEXO 8: Código para la herramienta de visualización de datos 245
5. Bibliografía 360

ÍNDICE DE FIGURAS

Fig. 1: Representación del proyecto propuesto, sus aplicaciones y las principales responsabilidades. 20
Fig. 2: Diagrama de bloques genérico de un LCL [11] 20
Fig. 3: Flujo de test y caracterizaciones sobre los grupos LCL2A-Tx-L1 y LCL10A-Tx-L2 y posterior procesamiento de datos. Fases en verde realizadas en la Universidad de Valencia y fases en azul realizadas en la Universidad Miguel Hernández. 23
Fig. 4: Diagrama temporal genérico del LCL ante una sobrecarga, extraído del estándar ECSS [11]. 25
Fig. 5: Representación general de las curvas de sobrecarga resistiva obtenidas a lo largo del trabajo. Curva de corriente a través de la celda propuesta (superior) y curva de tensión entrada-salida de la misma celda(inferior). 25
Fig. 6: Representación general de las curvas de cortocircuito obtenidas a lo largo del trabajo. Curva de corriente a través de la celda propuesta (superior) y curva de tensión entrada-salida de la misma celda(inferior). 26
Fig. 7: Representación general de las curvas de sobrecarga capacitiva obtenidas a lo largo del trabajo. Curva de corriente a través de la celda propuesta (superior) y curva de tensión entrada-salida de la misma celda(inferior). 27
Fig. 8: Tiempo de ejecución de diversos algoritmos de suavizado digital de señales de una dimensión, los algoritmos se ejecutaron en un PC con procesador Intel Core i7, en un entorno Python con soporte de las bibliotecas NumPy y SciPy. [19]. 35
Fig. 9: Diagrama de conexiones de control y adquisición sobre el sistema electrónico. 39
Fig. 10: Esquema de conexiones del sistema electrónico: sobrecarga resistiva (superior), cortocircuito (central) y sobrecarga capacitiva (inferior). 40
Fig. 11: Banco de trabajo completo. 43
Fig. 12: Secuencia de pulsos de sobrecarga. En el caso de la sobrecarga capacitiva, únicamente se genera la señal correspondiente al DUT. 44
Fig. 13. Diagrama de Bloques de Comunicaciones de Instrumentos mediante VISA. .. 47
Fig. 14: Diagrama de bloques de la operación de los distintos modos del programa, donde se destaca el flujo de programa estándar (verde), el flujo de error (rojo y amarillo) y el flujo de finalización forzada del programa (azul). 48
Fig. 15: Ventana de configuración inicial del programa de control del banco de trabajo en LabVIEW 51
Fig. 16: Ventana de fase de pruebas del programa de control del banco de trabajo en LabVIEW 52
Fig. 17: Ventana de inicio de programa del programa de control del banco de trabajo en LabVIEW 54
Fig. 18: Ventana de error del programa de control del banco de trabajo en LabVIEW. 56
Fig. 19: Curva aproximada de tiempo de cálculo frente al tamaño de ventana en el filtro de SG, se destacan dos zonas searadas por un punto óptimo de tamaño de ventana, a partur del cual, el suavizado distorsiona los detalles importantes de la señal original. 62
Fig. 20: Curva de potencia disipada en el JFET en pruebas de sobrecarga sobre un LCL (verde), y curvas derivadas de este mediante el filtro de SG (naranja y azul). 64
Fig. 21: Detalle 1 (en el momento de la sobrecarga) de la Fig. 19. 64
Fig. 22: Detalle 2 de la Fig. 20. 65
Fig. 23: Comparativa de curvas de potencia entre algunos de los JFET incorporados en LCL de clase 10 para sobrecargas capacitivas. 66
Fig. 24: Curva de potencia disipada en el JFET en pruebas de sobrecarga capacitiva sobre un LCL (verde), y curvas derivadas de este mediante el filtro de SG (naranja y azul).. 66

Fig. 25: Detalle 1 (comportamiento ante oscilaciones de intensidad) de la Fig. 23. 67
Fig. 26: Representación de la curva de intensidad del JFET del grupo LCL3 en distintas repeticiones de la misma prueba de cortocircuito. Se ha aplicado a todas las curvas el filtro de SG. 68
Fig. 27: Detalle de la Fig. 25. 68
Fig. 28: Diagrama de flujo general de la detección y corrección del error de offset para sobrecargas capacitivas..... 70
Fig. 29: Estructura de diccionarios confeccionada por el programa principal. 73
Fig. 30: Diagrama de flujo simplificado del programa principal, donde solo se muestra la interacción con el módulo de análisis de máximos. Parte 1_2. 74
Fig. 31: Diagrama de flujo simplificado del programa principal, donde solo se muestra la interacción con el módulo de análisis de máximos. Parte 2_2. 75
Fig. 32: Diagrama de flujo de la función de detección de valor máximo..... 77
Fig. 33: Estado inicial del proceso de obtención del inicio del intervalo de carga de la sobrecarga capacitiva. Vista general (izquierda) y vista en detalle (derecha). 80
Fig. 34: Estado intermedio del proceso de obtención del inicio del intervalo de carga de la sobrecarga capacitiva. Vista general (izquierda) y vista en detalle (derecha). 81
Fig. 35: Estado final del proceso de obtención del inicio del intervalo de carga de la sobrecarga capacitiva. Vista general (izquierda) y vista en detalle (derecha). 81
Fig. 36: Estado inicial del proceso de obtención del final del intervalo de carga de la sobrecarga capacitiva. Vista general (izquierda) y vista en detalle (derecha). 82
Fig. 37: Estado intermedio del proceso de obtención del final del intervalo de carga de la sobrecarga capacitiva. Vista general (izquierda) y vista en detalle (derecha). 82
Fig. 38: Estado final del proceso de obtención del final del intervalo de carga de la sobrecarga capacitiva. Vista general (izquierda) y vista en detalle (derecha). 82
Fig. 39: Diagrama de flujo de la función de detección de intervalo de carga. Parte 1_2. 83
Fig. 40: Diagrama de flujo de la función de detección de intervalo de carga. Parte 2_2. 84
Fig. 41: Estado inicial del proceso de obtención del inicio del intervalo de desconexión de la sobrecarga resistiva. Vista general (izquierda) y vista en detalle (derecha). 85
Fig. 42: Estado anterior al final del proceso de obtención del inicio del intervalo de desconexión de la sobrecarga resistiva. Vista general (izquierda) y vista en detalle (derecha). 86
Fig. 43: Estado inicial del proceso de obtención del final del intervalo de desconexión de la sobrecarga resistiva. Vista general (izquierda) y vista en detalle (derecha). 86
Fig. 44: Estado final del proceso de obtención del final del intervalo de desconexión de la sobrecarga resistiva. Vista general (izquierda) y vista en detalle (derecha). 87
Fig. 45: Comparativa de intervalos de desconexión en curvas de sobrecarga resistiva de varios grupos de LCL, tanto de clase 2 como de clase 10. 87
Fig. 46: Diagrama de flujo de la función de detección de intervalo de desconexión para sobrecarga resistiva. Parte 1_2. 88
Fig. 47: Diagrama de flujo de la función de detección de intervalo de desconexión para sobrecarga resistiva. Parte 2_2. 89
Fig. 48: Diagrama de la función de análisis de energía disipada. 91
Fig. 49: Diagrama de flujo de la función de análisis de corriente media de limitación. 93
Fig. 50: Ejemplo de representación visual de los resultados de energía disipada en el JFET obtenidos mediante los programas de análisis empelando el primer método de visualización. 95

Fig. 51: Ejemplo de representación visual de los resultados de tiempo de desconexión obtenidos mediante los programas de análisis empelando el primer método de visualización. 95
Fig. 52: Ejemplo de representación visual de los resultados de tiempo de desconexión obtenidos mediante los programas de análisis empelando el segundo enfoque de visualización, en este caso se representan cuatro grupos en pruebas de cortocircuito... 97
Fig. 53: Representación de la estructura de diccionarios jerárquicos obtenida con la función de organización de datos. 100
Fig. 54: Representación genérica del contenido de los archivos de los datos de caracterización provistos. 101
Fig. 55: Gráfica de caracterización de JFET de referencia IJW120R100T1, empleados en grupos de clase 10, de corriente de fuga respecto de tensión entre drenador y surtidor con comportamiento anómalo de un miembro (arriba) y misma gráfica una vez ha sido eliminado el dispositivo del grupo (abajo). 103
Fig. 56: Gráfica de caracterización de JFET de referencia IJW120R100T1, empleados en grupos de clase 2, de resistencia de canal respecto de la intensidad con comportamiento anómalo de varios miembros en una fecha específica (arriba) y misma gráfica una vez ha sido eliminado la fecha problemática(abajo). 104
Fig. 57: Gráfica de caracterización de grupo de LCL de clase 10, de resistencia de canal respecto de la intensidad en diferentes fechas. 106
Fig. 58: Gráfica de violín equivalente de la representación de la Fig. 56. 107
Fig. 59: Gráfica $Id - Vgs$ de JFET con comportamiento anómalo (rojo) y gráficas de transconductancia empleando distintos métodos (azules). 109
Fig. 60: Detalle 1 de la Fig. 58. 109
Fig. 61: Detalle 2 de la Fig. 58. 109
Fig. 62: Diagrama de flujo del programa que encuentra las regiones extrapolables lineales de las curvas de transconductancia con mesetas. 110
Fig. 63: Curvas de $Id - Vgs$ de JFET de referencia IJW120R100T1 en las fechas de enero/marzojunto con sus rectas de extrapolación lineal y la intersección de estas con el eje de abscisas. 111
Fig. 64: Gráfica $Id - Vgs$ con comportamiento estándar (rojo) y gráficas de transconductancia empleando distintos métodos (azules). 112
Fig. 65: Detalle de la Fig. 63. 112
Fig. 66: Gráfica de barras del valor del voltaje umbral en JFET de la familia UJN1205K empleando el método de corriente constante (10mA). 114
Fig. 67: Gráfica de barras del valor del voltaje umbral en JFET de la familia UJN1205K empleando el método de extrapolación lineal en la región de la meseta —método específico para curvas anómalas obtenidas de varios JFET—. 114
Fig. 68: Pantalla principal de la herramienta de visualización de datos. 117
Fig. 69: Diagrama de tratamiento de datos del programa. 118
Fig. 70: Ventana de carga de archivos CSV. 119
Fig. 71: Ventana de carga de estructuras jerarquizadas de dataframes 120
Fig. 72: Ventana de selección de dataframes dentro de la estructura jerarquizada del diccionario 120
Fig. 73: Ventana de información de datos (izquierda) y ventana de vista previa de dataframe (derecha). 121
Fig. 74: Ventana de selección de dataframes. 122
Fig. 75: Ventana de selección de subgráfico (izquierda) y ventana de asignación de nombres de columnas 124
Fig. 76: Ventana de análisis, sección de análisis de sobrecarga. 126

Fig. 77: Ventana de análisis, sección de análisis temporal..... 126
Fig. 78: Diagrama de bloques del funcionamiento del programa..... 128
Fig. 79: Diagrama del módulo común de control de memoria..... 135
Fig. 80: Ventana de aviso de exceso de RAM en método directo 1..... 136
Fig. 81: Diagrama de flujo del control indirecto de memoria. 136
Fig. 82: Diagrama de flujo del control directo de memoria 1. 137
Fig. 83: Diagrama de flujo del control directo de memoria 2. 138
Fig. 84: Proceso de carga de datos y personalización de contenido..... 139
Fig. 85: Proceso de comprobación de la integridad de los datos..... 140
Fig. 86: Proceso de representación de potencia..... 141
Fig. 87: Proceso de comprobación de curva defectuosa y eliminación..... 141
Fig. 88: Proceso de preparación de los datos del análisis de intervalo..... 142
Fig. 89: Resultado de la aplicación de un subgráfico inferior a la representación de las curvas..... 143
Fig. 90: Aplicación del zoom en la gráfica de las curvas de potencia para la comprobación de la veracidad del análisis de intervalo. 143
Fig. 91: Proceso de preparación de datos de caracterización extraídos de archivos '.spydata'. 144
Fig. 92: Proceso de comprobación de la integridad de los datos de caracterización.... 145
Fig. 93: Proceso de representación de curvas $Id - Vgs$. 146
Fig. 94: Representación de curvas y derivadas. 146
Fig. 95: Esquemático de la placa del shield para la tarjeta de Arduino Uno..... 155
Fig. 96: Capa Top de la placa del shield para la tarjeta de Arduino Uno..... 156
Fig. 97: Capa Bottom de la placa del shield para la tarjeta de Arduino Uno. 157
Fig. 98: Capa Silkscreen de la placa del shield para la tarjeta de Arduino Uno..... 158
Fig. 99: Diagrama de bloques del programa de LabView en el banco de trabajo (vista general)..... 160
Fig. 100: Diagrama de bloques del programa de LabView en el banco de trabajo (vista 1-1)..... 161
Fig. 101: Diagrama de bloques del programa de LabView en el banco de trabajo (vista 1-2)..... 162
Fig. 102: Diagrama de bloques del programa de LabView en el banco de trabajo (vista 1-3)..... 163
Fig. 103: Diagrama de bloques del programa de LabView en el banco de trabajo (vista 1-4)..... 164
Fig. 104: Diagrama de bloques del programa de LabView en el banco de trabajo (vista 1-5)..... 165
Fig. 105: Diagrama de bloques del programa de LabView en el banco de trabajo (vista 1-6)..... 166
Fig. 106: Diagrama de bloques del programa de LabView en el banco de trabajo (vista 2-1)..... 167
Fig. 107: Diagrama de bloques del programa de LabView en el banco de trabajo (vista 2-2)..... 168
Fig. 108: Diagrama de bloques del programa de LabView en el banco de trabajo (vista 2-3)..... 169
Fig. 109: Diagrama de bloques del programa de LabView en el banco de trabajo (vista 2-4)..... 170
Fig. 110: Diagrama de bloques del programa de LabView en el banco de trabajo (vista 2-5)..... 171

Fig. 111: Diagrama de bloques del programa de LabView en el banco de trabajo (vista 2-6). 172

ÍNDICE DE TABLAS

Tabla 1: Estructura de comandos de comunicación VISA. 47
Tabla 2: Intervalos sobre el eje temporal donde se extrae la muestra del offset en los distintos tipos de sobrecarga. 69
Tabla 3: Porcentajes sobre la base temporal de las curvas donde se produce la sobrecarga, obtenidos de forma empírica. 76
Tabla 4: Porcentajes sobre la base temporal de las curvas donde se busca el inicio y el final del intervalo de desconexión/carga. 78
Tabla 5: Grupos de dispositivos de clase 2. 152
Tabla 6: Grupos de dispositivos de clase 10 (parte 1). 153
Tabla 7: Grupos de dispositivos de clase 10 (parte 2). 154

ABREVIATURAS

AC — Alternate Current

API — Application Programming Interface

CSV — Comma Sepparated Values (file)

DAQ — Data Acquisition Systems

DC — Direct Current

EMI — Electromagnetic Interference

ESA — European Space Agency

FET — Field-Effect Transistors

FPGA — Field-Programmable Gate Array

GPIO — General-Purpose Input/Output

HTOL — High-Temperature Operating Life

GUI — Graphical User Interface

IGBT — Insulated Gate Bipolar Transistor

MOSFET — Metal Oxide Semiconductor Field-Effect Transistor

NI — National Instruments

NI MAX — National Instruments Measurement & Automation Explorer

PCB — Printed Circuit Board

SCPI — Standard Commands for Programmable Instruments

SG — Savitz-Golay

SiC— Carburo de Silicio

SPES — Space Power and Electronic Systems

TIC — Tecnologías de la Información y la Comunicación

UMH — Universidad Miguel Hernández

USB — Universal Serial Bus

VISA — Virtual instrument Software Architecture

WBG — Wide Band Gap

DUT — Dispositive Under Test

1. INTRODUCCIÓN

1.1. Motivación

Tal como se indica en el *European Green Deal* [1], la mejora en el bienestar y la salud de la población actual y la futura depende de acciones tomadas en áreas como la climatología, el transporte, el medio ambiente o la energía. La electrónica de potencia toma un papel crucial en la gestión de la energía, desde la producción, pasando por el almacenamiento, la transmisión y el uso final.

La ficha informativa "Supporting the Green Transition", publicada el 19 de febrero de 2020 [2], insta a mejorar las infraestructuras para que sean climáticamente neutrales para el 2030. Las tecnologías de semiconductores están ganando paulatinamente más relevancia para una transición verde y digital, especialmente el desarrollo de nuevos dispositivos semiconductores, tal como los *Wide Band Gap* (WBG).

Este desarrollo está alineado con los objetivos de fomentar la comercialización verde del espacio, ya que impacta en muchos aspectos de los subsistemas eléctricos espaciales. La Estrategia Tecnológica de la ESA [3] declara que las misiones espaciales y las comunicaciones satelitales deben ser energéticamente eficientes y requerir tecnologías y dispositivos de conversión de potencia avanzados. Las innovaciones tecnológicas en estos ámbitos apoyarán la economía circular y la neutralidad climática en Europa, contribuyendo también a otros sistemas eléctricos en DC de alta fiabilidad.

Los sistemas de propulsión ecológicos (propulsión eléctrica), la conectividad aumentada y la optimización de plataforma (reducción del coste por kilo lanzado) han impulsado desarrollos similares en otros sistemas de alta fiabilidad como la aeronáutica, donde se ha adoptado la distribución en ±270V DC, o en vehículos eléctricos, donde se emplean 400V DC o incluso 800V DC. Otros sistemas críticos en DC como centros de datos y TIC emplean también 380V DC. Considerando la máxima tensión en estos sistemas (800V), los dispositivos de 1.5kV representan un buen compromiso entre pérdidas por conducción, capacidad de bloqueo y rendimiento en conmutación.

El aumento de la tensión de distribución conlleva restricciones, siendo clave el desarrollo de semiconductores de potencia de alta tensión y fiabilidad para protección en DC. La propuesta técnica consiste en desarrollar nuevos dispositivos de potencia en carburo de silicio (SiC) de 1.5kV optimizados para un uso racional de la energía en estas tensiones.

Actualmente no existe ningún transistor de potencia en SiC cualificado por la ESA, y el único diodo SiC cualificado ha sido diseñado por el grupo del profesor Philippe Godignon (IMB-CNM), y comercializado por Alter Technology. Este diodo ha sido incluido en la European Preferred Part List (EPPL) de la ESA y utilizado en misiones como BepiColombo (ESA-JAXA), Solar Orbiter (ESA-NASA) y se considera para Juice-Gala y Comet I.

Hipótesis de partida y novedad

El aumento de la electrónica ubicua (ordenadores, comunicaciones, LED, vehículos eléctricos) y la generación renovable están impulsando el uso de distribución en DC. A diferencia de la distribución AC, la distribución en DC no presenta problemas de sincronización, ni potencia reactiva o armónicos, lo cual implica un control más simple.

La mayoría de los estudios se centran en el control y la gestión energética, dejando de lado la protección contra sobrecorrientes, la cual presenta dificultades específicas en DC como el arco continuo, mayor cortocircuito y alta $\frac{di}{dt}$. Estos problemas requieren dispositivos de estado sólido con respuesta rápida, en lugar de fusibles o protecciones electromecánicas. En la literatura se han propuesto varios dispositivos de protección en DC de estado sólido [4] y [5], operando entre 48V y 800V.

Estos dispositivos se conocen como:

- *Solid-State Circuit Breakers* (SSCBs), para interrupción rápida de fallos, en el rango de microsegundos.
- *Solid-State Power Controllers* (SSPCs) y *Latching Current Limiters* (LCLs), usados en aeronáutica y espacio, que incluyen, además de la protección contra sobrecorrientes, el control de corriente de irrupción a lo largo de un tiempo prestablecido (*Trip Off Time*), control remoto ON/OFF, protección contra subtensión y sobretensión, y protección térmica [6] y [7].

Tradicionalmente, estos dispositivos usan tecnología en silicio (Si), como IGBT (bipolares) o MOSFETs (monopolares). Sin embargo, estos presentan limitaciones en energía, potencia y comportamiento electrónico y térmico debido a las limitaciones físicas del silicio.

Los semiconductores de banda prohibida ancha como el SiC, en un estado ya maduro, permiten dispositivos SSCBs, SSPCs y LCLs hasta 400V con mayor conductividad térmica, campo eléctrico crítico y temperatura intrínseca. Se han reportado dispositivos de 400V, 5A nominales y 200A de corte en un tiempo inferior a 800ns [8]; una estructura en cascodo de SiC de 380V/80A con 9.4us de respuesta [9]; y limitadores de corriente autónomos de 380V/4.5A con un tiempo de respuesta inferior a 600ns [10].

Sin embargo, persisten las siguientes limitaciones:

- Falta de dispositivos de potencia tipo P (voltaje puerta-surtidor negativo).
- Diseños no óptimos tanto para el modo lineal como la conducción.
- Problemas alrededor de la alta temperatura y resistencia a radiación (*rad-hard*).

Propuesta del grupo SPES

Como respuesta a estas necesidades, surge un proyecto colaborativo liderado por el grupo de investigación SPES de la Universidad Miguel Hernández y la Universidad de Valencia, que plantea el desarrollo de una nueva generación de celdas de potencia inteligentes universales basadas en SiC, limitadoras de corriente, capaces de operar en 300V-DC y de alta fiabilidad. Estas celdas integrarán un MOSFET de canal P de baja tensión con un JFET SiC de alta tensión con control de corriente y sensor térmico integrados, optimizados para el modo lineal y con baja resistencia en conducción. Todos los dispositivos se diseñarán para operar a alta temperatura y ser resistentes a la radiación, cumpliendo requisitos de satélites (300V), aeronáutica (±270V DC) y vehículos eléctricos (hasta 800V).

La Fig. 1 resume las aplicaciones previstas y las responsabilidades de cada subproyecto.

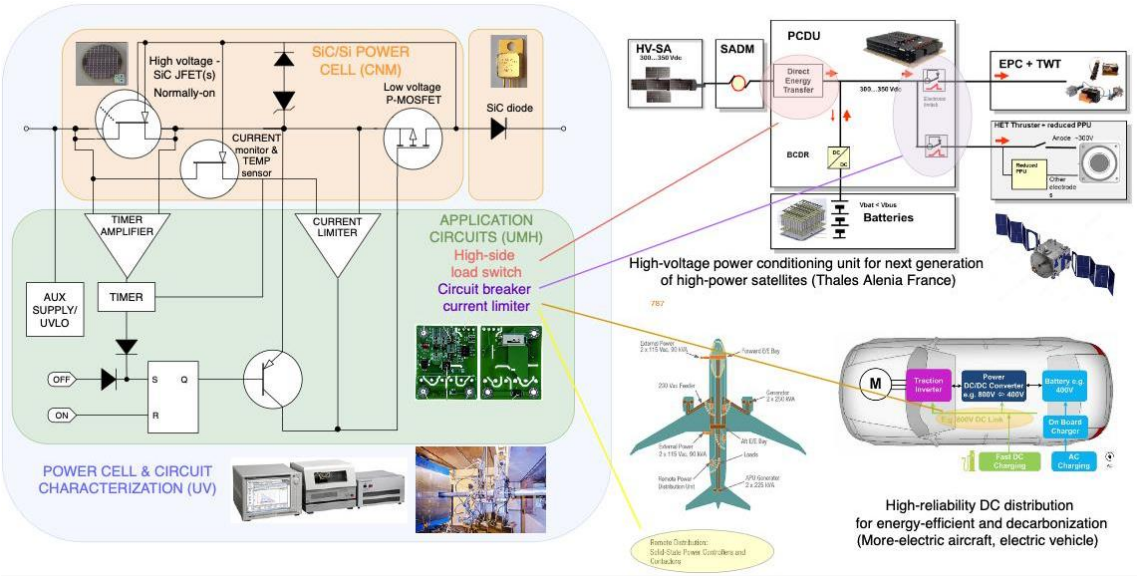


Fig. 1: Representación del proyecto propuesto, sus aplicaciones y las principales responsabilidades.

La celda se basa en un JFET SiC normalmente en conducción y un MOSFET Si de canal P de bajo voltaje normalmente en corte en configuración de cascodo. Incluye sensores integrados de corriente y temperatura para monitorización y protección. Puede configurarse como LCL, disyuntor o conmutador de carga con limitación de corriente.

La Fig. 2 muestra el diagrama de bloques de un LCL para aplicaciones espaciales según las directrices de la Agencia Espacial Europea [11].

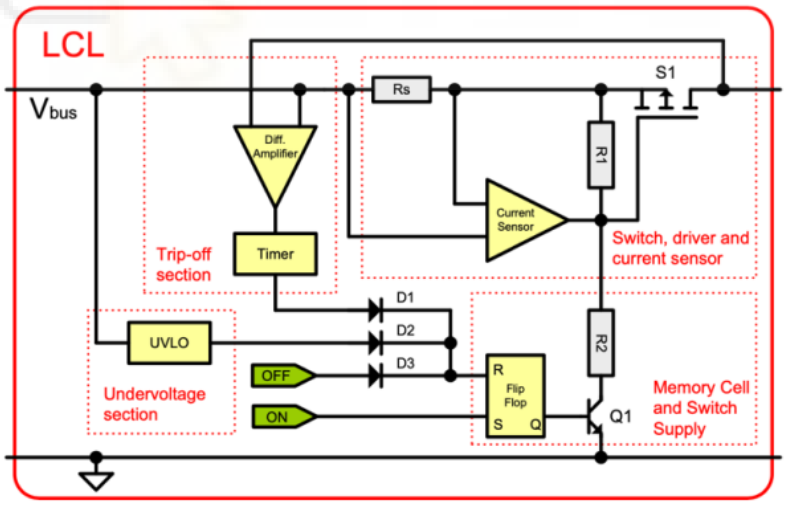


Fig. 2: Diagrama de bloques genérico de un LCL [11] .

Las ventajas principales de la celda propuesta por el grupo SPES con respecto a los circuitos de protección de sobrecorriente en DC son:

1. Mejora de sistemas de distribución DC de alta fiabilidad:
    - Control compatible con P-MOSFETs (dispositivo con requisitos de manejo simples).
    - Capacidad de bloqueo elevada con menores pérdidas debido al hecho de no utilizar resistencia de medida de corriente.
    - Mejora en la resistencia a la radiación [12].
    - Comportamiento óptimo en modo lineal y alta robustez.

2. Todos los sistemas de distribución DC se beneficiarán del uso de la celda de potencia inteligente: centros de datos, TIC, movilidad eléctrica, microredes, fuentes y almacenamiento.

Esta celda requiere validación mediante ensayos de vida acelerada para demostrar su fiabilidad y su respuesta en condiciones extremas; por ello, las actividades principales de verificación se organizan en dos sedes complementarias. En la Universidad de Valencia se desarrollarán las campañas de prueba HTOL (High-Temperature Operating Life) y la caracterización eléctrica de los dispositivos FET, asegurando el análisis de su comportamiento bajo estrés prolongado y distintos regímenes de temperatura.

De manera paralela, este trabajo atiende la fase de ensayos de sobrecarga repetitiva, que requiere un banco de pruebas en las instalaciones de Elche que permita realizar estos ensayos de forma fiable y sistemática, en diferentes tiempos y configuraciones, así como recopilar y sincronizar las mediciones eléctricas de interés. Asimismo, es preciso desarrollar un entorno de software que centralice el control de la instrumentación, gestione el almacenamiento estructurado de los datos experimentales y facilite el análisis estadístico de los resultados, tanto de sobrecarga como los datos de caracterización recopilados en Valencia. Con este enfoque, se cubren plenamente los requisitos de trazabilidad y fiabilidad del proyecto del grupo SPES, aportando el soporte técnico e informático necesario para completar la validación de las celdas de potencia propuestas, abarcando desde su respuesta ante sobrecargas hasta el análisis de las alteraciones sufridas por los dispositivos tras cada ensayo.

1.2. Fiabilidad de componentes

En el ámbito de la fiabilidad de dispositivos electrónicos, las pruebas de estrés desempeñan un papel fundamental para garantizar que los productos funcionen de forma segura y fiable durante toda su vida útil prevista. Estas pruebas consisten en someter los dispositivos a condiciones extremas de operación —como temperaturas elevadas, vibraciones mecánicas o tensiones eléctricas superiores a las nominales— con el objetivo de identificar posibles fallos latentes o debilidades de diseño y fabricación que podrían manifestarse durante su uso normal [13].

Una categoría específica dentro de estas pruebas son las denominadas pruebas de vida acelerada, que permiten estimar la vida útil y las tasas de fallo de un dispositivo en un plazo de tiempo reducido. Para ello, se aplican condiciones de estrés más severas que las reales, con el fin de acelerar los mecanismos de degradación física y química presentes en los componentes. De este modo, se obtiene información crucial sobre los modos de fallo, su frecuencia y su evolución, facilitando el desarrollo de diseños más robustos y fiables.

Según [13] existen principalmente dos tipos principales de pruebas de vida acelerada:

1. **Prueba de estrés constante:** En este método, el dispositivo se expone a un nivel constante de estrés que supera sus condiciones normales de operación. Se monitoriza el tiempo hasta que se produce un fallo para establecer una distribución estadística de vida útil bajo estrés. Este tipo de prueba es adecuado para estudiar fallos cuya tasa de ocurrencia se incrementa de forma predecible con la magnitud del estrés aplicado.

2. **Prueba de estrés escalonado:** Aquí, el estrés se incrementa de forma gradual en pasos predefinidos mientras se observa el comportamiento del dispositivo. El objetivo es identificar el nivel crítico de estrés a partir del cual se inicia el fallo, así como obtener estimaciones de vida útil bajo diferentes condiciones. Este enfoque es útil para definir los límites de tolerancia de un dispositivo y para entender mejor la resistencia de los materiales y estructuras internas.

El tipo de ensayos a desarrollar en el presente trabajo centran sus esfuerzos en las pruebas de estrés constante en diferentes fases, de tipo electrónico en base a sobrecargas repetitivas. El objetivo principal de estas pruebas no es encontrar el momento en el que se produce el fallo sino establecer una cantidad suficiente de sobrecargas y comprobar las posibles desviaciones en las características eléctricas de los componentes de la celda propuesta.

Además, el diseño de ensayos de vida acelerada es fundamental establecer condiciones de operación que garanticen tanto la fiabilidad de los resultados como la integridad de los dispositivos bajo prueba. Para ello, se aplica el concepto de *derating*, es decir, la reducción intencionada de los niveles de tensión, corriente y temperatura respecto a los máximos especificados por el fabricante. Esta práctica, ampliamente adoptada en entornos espaciales, tiene como objetivo disminuir la probabilidad de fallo, aumentar la vida útil de los componentes y asegurar márgenes de diseño adecuados. En este trabajo, se ha seguido la norma ECSS-Q-ST-30-11C [14], que establece criterios específicos de *derating* para distintos tipos de componentes electrónicos en aplicaciones espaciales, considerando su familia tecnológica y condiciones de operación prolongada.

Bajo estas directrices, las campañas HTOL, realizadas en paralelo al presente trabajo, se han llevado a cabo conforme a la norma JEDEC JESD22-A108G [15], aplicando condiciones de ensayo consistentes en 1.000 horas de operación a temperaturas de unión entre 100 °C y 120 °C, y al 75 % de la corriente máxima especificada para el dispositivo. Este enfoque permite activar los mecanismos de degradación relevantes sin exceder los límites de diseño, lo que garantiza que los resultados obtenidos sean extrapolables a condiciones reales de funcionamiento.

Del mismo modo, los ensayos de sobrecarga repetitiva desarrollados en este trabajo siguen principios de *derating* acordes con la citada normativa ECSS. Esta estrategia permite evaluar la respuesta de la celda y de sus dispositivos FET constituyentes ante eventos repetitivos de alta carga, obteniendo datos fiables y comparables para el análisis de fiabilidad a largo plazo, en consonancia con los requerimientos establecidos por el proyecto SPES.

La Fig. 3 muestra el itinerario seguido por los dos tipos de grupos principales para los análisis ejecutados a lo largo de este trabajo,

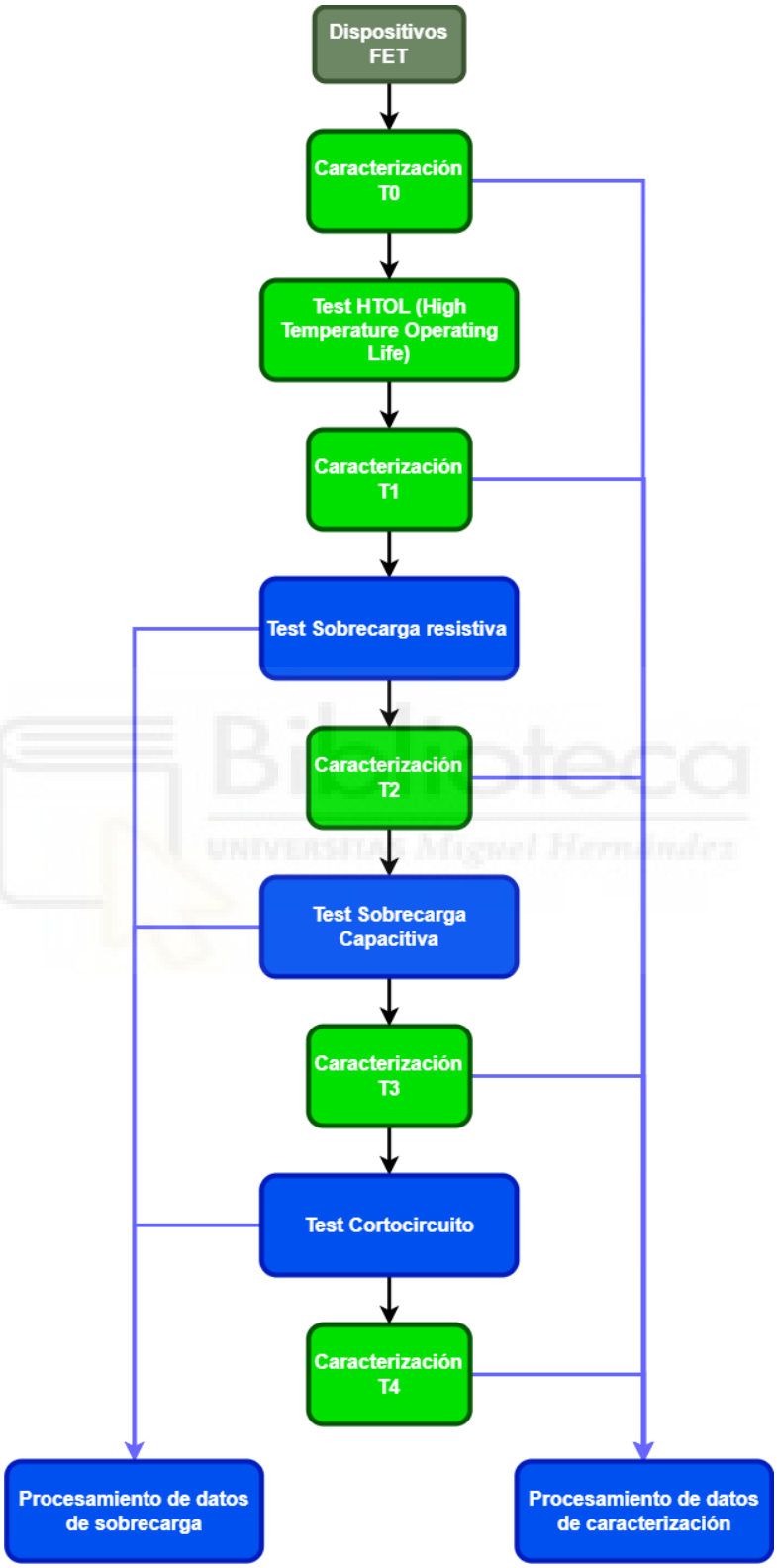


Fig. 3: Flujo de test y caracterizaciones sobre los grupos LCL2A-Tx-L1 y LCL10A-Tx-L2 y posterior procesamiento de datos. Fases en verde realizadas en la Universidad de Valencia y fases en azul realizadas en la Universidad Miguel Hernández.

1.3. Definiciones y eventos de sobrecarga

Durante parte de este trabajo se facilitará la representación para el análisis por parte del proyecto principal de la respuesta de las celdas propuestas incorporadas en circuitos de protección de tipo LCL, de forma que se han seleccionado una serie de dispositivos los cuales han sido clasificados en diferentes grupos, tal como pude verse en la Tabla 5, Tabla 6 y Tabla 7 del Anexo 1. Además, se incorporarán también celdas con tres JFET en paralelo de forma que se repartan la carga, perteneciendo a grupos de una corriente nominal superior a los grupos convencionales. El nombre que recibirá cada grupo será el de LCL más un número distintivo que irá desde el 1 hasta el 32, este nombre puede ir complementado con una referencia, estas referencias abarcan cuatro grupos de LCL distintos.

Para poder abarcar la extensión del trabajo, es necesario introducir una serie de definiciones acompañadas de una curva típica de sobrecarga (véase Fig. 4), desde las ECSS [11]:

- **Clase de LCL:** Corriente máxima permitida que puede fluir a través del propio LCL, bajo condiciones estándar.

- **Sobrecarga:** Cualquier condición en la cual un sistema eléctrico o electrónico se ve sometido a valores de corriente o tensión que exceden los niveles máximos especificados, ya sea de forma temporal o continua, poniendo en riesgo la funcionalidad o la integridad del sistema.

- **Corriente de limitación (*Limitation current*):** Es la corriente máxima que un circuito o sistema de protección permite en su salida durante una condición de sobrecorriente antes de que se active la desconexión o el sistema estabilice la corriente.

- **Pico de sobrecorriente (*Current Overshoot*):** Pico temporal de corriente que aparece antes de que el sistema alcance la corriente de limitación ante una sobrecarga.

- **Tiempo de desconexión (*Trip Off Time*):** Tiempo entre el cruce del valor real de limitación de corriente por parte del LCL y el evento de desconexión, en una condición de sobrecorriente permanente.

- **Tiempo de transición:** Tiempo entre el cruce del valor real de limitación de corriente por parte del LCL y el comienzo de la estabilización en una corriente de limitación.

- **Tiempo de limitación:** Tiempo entre el comienzo de la estabilización en una corriente de limitación y el evento de desconexión.

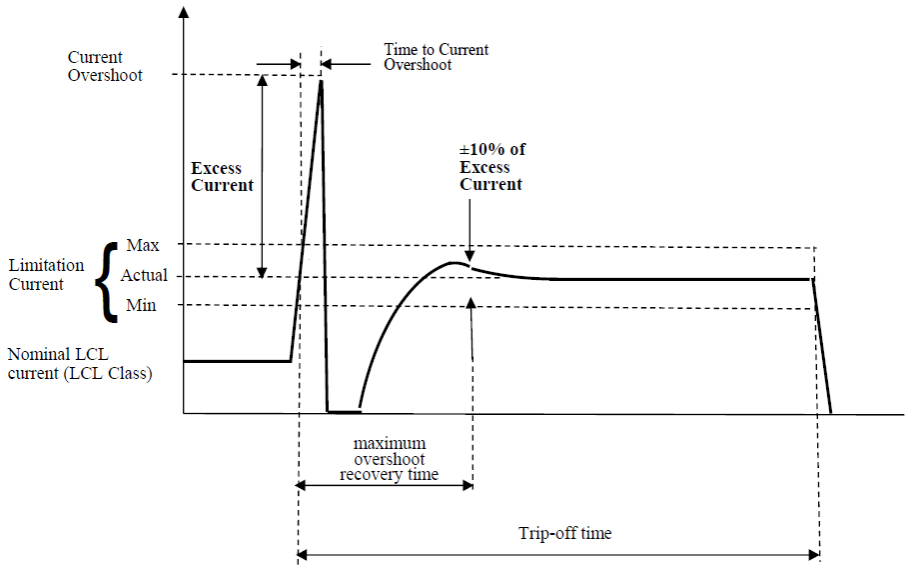


Fig. 4: Diagrama temporal genérico del LCL ante una sobrecarga, extraído del estándar ECSS [11].

A lo largo del trabajo se contemplarán tres tipos diferentes de sobrecarga, en la Fig. 5, Fig. 6 y Fig. 7 se muestran las curvas de voltaje y corriente a través de las celdas propuestas. Para estos tres tipos se pueden plantear definiciones en el ámbito específico de este trabajo:

- **Sobrecarga resistiva:** Se define como una condición de sobrecorriente generada por un cambio en la carga conectado al sistema, reduciendo su resistencia efectiva de forma controlada mediante un interruptor.

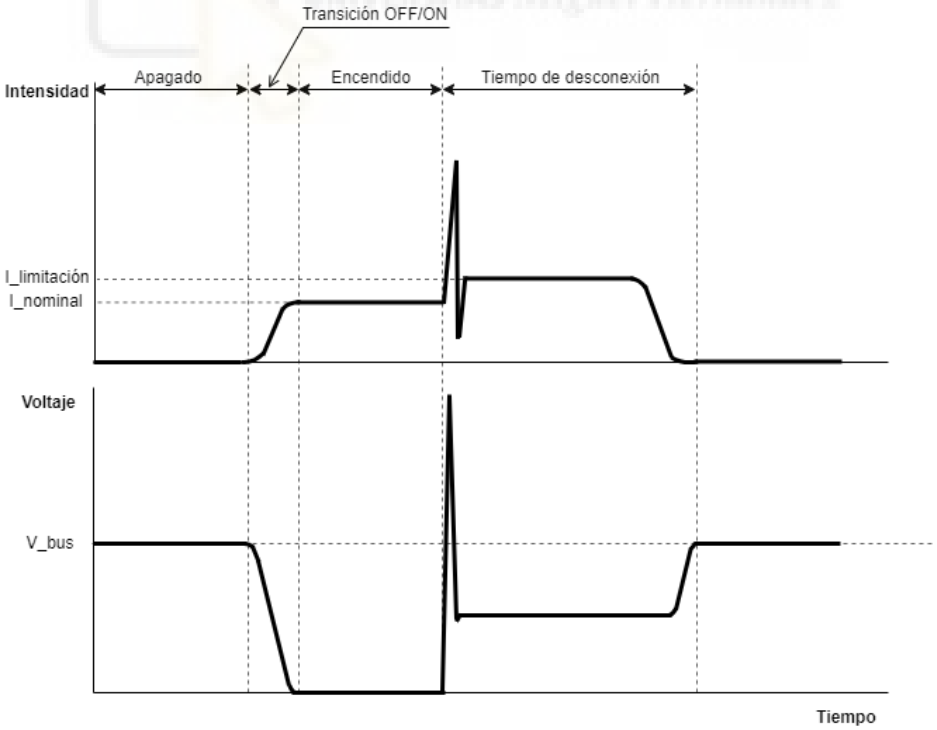


Fig. 5: Representación general de las curvas de sobrecarga resistiva obtenidas a lo largo del trabajo. Curva de corriente a través de la celda propuesta (superior) y curva de tensión entrada-salida de la misma celda(inferior).

- **Cortocircuito:** Se define como una condición de sobrecorriente generada por un cortocircuito en la carga conectado al sistema, mediante la activación de un interruptor que reduce la resistencia efectiva de la carga a la resistencia del cableado y del propio interruptor. La curva de corriente que se obtiene es muy parecida al caso de la sobrecarga, con la salvedad de que se alcanzan valores de corriente pico mucho más altos y que la zona de corriente de limitación no presenta una meseta tan clara, sino que tiene cierta inclinación, además, la tensión de bloqueo durante la limitación es mucho más alta.

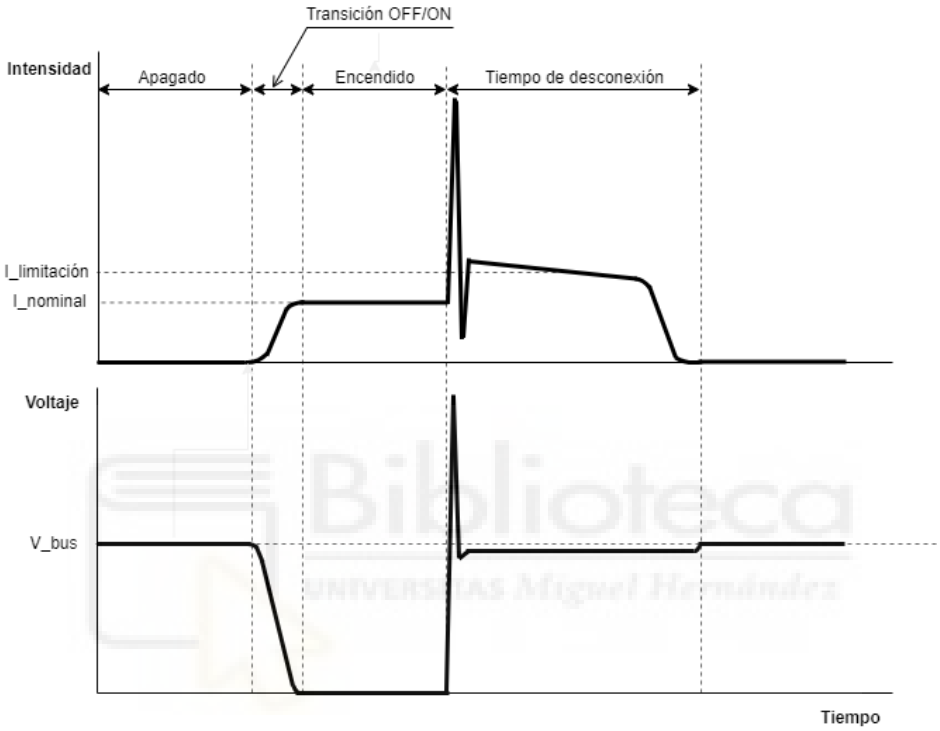


Fig. 6: Representación general de las curvas de cortocircuito obtenidas a lo largo del trabajo. Curva de corriente a través de la celda propuesta (superior) y curva de tensión entrada-salida de la misma celda(inferior).

- **Sobrecarga capacitiva:** se refiere a la corriente de irrupción (*inrush current*) que ocurre cuando se conecta una carga que incluye un condensador en paralelo con la línea de alimentación. A diferencia de los anteriores, en este caso no se analiza la respuesta del sistema ante una sobrecarga provocada, sino la respuesta del sistema a la carga del condensador en paralelo, el cual, inicialmente, demanda una alta cantidad de corriente.

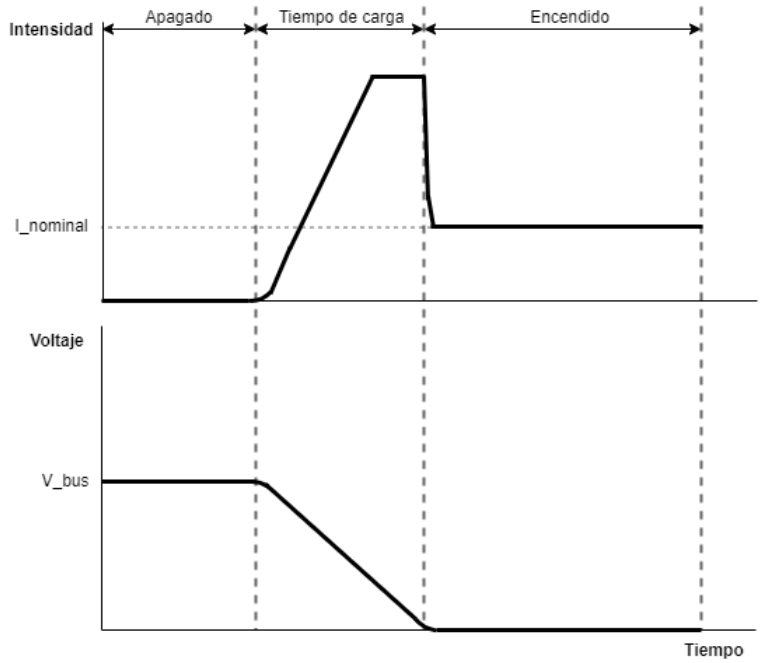


Fig. 7: Representación general de las curvas de sobrecarga capacitiva obtenidas a lo largo del trabajo. Curva de corriente a través de la celda propuesta (superior) y curva de tensión entrada-salida de la misma celda(inferior).

1.4. Caracterización de transistores FET

La caracterización de transistores es un proceso fundamental en la ingeniería electrónica que consiste en la medición y análisis de sus propiedades eléctricas para comprender y predecir su comportamiento bajo diferentes condiciones de operación. En particular, los transistores de efecto de campo (FET), ampliamente utilizados en amplificación y conmutación, requieren una caracterización precisa para asegurar el diseño eficiente y fiable de circuitos electrónicos.

Los FET se caracterizan por tres terminales principales: puerta, drenador y surtidor, donde la corriente entre drenador y surtidor se controla mediante el voltaje aplicado en la puerta. La caracterización eléctrica típicamente se realiza mediante la adquisición de diferentes curvas características, las cuales revelan las relaciones entre voltajes y corrientes en el dispositivo.

La caracterización permite una serie de objetivos:

- **Optimizar el diseño de circuitos**: Ajustar las condiciones de operación acorde al dispositivo para obtener el rendimiento deseado.

- **Validar modelos de simulación**: A partir de las curvas, se extraen una serie de parámetros relevantes, los cuales se usan en simuladores SPICE para prever el comportamiento en circuitos complejos.

- **Detectar variaciones y defectos**: Identificar degradación o fallos en dispositivos antes de su integración en sistemas finales.

A lo largo de este trabajo solo se incorporarán herramientas de procesado de tres tipos de curvas principales:

- **Curva $I_{d(off)} - V_{ds}$:** Esta curva muestra cómo varía la corriente de drenaje en estado apagado de un transistor FET en función del voltaje aplicado entre drenador y surtidor. Esta corriente, conocida como corriente de fuga, es idealmente, muy baja, pero su comportamiento frente a diferentes valores de V_{ds} es fundamental para evaluar la capacidad del transistor para evitar pérdidas y asegurar un apagado efectivo.

- **Curva $R_{ds} - I_d$:** Esta curva representa la relación entre la resistencia en estado de encendido del transistor FET y la corriente de drenaje. Esta resistencia es un parámetro clave que indica cuánto se opone el transistor al paso de corriente cuando está activado; generalmente, a medida que aumenta esta corriente, la resistencia puede variar debido a efectos térmicos y de saturación del dispositivo. Esta resistencia debe ser lo más baja posible para reducir pérdidas de conducción.

- **Curva $I_d - V_{gs}$:** Esta curva indica la relación entre la corriente del drenador y el voltaje aplicado entre puerta y surtidor en un transistor FET. Esta curva es fundamental para entender el comportamiento de encendido del transistor, ya que indica cómo varía la corriente cuando el voltaje de puerta supera el umbral de conducción. A partir de esta curva se determina el voltaje umbral (V_{th}), y permite evaluar parámetros clave como la transconductancia a partir de la derivada, que influyen en la ganancia y la sensibilidad del dispositivo.

Por lo anterior, además de realizar pruebas de vida acelerada, se llevarán a cabo caracterizaciones de los dispositivos individuales de cada grupo, así como de la celda en su conjunto, en diferentes fechas en la Universidad de Valencia, tras haber sometido los dispositivos a los diversos ensayos de estrés, incluyendo los propios ensayos de sobrecarga ejecutados en este trabajo.

1.5. Objetivos

El objetivo principal de este TFG es abarcar tres partes fundamentales del proceso general del diseño y validación del prototipo de celda propuesta:

1. **Objetivo General 1—Desarrollo de un banco de trabajo para la realización controlada y autónoma de ensayos de estés en dispositivos de protección**:
    - **Objetivo secundario 1.1 — Instrumentación:** Se pretende que el banco de trabajo esté compuesto y sea capaz de coordinar las siguientes partes principales:
        - Computador como unidad central de control, que disponga de un programa con una interfaz gráfica amigable para poder ser utilizado por personas ajenas al desarrollo del presente trabajo.
        - Unidad de generación de trenes de pulsos.
        - Dispositivo de control de potencia y placa de *Drivers*.
        - Fuentes de alimentación externas.
        - Banco de condensadores.
        - Conjuntos de resistencias de potencia combinables.
        - Instrumentos de adquisición y visualización de datos, entre los cuales se encontrará un instrumento principal que será controlado por el computador y uno o varios instrumentos secundarios.

    - **Objetivo secundario 1.1 — Configuración y manejo del dispositivo principal de adquisición y visualización de datos:** El programa deberá integrar una ventana en la GUI que facilite el ajuste inicial del dispositivo principal de adquisición, del cual obtendrá las mediciones.

    - **Objetivo secundario 1.2 — Ejecución y verificación de eventos de sobrecarga preliminares:** El programa debe incluir una ventana en la GUI que permita realizar eventos de sobrecarga de prueba para comprobar la integridad del sistema.

    - **Objetivo secundario 1.3 — Gestión de secuencias de eventos y temporización:** El programa debe incluir control sobre el flujo de sobrecargas repetitivas, además de permitir especificar temporización y cantidad.

    - **Objetivo secundario 1.4 — Monitorización y notificación de errores:** El programa debe incorporar un control activo sobre posibles fallos a lo largo del flujo de la sesión y aplicar un protocolo específico.

2. **Objetivo General 2—Desarrollo de programas para la interpretación de los datos generados por el banco de trabajo para la realización controlada de ensayos de estés en dispositivos de protección:**
    - **Objetivo secundario 2.1 — Mitigación de errores de ruido y offset en las curvas para obtener resultados fiables.**

    - **Objetivo secundario 2.2 — Diseño modular y escalable para facilitar futuras ampliaciones o adaptaciones.**

- **Objetivo secundario 2.3 — Organización de datos en estructuras jerárquicas para un acceso y análisis eficiente.**

- **Objetivo secundario 2.4 — Exclusión selectiva de datos o curvas atípicas para mantener la calidad del análisis.**

3. **Objetivo General 3—Desarrollo de un programa para la representación de los datos generados por el banco de trabajo para la realización controlada de ensayos de estés en dispositivos de protección y para datos de caracterización:**
   - **Objetivo secundario 3.1 — Funcionalidad versátil y adaptable:** El programa será capaz de analizar sobrecargas repetitivas y otros tipos de datos, adaptándose a diferentes contextos y fomentando la reutilización.

   - **Objetivo secundario 3.2 — Personalización y exportación de resultados:** El programa permitirá personalizar la visualización de gráficos y exportar resultados en varios formatos para su análisis posterior.

   - **Objetivo secundario 3.3 — Visualización múltiple para análisis comparativo:** El programa facilitará la comparación simultánea de múltiples conjuntos de datos mediante gráficos claros y eficientes.

   - **Objetivo secundario 3.4 — Operaciones matemáticas integradas:** El programa incluirá cálculos básicos y funciones estadísticas para realizar un análisis cuantitativo automatizado**.**

   - **Objetivo secundario 3.5 — Escalabilidad e integración con otros sistemas:** El programa estará diseñado para escalar y permitir la incorporación de nuevas funcionalidades, así como la integración con otros sistemas.

2. ESTADO DEL ARTE

Con el objetivo de establecer un marco contextual sólido para el desarrollo del presente trabajo, este apartado recopila y analiza las principales investigaciones, herramientas y metodologías existentes relacionadas con los distintos aspectos técnicos implicados. Se abordan, en primer lugar, entornos de trabajo similares que han servido como referencia o inspiración. Posteriormente, se revisan distintas técnicas empleadas para la obtención del voltaje umbral en transistores FET, así como métodos de análisis de señales mediante software especializado. Finalmente, se examinan diversas estrategias para el almacenamiento de datos y la gestión de la memoria RAM, con el fin de identificar enfoques relevantes y buenas prácticas aplicables para este trabajo.

2.1. Revisión de entornos de trabajo similares

En la actualidad, los bancos de pruebas para ensayos de vida acelerada han evolucionado hasta convertirse en sistemas integrados que combinan hardware de alta precisión y software avanzado, permitiendo replicar condiciones de estrés extremo de manera automatizada y coordinada entre los diferentes subsistemas. En contraste con métodos anteriores, que dependían en gran medida de instrumentos analógicos y configuraciones manuales, hoy en día se aprovecha la digitalización y la conectividad para obtener resultados precisos en tiempo real.

Actualmente, la generación de sobrecargas se realiza mediante circuitos que integran dispositivos de control de potencia que son capaces de manejar altas corrientes y voltajes—donde los más utilizados son los IGBT—. Estos dispositivos se controlan mediante señales digitales generadas por microcontroladores modernos, programados en lenguajes como C o Python, o incluso sistemas FPGA. Estos dispositivos permiten configurar perfiles de modulación muy precisos y flexibles.

Una parte esencial del sistema es la etapa de *drivers* para los dispositivos de potencia. Estos aseguran la carga y descarga rápida de capacitancias de puerta, y ajustan las condiciones necesarias del dispositivo permitiendo transiciones nítidas y reproducibles. La rápida conmutación es crucial para provocar condiciones súbitas de sobrecarga.

En los bancos de pruebas para ensayos de vida acelerada, la energía puede suministrarse al circuito bajo prueba mediante una fuente de tensión de alta potencia, la cual proporciona una alimentación continua y estable. Sin embargo, para simular condiciones de sobrecarga que requieren la entrega rápida de grandes corrientes, es común complementar la fuente principal con bancos de condensadores de alta capacidad. Estos condensadores pueden liberar su energía casi instantáneamente, lo que es esencial para replicar transitorios rápidos que las fuentes de tensión convencionales podrían no manejar eficazmente. Esta estrategia mejora la precisión de las pruebas y protege la integridad de la fuente de alimentación principal al evitar demandas repentinas que podrían afectar su estabilidad y rendimiento.

La configuración de la carga es otro aspecto clave. En los bancos de pruebas actuales se utilizan conjuntos de resistencias de potencia, que permitan ajustar de forma precisa el nivel de carga aplicado al dispositivo DUT. Esta modularidad en la configuración de la carga posibilita la simulación de diferentes escenarios de operación y facilita la adaptación a diferentes escenarios.

Asimismo, en la adquisición de datos se utilizan sistemas DAQ de alta velocidad y precisión, osciloscopios digitales y multímetros con interfaces de comunicación que permiten registrar, almacenar, y analizar de manera automática señales de voltaje y corriente en los puntos de interés.

La coordinación de todos estos elementos se logra a través de plataformas de software centralizadas —por ejemplo, LabVIEW, Matlab/Simulink o entornos desarrollados en Python— que integran la simulación, el control de la generación de señales y la adquisición de datos. Estos sistemas permiten programar secuencias de prueba, ajustar parámetros en tiempo real y sincronizar los distintos módulos mediante protocolos de comunicación modernos, como Ethernet, USB o interfaces industriales.

La confección del banco de trabajo implementado en este proyecto se basa en la implementada por David Marroquí en su tesis doctoral 'Diseño e implementación de controladores de potencia de estado sólido SIC para aplicaciones DC' [16]. La elección de este banco de trabajo se basa en la experiencia en pruebas de estrés repetitivo de este estilo y la fiabilidad que ha mostrado, el uso de los diferentes subsistemas del banco se verá adaptado a la situación de conocimientos y disponibilidad de materiales del momento en el que se comenzó a desarrollar.

2.2. Revisión de técnicas de obtención de voltaje umbral de transistores FET

El voltaje umbral (V_{th}) constituye uno de los parámetros más representativos del funcionamiento de los transistores FET, al definir el punto a partir del cual se establece una conducción significativa entre el drenador y el surtidor. Su conocimiento preciso es crucial para el diseño de circuitos, el modelado de dispositivos y la evaluación de su fiabilidad ante factores como el envejecimiento, la variabilidad del proceso o el estrés térmico.

En la obra de Ortiz-Conde et al. [17] se presenta una revisión detallada de los métodos más utilizados para la obtención de V_{th}, tanto en dispositivos monocristalinos como no cristalinos, considerando su aplicación en las regiones lineal y de saturación, así como su sensibilidad frente a parámetros parásitos como la resistencia serie o la degradación de movilidad. A continuación, se resumen los principales enfoques revisados:

- **Método de corriente constante (*Constant Current, CC*):** En este método se define V_{th} como el valor de voltaje de compuerta correspondiente a un nivel fijo de corriente de drenaje. Es ampliamente usado en entornos industriales por su simplicidad, aunque presenta dependencia del valor arbitrario de corriente elegido.

- **Método de extrapolación lineal (*Linear Extrapolation, LE*):** Este procedimiento consiste en extender linealmente la curva $I_d - V_{gs}$ desde su punto de máxima pendiente (máxima transconductancia) hasta cortar el eje de voltaje. Es uno de los métodos más aceptados, pero puede verse afectado por efectos parásitos y dificultades al obtener adecuadamente dicho punto.

- **Método del punto de coincidencia (*Match Point, MP*):** Este enfoque define V_{th} como el punto donde la corriente de drenador medida se desvía un porcentaje específico (usualmente 5%) del comportamiento esperado en la región de subumbral. Se pueden obtener diferentes valores de voltaje umbral al definir

diferentes valores de porcentaje específico de desviación. La obtención de este punto conlleva realizar una representación en escala semilogarítmica.

- **Método de la segunda derivada (*Second Derivative, SD*):** Este enfoque establece V_{th} como el voltaje donde la derivada segunda de la corriente respecto a la compuerta es máxima. Este método evita la dependencia frente a resistencias en serie, pero es altamente sensible al ruido de medición por el hecho de tener que aplicar dos derivadas sobre una curva que presumiblemente cuenta con ciertos errores de medición.

La evaluación comparativa de estos métodos muestra que la elección apropiada depende del tipo de dispositivo, la región de operación y el nivel de precisión requerido. Para este trabajo, se propone emplear dos métodos fundamentales, el método de la corriente constante y el método de la extrapolación lineal, analizando sus virtudes y sus desventajas en torno a las curvas de caracterización recibidas.

2.3. Revisión de técnicas de tratamiento de señal

Al analizar curvas obtenidas mediante un osciloscopio y transferidas a un ordenador, es fundamental identificar y comprender las posibles fuentes de error que pueden comprometer la precisión de los resultados. Entre las más relevantes se encuentran la resolución del ADC, la impedancia y carga de la sonda y la interferencia electromagnética (EMI), la deriva térmica, los errores de sincronización y muestreo, las limitaciones de ancho de banda, la pérdida de precisión durante la exportación de datos y los errores de calibración. Detectar y mitigar estos factores es esencial para garantizar la validez de las mediciones y evitar interpretaciones erróneas en el análisis posterior. En lo que respecta a este trabajo, tan solo se hará énfasis en dos de las manifestaciones más comunes de error para el análisis de curvas, ya sean provocados por una sola o varias de las fuentes comentadas anteriormente, estos son el ruido en la señal y el error de offset.

Ruido

El ruido en sistemas electrónicos se define como cualquier señal no deseada superpuesta a la señal útil, que degrada la precisión de la medida. Se origina tanto en fenómenos físicos inherentes a los componentes como en interferencias externas, y resulta el principal factor limitante en aplicaciones de baja señal y amplificación de precisión. El principal obstáculo que presenta el ruido en este trabajo no tiene que ver con la precisión de la señal puesto que se trabaja lejos del ámbito de la baja señal, en realidad tiene que ver con la dificultad de encontrar zonas con cambios de tendencia para distinguir programáticamente las distintas zonas de la curva.

Tal como comenta David Harvey, en su obra *Instrumental Analysis [18]*, existen diversas fuentes de ruido que afectan a la calidad de los datos obtenidos, siendo especialmente relevantes aquellas que se originan en el propio instrumental, entre estas, se encuentran:

- **Ruido térmico (Johnson–Nyquist):** Proviene de la agitación térmica de los portadores de carga en resistencias y elementos pasivos. Es de espectro plano (“blanco”) y su amplitud aumenta con la temperatura.

- **Ruido de disparo (*shot noise*):** Se debe a la naturaleza discreta de la carga eléctrica cuando los portadores atraviesan barreras de potencial, como en diodos y transistores polarizados. Su densidad espectral es proporcional a la corriente.

- **Ruido flicker ($\frac{1}{f}$):** Predomina a bajas frecuencias y está asociado a fluctuaciones en las trayectorias de los portadores y defectos en los semiconductores.

- **Ruido inducido y acoplado:** Surge de interferencias electromagnéticas externas (EMI), diafonía entre pistas y lazos de tierra mal diseñados. Puede introducirse a través de cables o el propio entorno de la placa.

En la literatura se encuentran diversas técnicas empleadas para reducir la influencia de este tipo de ruido, en este trabajo el enfoque principal es el del suavizamiento de la señal durante el procesamiento digital por su facilidad de aplicación.

Asimismo, pueden encontrarse diversos estudios comparativos de filtros de suavizado digital de señales, en la obra de Pawel Kowalski y Robert Smyk [19], donde se comentan algunos de los siguientes filtros:

- **Filtro de la media (*mean filter*):** Este filtro calcula la media aritmética de los valores dentro de una ventana deslizante para reemplazar cada punto de la señal.

- **Filtro de la mediana (*median filter*):** Este método, a diferencia del anterior, sustituye cada punto por la mediana de los valores dentro de una ventana ordenada, lo que reduce ruido puntual.

- **Filtro discreto de Kalman:** En este caso se estima el estado de un sistema dinámico a partir de medidas ruidosas, combinando predicción del modelo y corrección con la observación.

- **Filtro Gaussiano:** Este filtro se basa en aplicar una convolución entre la señal y una máscara basada en la distribución gaussiana, aplicando una mayor ponderación más los valores cercanos al centro de la ventana.

- **Kernel Density Estimation (*KDE*):** Este método emplea el cálculo de una estimación no paramétrica de la densidad de probabilidad, y suaviza la señal usando regresión ponderada con núcleos, siendo uno de los núcleos más populares el de Epanechnikov.

En la obra comentada anteriormente [19], se pueden encontrar diversas razones para elegir el filtro de SG sobre otros tipos de filtro, entre ellas, se destacan:

- **Comparativa de rendimiento:** Tal como puede observarse en la Fig. 8 donde se muestra una comparativa de tamaño de ventana frente a tiempo de procesamiento, se pueden extraer ciertas conclusiones:
    - Frente al filtro de la mediana, el filtro de SG es igual de rápido en ventanas de tamaño cercano a 25 y aproximadamente 3 veces más veloz en ventanas de tamaño cercano a 63.

    - En términos generales, el tiempo de ejecución es muy similar al del filtro de Kalman y solo es superado por el filtro Gaussiano y el filtro de la media.

- **Balance ruido–detalle muy competitivo:** frente a otro tipo de filtros como el de Kalman, SG ofrece una parametrización más sencilla (solo se han de seleccionar ventana y grado). Por el contrario, frente a otros tipos de filtro como el de la media o mediana, permite un ajuste más fino gracias a su capacidad para preservar la forma local de la señal mediante el uso de un polinomio de grado configurable.

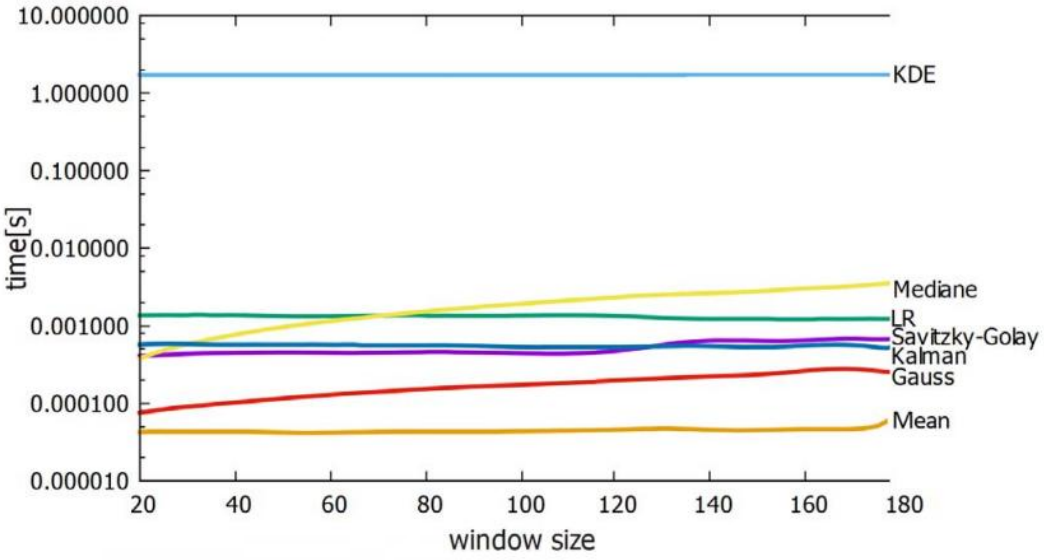


Fig. 8: Tiempo de ejecución de diversos algoritmos de suavizado digital de señales de una dimensión, los algoritmos se ejecutaron en un PC con procesador Intel Core i7, en un entorno Python con soporte de las bibliotecas NumPy y SciPy. [19].

Un aspecto fundamental para el desarrollo de los programas de análisis de eventos de sobrecarga será, por tanto, la aplicación de un filtro de suavizado digital, en este caso, se ha escogido el filtro de Savitzky–Golay atendiendo a diversas razones de diseño y adecuación del filtro a los datos a procesar. El filtro de SG, presentado por Abraham Savitz y Marcel Golay en 1964 [20] es un método de suavizado y diferenciación de señales basado en un ajuste polinomial por mínimos cuadrados sobre una ventana móvil de datos. Las principales virtudes de este filtro expuestas en la obra de sus autores son:

- **Fundamento matemático:** El filtro trabaja sobre ventanas con una cantidad impar de muestras (2m+1), a partir de esta ventana se ajusta un polinomio de grado n que minimiza el error cuadrático, y el valor suavizado (o su derivada) en el punto central se obtiene por convolución con coeficientes enteros normalizados, evitando resolver el sistema de ecuaciones en cada paso.

- **Preservación de formas y picos:** a diferencia de un promedio móvil clásico, conserva la altura y anchura de los picos casi intactas, ya que el ajuste local polinomial adapta la curva sin aplanarla excesivamente.

- **Cálculo de derivadas con alta fidelidad:** el mismo esquema de convolución permite estimar derivadas de primer y segundo orden en el punto central, coincidiendo con la derivada del polinomio ajustado y sin amplificar el ruido.

- **Control de la suavidad frente a la resolución:** el usuario define el tamaño de la ventana y el grado de polinomio para ajustar el compromiso reducción de ruido

frente a la preservación de detalles. Generalmente se trabaja con ventanas de grado bajo (por ejemplo, entre 2 y 4).

- **Eficiencia computacional:** El hecho de que cada punto se obtenga por convolución, la cual se puede implementar con desplazamientos y restas sucesivas, las cuales son operaciones sencillas, reduce la capacidad computacional necesaria.

Offset

En sistemas de medición, el error de ***offset*** se refiere a una desviación sistemática en la señal medida, representando un desplazamiento constante en el valor de la medición, incluso cuando la entrada real es cero. Este fenómeno es inherente a las características de los instrumentos de medición y puede originarse en diversas etapas del sistema, desde la sonda hasta el dispositivo de adquisición de datos.

El *offset* puede manifestarse como una desviación en la línea base de la señal, afectando la precisión de las mediciones. Además, el offset puede variar con el tiempo y las condiciones ambientales, lo que introduce una deriva que requiere atención y corrección periódica. Existen varias fuentes de error de offset, las cuales pueden provocan un error constante o variable a lo largo de sesiones de pruebas, algunas de las principales fuentes que se pueden asociar a las pruebas realizadas en el banco de trabajo del proyecto son:

- **Deriva Térmica**: El calentamiento interno de la sonda y las fluctuaciones de temperatura ambiente pueden causar cambios en las características eléctricas de los componentes, resultando en una variación del offset con el tiempo [21].

- **Interferencia Electromagnética (EMI)**: Campos electromagnéticos externos pueden inducir corrientes parásitas en la etapa de entrada de la sonda, desplazando la línea base de la señal medida [22].

- **Fluctuaciones de Alimentación y Acondicionamiento:** Variaciones en la tensión de alimentación de la sonda o en los circuitos de acondicionamiento de señal pueden alterar el punto de operación de los amplificadores internos, introduciendo un offset variable.

La forma de afrontar este tipo de error será, de nuevo, la corrección digital de este fenómeno, implementando una serie de técnicas basadas en cada tipo de curva que se han obtenido a lo largo de las campañas de prueba.

2.4. Revisión de técnicas de almacenamiento y gestión de datos

Estructuras de almacenamiento de datos

El almacenamiento eficiente y la manipulación de grandes volúmenes de datos representan un desafío fundamental en el ámbito del análisis de datos. Tradicionalmente, este problema ha sido abordado en la industria mediante el uso de sistemas de gestión de bases de datos, especialmente bases de datos relacionales —las cuales se basan en una organización de la información en forma de tablas, con una estructura de claves diferenciales— como MySQL o PostgreSQL, que ofrecen integridad referencial, consultas optimizadas mediante SQL y mecanismos robustos de búsqueda. Estos sistemas permiten una gestión escalable de información estructurada, sin embargo, su integración

eficiente con entornos de análisis como Python requiere capas intermedias como *SQLAlchemy*, las cuales pueden añadir complejidad al flujo de trabajo.

No obstante, en entornos científicos e interactivos, la industria opta en muchos casos por opciones más directas para el manejo en memoria, como los *dataframes* de Pandas. Estas estructuras cargan los datos desde archivos planos estructurados como CSV o desde ciertos tipos de bases de datos, proporcionando una vista tabular de fácil manipulación. Sin embargo, cuando el volumen de datos es elevado, los *dataframes* pueden consumir una gran cantidad de memoria, poniendo en compromiso la capacidad disponible del sistema.

En el contexto de este trabajo se ha optado por una solución simple, alejada del tratamiento de bases de datos, por lo tanto, se emplearán objetos de tipo *dataframe* para la lectura de diferentes tipos de archivos generados en el propio banco de trabajo o recibidos externamente. Para el almacenamiento de los datos se ha empleado una estructura de diccionarios jerarquizados, que facilitan el proceso de ordenar datos bien diferenciados, estos diccionarios se guardan, a su vez, en el espacio de trabajo del entorno de desarrollo integrado, que en este caso es Spyder, en archivos denominados '.spydata'. Aunque esta metodología de almacenamiento de datos carece del rigor de las formas de almacenamiento estándar, representa una solución flexible y eficiente para el entorno del trabajo.

Control de memoria RAM

Controlar el uso de la memoria RAM es esencial para garantizar que las aplicaciones —y muy especialmente las de análisis de datos en Python— funcionen de manera eficiente y estable. Sin una gestión adecuada, el sistema puede verse obligado a recurrir a la paginación, lo que degrada drásticamente el rendimiento, o incluso activar el *Out-Of-Memory Killer (OOM Killer)* del núcleo, provocando la terminación inesperada de procesos críticos.

Entre las consecuencias más relevantes de un consumo descontrolado de memoria destacan:

- **Ralentización por intercambio de páginas**: cuando la RAM se llena, el sistema mueve datos a disco; dado que el acceso a almacenamiento secundario es órdenes de magnitud más lento que a memoria principal, las operaciones se vuelven notablemente lentas y la aplicación puede dar la impresión de "congelarse".

- **Terminación de procesos por *OOM Killer***: si la memoria física y el espacio de *swap* —el cual constituye una porción del disco duro que actúa como memoria virtual cuando se llena la RAM— se agotan, el *kernel* —núcleo del sistema operativo que gestiona el hardware y los procesos— selecciona procesos para eliminar y liberar recursos; esto puede interrumpir servicios esenciales o hacer caer la propia aplicación.

- **Bloqueos y mala experiencia de usuario**: un uso excesivo de recursos provoca bloqueos, errores de memoria y tiempos de respuesta elevados, comprometiendo la fiabilidad y usabilidad del software.

- **Fragmentación y desperdicio de memoria**: sin políticas de limpieza o liberación (*garbage collection* eficiente), los objetos no referenciados y la fragmentación reducen la memoria realmente disponible para tareas críticas.

Para hacer frente a estos problemas, se plantean diferentes técnicas enfocadas en el control del uso de memoria a lo largo del programa y en la eficiencia en cuanto a los elementos cargados en ella. En el contexto de la eficiencia en el manejo de grandes volúmenes de datos tabulares se pueden encontrar, entre otros, trabajos como el de Kakaraparthy y Patel, '*SplitDF: Splitting Dataframes for Memory-Efficient Data Analysis'* [23], que proponen una técnica denominada *splitting*, la cual permite reducir el uso de memoria mediante la descomposición sin pérdida de *dataframes* en subconjuntos más compactos. A diferencia de la normalización tradicional, esta técnica no requiere el descubrimiento de dependencias funcionales y mantiene una vista unificada del *dataframe*, permitiendo al usuario operar sobre los datos sin modificar su flujo de trabajo habitual. La implementación de esta técnica ha demostrado reducir el uso de memoria entre un 19% y un 61% en distintos conjuntos de datos reales, lo cual la convierte en una solución eficaz para entornos de análisis donde los recursos de hardware son limitados.

En el ámbito de este trabajo, se propone el empleo de técnicas sencillas de control del uso de memoria a lo largo del programa y no se aplican técnicas de reducción del peso en memoria de los diferentes elementos de este en base a criterios de simplificad y eficiencia del empleo del tiempo disponible.

3. DESARROLLO DEL BANCO DE TRABAJO PARA EL ENEVEJECIMIENTO ACELERADO DE TRANSISTORES

El desarrollo de un banco de trabajo eficaz para la realización de ensayos de envejecimiento acelerado en las celdas propuestas requiere una definición precisa tanto del hardware como del software implicado. En este contexto, se definen una serie de objetivos específicos que orientan el diseño de la arquitectura física y lógica del sistema, abarcando desde la configuración de los módulos electrónicos hasta la implementación de protocolos de comunicación y una interfaz operativa adaptada a las necesidades del proyecto principal.

3.1. Descripción del hardware para el banco de trabajo para la realización controlada de ensayos de estés en dispositivos de protección

El banco de trabajo se ha diseñado para la experimentación, prueba y almacenamiento de resultados de los grupos de celdas propuestas dentro de un circuito especializado del tipo LCL, tecnología introducida en el apartado 1.1, operando bajo condiciones controladas. Su propósito principal es proporcionar un entorno adecuado para la realización y almacenamiento de mediciones que reflejen el comportamiento de los componentes bajo estas condiciones, facilitando la evaluación de su rendimiento y respuesta ante los distintos escenarios de sobrecarga ante un envejecimiento acelerado de sus componentes. En la Fig. 9 puede verse una representación del diagrama de conexiones del banco de trabajo desarrollado y en la Fig. 10 se muestra el esquema electrónico para los tres tipos de sobrecarga tratados en este trabajo, en estos esquemas se ha omitido la participación de instrumentos menos relevantes como multímetros o fuentes de alimentación para la tarjeta de *drivers*.

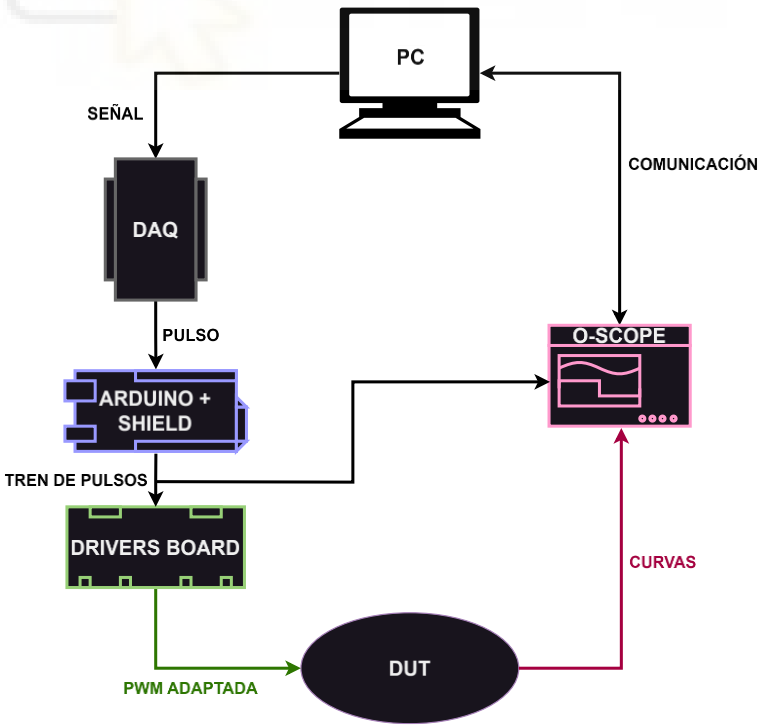


Fig. 9: Diagrama de conexiones de control y adquisición sobre el sistema electrónico.

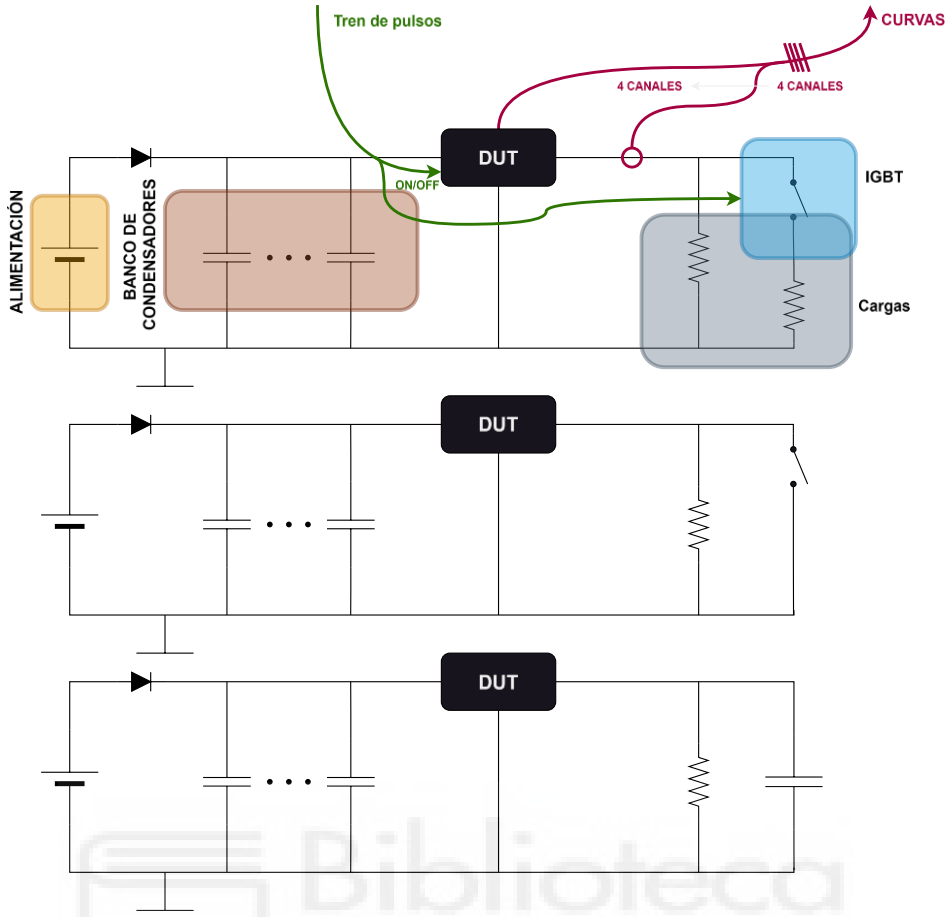


Fig. 10: Esquema de conexiones del sistema electrónico: sobrecarga resistiva (superior), cortocircuito (central) y sobrecarga capacitiva (inferior).

Los requerimientos mencionados deben ligarse con tres limitaciones fundamentales: las limitaciones temporales del proyecto, la disponibilidad de equipos y componentes en el laboratorio y el nivel de conocimiento técnico disponible en ese momento. Estos factores han guiado la selección y configuración de los elementos necesarios para llevar a cabo las pruebas de la manera más eficiente y segura posible dentro del marco especificado.

Bajo estas premisas, se definen como incorporaciones fundamentales:

- Sistema centralizado de control, en el que un ordenador actúa como unidad principal de gestión. Su función es coordinar y supervisar el funcionamiento de los distintos subsistemas, asegurando una operación sincronizada. Para complementar esta estructura, se desarrollará un programa con interfaz intuitiva que facilite la configuración rápida de las sesiones experimentales, permitiendo tanto pruebas individuales como cíclicas. Este software integrará visualización gráfica en tiempo real de las mediciones y contará con mecanismos para actualizaciones periódicas y registro de eventos, fortaleciendo así el control y análisis del banco de trabajo.

- Unidad de generación de trenes de pulsos, cuyo objetivo es activar y desactivar dispositivos de control de potencia y permitir o no la alimentación del DUT, los cuales permiten generar las condiciones controladas necesarias para la experimentación. La cantidad, duración y sincronización de los pulsos debe ser

configurable y flexible. Para este propósito se ha seleccionado el siguiente equipamiento:

- Placa Arduino Uno: se encarga de generar las señales de control necesarias en función de una señal de disparo digital recibida, el programa incorporado se puede observar en el Anexo 2. Esta configuración permite un control seguro y coordinado del DUT y de los dispositivos de conmutación. Se utiliza una placa *shield* adicional que facilita las conexiones y estabiliza las señales digitales.

  Para asegurar una correcta adaptación de las señales entre el Arduino UNO, el sistema de adquisición (DAQ) y los módulos de control, se ha diseñado un *shield* específico empleando el software KiCad, el cual se ha fabricado en laboratorio utilizando técnicas de insolado fotográfico. Este *shield* incorpora conectores y componentes pasivos disponibles en el laboratorio, como resistencias de *pull-down*, optimizando recursos sin afectar la funcionalidad. Estas resistencias son especialmente importantes para las entradas desde el DAQ, ya que este puede permanecer en un estado de alta impedancia al arrancar, lo que podría generar señales no deseadas en el Arduino. Estas resistencias garantizan que las entradas se mantengan en estado bajo hasta que el DAQ esté plenamente operativo, evitando activaciones espurias.

  En cuanto a la salida hacia los drivers, el *shield* conecta las señales desde los pines del Arduino a través de un conector de faja, donde igualmente se emplean resistencias de *pull-down* para mantener las líneas en bajo durante el arranque. Además, dado que los drivers requieren una alimentación de 5 V con una demanda considerable de corriente, no se utiliza la salida de 5 V del propio Arduino, sino que el shield dispone de una entrada de alimentación externa mediante conector de tornillo. Esta entrada se distribuye internamente hacia los conectores correspondientes, asegurando un suministro energético estable para todos los componentes implicados. El esquemático de este *shield* y las diferentes capas que componen la placa pueden verse en el Anexo 2.

- Tarjeta NI USB-6211: dispositivo de adquisición de datos de 16 entradas analógicas (16 bits), 2 salidas analógicas, 12 líneas digitales y 2 contadores. En este trabajo se utiliza para generar un pulso digital configurado desde LabVIEW, que actúa como señal de disparo para la tarjeta Arduino. Esta integración permite una sincronización precisa entre el software de control y el hardware de conmutación, asegurando una ejecución repetible de los ensayos.

- Interruptor de control de potencia. Necesario para provocar sobrecargas resistivas y simular cortocircuitos en condiciones controladas durante los ensayos.

  Para ello, se ha empleado el módulo IGBT FZ600R12KS4HOSA1 de Infineon, capaz de soportar hasta 1200 V y 600 A. Este componente permite conmutar cargas elevadas de forma segura, facilitando la simulación de fallos y maniobras críticas en los ensayos realizados en este trabajo.

- *Drivers* para los dispositivos de control de potencia, imprescindibles para adaptar la señal de control, asegurando su correcto funcionamiento con alta precisión y fiabilidad.

  Para este fin, se ha seleccionado la tarjeta de drivers aislada RDHP-1702 de Power Integrations, diseñada específicamente para aplicaciones en medio puente. Este dispositivo ofrece dos canales independientes y protección por hardware contra solapamiento, asegurando una activación robusta y segura del interruptor IGBT y el control de alimentación del DUT.

- Instrumento principal de adquisición de señales electrónicas y visualización directa, encargado de realizar las mediciones en los puntos de interés y enviarlas al ordenador para su procesamiento en el programa. La posibilidad de visualizar señales directamente en el instrumento, de manera independiente al software, es fundamental para garantizar la fiabilidad del proceso de experimentación. Dado que el sistema opera con transiciones agresivas, esta doble verificación contribuye a la detección temprana de posibles anomalías que podrían representar un riesgo para la seguridad.

  Para cubrir esta función, se ha empleado un osciloscopio Tektronix MDO3104 de dominio mixto, que permite la adquisición simultánea de señales analógicas y digitales, ofreciendo una visión completa del comportamiento del sistema bajo prueba. Este equipo resulta especialmente útil para monitorizar las señales de activación del interruptor de sobrecarga y de alimentación del DUT. La instrumentación se complementa con sondas de altas prestaciones: las sondas de corriente TCP303 y TCP300, capaces de medir hasta 150 A con 15 MHz de ancho de banda; las sondas diferenciales THDP0100 y THDP0200, que permiten medir tensiones de hasta 6000 V y 1500 V respectivamente, con anchos de banda de hasta 200 MHz; y la sonda digital P6316, apta para captar hasta 16 canales digitales con resolución temporal de hasta 5 ns.

- Multímetros digitales, usados como fuente secundaria de medición para garantizar la seguridad en el proceso experimental. Serán especialmente importantes para verificar parámetros críticos como el voltaje del banco de condensadores, asegurando que se mantengan dentro de rangos seguros durante las pruebas, y al finalizarlas.

  Para cumplir este objetivo, se ha seleccionado el multímetro Fluke 175 por criterio de disponibilidad en laboratorio y facilidad de uso.

- Fuentes de alimentación externas no controladas por el software. Se requiere una fuente de alta potencia para suministrar la energía al sistema además de una fuente secundaria de menor potencia para la alimentación de los *drivers* para los dispositivos de control de potencia. Estas fuentes no deben ser manejadas por el software, la conexión, especificación de parámetros y desconexión deben ser realizadas manualmente.

  Para la alimentación principal se ha utilizado la fuente Keysight N8937A, capaz de suministrar hasta 1500 V y 15 kW. Como fuente secundaria se ha empleado

una Tektronix PS280, adecuada para alimentar los drivers de los dispositivos de control de potencia, con salida de hasta 30V y 2A.

- Banco de condensadores para almacenamiento y posterior liberación de energía directamente al circuito. Este debe contar con un sistema seguro para liberar su energía cuando se haya terminado la sesión de experimentación.

  Para esta función, se ha integrado un banco de condensadores de Cornell Dubilier 947D591K132DJRSN conectado en paralelo a la fuente de alimentación, diseñado para absorber picos de corriente durante las pruebas y proteger el sistema. El conjunto, fabricado en el IE-g, está compuesto por seis unidades de 590 µF capaces de operar hasta 1300 V, ofreciendo una solución estable y robusta.

- Conjunto de resistencias de potencia combinables. Necesarias para ajustar la carga en el circuito según las necesidades experimentales.

  En este sentido se ha utilizado un banco de resistencias diseñado en el laboratorio IE-g, formado por cinco resistencias de 470 Ω cada una, configurables manualmente para simular diferentes cargas. El banco incorpora ventilación forzada para disipar el calor generado, garantizando un funcionamiento seguro y estable durante las pruebas de sobrecarga.

En la Fig. 11 se puede observar el banco de trabajo instalado en los laboratorios de la UMH.

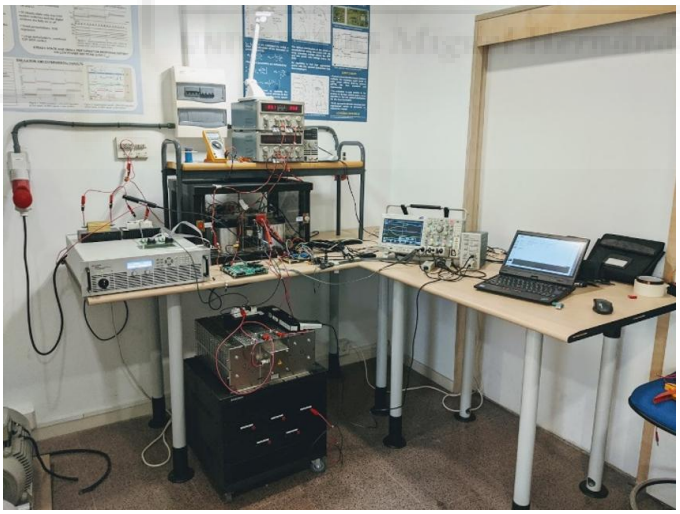


Fig. 11: Banco de trabajo completo.

A grandes rasgos, el banco de trabajo debe cumplir con el objetivo de ejecutar un número determinado de sobrecargas de forma controlada, registrando las curvas eléctricas más relevantes del sistema para su posterior análisis. Para lograrlo de manera efectiva, resulta fundamental crear una secuencia de pulsos adecuada para cada tipo de sobrecarga, ya que la topología de activación varía entre ellas. Por ejemplo, en el caso de la sobrecarga capacitiva, no se incluye el IGBT de la carga, lo que modifica el comportamiento del sistema.

El tren de pulsos afecta principalmente a dos zonas: por un lado, el propio dispositivo bajo test (DUT), al que se accede mediante su pin de alimentación. Cuando este pin se lleva a nivel alto, se activa la alimentación del sistema y el LCL entra en funcionamiento, permitiendo el paso de corriente hacia la carga. Por tanto, el funcionamiento normal del DUT se inicia cuando esta señal permanece en estado alto. Por otro lado, controla el IGBT de la carga. Este componente es el encargado de provocar el salto de carga que genera la sobrecarga y desencadena el comportamiento de protección en el LCL. De este modo, mediante el ajuste del tiempo de activación (tiempo en ON) de ambas señales, se controla con precisión el instante de aparición de la corriente nominal y el tiempo de aparición de la sobrecarga.

Una correcta temporización de la señal de sobrecarga o del IGBT permite asegurar que el LCL dispone del tiempo necesario para reaccionar y cortar la corriente (*Trip Off Time*). Si los tiempos están bien definidos tal como se muestra en el Anexo 3, puede utilizarse una misma secuencia de pulsos para los tres tipos de ensayo. En el caso de la sobrecarga capacitiva, basta con omitir la conexión de la señal de activación del IGBT, ya que dicho componente no está presente en ese modo de prueba.

Como estrategia para detectar con precisión el instante de sobrecarga mediante el osciloscopio, se puede utilizar directamente una de las señales del tren de pulsos como disparador (*trigger*), conectándola a uno de los canales digitales del osciloscopio y configurando el flanco de subida como evento de sincronización.

En la Fig. 12 se muestra una representación visual del tren de pulsos generado en el banco de trabajo. Con respecto a la Fig. 5, la Fig. 6 y la Fig. 7 —que muestran las curvas de sobrecarga obtenidas—, introducidas en el apartado 1.3, los tres tipos de sobrecarga sufren una transición de apagado a encendido una vez se pone a nivel alto la señal de activación del DUT, esta transición se denomina 'tiempo de carga' para las sobrecargas capacitivas debido a que es el momento en el que comienza a cargarse el condensador de salida. En cuanto al pulso del IGBT, que solo está presente en sobrecargas resistivas y de cortocircuito, una vez se pone a nivel alto, se activa el interruptor de potencia y ocurre un transitorio donde se alcanza el pico de sobrecorriente, después del cual el LCL entra en limitación durante el llamado 'tiempo de desconexión'.

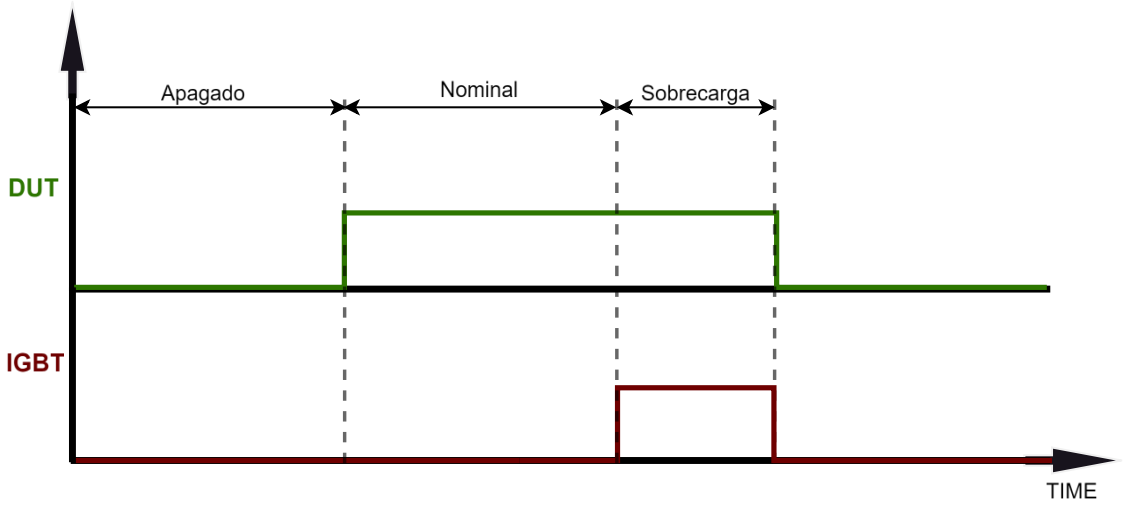


Fig. 12: Secuencia de pulsos de sobrecarga. En el caso de la sobrecarga capacitiva, únicamente se genera la señal correspondiente al DUT.

3.2. Descripción del software para el banco de trabajo para la realización controlada de ensayos de estés en dispositivos de protección

Para la gestión centralizada y eficiente de este trabajo, se ha elegido LabVIEW de National Instruments como plataforma principal de desarrollo. Este entorno de programación gráfica ofrece una interfaz intuitiva que facilita la creación y depuración de aplicaciones complejas, incluso sin conocimientos avanzados en lenguajes tradicionales. Además, LabVIEW destaca por su amplia compatibilidad con diversos dispositivos de hardware, lo que simplifica la integración de componentes heterogéneos dentro del sistema. Su capacidad para manejar proyectos de gran escala y reutilizar código contribuye a optimizar los tiempos de desarrollo y mantenimiento. Complementariamente, para la configuración y gestión del hardware de adquisición de datos se ha empleado el Explorador de Medición y Automatización (NI MAX), que permite una administración sencilla y efectiva de los dispositivos, asegurando así un rendimiento óptimo y una integración fluida con LabVIEW.

En base a esta plataforma, se desarrollará un programa con una serie de objetivos específicos, orientado a un manejo adecuado de los protocolos de comunicación y a facilitar la operación del banco de trabajo. A continuación, se presenta una descripción detallada del programa desarrollado.

3.2.1. Objetivos específicos y limitaciones

A partir de los objetivos generales presentados en el apartado 1.5, se desarrollan en profundidad los objetivos específicos del desarrollo del programa de control del banco de trabajo de sobrecarga sobre el LCL mediante LabVIEW:

- **Configuración y manejo del dispositivo principal de adquisición y visualización de datos:**
    - Integrar la comunicación con el osciloscopio Tektronix MDO3104 y las sondas especificadas, garantizando la correcta sincronización y transmisión de datos.
    - Permitir al usuario modificar la configuración de cada sonda tanto a través del programa como de forma manual, abarcando ajustes en parámetros críticos como son la sensibilidad y la impedancia.
    - Facilitar la personalización del *trigger* del osciloscopio, permitiendo ajustar el umbral, la pendiente y la posición del disparo según distintos escenarios de prueba.
    - Ofrecer opciones para configurar el eje horizontal, como el tiempo de muestreo y la escala temporal, de forma que se permita optimizar la visualización y el análisis de las señales.
    - Incluir la generación de archivos de información de la prueba de aportar información sobre las condiciones de configuración del osciloscopio y las sondas en las que se realizaron.
- **Ejecución y verificación de eventos de sobrecarga preliminares:**

- Implementar la opción de generar eventos de sobrecarga puntuales para comprobar la adecuación de la configuración del banco de trabajo y del osciloscopio.

- Incluir rutinas de validación que confirmen la correcta ejecución de los eventos y permitan detectar posibles desajustes o errores en la configuración.

- **Gestión de secuencias de eventos y temporización:**
  - Ofrecer la posibilidad de especificar y ajustar el tiempo entre repeticiones de eventos de sobrecarga, adaptándose a los requisitos específicos de rearme del circuito.

  - Implementar la funcionalidad para detener manualmente el flujo de eventos de sobrecarga, lo que permitirá realizar ajustes de forma inmediata en caso de detectar algún malfuncionamiento en el sistema.

- **Monitorización y notificación de errores:**
  - Incorporar un sistema de monitoreo que identifique tempranamente cualquier error o anomalía durante la ejecución del flujo de pruebas. Además, ofrecer la posibilidad de rearmar el sistema para casos de errores leves, permitiendo continuar con el flujo de eventos.

  - Integrar un mecanismo de notificación vía correo electrónico que alerte al usuario en caso de ocurrir algún error, permitiendo una rápida intervención y corrección de incidencias.

3.2.2. Protocolos de comunicación

De entre las diferentes posibilidades de programación que ofrece el osciloscopio a utilizar, se ha optado por utilizar la interfaz VISA. Esta API estándar facilita la comunicación entre el ordenador y dispositivos de medición, permitiendo el control y la configuración del osciloscopio mediante comandos específicos.

El osciloscopio Tektronix MDO3104 ofrece varias alternativas para su programación, entre las cuales se destacan:

- ***VISA drivers*:** Permiten una comunicación directa con el dispositivo a través de comandos estandarizados, ofreciendo una integración robusta y flexible.

- ***e*Scope Web-enabled tools*:** Herramientas web que facilitan el acceso y control remoto del dispositivo mediante un navegador, ideales para entornos en los que se requiera supervisión a distancia.

- ***Socket server*:** Establece una conexión basada en TCP/IP, útil en configuraciones de red específicas.

En este trabajo se utilizará la conexión USB, ya que, además de facilitar la configuración, ofrece una alta velocidad de comunicación y fiabilidad. Sin embargo, es importante destacar que la interfaz VISA también permite conexiones vía Ethernet o mediante cables GPIB, ampliando así las posibilidades de integración en otros entornos.

Para utilizar VISA es imprescindible instalar los drivers correspondientes, que permiten la conexión con distintos tipos de hardware a través de interfaces. En este caso, se empleará NI-VISA, la implementación de National Instruments del estándar VISA. NI-VISA no solo proporciona la API necesaria para el envío y recepción de comandos, sino que también incluye los drivers específicos para garantizar una comunicación estable y eficiente con los instrumentos de medición.

El funcionamiento de VISA se fundamenta en el uso de comandos estandarizados, habitualmente basados en el protocolo SCPI. Estos comandos permiten enviar instrucciones estructuradas y coherentes para configurar parámetros críticos del dispositivo, tales como ajustes de *trigger*, escalas temporales y la adquisición de datos. La estructura jerárquica de los comandos SCPI facilita la navegación a través de las funciones del osciloscopio, permitiendo operaciones específicas como definir la fuente de *trigger*, ajustar la escala temporal o iniciar la captura de datos.

En la Fig. 13 se puede visualizar el diagrama de bloques del flujo de comunicación entre los instrumentos mediante VISA, destacando el rol de cada uno de los sistemas involucrados

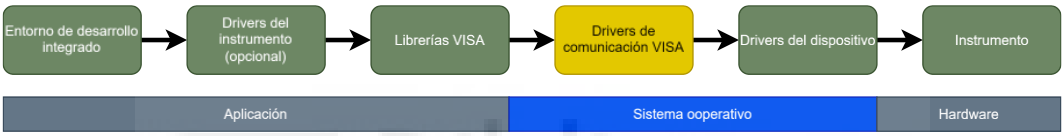


Fig. 13. Diagrama de Bloques de Comunicaciones de Instrumentos mediante VISA.

Esta metodología no solo simplifica la integración en sistemas automatizados, sino que también posibilita el desarrollo de aplicaciones a medida. Por ejemplo, se puede crear, como en el caso del software desarrollado para el banco de trabajo, una interfaz de usuario intuitiva que oculte la complejidad de los comandos subyacentes, permitiendo que incluso usuarios sin conocimientos profundos de SCPI interactúen con el osciloscopio de manera sencilla y eficiente.

En la Tabla 1 se muestra la correcta formulación de las instrucciones encomendadas al osciloscopio a lo largo de la ejecución del programa, un ejemplo de instrucción utilizado en el programa es 'TRIGger:MODe NORMal', instrucción la cual contiene un encabezado, un solo mnemónico y un argumento, que especifica que el modo de funcionamiento del *trigger* del osciloscopio debe establecerse en 'normal'.

Tabla 1: Estructura de comandos de comunicación VISA.

Símbolo	Significado
Encabezado	Es el nombre básico del comando. Si el encabezado termina con un signo de interrogación, el comando es una consulta. El encabezado puede comenzar con un carácter de dos puntos (**:**), si el comando se concatena con otros comandos, es obligatorio.
Mnemónico	Es una subfunción del encabezado. Algunos comandos tienen un solo mnemónico, mientras que otros pueden tener múltiples mnemónicos, separados siempre por un carácter de dos puntos (**:**).

Argumento	Es una cantidad, calidad, restricción o límite asociado al encabezado. Algunos comandos no tienen argumentos, mientras que otros pueden tener múltiples argumentos.
Coma	Una sola coma se usa entre argumentos en comandos con múltiples argumentos. Opcionalmente, puede haber espacios en blanco antes y después de la coma.
Espacio en blanco	Un carácter de espacio en blanco se usa entre un encabezado de comando y su argumento relacionado. Opcionalmente, el espacio en blanco puede estar compuesto por múltiples caracteres de espacio.

3.2.3. Interfaz principal, modos de operación y funcionamiento

El programa ha sido diseñado con cuatro módulos o modos de operación junto con otro módulo común que estructuran de manera clara y concisa la secuencia de procesos necesarios para llevar a cabo las pruebas de sobrecarga repetitivas. Estos módulos representan los cuatro escenarios fundamentales en el flujo de trabajo tal como puede verse en la Fig. 14, garantizando una transición ordenada entre la configuración, la validación de parámetros, la ejecución de pruebas y la gestión de errores. Esta segmentación no solo facilita el uso del sistema, sino que también permite un control preciso sobre cada fase del proceso, optimizando la fiabilidad y repetibilidad de las pruebas. Además, cada uno de los cuatro módulos principales tienen asociada una ventana interactiva. El diagrama de bloques general se puede encontrar en el Anexo 4.

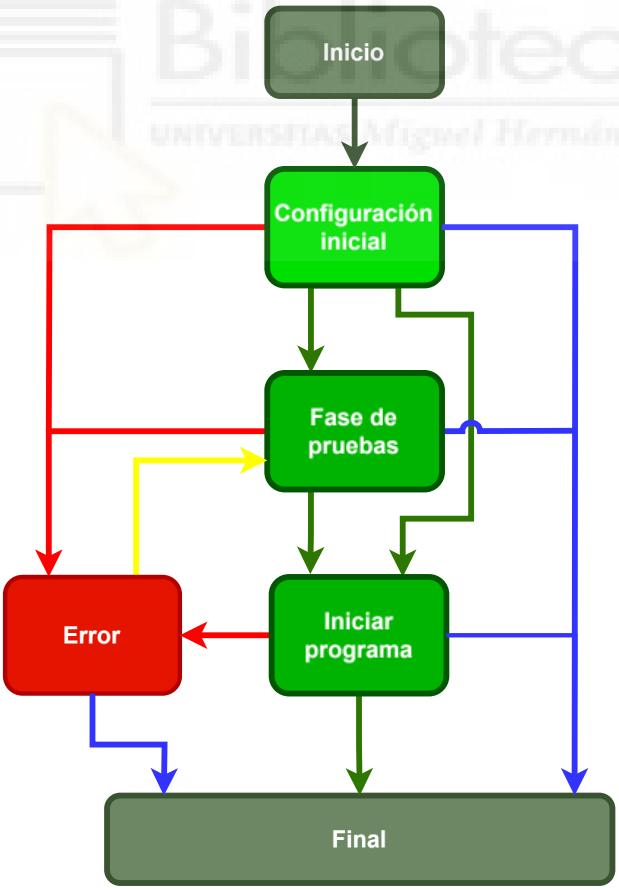


Fig. 14: Diagrama de bloques de la operación de los distintos modos del programa, donde se destaca el flujo de programa estándar (verde), el flujo de error (rojo y amarillo) y el flujo de finalización forzada del programa (azul).

3.4.4.1. Módulo común de control

En la parte superior derecha de todas las ventanas del programa desarrollado en LabVIEW se encuentra una pestaña común, presente en todos los módulos de operación. Este módulo ofrece al usuario acceso permanente a una serie de funcionalidades durante la ejecución del programa. Su diseño responde a la necesidad de garantizar un control centralizado y seguro sobre los elementos críticos del sistema de pruebas.

Una de las funciones principales del módulo es el botón llamado 'Parar programa'. Este botón permite finalizar de forma segura la sesión de trabajo. Es importante recalcar que se debe utilizar este botón en lugar del botón de 'Abortar ejecución' propio del entorno de LabVIEW, ya que su activación desencadena una serie de procesos que aseguran la desconexión controlada de los distintos instrumentos conectados. Este cierre ordenado evita posibles errores de comunicación o bloqueos en los canales de conexión, particularmente con la tarjeta de adquisición de datos (*DAQ*) y el osciloscopio.

Justo encima del botón de parada se encuentra el campo denominado 'Visa resource name del osciloscopio', en el cual debe especificarse el identificador VISA del osciloscopio. Este identificador, que puede obtenerse fácilmente a través del software NI MAX es indispensable para establecer una conexión correcta con el dispositivo. La correcta introducción de este valor garantiza que el programa pueda comunicarse de forma estable y efectiva con el osciloscopio. La necesidad de incorporar un campo dedicado exclusivamente a este identificador se debe a que a lo largo de las campañas de pruebas de este trabajo ha resultado ser un parámetro conflictivo.

A continuación, el módulo presenta tres elementos de configuración destinados al control de la tarjeta DAQ. Estos permiten:

- Seleccionar el terminal de salida por el cual se emitirá el pulso de control hacia el sistema.
- Establecer el contador interno de la DAQ encargado de generar dicho pulso.
- Especificar la duración (ancho) del pulso que se enviará desde el DAQ hasta la tarjeta de Arduino.

Este último parámetro —el ancho del pulso— resulta especialmente relevante, ya que debe ajustarse con precisión para asegurar que la señal generada sea correctamente detectada por la placa Arduino, que actúa como módulo principal de conmutación.

Cabe señalar que es durante la **Fase de Pruebas** cuando el usuario tiene mayor capacidad ajustar manualmente estos parámetros del módulo común, aunque pueden cambiarse desde cualquier módulo excepto durante la fase de **Iniciar Programa**.

3.4.4.2. Módulo de Configuración inicial

El primer módulo que se presenta al usuario al iniciar una sesión de trabajo en el entorno de pruebas automatizado está destinado a la configuración del osciloscopio. Esta etapa es fundamental, ya que una correcta preparación del instrumento garantiza la calidad y precisión de los datos adquiridos durante las sesiones de ensayo. En la Fig. 15 puede verse la ventana implementada en LabView.

Este módulo permite al usuario ajustar distintos parámetros del osciloscopio antes de que comience la adquisición automatizada. La primera sección está dedicada a la configuración del eje temporal, donde pueden modificarse opciones como la base temporal o escala de tiempo (*timebase*) y el retardo horizontal (*horizontal delay*). Estos parámetros son esenciales para adaptar la visualización de la señal, permitiendo centrar eventos de interés en la pantalla, y ajustar la resolución temporal para un análisis más detallado.

Uno de los parámetros más relevantes es la cantidad de puntos de muestreo que el osciloscopio almacenará por adquisición (*record lenght*). Esta opción tiene un impacto directo tanto en la calidad como en el rendimiento del sistema. Un mayor número de puntos permite una resolución más alta, lo que resulta especialmente útil para analizar fenómenos con cambios abruptos, como los transitorios asociados a la limitación de corriente del LCL. Sin embargo, este aumento de resolución tiene como contrapartida una mayor carga de datos, lo que puede ralentizar significativamente el proceso de transferencia desde el osciloscopio al ordenador. Esto, a su vez, incrementa el tiempo mínimo entre repeticiones de sobrecarga, y complica el manejo de los datos en el posprocesamiento, tanto por su tamaño como por el consumo de recursos que conlleva.

A continuación, se ofrece al usuario la posibilidad de modificar la configuración del disparo (*trigger*) del osciloscopio. Este mecanismo determina cuándo debe comenzar la adquisición de una señal, lo cual es clave para asegurar la repetibilidad y la alineación temporal entre distintas pruebas. Entre las opciones disponibles, destaca la posibilidad de seleccionar el tipo de disparo (*trigger mode*), siendo los más habituales:

- **Auto (automático):** El osciloscopio continúa refrescando la señal incluso si no se detecta el evento de disparo, útil para sostener una representación constante de la señal que capta el osciloscopio, aunque con menor precisión temporal del evento.

- **Normal:** El dispositivo espera indefinidamente hasta que se produce el evento de disparo, garantizando que la captura comience justo en el punto de interés, siendo ideal para eventos raros o transitorios definidos.

En la siguiente sección del módulo, se proporciona al usuario un control detallado sobre los canales del osciloscopio. Es posible seleccionar qué canales estarán activos durante la sesión, así como configurar sus propiedades fundamentales. Estas propiedades incluyen el rango de tensión, el desplazamiento del nivel de la sonda (*offset*), el tipo de acoplamiento (AC o DC) y la atenuación de la sonda, esta última es ajustable en función del tipo de sonda empleada y la configuración admitida por el osciloscopio.

Una luz verde a la derecha del nombre del canal indica visualmente que el canal ha sido activado para la sesión. Adicionalmente, el módulo cuenta con un botón que permite mantener activa la selección de un canal sin aplicar cambios sobre su configuración previa. Esta funcionalidad es especialmente útil en situaciones donde la configuración del osciloscopio ha sido guardada previamente —por ejemplo, en un dispositivo USB— y se desea cargar directamente, evitando una reconfiguración manual.

Finalmente, desde esta misma ventana se puede acceder a dos módulos principales de operación:

- **Fase de pruebas:** Modo recomendado como paso previo a cualquier sesión. Permite verificar el correcto funcionamiento del banco de trabajo, incluyendo la

activación del tren de pulsos por parte del sistema Arduino y la adecuada recepción de eventos de disparo por parte del osciloscopio.

- **Inicio del programa:** Modo que lanza directamente el experimento programado, asumiendo que todos los componentes han sido previamente validados.

Fig. 15: Ventana de configuración inicial del programa de control del banco de trabajo en LabVIEW.

3.4.4.3. Módulo de Fase de pruebas

La ventana correspondiente a la **fase de pruebas** presenta una interfaz sencilla orientada a verificar que los componentes clave del sistema funcionan correctamente antes de iniciar la sesión de adquisición de datos. La implementación de esta ventana puede verse en la Fig. 16.

En primer lugar, se dispone de un botón cuya función es generar, a través de la tarjeta DAQ, una señal digital que actúa como disparador. Esta señal es recibida por la placa Arduino, que, en respuesta, comienza a emitir el tren de pulsos necesario para controlar los interruptores del sistema durante los ensayos. Esta interacción debe comprobarse cuidadosamente para asegurarse de que la secuencia de activación se produce de forma correcta.

Durante esta fase, es responsabilidad del usuario confirmar que:

- La señal de disparo generada por la DAQ es detectada correctamente por la placa Arduino.
- Ambas señales del tren de pulsos se generan con la forma y temporización esperadas.

- Los dispositivos controlados responden de forma coherente al estímulo.

Si se detecta cualquier anomalía —por ejemplo, la no generación de pulsos, respuestas inconsistentes o retardo anómalo— se deben revisar los parámetros configurados en el módulo común, como el puerto de salida del DAQ o el ancho del pulso. Una vez realizadas las correcciones necesarias, se puede volver a pulsar el botón de "Mandar pulso" para comprobar si el problema ha sido resuelto.

Además de esta funcionalidad principal, se incorpora un segundo botón destinado a verificar el sistema de envío de correos electrónicos, utilizado por el programa para notificar eventos importantes al usuario. Al activarlo, se envía un correo de prueba y, justo debajo, un recuadro informa si el envío ha sido exitoso. Esta prueba es esencial, ya que el sistema de correo puede fallar en ciertas configuraciones, o incluso ser bloqueado por el servidor receptor si los mensajes son clasificados como spam.

Finalmente, desde esta ventana se ofrece acceso directo al modo **Iniciar programa**, lo que implica que el usuario ha llevado a cabo con éxito todas las comprobaciones necesarias para garantizar que el banco de trabajo se encuentra en condiciones óptimas para comenzar las pruebas.

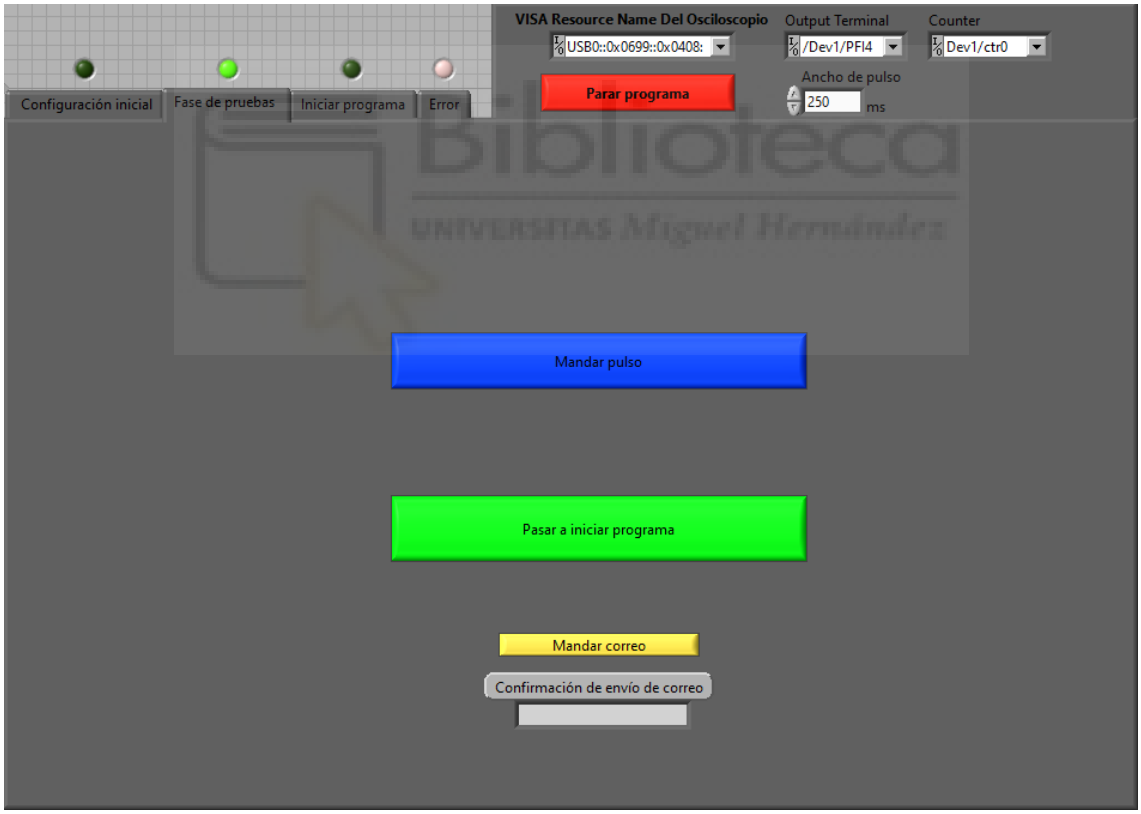


Fig. 16: Ventana de fase de pruebas del programa de control del banco de trabajo en LabVIEW.

3.4.4.3. Módulo de iniciar programa

Una vez finalizadas las configuraciones iniciales y verificado el correcto funcionamiento del banco de trabajo, se procede a ejecutar la sesión de adquisición de

datos a través de una ventana especialmente diseñada para este propósito. La Fig. 17 muestra la ventana implementada en LabView.

En la esquina superior izquierda de la interfaz se encuentra un recuadro destinado a la gestión de repeticiones. En este apartado, el usuario puede especificar el número total de sobrecargas que se desean realizar durante la sesión. A su vez, el mismo recuadro proporciona información en tiempo real acerca del estado del proceso, como el número de repetición actual y el tiempo que ha durado la repetición inmediatamente anterior. El indicador de tiempo de duración de repetición resulta especialmente útil para evaluar cómo influyen distintos factores —como el número de puntos por curva, el número total de curvas adquiridas o el tiempo de espera entre repeticiones— en el rendimiento global del sistema.

En la zona central superior de la ventana se presenta un panel destinado a la configuración de guardado de ficheros, destinado a asegurar la trazabilidad de los resultados y facilitar su posterior análisis. En esta zona se puede definir:

- La ruta de destino para los archivos CSV que se generarán tras cada repetición.
- El nombre base de los archivos.
- La extensión (habitualmente, la extensión '.csv').
- El número de partida, a partir del cual se irá incrementando automáticamente un índice en el nombre de los archivos para diferenciar cada repetición (usualmente, el 0).

En la parte inferior derecha se encuentra un visor gráfico en tiempo real, el cual muestra las curvas obtenidas del osciloscopio tras cada repetición. Esta representación visual proporciona al usuario una segunda capa de verificación sobre la validez de los datos capturados. Asimismo, dado que el sistema de envío de alertas por correo electrónico incorpora automáticamente una captura de pantalla de esta ventana, las gráficas mostradas pueden ser de gran utilidad para diagnosticar remotamente el estado del dispositivo bajo test en caso de error o comportamiento anómalo.

Por último, se disponen los controles de ejecución:

- Un botón para iniciar la sesión de adquisición.
- Un indicador visual que señala si el sistema se encuentra en funcionamiento.
- Un botón de parada segura, que permite al usuario interrumpir la sesión en curso ante la aparición de eventos inesperados o fallos menores, como medida preventiva, pulsar este botón envía tan rápido como sea posible al **módulo de tratamiento de errores**.

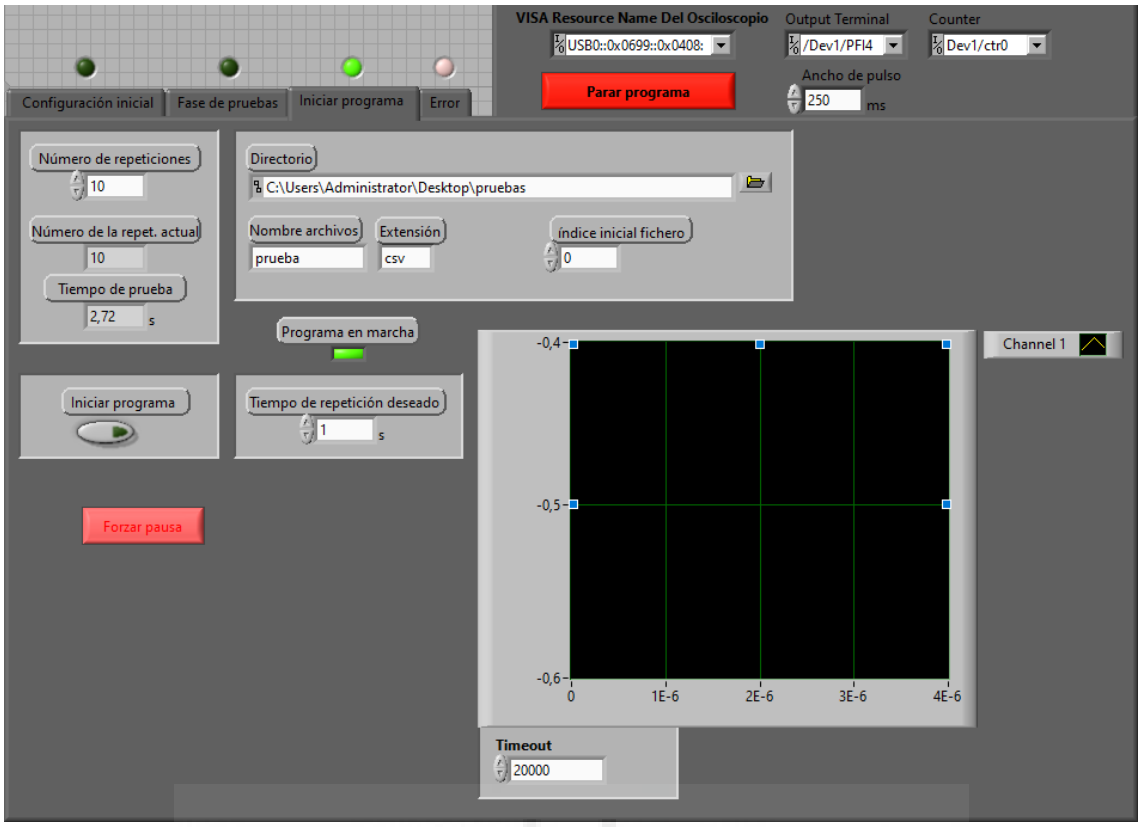


Fig. 17: Ventana de inicio de programa del programa de control del banco de trabajo en LabVIEW.

3.4.4.4. Módulo de Tratamiento de errores

El sistema de adquisición incorpora un módulo específico para la gestión de errores durante la ejecución. Esta funcionalidad está diseñada con el objetivo de garantizar la seguridad del equipo de laboratorio, y ofrecer al usuario una vía clara de actuación en caso de fallo. La ventana asociada este tratamiento de errores se muestra en la Fig. 18.

Esta **ventana de tratamiento de errores** constituye la cuarta interfaz principal del programa, y se accede a ella automáticamente desde cualquiera de las otras fases si se detecta una anomalía que interrumpe el flujo normal del sistema o mediante la pausa de la adquisición de datos, la cual se ejerce de forma manual. Una vez redirigido a esta ventana, el usuario encontrará, en la zona central superior, un código identificador del error producido, el cual pude ayudar a identificar el tipo de fallo en la mayoría de los casos. De forma paralela, el sistema envía automáticamente un correo electrónico al usuario, el cual incluye una captura de pantalla de la ventana activa justo antes de que ocurriera el error. Esta medida permite disponer de un registro visual de la situación previa al fallo, muy útil para realizar un análisis posterior o para asistencia remota.

Desde esta ventana, el usuario dispone de dos opciones de actuación:

- **Rearmar el sistema**: Una vez identificado el error, el usuario puede tratar de corregirlo (por ejemplo, reconectando cables o ajustando parámetros incorrectos). Al pulsar el botón de rearmado, el programa redirige automáticamente a la fase de pruebas, donde se puede comprobar nuevamente el funcionamiento del sistema antes de volver a iniciar la rutina de adquisición. Si se ha seleccionado esta opción, y el flujo del programa llegó a la fase de error mientras se encontraba activa la

rutina de adquisición de datos, el número de iteración del momento del fallo se guarda en memoria para que, una vez se acceda de nuevo a la ventana de **iniciar programa**, pueda reanudarse la rutina con el identificador adecuado, aunque esto puede cambiarse manualmente.

- **Finalizar la sesión**: Si no es posible corregir el error o si se considera oportuno detener la operación, el usuario puede optar por parar el programa de forma controlada mediante el botón de 'Parar Programa'.

Este módulo se activa automáticamente ante diversos tipos de errores, que se pueden clasificar según la fase desde la cual se haya producido el salto:

- **Desde la fase de configuración**: El caso más común de fallo en esta fase es el de conexión con el osciloscopio, generalmente debido a un identificador VISA incorrecto, una configuración mal introducida para ciertos tipos de sondas o un problema físico en la conexión. En esta situación, el usuario puede revisar los parámetros correspondientes o volver a ejecutar el software NI-MAX para verificar la disponibilidad del instrumento.

- **Desde la fase de pruebas**: Se accede a la ventana de errores cuando existe un problema con el DAQ, como una mala configuración del canal de salida o una incompatibilidad en los pulsos enviados al Arduino.

- **Desde la fase de adquisición**: Es la fase más propensa a los errores por su complejidad. Entre este tipo de fallos se encuentran:
  - Pérdida de conexión con el osciloscopio durante la rutina.
  - Mensajes de error generados por el osciloscopio en respuesta a comandos enviados por el programa en situaciones inadecuadas, normalmente cuando el osciloscopio sufre algún tipo de bloqueo.
  - Fallos en la escritura de archivos CSV, ya sea por errores de permisos, saturación del directorio o conflictos de nombres de archivos.

Asimismo, pueden producirse otros tipos de errores en subrutinas secundarias o debido a condiciones excepcionales durante el flujo del programa, como la desconexión accidental del DAQ o interrupciones de alimentación.

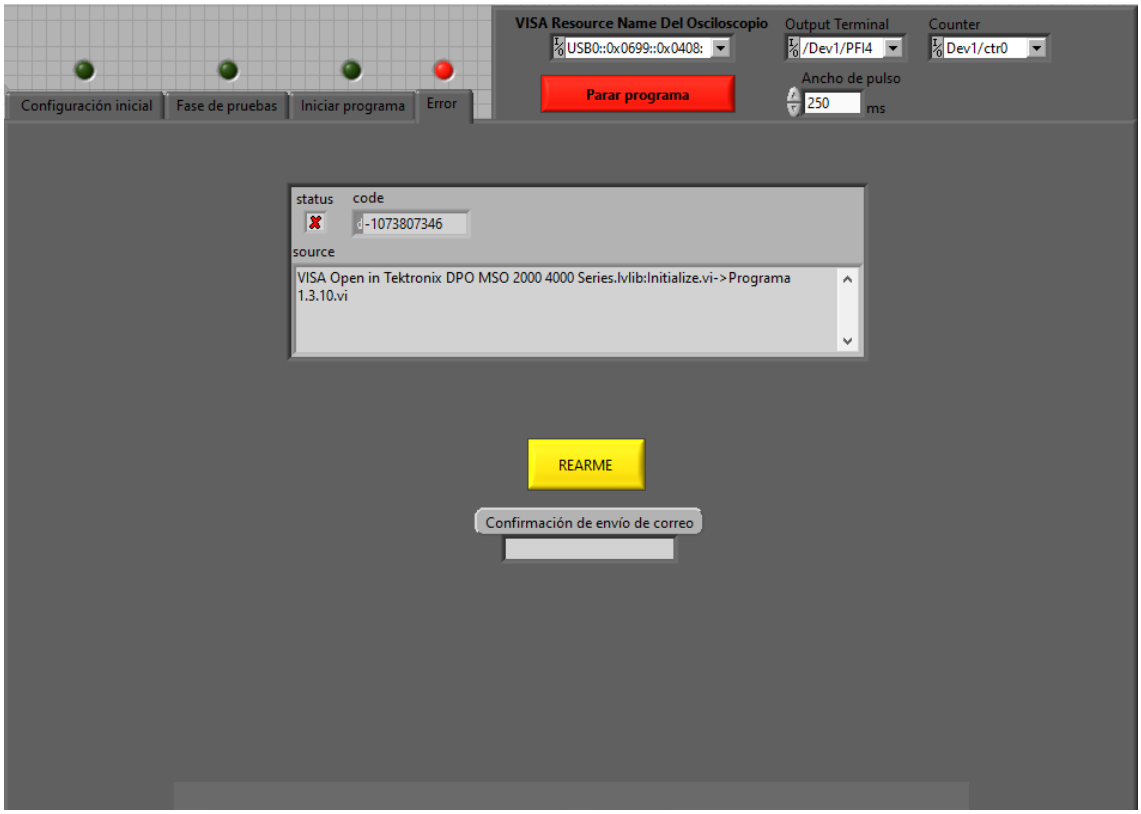


Fig. 18: Ventana de error del programa de control del banco de trabajo en LabVIEW.

3.4.4.5. Estructura interna

Como se comentó en los objetivos del programa, la arquitectura interna del software desarrollado ha sido diseñada siguiendo criterios de robustez, mantenibilidad y eficiencia. Para ello, se ha optado por basar la implementación principalmente en funciones pertenecientes a las librerías oficiales de LabVIEW, proporcionadas por National Instruments. Esta elección responde a varios motivos fundamentales:

- **Fiabilidad y rendimiento**: Las funciones oficiales están optimizadas y ampliamente probadas por la comunidad y, en ciertos casos, por el fabricante, lo que garantiza un comportamiento predecible en la mayoría de los escenarios y una mayor tolerancia a fallos durante la implementación final en comparación con soluciones personalizadas o desarrollos de terceros.

- **Rapidez de desarrollo**: Al emplear bloques funcionales ya disponibles en el entorno de LabVIEW, se ha logrado reducir significativamente el tiempo de implementación, lo que ha permitido centrar los esfuerzos en la lógica de control, la interfaz de usuario y la integración entre dispositivos.

Uno de los aspectos más relevantes de esta decisión se refleja en la forma en que se gestiona la comunicación con la placa Arduino. Aunque existen alternativas para comunicarse directamente con esta a través de protocolos como USB o serie, se ha optado por utilizar el módulo de adquisición de datos NI DAQ como intermediario. Esta decisión se justifica por la razón de que las librerías específicas para DAQ incluidas en LabVIEW permiten una gestión más completa y directa de señales digitales, facilitando tanto la configuración de canales como la temporización de eventos y el manejo de pulsos

Por otro lado, se ha hecho hincapié en el control y validación de las entradas de usuario dentro del programa. Para evitar errores de ejecución derivados de configuraciones incorrectas, se han restringido los rangos de valores que el usuario puede introducir. Estas medidas de validación no solo mejoran la usabilidad del sistema, reduciendo el número de errores humanos, sino que también contribuyen a mantener la integridad del flujo de trabajo y la coherencia de los datos adquiridos. Entre estas restricciones se encuentran las referidas a:

- **Valores negativos:** No se permite la introducción de valores negativos en parámetros como la amplitud, duración o número de repeticiones, ya que no tienen sentido físico en este contexto.

- **Límites:** Se establecen límites superiores e inferiores razonables para evitar la saturación de memoria o la sobrecarga de procesamiento tanto en el DAQ como en el osciloscopio.

El propósito principal del programa desarrollado en LabVIEW es la adquisición automatizada de datos experimentales durante las pruebas de sobrecarga realizadas sobre los DUT, y su posterior almacenamiento estructurado en archivos de tipo CSV. Estos archivos son organizados en un sistema jerárquico de carpetas, cuya estructura responde a criterios definidos (como el tipo de sobrecarga o la familia de LCL), permitiendo así una fácil trazabilidad y un acceso automatizado por parte de los programas de análisis desarrollados en Python.

La finalidad de esta organización es doble:

- Facilitar el procesamiento masivo de datos sin intervención manual, manteniendo un flujo de trabajo reproducible.
- Proporcionar una base de datos estructurada desde la que extraer, analizar y visualizar información crítica sobre el comportamiento del sistema bajo prueba.

Además de los archivos CSV que contienen los datos experimentales de cada repetición, el sistema genera automáticamente archivos de texto complementarios en cada una de las carpetas correspondientes. Estos archivos incluyen información detallada sobre las condiciones específicas bajo las que se han realizado las pruebas, tales como los parámetros de configuración seleccionados en el software, la identificación de las curvas capturadas, las unidades empleadas y, especialmente, la configuración de las sondas de medida. Asimismo, se registra el valor del tiempo de refresco establecido entre repeticiones, un parámetro crítico en determinadas pruebas como la sobrecarga capacitiva, donde es imprescindible asegurar la descarga completa del condensador de salida antes de iniciar una nueva repetición para evitar errores en los resultados o condiciones no seguras de operación.

En cada sesión típica del trabajo se ejecutan alrededor de mil repeticiones de sobrecarga, lo que implica la generación de mil archivos CSV. Cada archivo contiene los datos de una única repetición y está compuesto por un número de columnas igual al de los canales de adquisición activados, que habitualmente es cuatro. Estos canales recogen señales clave del sistema:

- Tensión drenador-surtidor en el JFET.
- Tensión drenador-surtidor en el PMOS.

- Corriente que atraviesa la línea principal del LCL, que es la misma para el JFET y el PMOS.
- Tensión del bus que recorre el LCL.

Cada curva registrada cuenta con una resolución temporal de un millón de puntos, lo que significa que cada archivo contiene cuatro millones de valores en coma flotante. Esta densidad de información es necesaria para capturar con precisión eventos transitorios de muy corta duración, como la respuesta transitoria del sistema ante la propia sobrecarga. Esto implica que, como consecuencia del elevado volumen de datos generado —aproximadamente cuatro mil millones de datos por sesión—, los archivos se almacenan directamente en discos duros de alta capacidad, dado que el uso de almacenamiento temporal o memoria interna del sistema resultaría insuficiente. Esta solución garantiza una correcta conservación de los datos y evita cuellos de botella durante el proceso de adquisición.

Posteriormente, los programas de análisis desarrollados en Python acceden a las carpetas generadas y procesan los archivos de forma automatizada, extrayendo de cada uno la información necesaria. Los resultados se almacenan en estructuras jerarquizadas, generalmente en forma de diccionarios anidados.

4. DESARROLLO DE HERRAMIENTAS DE ANÁLISIS DE DATOS

El segundo de los objetivos principales de este trabajo es el desarrollo de herramientas de análisis que permitan interpretar adecuadamente los datos generados durante los ensayos de envejecimiento acelerado. El banco de trabajo por sí solo no resulta suficiente si no se cuenta con un conjunto robusto de utilidades capaces de procesar, analizar y visualizar las curvas obtenidas. Estas herramientas son esenciales para detectar eventos significativos como sobrecargas, corregir desviaciones sistemáticas, reducir el impacto del ruido en la señal y, en definitiva, facilitar una evaluación técnica rigurosa del comportamiento de los dispositivos sometidos a prueba.

Junto con el tratamiento de los datos generados en el laboratorio, se considera la necesidad de incorporar herramientas de análisis para los datos de caracterización de dispositivos FET previamente obtenidos y aportados por la Universidad de Valencia. Como apoyo transversal al resto de herramientas, se ha desarrollado una programa con interfaz gráfica de usuario, concebida para facilitar el desarrollo del software de análisis a través de aportar una manera sencilla y adaptada de cargar y representar datos en bruto.

4.1. Análisis de eventos de sobrecarga

Analizar las curvas de los JFET y PMOS dentro del grupo incorporado en el LCL a lo largo de las repeticiones nos permite hacernos una idea de la degradación del componente, ya que cada evento de sobrecarga produce un estrés térmico y eléctrico que puede afectar su desempeño y, a la larga, su fiabilidad. Al estudiar las curvas de tensión y corriente ante cada sobrecarga, se pueden identificar cambios en el comportamiento transitorio, como un aumento en el pico de corriente o en el comportamiento de limitación, como el tiempo de desconexión.

4.1.1. Introducción y objetivos

Para realizar esta tarea, se deben realizar un programa con una serie de funciones que se adapten a los tipos de curvas a analizar. Con respecto a los objetivos introducidos en el apartado 1.5, se expanden y se desarrollan en profundidad los objetivos referentes al programa de análisis de sobrecargas y las funciones que contiene:

- Las funciones deberán ser capaces de detectar y corregir posibles errores de offset introducidos por las sondas de medida, evitando así un sobredimensionamiento de los datos.

- Estas funciones deben ser lo suficientemente robustas como para analizar todo el conjunto de curvas obtenidas, incluyendo aquellas que presenten variaciones dentro de los márgenes admisibles, sin desvirtuar las formas características esperadas.

- Deben incorporar mecanismos que permitan mitigar el efecto del ruido en las señales, evitando que éste afecte negativamente al resultado del análisis.

- Las funciones deberán ser compatibles con una herramienta de visualización de curvas, que se desarrollará de forma paralela, con el objetivo de facilitar tanto la depuración de errores como la validación de los resultados.

- Deberán optimizarse para alcanzar un nivel de eficiencia que garantice tiempos de procesamiento razonables y adecuados al volumen de datos esperado.

- Los programas deberán diseñarse siguiendo principios de modularidad y escalabilidad, de modo que futuras ampliaciones o adaptaciones a nuevos tipos de curvas o condiciones de medida puedan implementarse de forma sencilla y eficiente.

- Las funciones deberán ser capaces de adaptarse dinámicamente al análisis de curvas tanto de corriente como de tensión, correspondientes a dispositivos JFET y PMOS, ajustando de manera automática ciertos parámetros de análisis si fuese necesario.

- La salida de los programas consistirá en una estructura de datos organizada y flexible que facilite su posterior tratamiento en análisis complementarios.

Para este trabajo, se consideran cuatro parámetros fundamentales de las curvas de sobrecarga en un LCL introducidos en el apartado 1.2, pero adaptados a los tres tipos de sobrecarga tratados:

- **Pico de sobrecorriente y pico de tensión:** Valores los cuales permiten identificar el esfuerzo instantáneo máximo al que se somete el dispositivo, comprobando que no se excedan sus límites de operación segura. A efectos de analizar este parámetro en las curvas, será suficiente con encontrar computacionalmente el valor máximo de las señales, buscando en ciertas zonas donde se sabe, con seguridad que se encontrarán estos valores. Encontrar la zona de pico es especialmente fácil ya que la obtención de las curvas mediante es osciloscopio se realizó empelando el *trigger* en las señales de activación de la placa y del IGBT en los casos que corresponda, como estas señales tienen una relación de tiempos definida en el programa cargado a la tarjeta de Arduino, el instante donde ocurre la sobrecarga está totalmente determinado.

- **Tiempo de desconexión:** Este parámetro permite determinar el perfil de carga energética aplicada al dispositivo y sirviendo de ventana de integración para otros parámetros derivados. En el ámbito del trabajo, en lugar de tomar como referencia de inicio y final el valor de la corriente de limitación, por simplicidad, se tomará como inicio el punto en el cual se produce el aumento repentino de corriente (inicio de la sobrecarga) y como punto final, el cual se produce justo cuando la corriente baja cero. En el caso de las curvas de sobrecarga capacitiva, este se definirá como el tiempo entre el que se carga totalmente el condensador de la carga y se definirá como '**tiempo de carga**'.

- **Corriente media de limitación:** Este parámetro permite comprobar la respuesta del sistema ante eventos de sobrecarga y es proporcional a la energía que absorberán los componentes de protección. Debido a la naturaleza de los tres tipos de sobrecarga tratados en este trabajo, este valor solo se obtendrá para sobrecargas resistivas y de cortocircuito. Este valor se obtendrá según la expresión (1), donde $I(t)$ es la corriente que atraviesa el dispositivo FET, el intervalo de integración deriva del **tiempo de desconexión**, el cual se reduce de forma proporcional por

ambos extremos para evitar los transitorios inicial y final de la sobrecarga, donde la corriente toma valores muy distintos al de limitación.

$$I_{lim} = \frac{\int_{t0'}^{tf'} I(t)dt}{tf' - t0'} \quad (1)$$

- **Energía disipada:** Este valor, en el ámbito del trabajo, representa la cantidad total de energía que el dispositivo disipa durante el evento de sobrecarga. Esta energía se obtendrá según la expresión general (2), donde $V_{ds}(t)$ es la tensión entre el drenador y el surtidor del dispositivo FET e $I(t)$ la corriente que lo atraviesa, el intervalo de integración coincide con el **tiempo de desconexión** determinado con anterioridad para las sobrecargas resistiva y de cortocircuito, y con el **tiempo de carga** para la sobrecarga capacitiva.

$$E_{lim} = \int_{t0}^{tf} V_{ds}(t) * I(t)dt \quad (2)$$

4.1.2. Ajuste del filtrado de ruido

Como ya se indicó en el apartado 2.3.1, el ruido es una de las principales fuentes de error que dificultan el análisis de las curvas por medio de software, en consecuencia, se ha seleccionado el filtro de Savitzky–Golay para reducir la influencia de este error. Para realizar un ajuste eficiente del filtro utilizado en estos programas, es preciso establecer de forma adecuada sus dos parámetros: el tamaño de la ventana y el grado del polinomio ajustado localmente en cada intervalo de datos.

De estos dos parámetros, el **tamaño de la ventana** es el que ejerce una influencia crítica sobre el comportamiento del filtro. El número de puntos considerados en cada ventana determina directamente el nivel de suavizado: una ventana de mayor tamaño permite reducir con más eficacia el ruido de alta frecuencia, a costa de un posible deterioro en la resolución de detalles finos de la señal. Por el contrario, un tamaño de ventana reducido conserva mejor las variaciones locales de la curva, pero atenúa el ruido en menor medida.

Además, existe un compromiso entre la calidad de la señal filtrada y el coste computacional. El tamaño de la ventana afecta directamente al número de operaciones necesarias para aplicar el filtro: a mayor tamaño, mayor número de convoluciones y, por tanto, mayor tiempo de procesamiento, esta relación resulta ser una curva muy suave, prácticamente lineal (véase Fig. 19). Por este motivo, debe buscarse un equilibrio adecuado que permita reducir eficazmente el ruido sin provocar un incremento excesivo del tiempo de cálculo, lo cual resulta especialmente importante cuando se trabaja con grandes volúmenes de datos como en el caso de este trabajo.

Cabe destacar que, como ya se ha comentado, un tamaño de ventana excesivamente grande puede degradar la calidad de la señal analizada. Al abarcar una región demasiado amplia, el polinomio ajustado tiende a "aplanar" o "suavizar en exceso" la curva, lo que puede llevar a la pérdida de información clave, como pequeños picos, caídas rápidas o

transitorios importantes en la forma de la señal. Estos detalles son frecuentemente críticos en el análisis de curvas de dispositivos electrónicos en el ámbito del presente trabajo, donde la identificación precisa de eventos de sobrecarga depende de la correcta conservación de estas estructuras.

En cuanto al **grado del polinomio**, su función principal es definir la "flexibilidad" del ajuste local. Un polinomio de mayor grado puede capturar cambios más rápidos en la señal, pero también corre el riesgo de ajustar en exceso el ruido. Por ello, en la práctica, se suelen emplear polinomios de bajo grado (generalmente entre 2 y 4), manteniendo el foco del ajuste en la ventana como el principal mecanismo de control del filtrado.

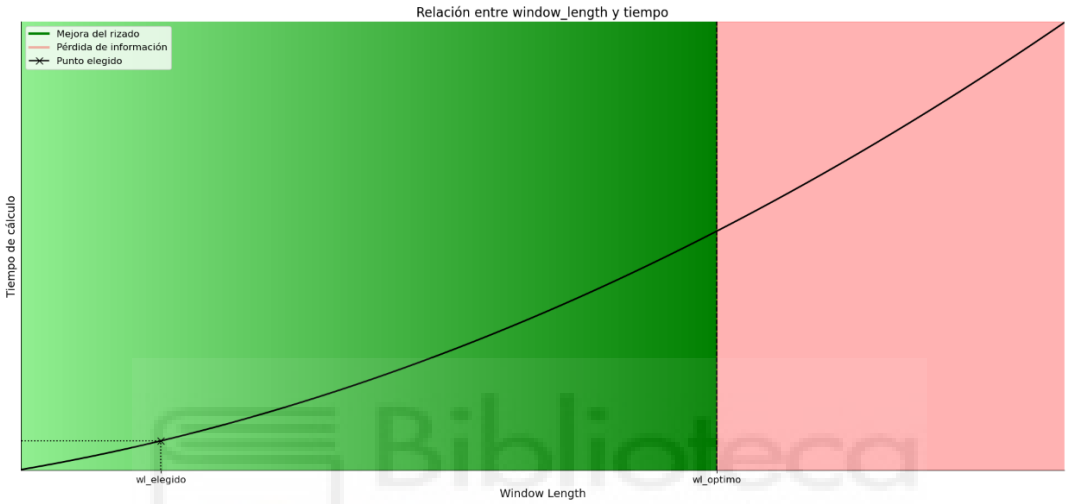


Fig. 19: Curva aproximada de tiempo de cálculo frente al tamaño de ventana en el filtro de SG, se destacan dos zonas searadas por un punto óptimo de tamaño de ventana, a partur del cual, el suavizado distorsiona los detalles importantes de la señal original.

Con el objetivo de encontrar los valores óptimos de los parámetros de filtrado en el método de SG, se ha llevado a cabo un análisis sistemático basado en la generación de distintas curvas —obtenidas a lo largo del trabajo—, las cuales han sido suavizadas utilizando diferentes tamaños de ventana, manteniendo fijo el grado del polinomio en 3. Esta elección responde al compromiso habitual en el filtrado de señales, donde un polinomio de tercer grado ofrece suficiente flexibilidad para modelar transitorios locales sin incurrir en sobreajustes del ruido.

Se han empleado curvas representativas de los tres tipos de sobrecarga analizados en el trabajo, ya que presentan características dinámicas distintas y, por tanto, diferentes exigencias de filtrado. El objetivo fundamental es alcanzar una calidad de suavizado aceptable, minimizando el ruido presente en las señales, pero sin comprometer de forma significativa el tiempo de procesamiento. Este criterio es esencial debido a que los programas desarrollados están destinados a analizar miles de curvas de forma automática, y un filtrado excesivamente lento podría multiplicar el tiempo total del análisis, afectando negativamente a la eficiencia general del sistema. En este tipo de curvas se ha puesto el foco en la integridad de los intervalos de desconexión o carga ya que son los más sensibles al aplicar este tipo de filtro.

En primer lugar, se analizaron las curvas correspondientes a eventos de **sobrecarga resistiva**, ya que, junto a las de cortocircuito, representan los casos más exigentes en cuanto al diseño del filtrado. Estas señales presentan transitorios muy rápidos en respuesta a la condición de fallo, lo que exige un filtrado suficientemente selectivo: capaz de atenuar el ruido, pero sin distorsionar o eliminar los detalles críticos de la dinámica del dispositivo.

Analizando específicamente las curvas de potencia disipada en el JFET —las cuales son las más indicadas para la obtención del intervalo de limitación—, se observa que un tamaño de ventana de 99 puntos resulta muy adecuado para señales de este tipo con aproximadamente un millón de puntos. Con esta configuración se consigue una reducción significativa del ruido, que permitirá realizar un análisis más preciso y robusto de las curvas, especialmente en la detección de intervalos de limitación. Como efectos secundarios del filtrado con esta ventana, se detecta una ligera atenuación del valor pico de la sobrecarga (véase Fig. 20), así como una transición no real hacia valores negativos en la potencia (véase Fig. 21), efectos esperables derivados del suavizado y que, sin embargo, no afectan al objetivo principal del análisis, dado que el intervalo de limitación permanece bien definido y claramente identificable.

Por otro lado, utilizando un tamaño de ventana muy grande, por ejemplo, de 9999 puntos, si bien el filtrado logra prácticamente eliminar todo el ruido visible en la señal (véase Fig. 22), se produce una pérdida notable de información importante. En particular, el transitorio asociado a la activación de la protección de sobrecarga queda tan suavizado que resulta imposible delimitar con precisión el inicio y el final del intervalo de limitación. Este exceso de suavizado supone una pérdida crítica de la información necesaria para el análisis de fallos, por lo que tamaños de ventana tan grandes deben descartarse en aplicaciones donde se requiera conservar la integridad de transitorios rápidos. El efecto de la pérdida de la forma del transitorio comentado se ilustra en detalle en la Fig. 21.

De forma análoga, el análisis de las curvas correspondientes a **condiciones de cortocircuito** sigue un planteamiento muy similar al descrito para la sobrecarga resistiva. Debido a la similitud en la forma general de las señales —caracterizadas ambas por un transitorio muy rápido seguido de una fase de limitación de corriente—, las exigencias en cuanto a filtrado son prácticamente equivalentes. Así, el tamaño de ventana seleccionado para la sobrecarga resistiva resulta igualmente adecuado para la caracterización de los cortocircuitos.

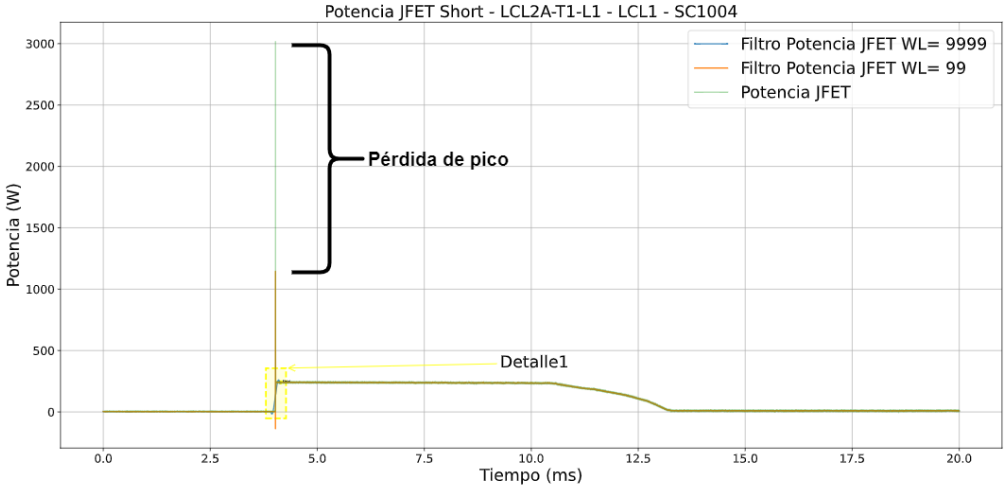


Fig. 20: Curva de potencia disipada en el JFET en pruebas de sobrecarga sobre un LCL (verde), y curvas derivadas de este mediante el filtro de SG (naranja y azul).

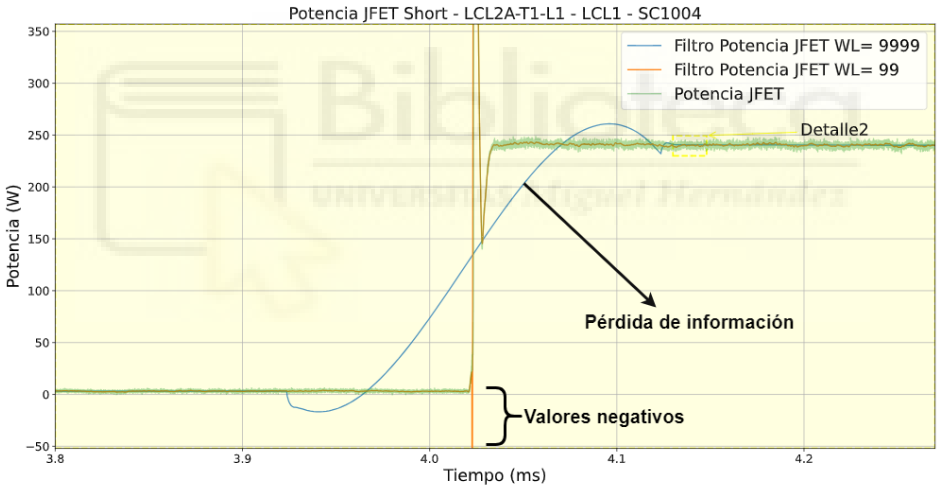


Fig. 21: Detalle 1 (en el momento de la sobrecarga) de la Fig. 20.

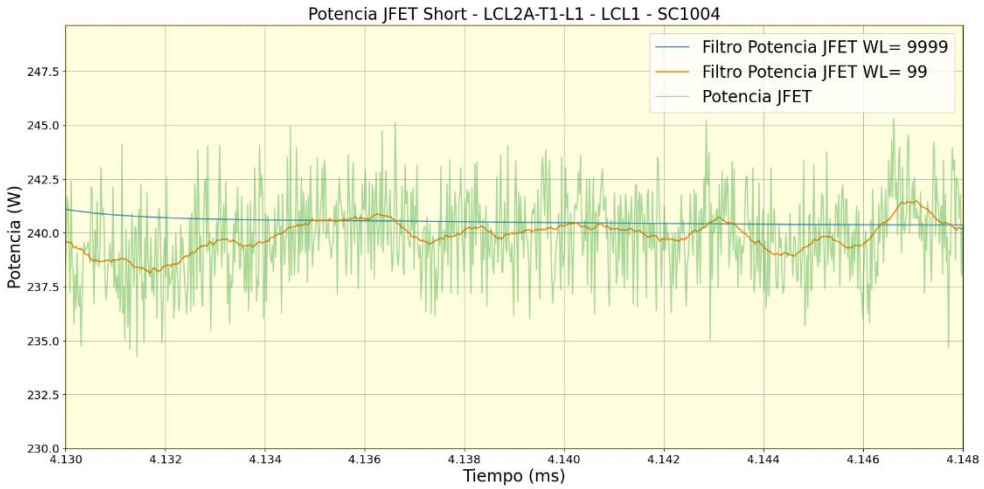


Fig. 22: Detalle 2 de la Fig. 21.

En segundo lugar, se ha aplicado el mismo proceso de optimización de parámetros de filtrado a las **curvas de arranque capacitivo**. El objetivo, de nuevo, es determinar un tamaño de ventana adecuado que permita una correcta obtención del intervalo de desconexión definido para este tipo de pruebas. En este escenario, las curvas de potencia disipada no resultan ser las más adecuadas para la identificación del intervalo de carga. Esto se debe a que, en ciertas combinaciones de JFET y PMOS, se observan oscilaciones agresivas tanto en corriente como en tensión durante el arranque, cuyo efecto se ve amplificado en la curva de potencia y que dificultan encontrar el inicio del intervalo de carga. Estas oscilaciones no son de igual amplitud en todos los grupos de LCL, en la Fig. 23 se puede observar la comparativa de algunas curvas de potencia de JFET incorporados en LCL de clase 10. El hecho de que las oscilaciones sean poco consistentes incluso en grupos de la misma clase implica que la elección de un tamaño de ventana del filtro unificado para todos no sea una tarea trivial, por este motivo, en el caso del arranque capacitivo se ha optado por emplear como referencia la curva de intensidad, más fácilmente analizable ante estas oscilaciones al aplicar el filtro.

A la hora de poder analizar las curvas de corriente del arranque capacitivo, debido a la presencia de un condensador en paralelo con la carga, la dinámica de la curva de intensidad presenta características diferenciadas: la corriente no presenta transitorios tan bruscos, sino una subida progresiva hasta un máximo, seguida de una bajada más suave a medida que el condensador se va cargando. Esta morfología facilita notablemente el proceso de filtrado, ya que permite una mayor flexibilidad en la elección del tamaño de ventana sin riesgo de pérdida severa de información relevante.

Aplicando un tamaño de ventana de 99 puntos, tal como puede verse en la Fig. 24, existe una reducción de las oscilaciones muy adecuada, incluso en las zonas donde persisten ciertas oscilaciones residuales. La forma general de la curva se mantiene fiel al comportamiento esperado, sin producirse fenómenos indeseados como desplazamiento de picos, pérdidas de pendiente o aparición de artefactos. Este ajuste permite, por tanto, delimitar de forma precisa el intervalo de carga.

Al aumentar el tamaño de ventana hasta aproximadamente 20.000 puntos, se consigue una mejora aún más notable en la calidad visual de la señal, logrando una casi completa eliminación de las oscilaciones. Sin embargo, esta mejora viene acompañada de una incipiente pérdida de información, especialmente visible en la fase de descenso de la curva, cuando el condensador comienza a estar completamente cargado (véase Fig. 24 y Fig. 25). Estos efectos, de nuevo, indican que existe un límite práctico para el aumento del tamaño de ventana en este tipo de curvas.

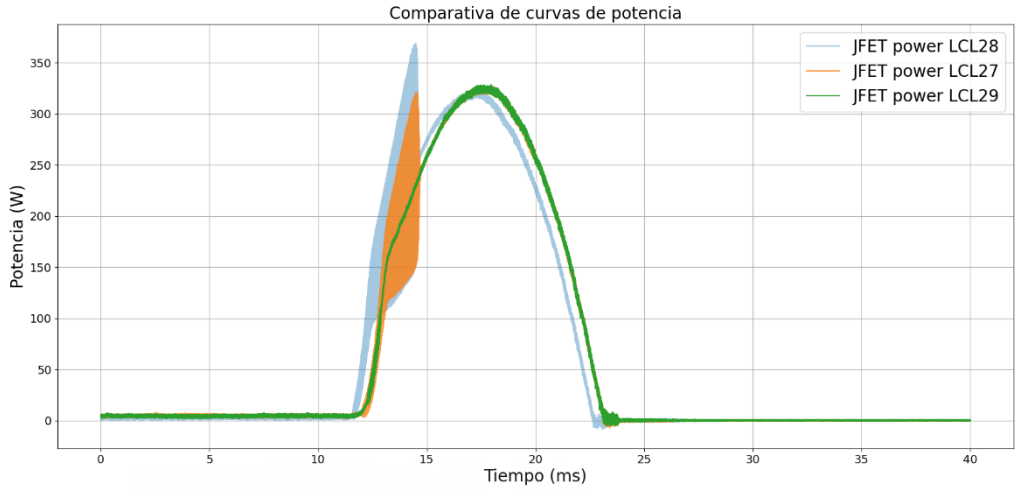


Fig. 23: Comparativa de curvas de potencia entre algunos de los JFET incorporados en LCL de clase 10 para sobrecargas capacitivas.

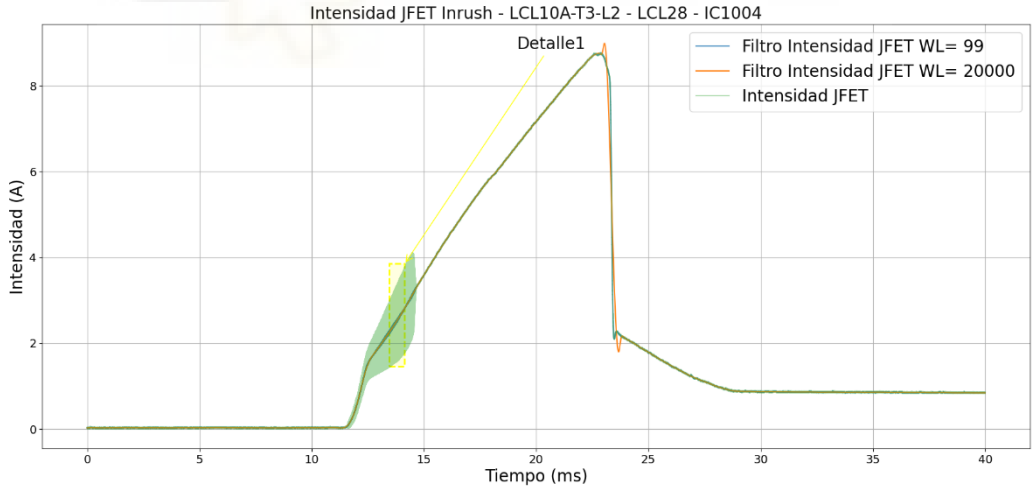


Fig. 24: Curva de potencia disipada en el JFET en pruebas de sobrecarga capacitiva sobre un LCL (verde), y curvas derivadas de este mediante el filtro de SG (naranja y azul).

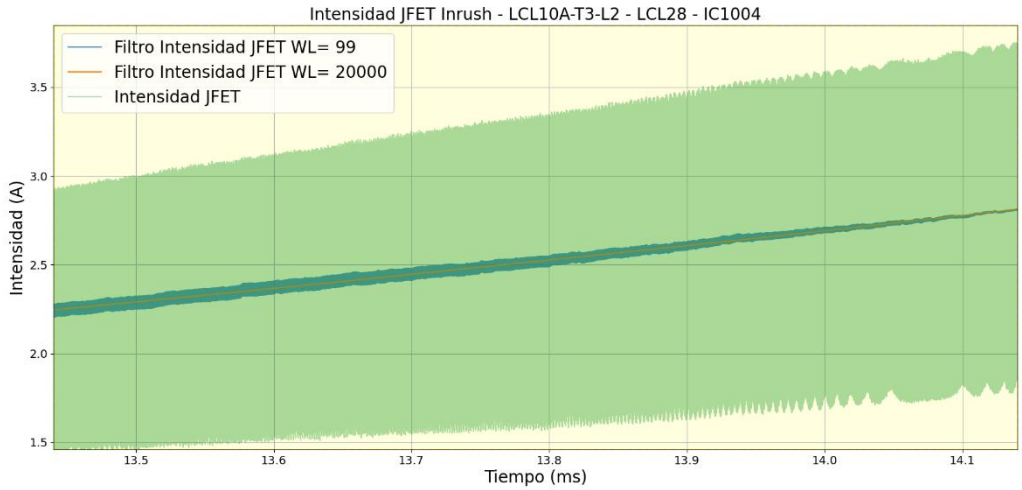


Fig. 25: Detalle 1 (comportamiento ante oscilaciones de intensidad) de la Fig. 24.

4.1.3. Ajuste del error de *offset*

Tal como se señaló en el apartado 2.3.1, el error de offset puede manifestarse de diversas formas, a la hora de analizar las cuevas de sobrecarga obtenidas durante el trabajo, este error resulta especialmente molesto, ya que desvirtúa el valor de pico de corriente y de tensión, dificulta la forma de encontrar los diferentes tiempos de desconexión, y provoca error a la hora de determinar la energía disipada en los transistores. Esta corrección se realizará de forma digital, durante el procesamiento de los datos, para asegurar mediciones precisas y un análisis fiable de las curvas características.

Esta corrección consiste en restar a toda la curva un valor constante, previamente calculado antes de proceder con los cuatro tipos de análisis realizados sobre las curvas de sobrecarga. Dicho valor de *offset* representa una desviación constante de la señal respecto a su referencia ideal, la cual puede introducir errores significativos si no se compensa adecuadamente. Un pequeño error constante en la señal se traduce en una acumulación considerable de energía ficticia o corriente errónea, lo cual podría llevar a conclusiones incorrectas sobre el comportamiento energético del dispositivo o su adecuación a ciertas condiciones de operación.

Por otro lado, existen ciertos casos en los que esta corrección de *offset* no se aplica, como ocurre durante el análisis del intervalo de carga en el arranque capacitivo. En esta fase, se utilizan ventanas de datos móviles y se analizan las variaciones relativas entre ellas. Dado que el *offset* representa un desplazamiento uniforme en toda la curva, no afecta al análisis diferencial entre ventanas consecutivas. Es decir, aunque la curva se desplace verticalmente, la forma relativa entre segmentos permanece inalterada, por lo que no se ve comprometida la integridad del análisis.

Cabe señalar que estas variaciones de *offset* no son constantes, sino que difieren entre todas las combinaciones posibles de dispositivos JFET y PMOS, e incluso pueden variar —más notablemente— entre distintas repeticiones del mismo montaje y combinación de

dispositivos, este comportamiento puede verse en la Fig. 26 y la Fig. 27 para curvas de cortocircuito.

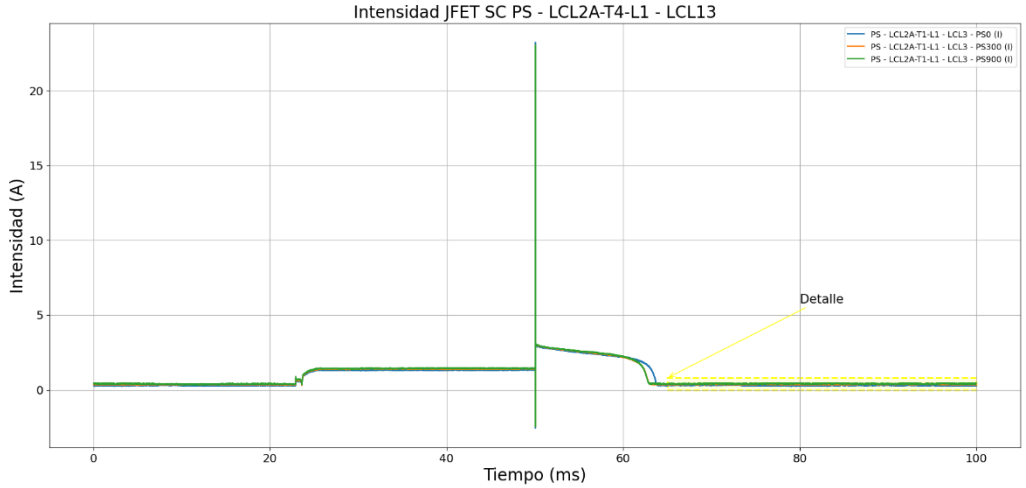


Fig. 26: Representación de la curva de intensidad del JFET del grupo LCL3 en distintas repeticiones de la misma prueba de cortocircuito. Se ha aplicado a todas las curvas el filtro de SG.

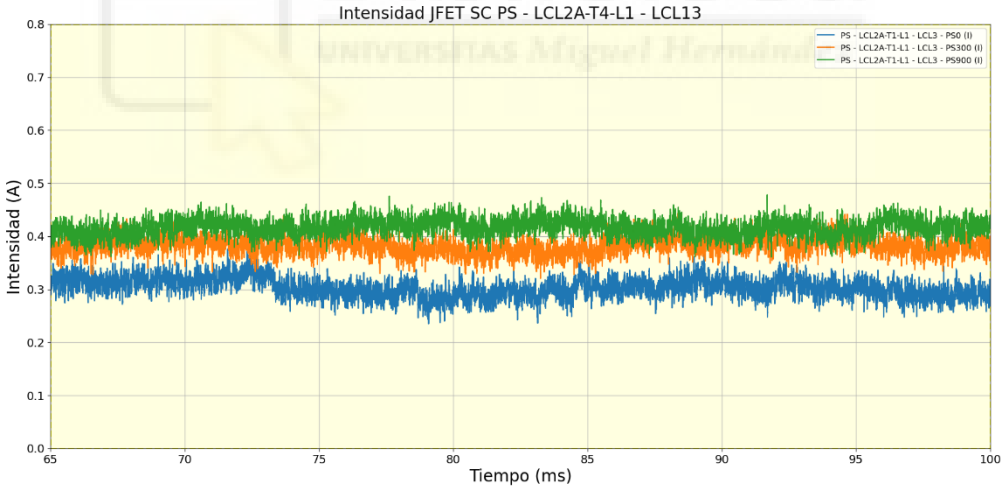


Fig. 27: Detalle de la Fig. 26.

El método de corrección digital de *offset* comienza con una fase previa de caracterización, en la cual se lleva a cabo una inspección detallada de múltiples curvas representativas de tensión y corriente, tanto para dispositivos PMOS como JFET, y para los tres tipos de sobrecarga considerados en el estudio. Esta inspección permite identificar tramos específicos de las señales en los que, debido a las condiciones de configuración del *set-up* experimental, se espera que la tensión o la corriente sean prácticamente nulas. Estas regiones, que se consideran como referencias de estado inactivo o reposo, son fundamentales para detectar y cuantificar los desplazamientos sistemáticos que

constituyen el *offset*. En la práctica, se toman los valores de la curva en ese intervalo y se obtiene la media de estos valores. En la Tabla 2 se muestran los diferentes porcentajes sobre el tiempo de las curvas donde se toma la muestra del offset en los diferentes tipos de sobrecarga, como ejemplo, en la ya comentada Fig. 26, para curvas de cortocircuito resulta evidente escoger una ventana de datos que comprenda el tramo inicial o final, si se escoge una ventana desde el 96% hasta el 96.5% de los datos, al contar con un millón de puntos, esta ventana constará de 5000 puntos.

Tabla 2: Intervalos sobre el eje temporal donde se extrae la muestra del offset en los distintos tipos de sobrecarga.

Tipo de sobrecarga	**Intervalo porcentual sobre el eje temporal (% de la cantidad de puntos de la curva)**					
	$V_{ds}JFET$		$V_{ds}PMOS$		**I**	
	inicio	**final**	**inicio**	**final**	**inicio**	**final**
Sobrecarga resistiva	0	1	0	1	96	96.5
Cortocircuito	40	40.5	40	40.5	0	1
Sobrecarga capacitiva	96	96.5	96	96.5	0	1

Una vez identificadas estas ventanas de datos representativas, se seleccionan de forma cuidadosa y se introducen como referencia dentro del software de análisis. Como se ha comentado estas ventanas sirven como base para calcular el valor medio de la señal en una zona donde idealmente no debería existir actividad eléctrica significativa.

Durante la ejecución del análisis en cada uno de los programas desarrollados, se aplica esta corrección cuando resulta necesario. La selección de la ventana de referencia adecuada se realiza en función del tipo de sobrecarga y del tipo de señal analizada (tensión o corriente), así como del tipo de dispositivo (JFET o PMOS). Una vez seleccionada la ventana correspondiente, se calcula la media de los datos contenidos en ella, y dicho valor se sustrae de toda la curva, eliminando así el *offset* de manera efectiva.

En la Fig. 28 se representa el diagrama de flujo del proceso de corrección del error de *offset* descrito a lo largo del apartado, en este caso, por simplicidad el diagrama de flujo se ha representado para las curvas de sobrecarga capacitiva.

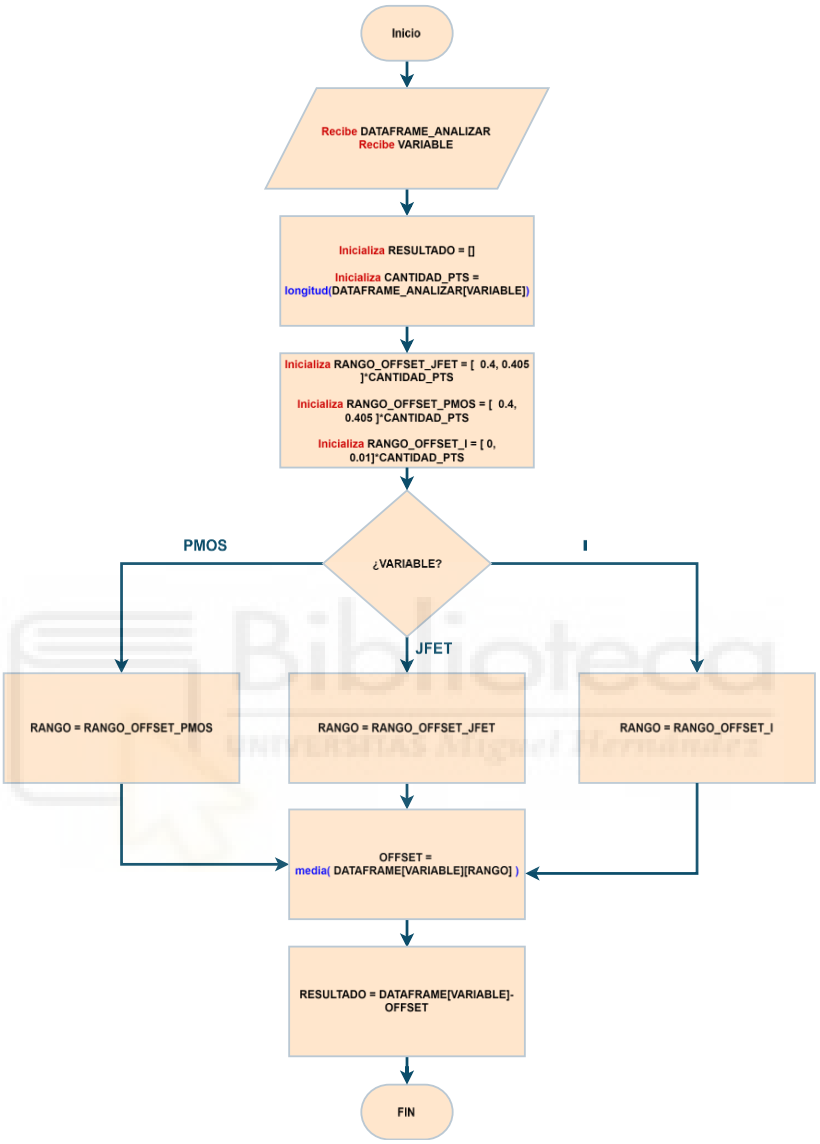


Fig. 28: Diagrama de flujo general de la detección y corrección del error de offset para sobrecargas capacitivas.

4.1.4. Funciones

Dado el elevado volumen de datos generado durante las campañas experimentales —que han sido realizadas a lo largo de varios meses y abarcando múltiples combinaciones de dispositivos JFET y PMOS bajo diferentes tipos de sobrecarga—, se ha desarrollado un programa general de análisis que integra de forma modular varias funciones orientadas a los cuatro parámetros fundamentales comentados en el apartado 4.1.1, pico de corriente/sobretensión, tiempo de desconexión/carga, corriente media de limitación y energía disipada. El programa principal debe coordinar estas funciones según las

preferencias del usuario y devolver unas estructuras jerarquizadas para organizar la información.

Las funciones que integran el análisis de eventos de sobrecarga se pueden ver en el Anexo 6.

4.1.4.1. Estructura de organización de los archivos

Con el fin de facilitar esta automatización y asegurar la trazabilidad de los datos, se propone una organización jerárquica de los archivos CSV generados por los ensayos del banco de trabajo. Como aproximación práctica y eficiente, se establece una estructura de carpetas con los siguientes niveles:

1. **Carpeta principal para cada tipo de sobrecarga**: Para comenzar, se crean tres carpetas principales, cada una correspondiente a un tipo de sobrecarga (capacitiva, resistiva o cortocircuito).

2. **Subcarpetas mediante etiquetas normalizadas de los diferentes grupos**: Dentro de cada carpeta de sobrecarga, se encuentran varias subcarpetas para cada celda a la que se le realiza el test. Debido a que estas celdas son una combinación de JFET y PMOS, para simplificar la referencia a estas combinaciones sin depender de los identificadores individuales de cada componente, se utiliza una nomenclatura basada en el formato Referencia + LCL + número, donde número varía del 1 al 32. Esta codificación permite distinguir fácilmente los grupos de dispositivos de forma abstracta y organizada. En el Anexo 1 se pueden encontrar las referencias empeladas para cada combinación, un ejemplo de esta nomenclatura es 'LCL2A-T1-L1 - LCL1'.

3. **Nombres de los archivos CSV**: Finalmente, dentro de cada subcarpeta se encuentran los archivos CSV que contienen los datos de cada repetición, estos tendrán una nomenclatura igual a la de la carpeta que los contiene, añadiéndole un número distintivo que representa el número de la repetición a la que corresponde —en las pruebas desarrolladas en el trabajo, desde 0 hasta 999—. Como ejemplo de un nombre de archivo, siguiendo el ejemplo introducido anteriormente sería 'LCL2A-T1-L1 - LCL1 - 0'.

Uno de los principales desafíos técnicos a resolver es el tratamiento eficiente de la gran cantidad de datos generados en cada campaña. Dado que cada campaña de sobrecarga consta de, generalmente, mil repeticiones experimentales, y cada repetición genera un archivo en formato CSV con cuatro columnas de datos: tensión JFET, tensión PMOS, corriente común y tensión conjunta, teniendo en cuenta que cada una de estas columnas contiene un millón de puntos medidos con precisión de punto flotante (al menos dos decimales), esto se traduce en archivos individuales con un peso medio de 26 MB, lo que supone un volumen total de aproximadamente 26 GB por campaña.

Ante esta magnitud de datos, la primera decisión clave en el diseño del programa es el uso de estructuras de datos optimizadas en memoria, en particular los objetos *dataframe* de la biblioteca 'pandas', ampliamente utilizados en análisis de datos científicos e industriales. Como se expuso en el apartado 2.3, estos objetos permiten manipular grandes volúmenes de información con una eficiencia razonable en cuanto al consumo de memoria y tiempo de procesamiento.

No obstante, incluso con el uso de *dataframe*, cargar los mil archivos simultáneamente en memoria resulta inviable en la mayoría de los sistemas, debido al alto consumo de recursos que implicaría. Para solventar esta limitación, se incorpora en el programa una funcionalidad que permite al usuario definir en cuántos bloques desea dividir la carga y análisis de los archivos CSV. Por ejemplo, si el usuario especifica que desea procesar los datos en diez bloques, el programa cargará secuencialmente cien archivos por bloque. Cada grupo de archivos se carga en una lista de *dataframes*, se analiza individualmente dentro de ese lote, y los resultados se almacenan de forma incremental, permitiendo continuar con el siguiente bloque sin necesidad de mantener todos los datos anteriores en memoria.

Este enfoque por bloques de procesamiento ofrece la ventaja de permitir ejecutar el análisis en equipos con recursos de hardware limitados. De esta forma, es posible realizarlos independientemente de la capacidad del ordenador utilizado, simplemente adaptando la cantidad de archivos que se cargarán por bloque. Además, es importante señalar que una gran cantidad de bloques —lo que implica una pequeña cantidad de ficheros cargados a la vez, pero también comunicarse más veces con el disco duro— puede aliviar en mayor medida la carga en memoria, sin embargo, resulta considerablemente más lento, debido a que antes de iniciar la comunicación plena con dispositivos de almacenamiento externo de gran capacidad, es usual que haya un período de inicialización de la comunicación que puede tardar varios segundos en algunos casos.

4.1.4.2. Descripción general del programa

El programa permite al usuario seleccionar de forma flexible cuáles de las cuatro características (tensión y corriente pico, tiempo de desconexión/carga, energía disipada y corriente media de limitación) desea analizar. Esta selección determina qué análisis se ejecutarán en cada ciclo de carga. Cabe destacar que, debido a la interdependencia lógica entre parámetros, los análisis de energía disipada y corriente media de limitación requieren obligatoriamente haber ejecutado previamente el análisis del intervalo de desconexión/carga, ya que este determina el rango temporal sobre el que se integran dichos parámetros.

Tal como puede verse en la Fig. 29, el objetivo final de cada lote procesado es generar un diccionario estructurado de forma sencilla y eficiente. En este diccionario, cada clave representa una combinación específica de dispositivos, en el final de la cadena se utiliza el identificador normalizado LCL + número —sin incluir la referencia directa a los modelos individuales de JFET o PMOS por simplicidad—, y su valor asociado es otro *dataframe* que contiene tantas columnas como características hayan sido seleccionadas para analizar, además de una cadena de texto a modo de información, que contendrá la misma información que el fichero de texto que generaba el propio banco de trabajo, aportando información útil sobre las condiciones en las que se obtuvieron los datos.

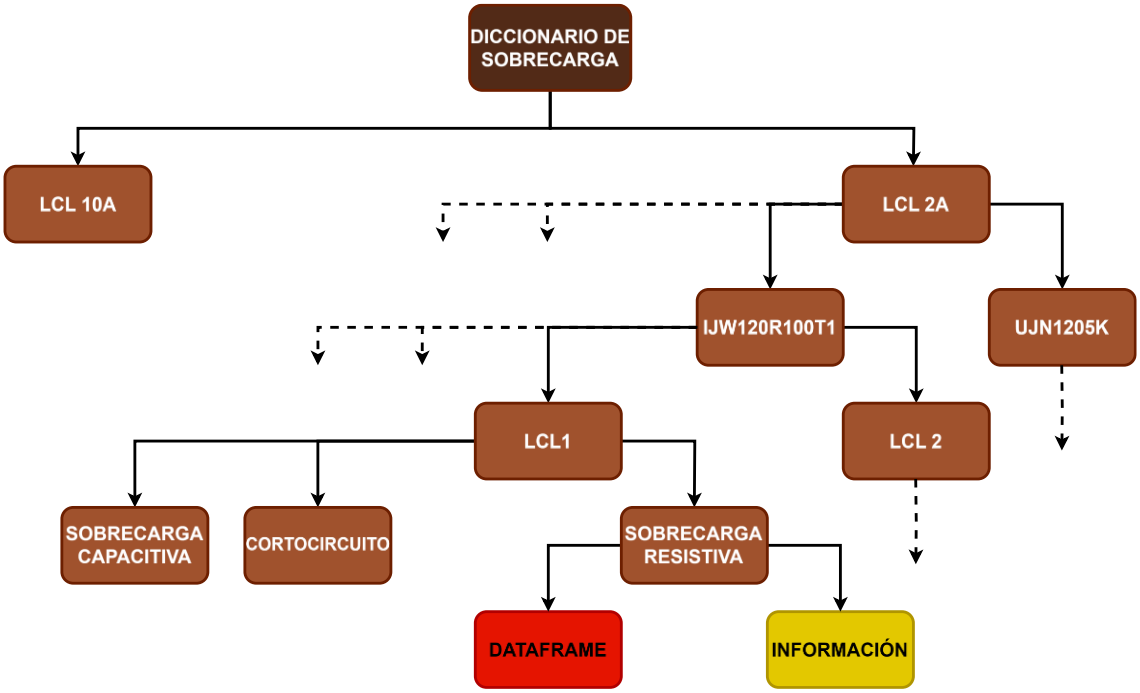


Fig. 29: Estructura de diccionarios confeccionada por el programa principal.

Cada fila de ese *dataframe* representa los resultados numéricos obtenidos para una repetición específica, esto implica que, de mil ficheros CSV —cuyos valores han sido cargados previamente en otro objeto de tipo *dataframe* — con las cuatro curvas obtenidas, se obtiene un *dataframe* con mil filas, donde cada fila contiene el resultado del análisis para cada fichero. De este modo, se construye progresivamente una base de datos tabulada no estándar que recoge de forma estructurada los resultados parciales de cada campaña.

El programa principal se apoya en una serie de funciones auxiliares, cada una de las cuales está diseñado para realizar uno de los análisis específicos sobre los lotes de datos cargados. Según las preferencias del usuario, se ejecutarán las cuatro funciones de análisis correspondientes y, a medida que se vayan cargando los *dataframes* en lotes, se irán realizando los análisis pertinentes, cuyos resultados se irán incorporando progresivamente en el diccionario. Este comportamiento puede verse, de forma simplificada en el diagrama de flujo de las Fig. 30 y Fig. 31, en este diagrama tan solo se introduce la interacción con la función de análisis de máximos por simplicidad, sin embargo, las otras tres funciones tienen un trato equivalente al mostrado en estaa figura.

Como forma de guardado de las estructuras de diccionarios que se van a generar se empleará el guardado del entorno de trabajo de Python en archivos '.spydata', propio del entorno de Spider, tal como se indicó en el apartado 2.3.

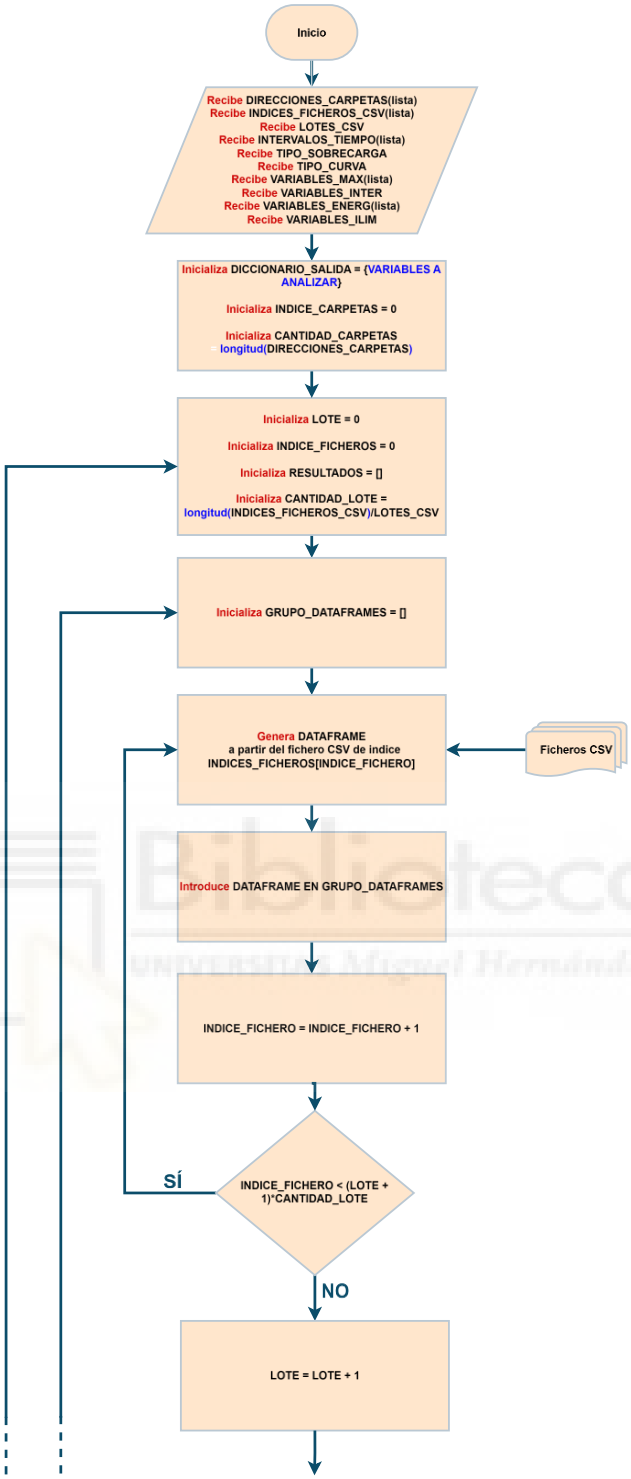


Fig. 30: Diagrama de flujo simplificado del programa principal, donde solo se muestra la interacción con el módulo de análisis de máximos. Parte 1_2.

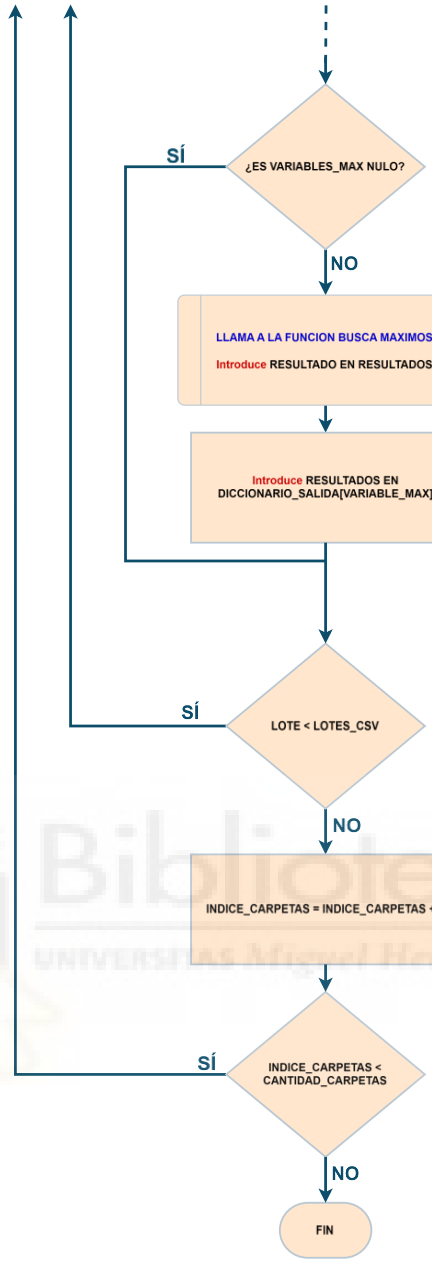


Fig. 31: Diagrama de flujo simplificado del programa principal, donde solo se muestra la interacción con el módulo de análisis de máximos. Parte 2_2.

4.1.4.3. Función de análisis de valor máximo

A continuación, se describe el comportamiento de la primera de las funciones: el análisis de máximos. Este análisis es el más sencillo de los cuatro, y su lógica se mantiene prácticamente inalterada para los tres tipos de sobrecarga Su objetivo es identificar el valor máximo alcanzado por una variable concreta (tensión en JFET, tensión en PMOS o corriente común) en el intervalo donde se produjo la sobrecarga, dentro de cada repetición del lote.

Antes de ejecutar el análisis, se lleva a cabo un proceso de *precaracterización* similar al utilizado en la corrección del offset. En este paso previo, se visualizan representaciones gráficas de curvas típicas para cada tipo de sobrecarga, con el fin de determinar de forma empírica una ventana temporal en la que, con seguridad, se alcanzará el valor máximo de

la variable analizada. Este enfoque permite restringir la búsqueda del máximo a un subconjunto pequeño de los datos (en lugar de procesar el millón de puntos completo), lo que mejora significativamente el rendimiento computacional del análisis. El hecho de que el momento en el que se produce la sobrecarga pueda ser controlado mediante los tiempos de las señales del tren de pulsos generadas por la tarjeta de Arduino hace que encontrar este intervalo sea una tarea muy sencilla. En la Tabla 3 se muestra la ventana de datos que se emplea para cada tipo de sobrecarga, expresando el punto inicial y el final de la ventana en porcentaje, para un mismo tipo de sobrecarga. El módulo recibe dos parámetros de entrada:

- Una lista de objetos *dataframe*, que contiene la información de los archivos CSV, correspondiente al lote de repeticiones cargado.
- Una cadena de texto que identifica la variable a analizar —por ejemplo, "V_JFET", "V_PMOS" o "I"—.

Tabla 3: Porcentajes sobre la base temporal de las curvas donde se produce la sobrecarga, obtenidos de forma empírica.

Tipo de sobrecarga	**Intervalo porcentual sobre el eje temporal (% de la cantidad de puntos de la curva)**	
	Inicio	**Final**
Sobrecarga resistiva	10	20
Cortocircuito	45	50
Sobrecarga capacitiva	20	40

Para cada *dataframe* de la lista, el programa identifica automáticamente la columna correspondiente según el nombre de la variable, y a continuación procede en dos etapas principales:

1. **Corrección del offset**: Al conocer la naturaleza de la variable (tensión o corriente) y el tipo de sobrecarga, se selecciona automáticamente la ventana adecuada para el cálculo del offset. El valor medio en esa región se calcula y se sustrae de toda la curva para obtener una señal ajustada.

2. **Cálculo del valor máximo**: Utilizando la librería 'NumPy' de Python, se obtiene el valor máximo en una ventana de datos que ha sido restringida según el tipo de sobrecarga.

Cada valor máximo obtenido se almacena en una lista de resultados, que posteriormente será recogida por el programa general para su integración en la estructura de datos final.

El diagrama de flujo del proceso de obtención de valor máximo descrito en este apartado se muestra en la Fig. 32.

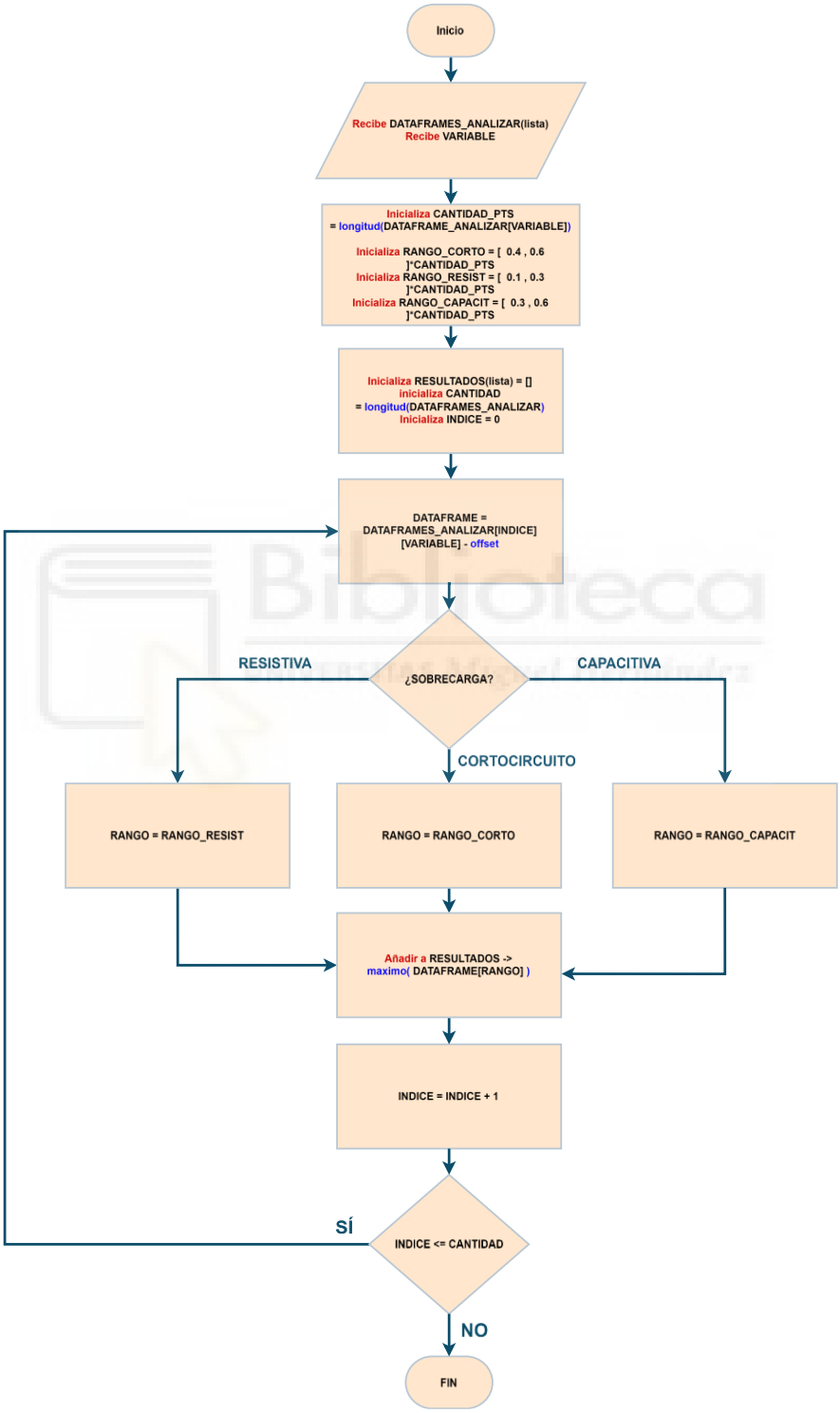


Fig. 32: Diagrama de flujo de la función de detección de valor máximo.

4.1.4.4. Función de análisis de intervalo de transición y desconexión

La segunda función en la arquitectura del programa principal es el **análisis del intervalo de carga o desconexión**, una parte crítica del tratamiento de los datos, ya que este intervalo define el periodo durante el cual el dispositivo está sometido a condiciones exigentes de operación. A diferencia del análisis de máximos, este módulo requiere una lógica diferenciada según el tipo de sobrecarga, ya que existen diferencias sustanciales entre las sobrecargas resistivas o de cortocircuito y las de arranque capacitivo.

Al igual que en los módulos anteriores, además de la corrección de offset, este análisis comienza con una fase de *precaracterización*, en la que se inspeccionan manualmente curvas representativas de los diferentes tipos de sobrecarga. El objetivo es identificar ventanas óptimas de búsqueda que permitan reducir el número de muestras a analizar en cada repetición. Esta etapa es especialmente importante aquí, ya que el método empleado se basa en la aplicación de medias móviles y comparaciones entre tramos de datos, lo cual implica operaciones repetitivas de coste computacional elevado.

En este caso se busca una ventana de datos para encontrar el punto inicial y otra para el punto final, sin una selección adecuada de estos puntos, el número de operaciones por *dataframe* se incrementaría de forma excesiva, haciendo que el procesamiento de varios miles de archivos resulte inviable en tiempo razonable. Por tanto, el acotamiento eficiente del rango temporal de búsqueda es un paso previo imprescindible para la viabilidad del análisis. La Tabla 4 presentan las ventanas de búsqueda de punto inicial o final, si el punto inicial es superior al punto final implica que la búsqueda se hace de delante hacia atrás. Cabe destacar que, para la búsqueda del punto final del intervalo, se suelen emplear ventanas ligeramente diferentes debido a la diferente duración del intervalo, esto se puede observar en la Fig. 45.

Tabla 4: Porcentajes sobre la base temporal de las curvas donde se busca el inicio y el final del intervalo de desconexión/carga.

Tipo de sobrecarga	**Intervalo porcentual sobre el eje temporal (% de la cantidad de puntos de la curva)**			
	Inicio sobrecarga		**Final sobrecarga**	
	Pto. Inic.	**Pto.Fin.**	**Pto. Inic.**	**Pto.Fin.**
Sobrecarga resistiva	20	21	40/70	100
Cortocircuito	50	51	60/70	100
Sobrecarga capacitiva	35	25	80/68	50

Durante esta fase, tal como se ha comentado anteriormente, es necesario distinguir entre las curvas correspondientes a corrientes nominales los de clase 2 y los de clase 10, ya que el comportamiento en la respuesta del circuito de protección varía notablemente con el nivel de corriente. En las sobrecargas resistivas, por ejemplo, pueden observarse diferencias en el tiempo de desconexión de hasta un 50%, mientras que en las de cortocircuito estas diferencias alcanzan el 45%. Esta variabilidad obliga a definir ventanas de búsqueda independientes para cada categoría de corriente, asegurando así que el análisis posterior sea preciso y aplicable a la curva que se está evaluando.

El objetivo de este módulo es detectar con precisión los puntos de inicio y fin del intervalo de desconexión, es decir, el periodo durante el cual el circuito de protección actúa para mantener las condiciones del dispositivo dentro de su zona segura. Sin embargo, este intervalo no está marcado por discontinuidades evidentes, y en muchas curvas la presencia de ruido, oscilaciones o transiciones suaves dificulta la detección directa.

Los métodos clásicos basados en la derivada de la señal, utilizados comúnmente para detectar cambios de pendiente, no son aplicables en este contexto, ya que:

- El ruido de alta frecuencia introduce múltiples zonas donde la derivada es engañosa.
- Las transiciones no siempre presentan inflexiones fácilmente detectables.
- La señal puede tener pequeñas oscilaciones superpuestas a una tendencia general.

Para afrontar estos retos, se recurre a una combinación de la aplicación del filtro de SG y **las medias móviles deslizantes**, estas se calculan sobre tramos acotados de la curva para identificar zonas donde se produce un cambio sostenido en los niveles medios de corriente o tensión. Estas medias pueden compararse entre sí o con umbrales fijos, definidos durante la fase de *precaracterización*.

El **análisis del intervalo de carga en casos de sobrecargas por arranque capacitivo** se diferencia notablemente del que se aplica a sobrecargas resistivas o por cortocircuito. Esta diferencia radica tanto en la naturaleza de la curva de corriente como en el enfoque metodológico necesario para identificar los puntos críticos.

En lugar de emplear la potencia disipada en el JFET, se analiza exclusivamente la corriente que circula a través del JFET. Esta elección responde a que la corriente ofrece una representación más clara y directa del comportamiento del sistema durante las fases inicial y final del transitorio. Además, como ya se expuso en el apartado 4.1.2, la aplicación del filtro de suavizado con unos parámetros únicos que engloben todas las curvas no es una tarea trivial para este tipo de sobrecargas debido a la presencia de oscilaciones excesivamente agresivas en ciertas curvas de potencia,

Otra característica importante de este análisis es que no se aplica corrección de *offset* a la curva de corriente. Esta decisión está justificada por el hecho de que el método de detección utilizado en este caso no se basa en valores absolutos, sino en comparaciones relativas entre medias móviles. Por lo tanto, un desplazamiento vertical constante en la señal no afecta a los puntos de cruce ni a la detección de eventos relevantes.

Para identificar el inicio del transitorio, el objetivo es determinar el momento en que la corriente comienza a aumentar, indicando el arranque capacitivo —es decir, la placa LCL ha sido alimentada, por lo tanto, permite el paso de corriente que alimenta a la carga resistiva y al condensador en paralelo, el cual comienza a cargarse—. Para ello, se selecciona una ventana de datos previa al inicio esperado, donde la corriente permanece estable y con un valor bajo (idealmente 0 sin embargo será superior debido al *offset*). A partir de esta ventana se calcula una media de referencia que representa el nivel de reposo.

El análisis se realiza recorriendo la curva de derecha a izquierda —esta elección no corresponde a ningún tipo de criterio, podría haberse realizado en el sentido contrario—, evaluando la corriente mediante una media móvil que se compara en cada paso con la media de referencia. El criterio de detección suele basarse en un factor de aumento; por ejemplo, cuando la media móvil cae por debajo de la media de ocho veces la media de referencia, se considera que ha comenzado el aumento de corriente y, por tanto, el transitorio. Este método resulta especialmente robusto incluso en condiciones de ruido, y se adapta bien a las características suaves de las curvas asociadas al arranque capacitivo, donde técnicas más tradicionales basadas en derivadas pierden efectividad. En la Fig. 33, la Fig. 34 y la Fig. 35 se puede observar el proceso descrito, partiendo desde un punto inicial, yendo punto a punto de la curva hacia la izquierda, se va comparando su media con respecto a la media de referencia multiplicada por el factor de ocho.

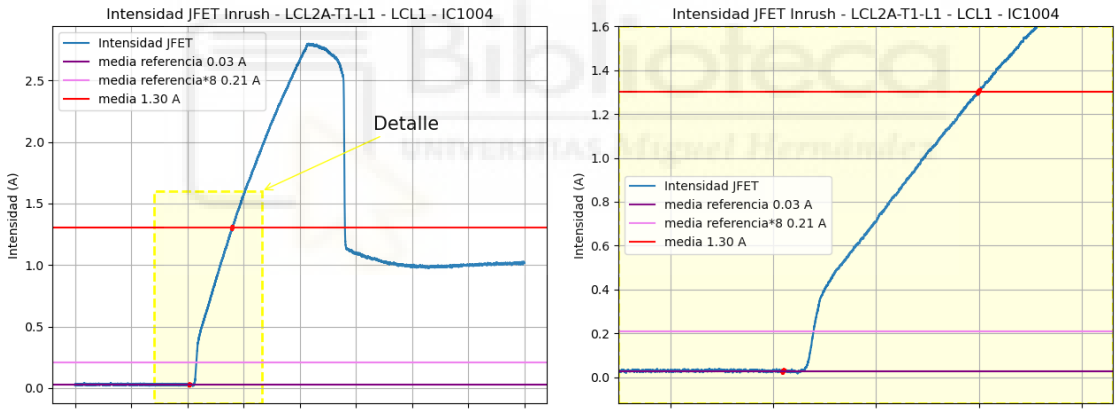


Fig. 33: Estado inicial del proceso de obtención del inicio del intervalo de carga de la sobrecarga capacitiva. Vista general (izquierda) y vista en detalle (derecha).

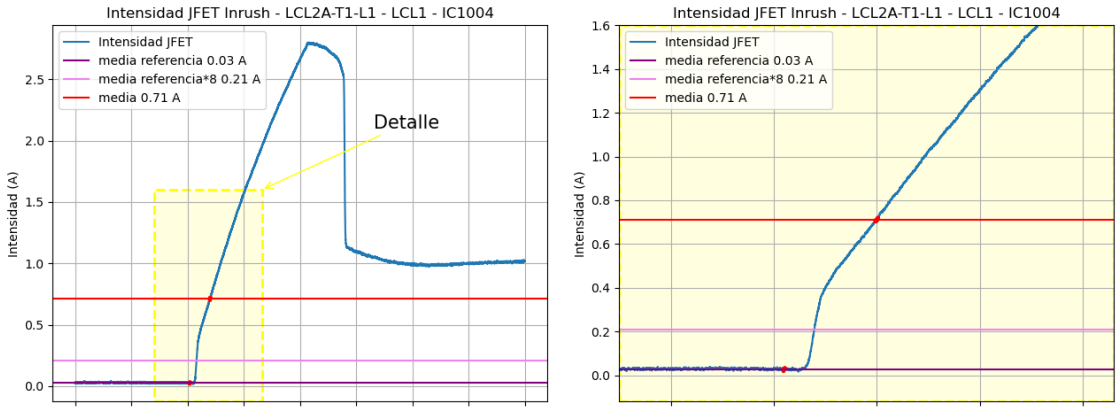


Fig. 34: Estado intermedio del proceso de obtención del inicio del intervalo de carga de la sobrecarga capacitiva. Vista general (izquierda) y vista en detalle (derecha).

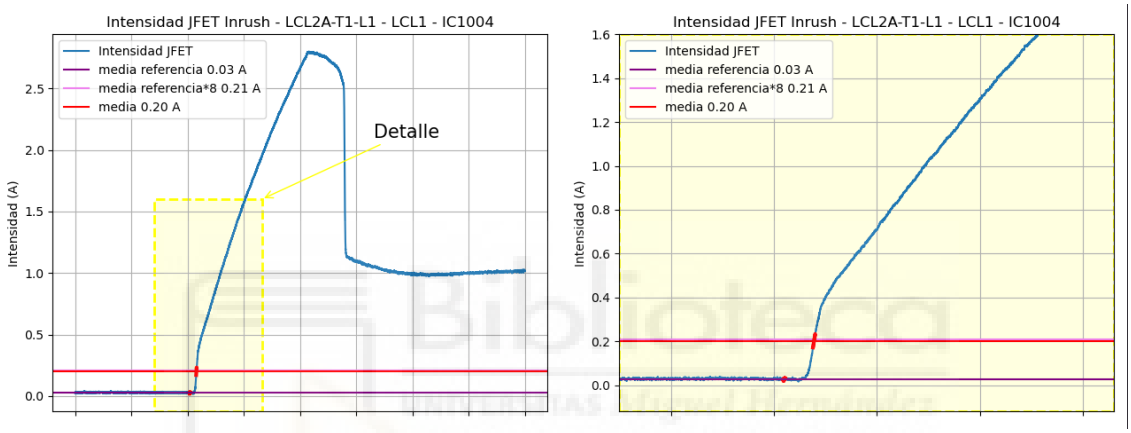


Fig. 35: Estado final del proceso de obtención del inicio del intervalo de carga de la sobrecarga capacitiva. Vista general (izquierda) y vista en detalle (derecha).

En cuanto al final del transitorio, se busca el instante en el que la corriente deja de disminuir y se estabiliza. Para detectar este punto se utilizan dos medias móviles: una centrada en el punto actual (denominada 'media de referencia') y otra desplazada hacia la izquierda, generalmente unos 200 puntos. Ambas se recorren simultáneamente de derecha a izquierda, evaluando la relación entre ellas en cada paso. Cuando la media secundaria supera a la principal en un 50 %, se considera que se ha producido el descenso de corriente que marca el fin del transitorio. De nuevo, en la Fig. 36, Fig. 37 y Fig. 38 se puede observar el proceso gráficamente, donde la vista en detalle se centra en el punto de la media de referencia, que, en este caso representa el punto que se escogerá como final de la carga.

Este enfoque, aunque efectivo, incrementa notablemente la carga computacional, ya que duplica el número de operaciones necesarias para suavizar y comparar las señales. Por ello, es fundamental realizar una buena precaracterización del transitorio: las ventanas de análisis deben estar bien delimitadas, incluyendo todos los segmentos de la zona de carga que contengan información relevante, pero excluyendo segmentos innecesarios. De lo contrario, una ventana demasiado amplia puede alargar considerablemente los tiempos de procesamiento, lo que repercute negativamente en el rendimiento general del análisis cuando se trabaja con grandes volúmenes de datos.

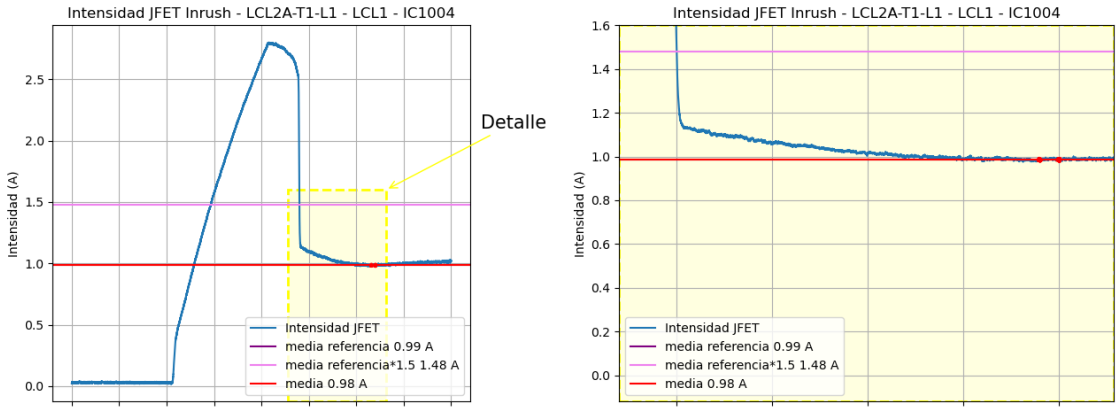


Fig. 36: Estado inicial del proceso de obtención del final del intervalo de carga de la sobrecarga capacitiva. Vista general (izquierda) y vista en detalle (derecha).

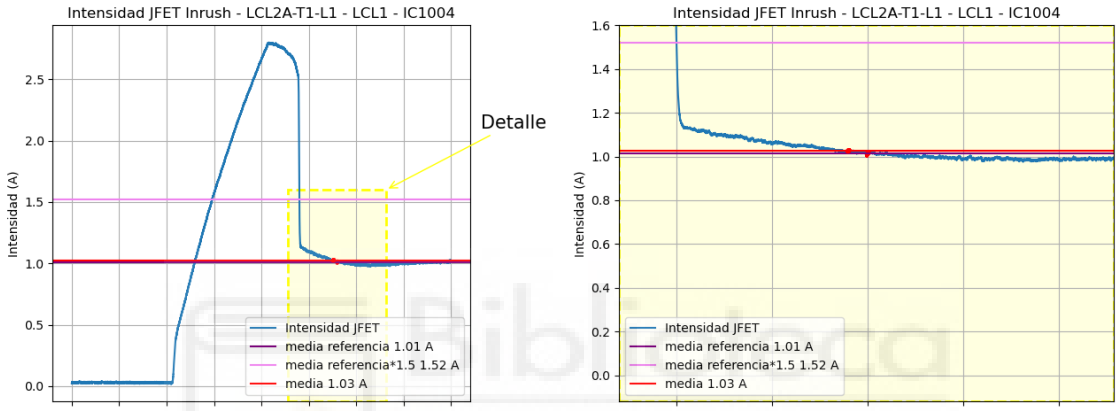


Fig. 37: Estado intermedio del proceso de obtención del final del intervalo de carga de la sobrecarga capacitiva. Vista general (izquierda) y vista en detalle (derecha).

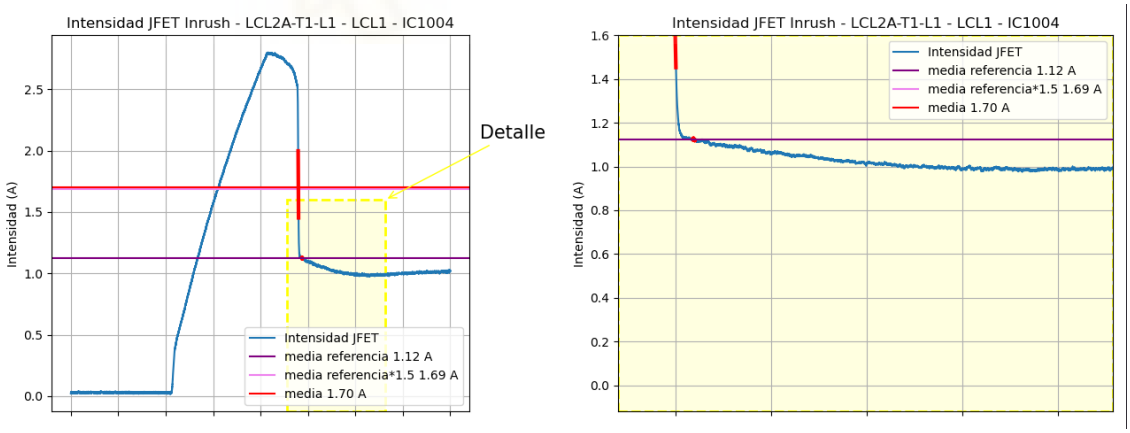


Fig. 38: Estado final del proceso de obtención del final del intervalo de carga de la sobrecarga capacitiva. Vista general (izquierda) y vista en detalle (derecha).

En la Fig. 39 y la Fig. 40 se muestra el diagrama de flujo de la función de detección de tiempo de carga para sobrecargas capacitivas.

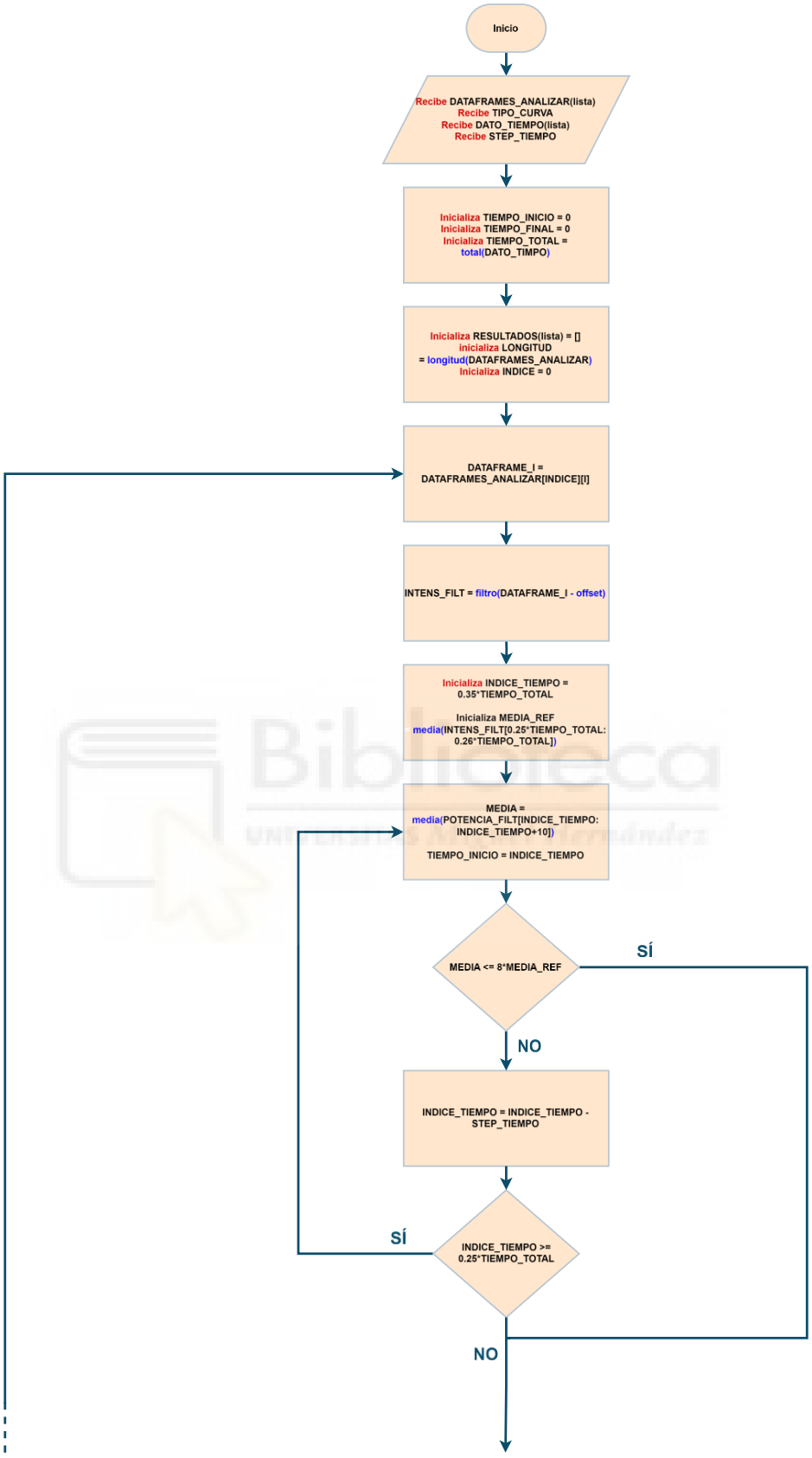


Fig. 39: Diagrama de flujo de la función de detección de intervalo de carga. Parte 1_2.

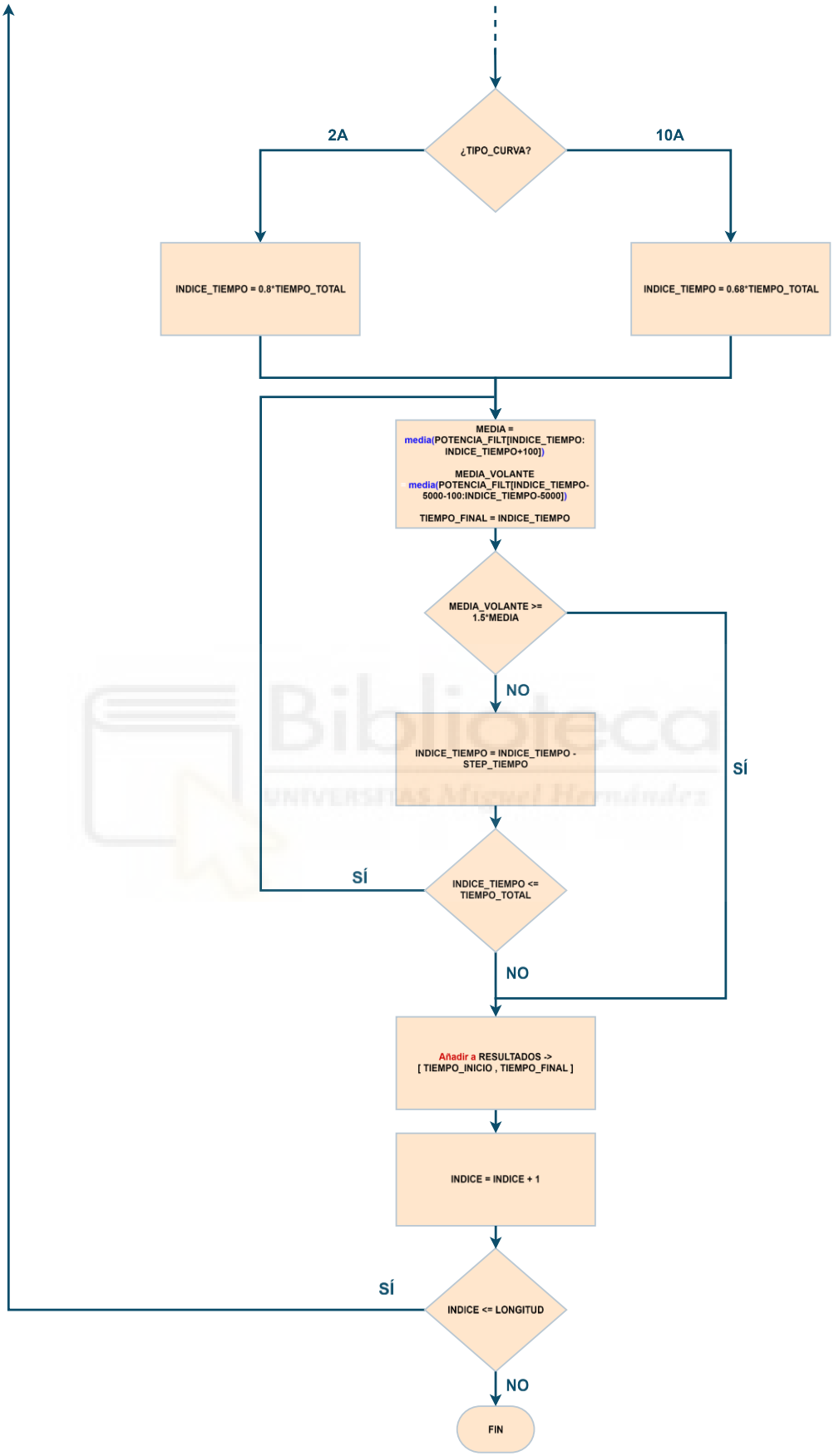


Fig. 40: Diagrama de flujo de la función de detección de intervalo de carga. Parte 2_2.

En el **análisis de sobrecargas resistivas y de cortocircuito**, el foco se pone exclusivamente en el JFET, ya que es el componente que soporta la mayor parte del esfuerzo energético durante estos eventos. En estos casos, la disipación de potencia significativa se concentra en dos momentos clave: primero, durante el transitorio inicial de la sobrecarga, donde la energía disipada es relativamente baja; y luego, durante el intervalo de limitación, en el cual el JFET actúa como un sistema de protección activa, disipando la energía, la suma de ambos intervalos es el intervalo de desconexión. Como este caso no se ejecutará en base a comparaciones de dos medias, sino en la comparación de una media con respecto a un valor de referencia fijado con anterioridad, en este análisis es esencial aplicar la corrección de *offset*.

La detección del punto inicial se realiza recorriendo la curva de potencia desde el comienzo hacia adelante (de izquierda a derecha), aplicando una media móvil sobre una ventana de datos fija. Se establece un umbral de referencia de 50 W, y el primer momento en el que esta media supera dicho valor se considera el comienzo del transitorio. Este punto es relativamente fácil de localizar, ya que la subida en la curva de potencia es abrupta. Además, este instante suele coincidir con el disparo del *trigger* del osciloscopio, lo que garantiza que todas las curvas estén sincronizadas temporalmente. Esto facilita aún más su identificación, ya que el evento se repite en una posición similar a lo largo de diferentes mediciones. A pesar de que el filtro distorsione ciertas zonas de la curva, especialmente los del transitorio, llegando a tener derivas hacia potencias negativas, la obtención de este punto no se ve afectada de forma alarmante en este proceso. En la Fig. 41 y la Fig. 42 se pueden ver dos momentos del proceso, el momento inicial y el momento anterior a la repetición cuya media superará los 50 W de potencia. Con un ajuste de ventana de la media que conste con un menor número de puntos se puede llegar a afinar mejor el punto, sin embargo, debido a la velocidad del transitorio y la resolución de la curva, no existe una gran cantidad de puntos en esa zona.

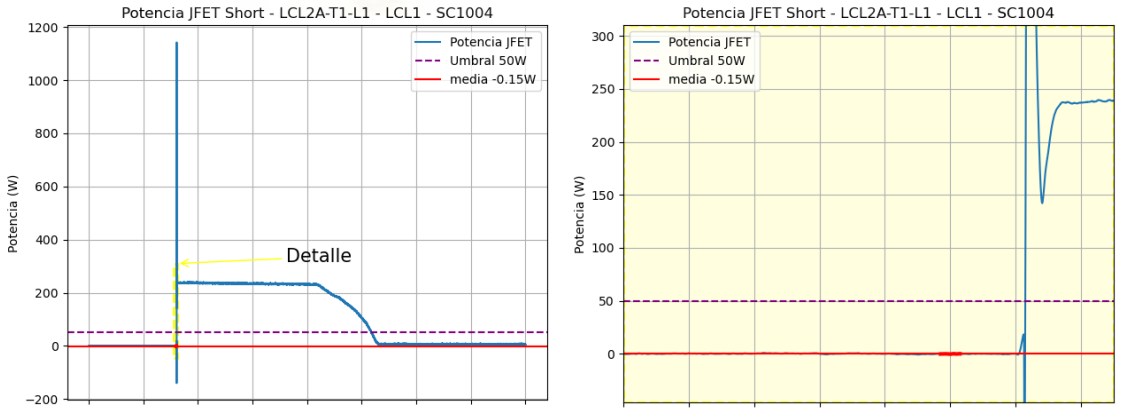


Fig. 41: Estado inicial del proceso de obtención del inicio del intervalo de desconexión de la sobrecarga resistiva. Vista general (izquierda) y vista en detalle (derecha).

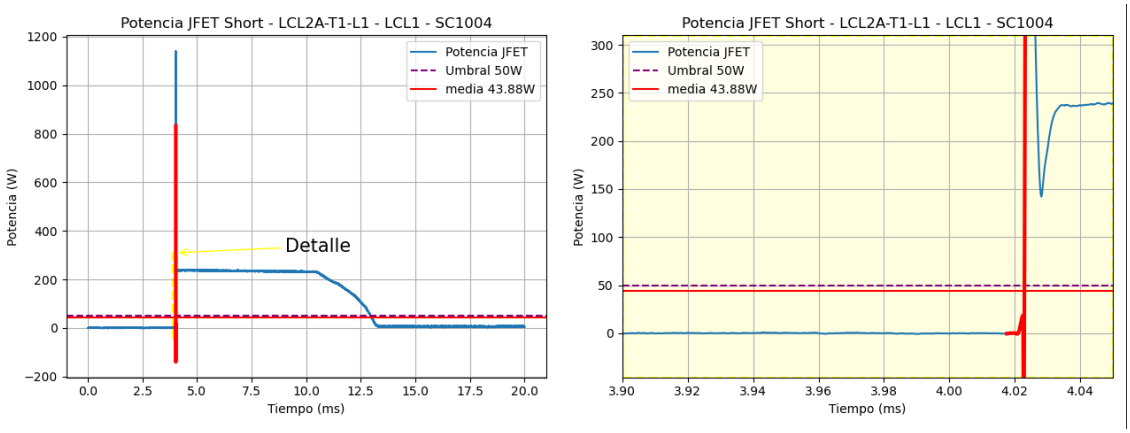


Fig. 42: Estado anterior al final del proceso de obtención del inicio del intervalo de desconexión de la sobrecarga resistiva. Vista general (izquierda) y vista en detalle (derecha).

En cambio, la detección del punto final presenta más desafíos. Aquí también se recorre la curva de izquierda a derecha y se utiliza una media móvil para detectar el primer punto en el que la potencia promedio cae por debajo del umbral de 50 W. Sin embargo, este descenso no suele ser tan claro como el ascenso inicial. En la Fig. 43 y la Fig. 44 se pueden ver dos momentos del proceso descrito.

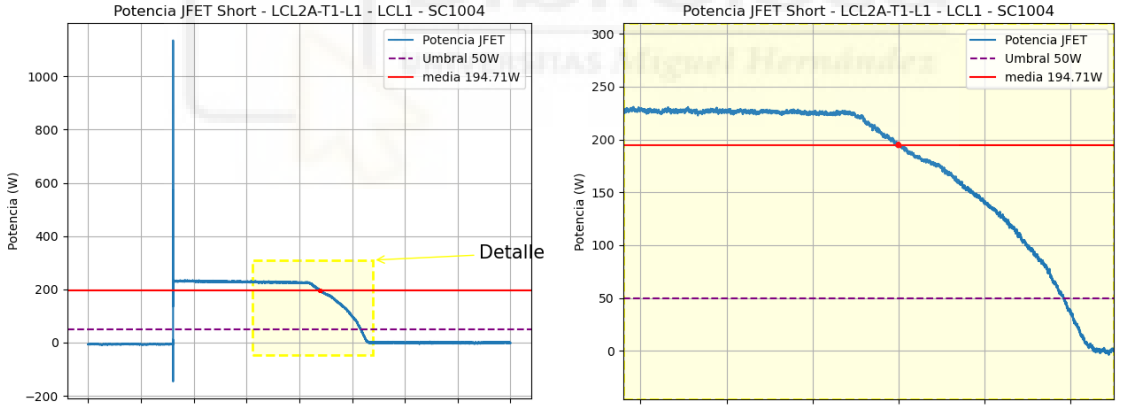


Fig. 43: Estado inicial del proceso de obtención del final del intervalo de desconexión de la sobrecarga resistiva. Vista general (izquierda) y vista en detalle (derecha).

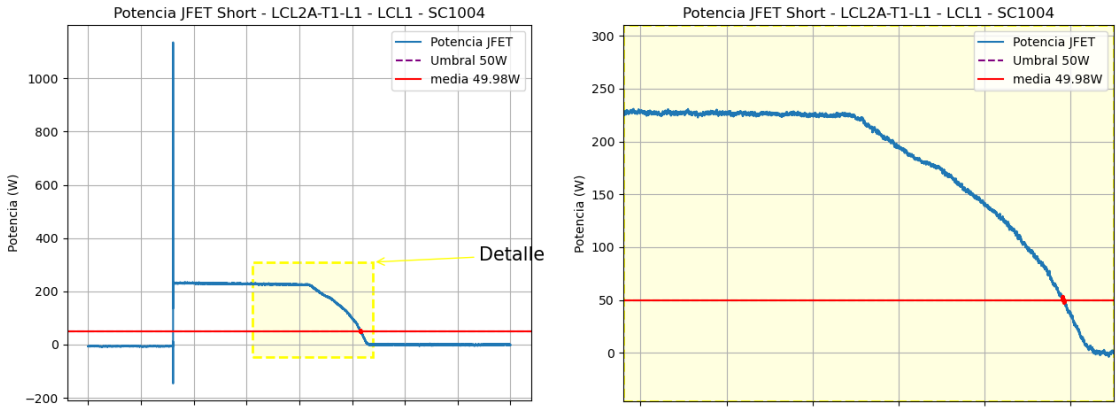


Fig. 44: Estado final del proceso de obtención del final del intervalo de desconexión de la sobrecarga resistiva. Vista general (izquierda) y vista en detalle (derecha).

Una de las principales dificultades radica en la alta variabilidad entre dispositivos (véase Fig. 45). La posición exacta del punto de descenso puede cambiar considerablemente según la referencia del JFET y del PMOS utilizado. Incluso dentro de una misma combinación de componentes, la duración del intervalo puede variar entre repeticiones. Además, la transición descendente suele ser mucho más gradual, lo que dificulta establecer un límite claro. A esto se suma que el perfil de bajada de la potencia no es uniforme entre curvas: en algunos casos puede presentar múltiples caídas parciales que podrían inducir a error si los parámetros de detección no están bien ajustados. Todo esto puede conducir tanto a falsas detecciones como a errores de estimación en la duración del evento, lo cual afecta directamente a la precisión del cálculo de la energía disipada.

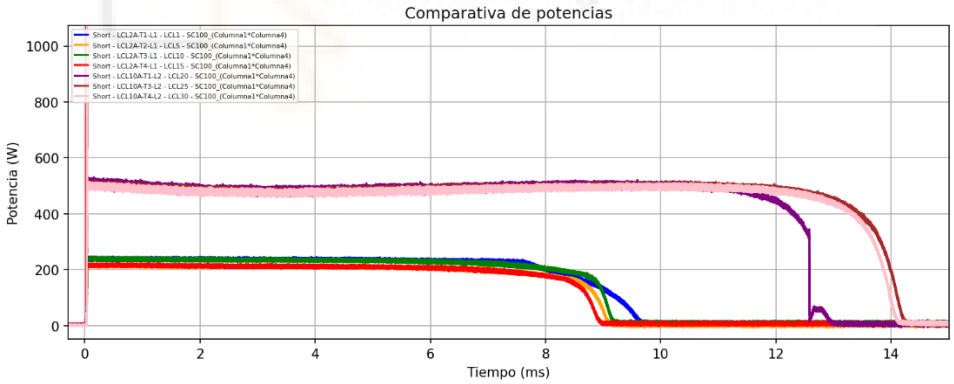


Fig. 45: Comparativa de intervalos de desconexión en curvas de sobrecarga resistiva de varios grupos de LCL, tanto de clase 2 como de clase 10.

En la Fig. 46 y la Fig. 47 se expone el diagrama de flujo de la función de detección de tiempo de carga para sobrecargas resistivas, el de cortocircuito es análogo.

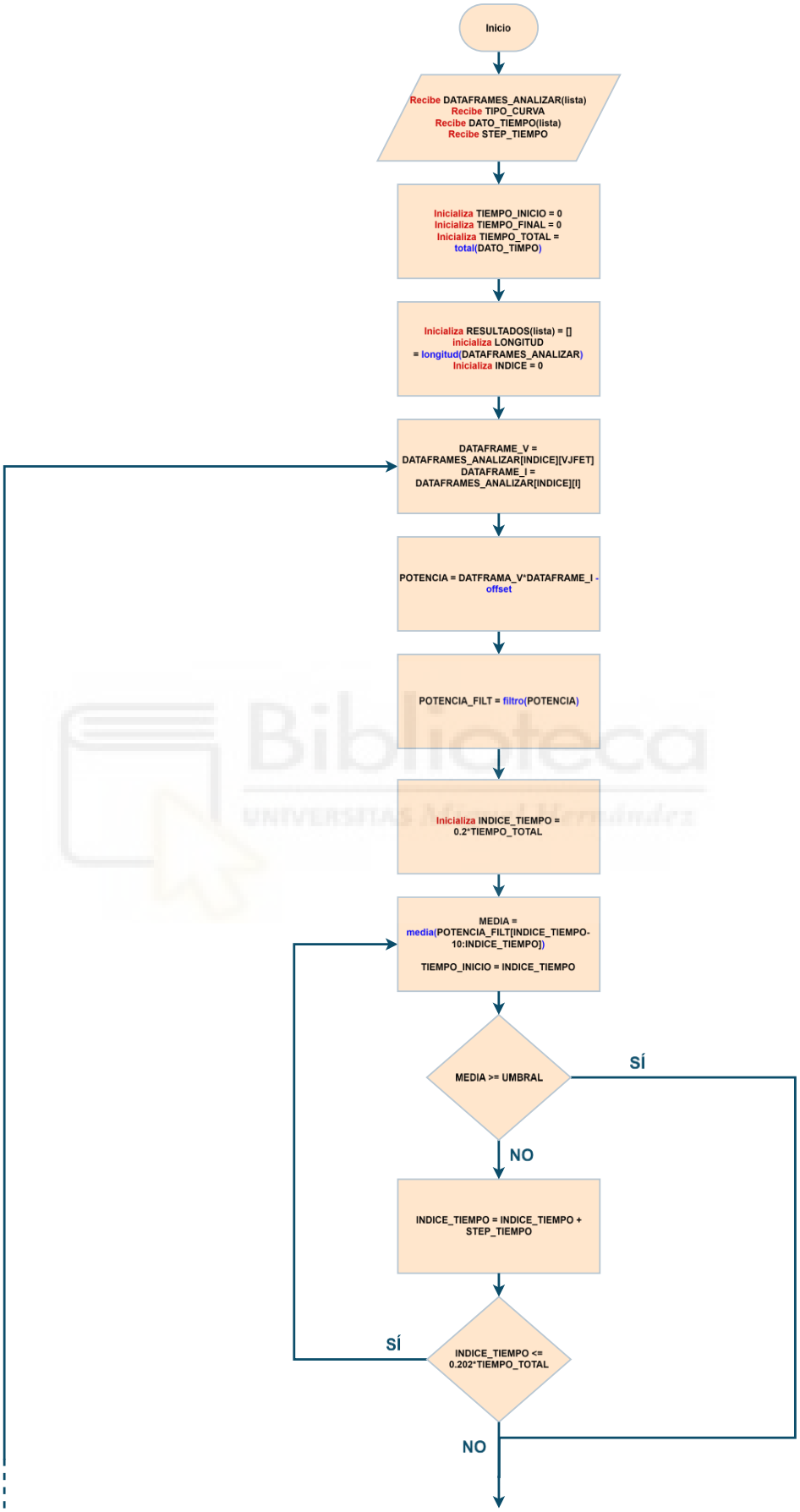


Fig. 46: Diagrama de flujo de la función de detección de intervalo de desconexión para sobrecarga resistiva. Parte 1_2.

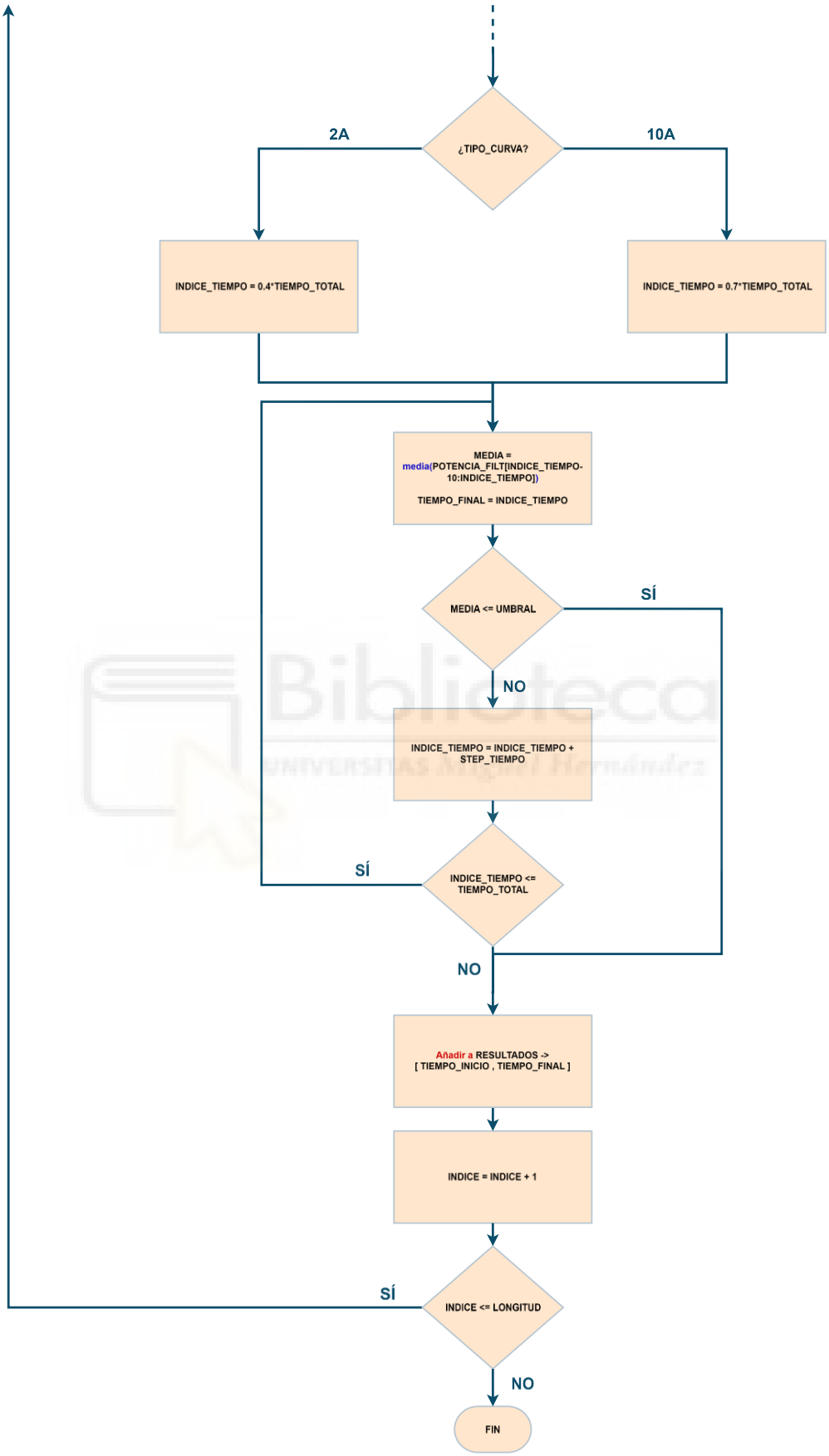


Fig. 47: Diagrama de flujo de la función de detección de intervalo de desconexión para sobrecarga resistiva. Parte 2_2.

4.1.4.5. Función de análisis de energía disipada

La tercera función implementada en el sistema tiene como finalidad calcular la energía total disipada en el JFET o en el PMOS durante una situación de sobrecarga —esto es, a lo largo del intervalo de desconexión para la sobrecarga resistiva y de cortocircuito y a lo largo del intervalo de carga para la sobrecarga capacitiva—. A diferencia del análisis del intervalo de desconexión/carga, este proceso resulta considerablemente más sencillo, tanto desde el punto de vista conceptual como en términos de carga computacional. Sin embargo, existe un requisito fundamental: el intervalo de carga o de desconexión debe haber sido previamente identificado, ya que la energía se calculará a lo largo del intervalo calculado, especialmente por ahorro de coste computacional ya que, en las otras áreas la potencia es casi nula, sin embargo, sigue siendo exigente su cálculo.

Una de las ventajas de este módulo es que no necesita una *precaracterización* adicional más allá de la definición del intervalo de análisis. Esto simplifica notablemente su ejecución y permite que el cálculo se lleve a cabo de forma automática tan pronto como se haya delimitado correctamente el intervalo. Aun así, es imprescindible que, antes de realizar cualquier cálculo, se haya aplicado una corrección de offset a las señales de corriente y tensión del dispositivo en cuestión. Como ya se ha comentado, cualquier desviación sistemática en estas curvas alteraría directamente la estimación de la potencia instantánea y, por consiguiente, del valor final de la energía disipada.

En cuanto al tratamiento del ruido, este análisis adopta un enfoque particular: no se aplica ningún tipo de filtrado o suavizado, como podría ser el filtro de SG. Esta decisión no es casual, sino que responde a una razón bien justificada. Dado que el cálculo de la potencia se realiza punto a punto, multiplicando la tensión por la corriente, y posteriormente se integra sobre un número elevado de muestras, el efecto del ruido de alta frecuencia no es especialmente significativo, además, el hecho de aplicar filtros sobre las curvas, por leve que sea su efecto, puede inducir cálculos erróneos por distorsionar las curvas.

Para llevar a cabo la integración numérica de la potencia y obtener la energía total, se utiliza el método de aproximación trapezoidal. Esta técnica ofrece un equilibrio muy favorable entre precisión, eficiencia y simplicidad. Es particularmente adecuada para datos muestreados, como los que se manejan en este tipo de análisis, ya que no requiere conocer una expresión analítica de la señal. Opera directamente sobre los valores discretos de potencia. Además, gracias al elevado número de puntos disponibles por curva —que en la mayoría de los casos llega al millón por medición—, el error asociado al método trapezoidal resulta insignificante.

En la Fig. 48 se ilustra el diagrama de flujo del proceso de obtención de la energía disipada en los dispositivos JFET.

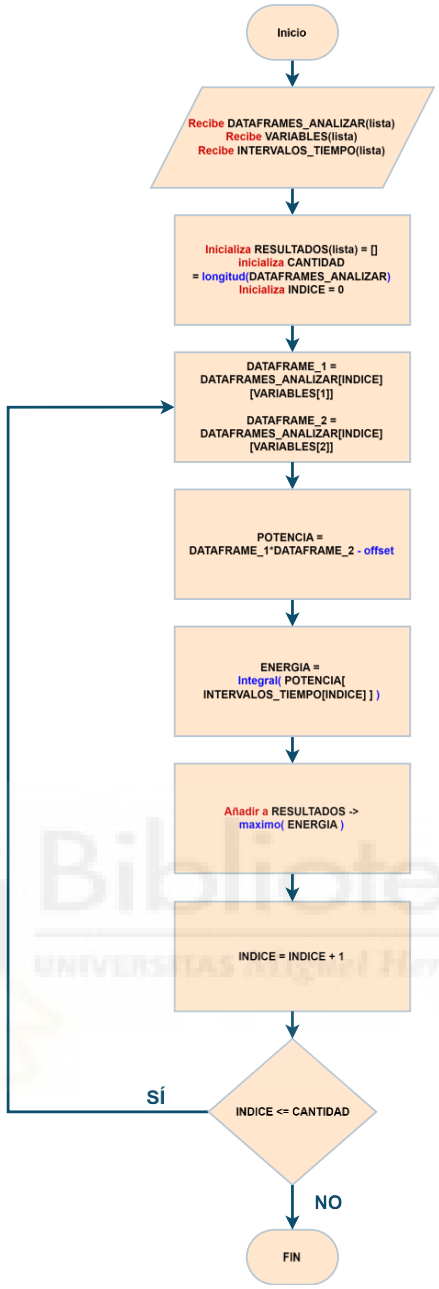


Fig. 48: Diagrama de la función de análisis de energía disipada.

4.1.4.6. Módulo de análisis de corriente media de limitación

El cuarto y último módulo desarrollado se encarga de calcular la corriente media durante el intervalo de limitación, es decir, la parte del intervalo de desconexión que no comprende el transitorio inicial. Este tipo de análisis se aplica exclusivamente a las sobrecargas resistivas o por cortocircuito, ya que no resulta adecuado en situaciones de arranque capacitivo debido a que no se alcanza el punto de limitación.

El procedimiento parte, necesariamente, de que el intervalo de limitación ya ha sido identificado mediante los análisis anteriores. Sobre ese segmento de la curva de corriente —que es común tanto para el JFET como para el PMOS— se lleva a cabo la estimación de la corriente media.

En la práctica, el intervalo se reduce considerablemente por ambos extremos, con el objetivo de no incluir en la integral el transitorio de sobrecorriente ni el transitorio ON/OFF del final del intervalo de desconexión, de forma que se aplican unos coeficientes al principio y al final del intervalo de 1.23 y 0.77 respectivamente.

Para ello, se selecciona el tramo de la señal correspondiente al intervalo detectado y se aplica una integración numérica, concretamente mediante el método de aproximación trapezoidal, con el objetivo de calcular el valor total de corriente acumulada en ese período. Posteriormente, este resultado se divide entre la duración del intervalo —es decir, la diferencia entre los instantes final e inicial— para obtener así el valor medio de la corriente durante la fase de limitación.

En la Fig. 49 se muestra el diagrama de flujo del proceso de obtención de la corriente media de limitación descrito en este apartado.

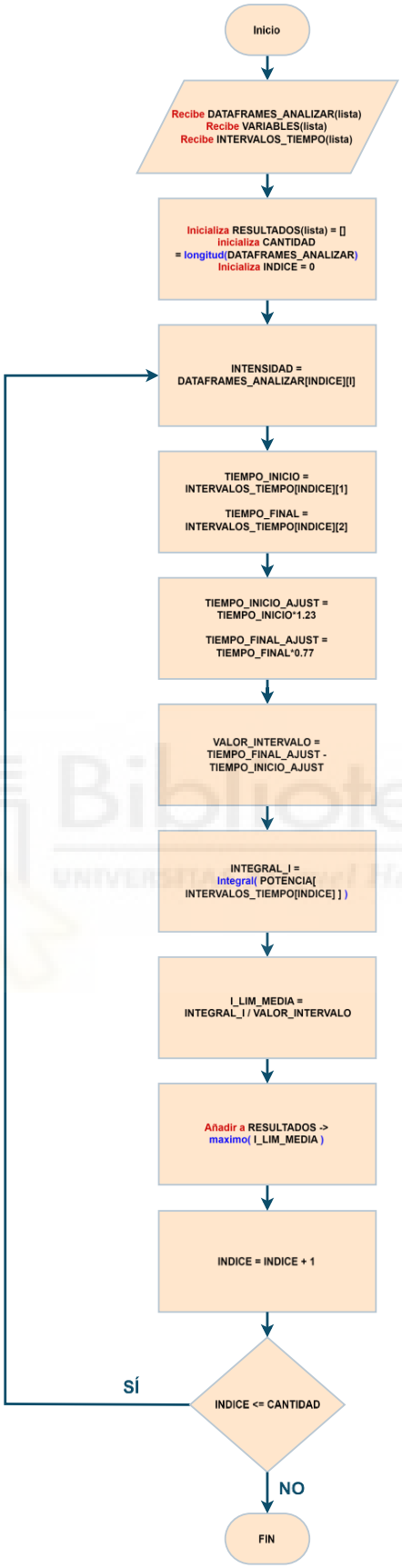


Fig. 49: Diagrama de flujo de la función de análisis de corriente media de limitación.

4.1.5. Visualización de los datos procesados

Una vez implementado el programa principal de análisis de las curvas de sobrecarga y obtenidos los resultados numéricos correspondientes a cada una de las características evaluadas, es necesario desarrollar herramientas de visualización que permitan interpretar estos datos de forma clara, comparativa y eficiente.

Para ello, se han concebido dos enfoques diferenciados de representación gráfica. El primero de ellos se basa en un esquema estructurado por referencia/LCL que busca representar de manera más directa los datos, mientras que el segundo tiene el objetivo de compactar más la información y mostrarla de manera más clara y eficiente.

Primer enfoque – Valor frente a nº repetición

En este primer enfoque de visualización, se genera una gráfica individual para cada característica analizada —como la corriente pico, la tensión pico, la energía disipada o la corriente media durante la limitación—, y se hace de forma específica para cada referencia, que agrupa un conjunto de cuatro LCLs. En cada una de estas gráficas se representan las curvas correspondientes a los cuatro LCLs asociados a la referencia, a las que se suma una quinta curva adicional que refleja el promedio de las anteriores. Esta última actúa como una referencia visual del comportamiento medio del grupo.

Esta estrategia ofrece varias ventajas. En primer lugar, permite una comparación interna eficaz dentro de cada referencia. Visualizar las curvas individuales de los LCLs facilita la detección de desviaciones, anomalías o patrones de comportamiento que puedan diferenciarse del conjunto. Además, la curva media aporta una referencia clara para valorar la estabilidad global del grupo y situar a cada dispositivo individual frente a ese comportamiento agregado. Otro aspecto positivo es que, al trazarse las curvas a lo largo de las repeticiones, se puede seguir la evolución de cada parámetro con el tiempo, lo que ayuda a identificar posibles degradaciones progresivas o comportamientos sistemáticos.

No obstante, este enfoque también presenta ciertos inconvenientes. Uno de los principales es la sobrecarga visual. Dado que cada gráfica contiene al menos cinco curvas, y considerando que este proceso se repite para cada característica analizada y para cada una de las referencias (que pueden llegar a ser hasta 32 grupos distintos), el volumen de información gráfica generado es elevado, lo que complica una revisión exhaustiva de los resultados. Además, desde el punto de vista operativo, la generación masiva de estas gráficas implica un coste computacional considerable. En sesiones con grandes volúmenes de datos, automatizar este proceso puede suponer un tiempo de procesamiento significativo, además de una ocupación elevada de recursos de almacenamiento.

Como ejemplo de este primer enfoque de representación de curvas, se aporta la Fig. 50 y la Fig. 51, las cuales muestran la evolución de la energía disipada en el JFET del grupo de ‘LCL1-4’ y la evolución tiempo de desconexión del grupo de ‘LCL25-28’ respectivamente.

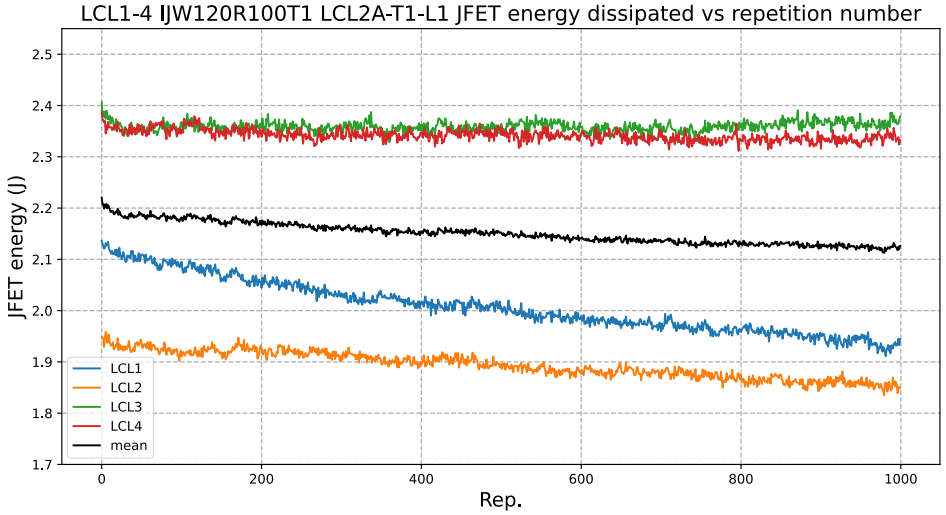


Fig. 50: Ejemplo de representación visual de los resultados de energía disipada en el JFET obtenidos mediante los programas de análisis empelando el primer método de visualización.

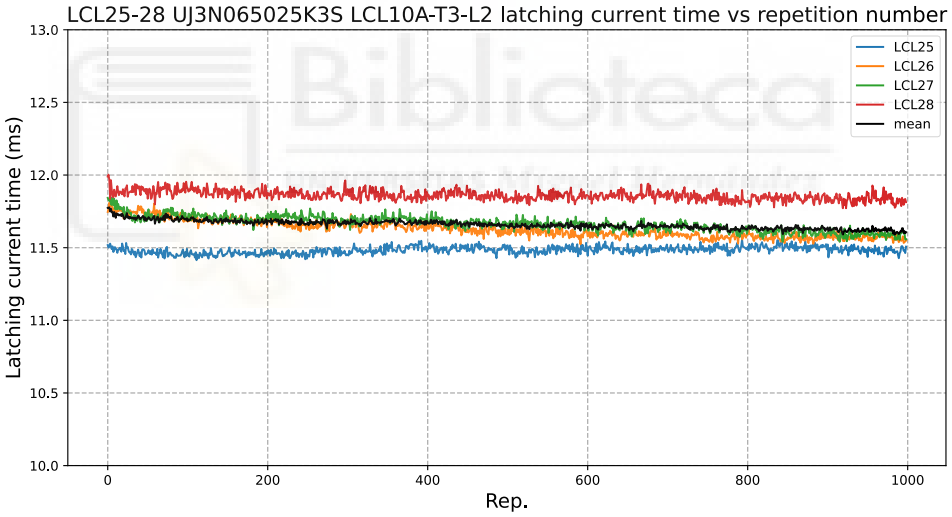


Fig. 51: Ejemplo de representación visual de los resultados de tiempo de desconexión obtenidos mediante los programas de análisis empelando el primer método de visualización.

Segundo enfoque – Diagrama de bigotes

Como respuesta a las limitaciones del primer enfoque de representación —especialmente la sobrecarga visual derivada de la cantidad de curvas y puntos mostrados—, se ha desarrollado una segunda vía de visualización centrada en la simplificación y condensación de la información. Esta segunda estrategia apuesta por reducir significativamente el número de elementos representados en cada gráfico, sin que ello suponga una pérdida sustancial de la información relevante contenida en las curvas.

El planteamiento parte de las curvas generadas en el primer enfoque, que cuentan con una resolución de mil puntos por curva (uno por repetición). En lugar de representar

directamente esos mil puntos, se aplica una reducción estructurada: cada curva se sintetiza en solo 10 puntos, dando lugar a una curva de valores medios. Cada uno de estos nuevos puntos se calcula como la media de una ventana de diez valores consecutivos, siendo los centros de estas ventanas los puntos 99, 199, 299, ..., hasta el 999. Es decir, el primer punto representado corresponde a la media del intervalo [90, 99], el segundo al intervalo [190, 199], y así sucesivamente. Esta técnica suaviza variaciones puntuales, ofreciendo una visión clara de la tendencia general sin necesidad de visualizar el conjunto completo de datos.

A continuación, todas las curvas reducidas se normalizan utilizando como referencia un único valor: el segundo punto de la curva media (correspondiente a la media de las ventanas de las cuatro curvas individuales). Este punto, que refleja el comportamiento promedio del sistema alrededor de la repetición número 199, se elige por estar situado tras la fase de estabilización inicial, pero aún cerca del comienzo de las repeticiones, y se considera representativo del rendimiento normalizado. Todas las curvas —incluida la propia curva media— se dividen por este valor, de modo que la curva media toma un valor de 1 en su segundo punto y el resto de las curvas se representan en relación con esta referencia. Esto permite observar la evolución relativa de las distintas curvas respecto a un estado base común.

Finalmente, para condensar la información del conjunto sin renunciar a la variabilidad, se identifican en cada uno de los 10 puntos los valores máximos y mínimos entre las cuatro curvas individuales (ya normalizadas). Estos extremos se representan como ‘bigotes’ de error alrededor de la curva media. Los bigotes se visualizan mediante líneas verticales discontinuas que unen el valor mínimo y el máximo en cada punto, rematadas por segmentos horizontales cortos de color, que heredan el color que les correspondería a las curvas originales si se hubieran representado. La curva media se dibuja con una línea sólida de color negro, y sus puntos se marcan con círculos negros para enfatizar su identidad y continuidad.

Este enfoque ofrece varias ventajas destacables:

- Se logra una representación condensada y legible, con solo una curva principal y 10 ‘bigotes’ por gráfico.

- La normalización respecto a un punto común permite observar de forma clara las variaciones relativas a lo largo del tiempo/repeticiones.

- La inclusión de los máximos y mínimos por punto proporciona una indicación visual de la dispersión y estabilidad entre las muestras de un mismo LCL.

No obstante, también presenta algunas desventajas:

- Al reducir los datos de 1000 a 10 puntos, se pierde resolución y se pueden ocultar tendencias locales o anomalías puntuales.

- Al representar únicamente la curva media, se produce una despersonalización de las curvas individuales, lo que dificulta analizar comportamientos específicos de un LCL concreto.

- La normalización puede dificultar la lectura del valor absoluto de las curvas. Esta limitación puede mitigarse incluyendo una anotación explícita en el gráfico que indique el valor del punto de referencia utilizado para la normalización.

Como ejemplo de este segundo enfoque se presenta la Fig. 52, que muestra la progresión del tiempo de limitación/desconexión de todos los grupos de clase 2 en pruebas de cortocircuito.

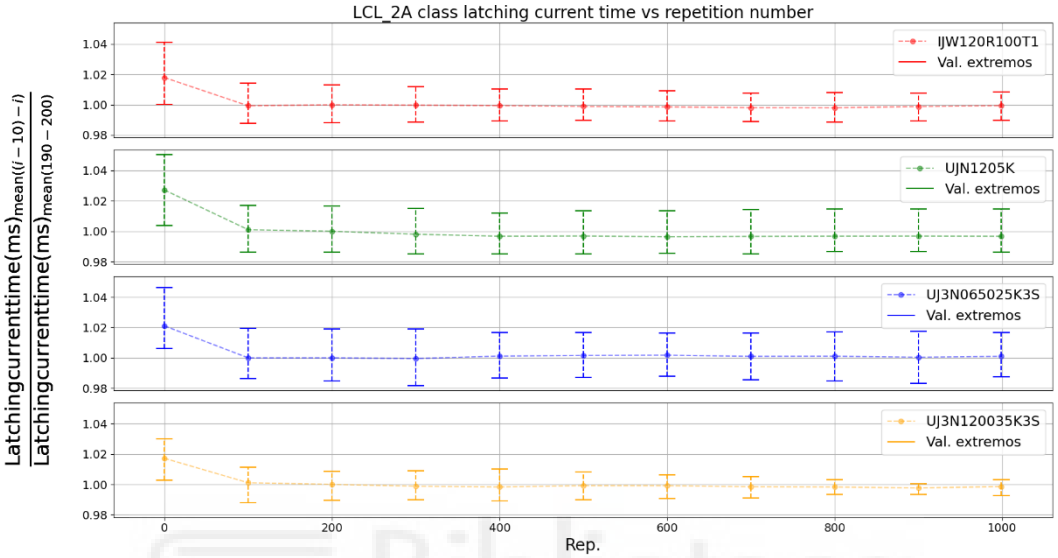


Fig. 52: Ejemplo de representación visual de los resultados de tiempo de desconexión obtenidos mediante los programas de análisis empelando el segundo enfoque de visualización, en este caso se representan cuatro grupos en pruebas de cortocircuito.

4.2. Análisis de datos de caracterización

Como se ha comentado anteriormente, los datos utilizados en el análisis de caracterización han sido proporcionados por la Universidad de Valencia. Debido a que el formato de los ficheros y la forma de organizarlos es sustancialmente diferente a la línea de datos de sobrecarga analizada hasta ahora, es necesario realizar programas específicos para este tipo de datos, de forma que se han desarrollado distintas funciones para su tratamiento, visualización y extracción de parámetros clave.

4.2.1. Introducción y objetivos

La representación de las curvas de caracterización de los dispositivos individuales —como el JFET y el PMOS—, así como del conjunto completo LCL en configuración cascodo, constituye una parte fundamental de este trabajo. Estudiar la forma adecuada de representar estas curvas a lo largo de las distintas campañas de pruebas permite obtener una visión más precisa del comportamiento y la degradación progresiva de los componentes.

En estudios anteriores se hacía hincapié en las curvas de sobrecarga, que proporcionan una perspectiva general del proceso de degradación. No obstante, las curvas de caracterización ofrecen una información mucho más detallada y específica sobre los

parámetros eléctricos clave de los dispositivos, permitiendo detectar con mayor claridad cambios sutiles en sus propiedades.

Es especialmente relevante considerar el conjunto en forma de cascodo incorporado en los LCL como una unidad funcional equivalente a un transistor en sí mismo. Esta visión permite tratar el sistema no solo como la suma de sus partes, sino como un bloque electrónico con características emergentes propias, cuyas curvas de caracterización reflejan de manera integrada la evolución del comportamiento eléctrico de los componentes del cascodo a lo largo del tiempo y del estrés inducido por las campañas de pruebas.

En este contexto, el objetivo principal no es tanto realizar un análisis automático de los datos, sino más bien desarrollar herramientas orientadas a la visualización y organización clara de la información. Esto permitirá una interpretación visual directa de cómo evoluciona la respuesta del sistema, facilitando la comparación entre distintas etapas del envejecimiento.

Para llevar a cabo el tratamiento y análisis de las curvas de caracterización de forma eficiente, es necesario desarrollar una serie de programas específicos que respondan a las particularidades de este tipo de datos. En relación con los objetivos presentados en el apartado 1.5, a continuación, se amplían y detallan exhaustivamente los objetivos relacionados con los programas de análisis de datos de curvas de caracterización:

- Organizar los datos en estructuras jerárquicas, como diccionarios anidados, que reflejen distintos niveles de agrupación como tipo de dispositivo, referencia, campaña y tipo de prueba, facilitando así el acceso y análisis de la información.

- Representar gráficamente grupos de curvas asociadas a dispositivos individuales o conjuntos como el introducido en el LCL en configuración cascodo, siguiendo la jerarquía de la información para mantener una representación coherente y estructurada.

- Incluir la posibilidad de excluir de forma selectiva curvas individuales en caso de que estas presenten anomalías, como errores en la obtención de los datos o comportamientos claramente atípicos, que puedan distorsionar el análisis general.

- Permitir tanto la visualización directa de las curvas como el análisis estadístico básico de los datos.

- Implementar el cálculo de derivadas, especialmente la derivada de la curva $I_D - V_{gs}$, con el fin de obtener la curva de transconductancia (g_m), una representación esencial para la evaluación funcional del dispositivo.

- Aplicar esquemas visuales eficaces que combinen gamas de colores para identificar referencias o dispositivos concretos, y tipos de línea diferenciados para representar distintas campañas, facilitando la interpretación de gráficas con alta densidad de información.

- Incluir herramientas para el cálculo automático del voltaje umbral (V_{th}) a partir de las curvas $I_D - V_{gs}$, empleando métodos como el de corriente umbral o el análisis del punto de máxima pendiente.

4.2.2. Funciones

Como respuesta a los objetivos planteados anteriormente, se han desarrollado una serie de funciones diferenciadas e interdependientes para en análisis de datos de caracterización.

Las funciones que integran el análisis de datos de caracterización se pueden ver en el Anexo 7.

4.2.2.1. Función de organización de datos

En primer lugar, se ha desarrollado una función cuyo objetivo principal es recorrer un conjunto de carpetas organizadas jerárquicamente y construir, a partir de ellas, una estructura de diccionarios anidados que refleje fielmente dicha jerarquía. Esta estructura se genera de forma automatizada y se puede almacenar en archivos '.spydata' —de igual forma que los diccionarios jerarquizados obtenidos en las pruebas de sobrecarga—, lo que permite conservar el estado del entorno de trabajo y facilita un acceso eficiente y organizado a los datos en futuras sesiones de análisis.

El enfoque consiste en replicar la organización de las carpetas físicas dentro de un diccionario en memoria, en el que cada nivel jerárquico representa un criterio específico como el tipo de dispositivo (JFET, PMOS o conjunto LCL), la familia a la que pertenece, la fecha de la campaña o el tipo de prueba realizada. De este modo, el acceso a los datos de caracterización se vuelve mucho más intuitivo y estructurado, permitiendo localizar rápidamente todas las mediciones asociadas a un determinado componente bajo determinadas condiciones. En la Fig. 53 se puede observar una representación visual de la estructura de diccionarios mencionada.

Esta organización no solo agiliza el procesamiento de la información, sino que resulta especialmente útil cuando se trabaja con grandes volúmenes de datos procedentes de múltiples campañas de pruebas, facilitando así el análisis comparativo entre dispositivos, fechas y configuraciones.

Para poder incorporar la información experimental contenida en las carpetas en los diccionarios jerarquizados que se utilizarán en este trabajo, es fundamental comprender la estructura interna de los archivos de datos obtenidos durante las pruebas de caracterización que se aportan de forma externa. Todos estos archivos presentan un formato común y bien definido, lo que permite automatizar su tratamiento de manera eficiente.

En primer lugar, cada archivo contiene diez líneas iniciales dedicadas a mostrar información descriptiva de la prueba, tales como el nombre del ensayo, la fecha, la hora y otras observaciones relevantes. A continuación, se encuentra una línea que especifica el nombre de las variables medidas durante la caracterización, seguida de otra línea que indica las unidades correspondientes a cada una de esas variables. Finalmente, se incluyen múltiples líneas de datos numéricos, en formato separado por espacios, que representan las distintas medidas registradas durante la prueba. En la Fig. 54 se muestra una representación de la estructura mencionada de los archivos recibidos.

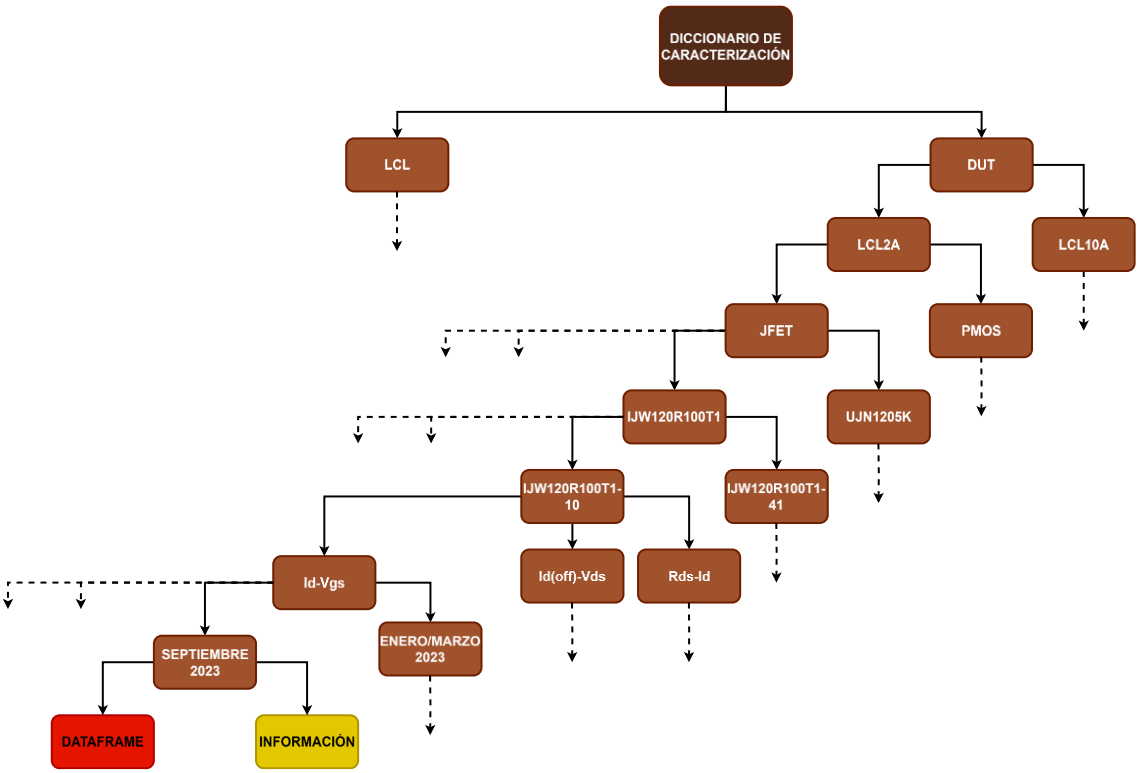


Fig. 53: Representación de la estructura de diccionarios jerárquicos obtenida con la función de organización de datos.

Esta herramienta ha sido diseñada para ser flexible, permitiendo al usuario filtrar qué datos desea cargar según distintos criterios. Entre las opciones disponibles, es posible acotar la carga por rango de fechas, tipo de dispositivo o variables concretas a analizar. Esto facilita trabajar únicamente con la información relevante, agiliza el procesamiento y mejora el control sobre los datos analizados. Además, resulta especialmente útil en contextos donde se generan continuamente nuevos resultados, ya que el sistema está preparado para incorporar información adicional de forma incremental.

Por cada archivo procesado, se generan dos elementos fundamentales dentro de cada eslabón final de la estructura:

1. **Un *dataframe* de Pandas**: Este objeto se compone tal que las columnas corresponden a los nombres de las variables, y las filas contienen los datos numéricos registrados durante la prueba de caracterización.

2. **Una cadena de texto completa**: Que conserva todo el contenido original del archivo, incluyendo las diez líneas iniciales, así como las líneas de nombres de variables y unidades. Esta cadena sirve como metainformación asociada a cada conjunto de datos.

En el caso de que algún archivo no se encuentre en la ruta esperada, o bien no cumpla con la estructura estándar descrita, el programa genera una notificación de error mediante la línea de comandos. Todas estas notificaciones se almacenan en una lista de cadenas de texto, lo que permite una revisión manual posterior y garantiza la trazabilidad de los posibles fallos durante el proceso de carga de datos, ya que incluye la ruta en la que se buscó el archivo donde se produjo el error. Los errores pueden ser de varios tipos, pero

principalmente se dividen entre errores de formato del archivo, donde el archivo no contiene la estructura de información esperada o de falta de archivo.

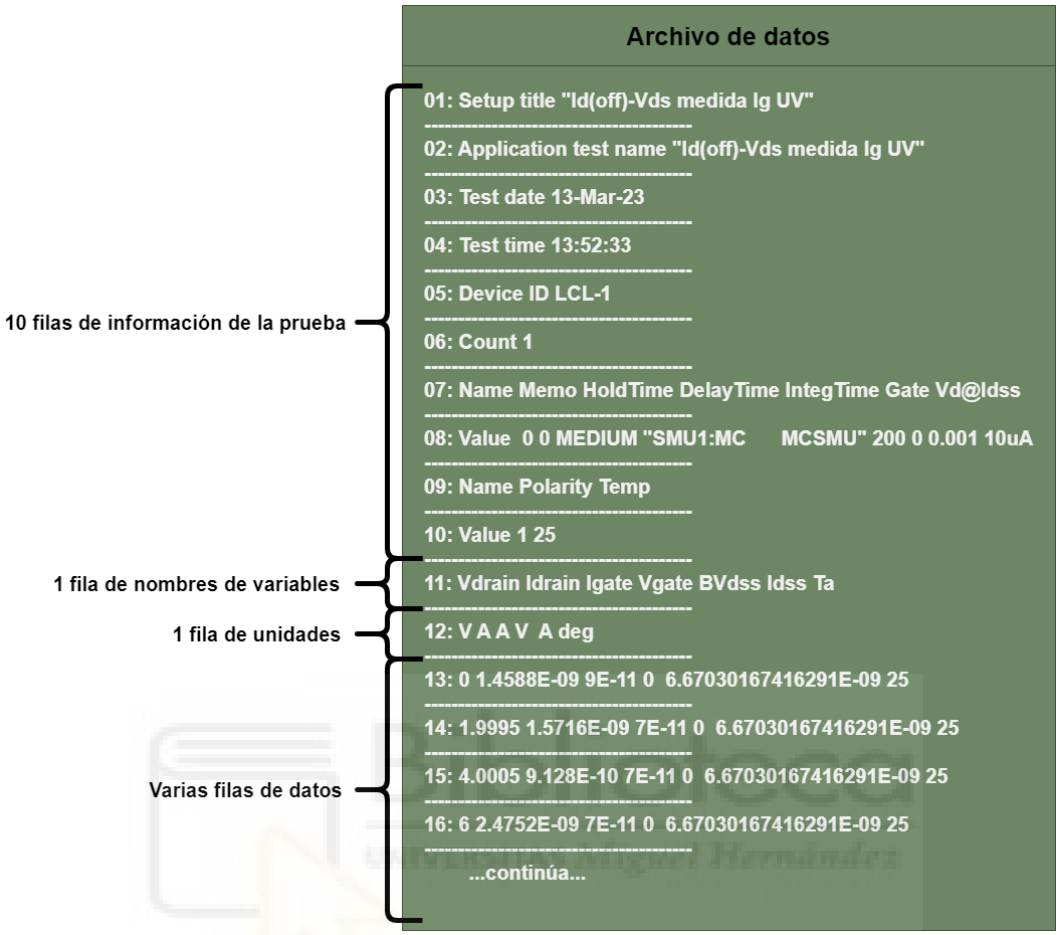


Fig. 54: Representación genérica del contenido de los archivos de los datos de caracterización provistos.

4.2.2.2. Representación gráfica de resultados, primer enfoque

Una vez se dispone del diccionario jerarquizado de datos, se abre la posibilidad de implementar diversas funcionalidades sobre estos. La más inmediata y útil es la representación gráfica de las curvas características obtenidas para los distintos dispositivos. A diferencia de la **herramienta de visualización avanzada** que se describe en el apartado 4.3, esta función está orientada a proporcionar una forma considerablemente más directa, rápida y operativa de representar los datos disponibles.

Esta herramienta permite al usuario seleccionar grupos de dispositivos, ya sean transistores PMOS, JFET, o incluso configuraciones en cascodo —que, para abreviar, se nombran con el grupo de LCL correspondiente— que integran ambos tipos. Además, ofrece la posibilidad de elegir las fechas concretas en las que se realizaron las medidas, así como de excluir determinados dispositivos cuya respuesta se desvíe significativamente del comportamiento general. Esto último es especialmente útil para eliminar curvas que podrían considerarse erróneas o atípicas y que, de estar presentes, dificultarían la interpretación global de los resultados.

El objetivo principal de esta representación no es analizar con detalle el comportamiento individual de cada dispositivo —ya que la superposición de múltiples curvas, correspondientes a distintas fechas y componentes, complica esta tarea—, sino más bien observar de manera conjunta el comportamiento general de una familia de dispositivos. En este sentido, se prioriza la identificación de tendencias globales sobre el análisis de cada componente de forma aislada.

Debido al elevado número de dispositivos que pueden representarse simultáneamente dentro de una misma fecha, se hace indispensable el uso de una codificación por colores que permita distinguir visualmente cada una de las curvas. Para este propósito, se ha recurrido al uso de una paleta de colores categórica denominada ***ColorBrewer***.

ColorBrewer ofrece esquemas de color efectivos inicialmente diseñados para mapas, pero ampliamente utilizados en visualización científica. Sus paletas, clasificadas en secuenciales, divergentes y categóricas, están diseñadas para ser perceptualmente claras, accesibles y útiles tanto en pantalla como en impresión. En este caso, se utiliza la paleta categórica 'Set3' de doce colores para distinguir diferentes dispositivos electrónicos.

Es importante destacar que esta codificación por color se aplica exclusivamente a la representación de los dispositivos de **clase 10**, ya que en este caso se llega a representar hasta 12 curvas diferentes por fecha. En cambio, para los dispositivos de **clase 2**, el número máximo de curvas representados simultáneamente por fecha es significativamente menor —normalmente no más de cuatro—, por lo que no resulta necesario recurrir a paletas tan complejas. En estos casos, se utilizan colores básicos suficientemente diferenciables entre sí sin comprometer la legibilidad del gráfico.

Por otro lado, para diferenciar las curvas correspondientes a distintas fechas de medida —ya que si además de representar varios dispositivos, también se representan esos mismos dispositivos en diferentes fechas, la distinción por colores resulta insuficientes—se han implementado también distintos estilos de línea (como sólidas, discontinuas, punteadas y combinaciones entre estas), lo que permite mantener la legibilidad y claridad de las representaciones incluso cuando se superponen varias curvas pertenecientes al mismo grupo de dispositivos y en diferentes fechas.

En la Fig. 55 se puede comprobar la aplicación de la paleta de colores y los diferentes tipos de línea en una representación con 5 fechas diferentes, además, se muestra la importancia de poder eliminar dispositivos específicos de la representación ya que, el JFET IJW120R100T1-4 presenta un comportamiento anómalo en una de las fechas. Asimismo, en la Fig. 56 se muestra un caso muy similar, en este caso con dispositivos de clase 2—donde no se emplea la paleta de colores específica—, siendo este caso específico el de la necesidad de eliminar una fecha en concreto ya que todos los dispositivos presentan un comportamiento diferente y dificulta la interpretación global.

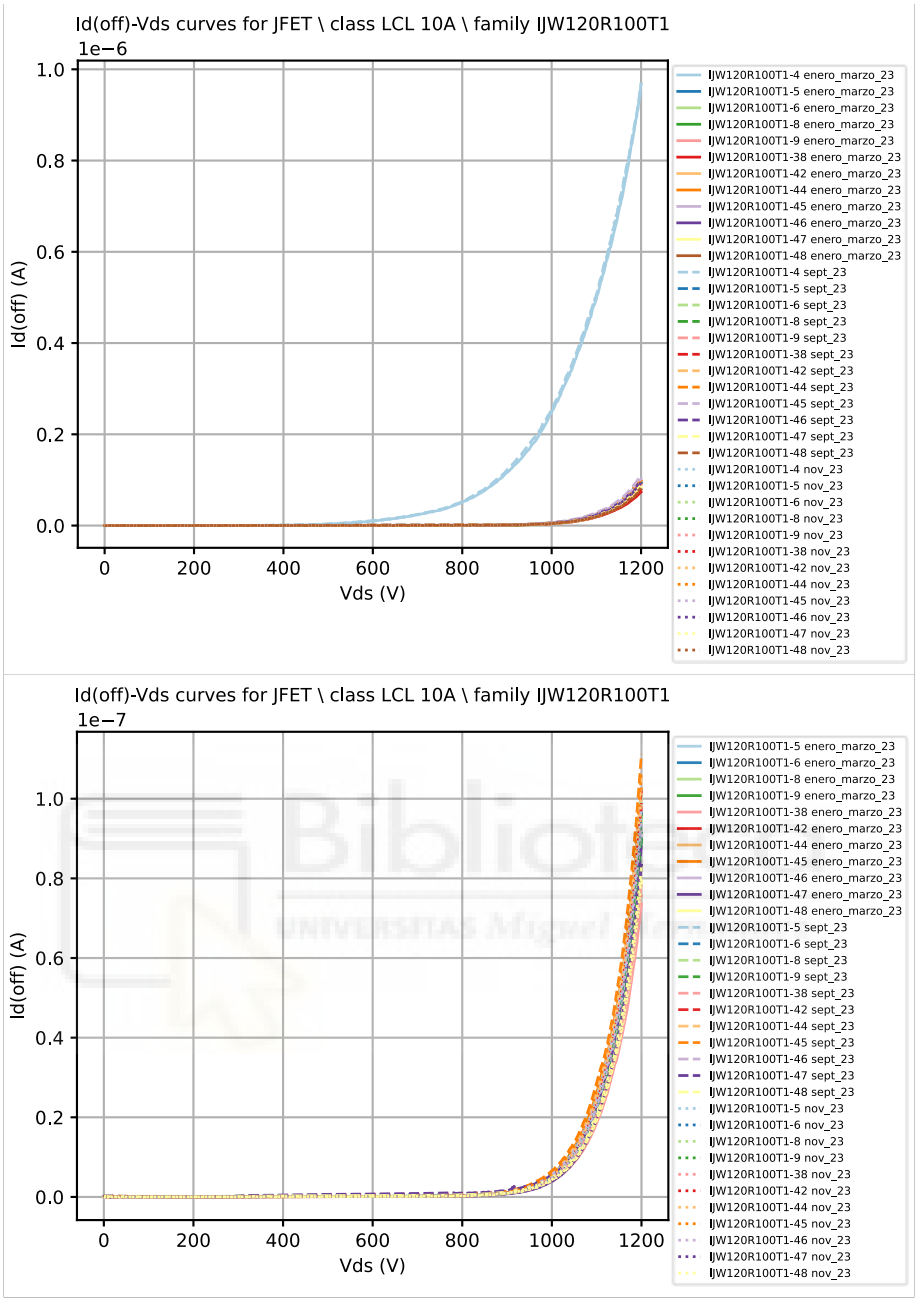


Fig. 55: Gráfica de caracterización de JFET de referencia IJW120R100T1, empleados en grupos de clase 10, de corriente de fuga respecto de tensión entre drenador y surtidor con comportamiento anómalo de un miembro (arriba) y misma gráfica una vez ha sido eliminado el dispositivo del grupo (abajo).

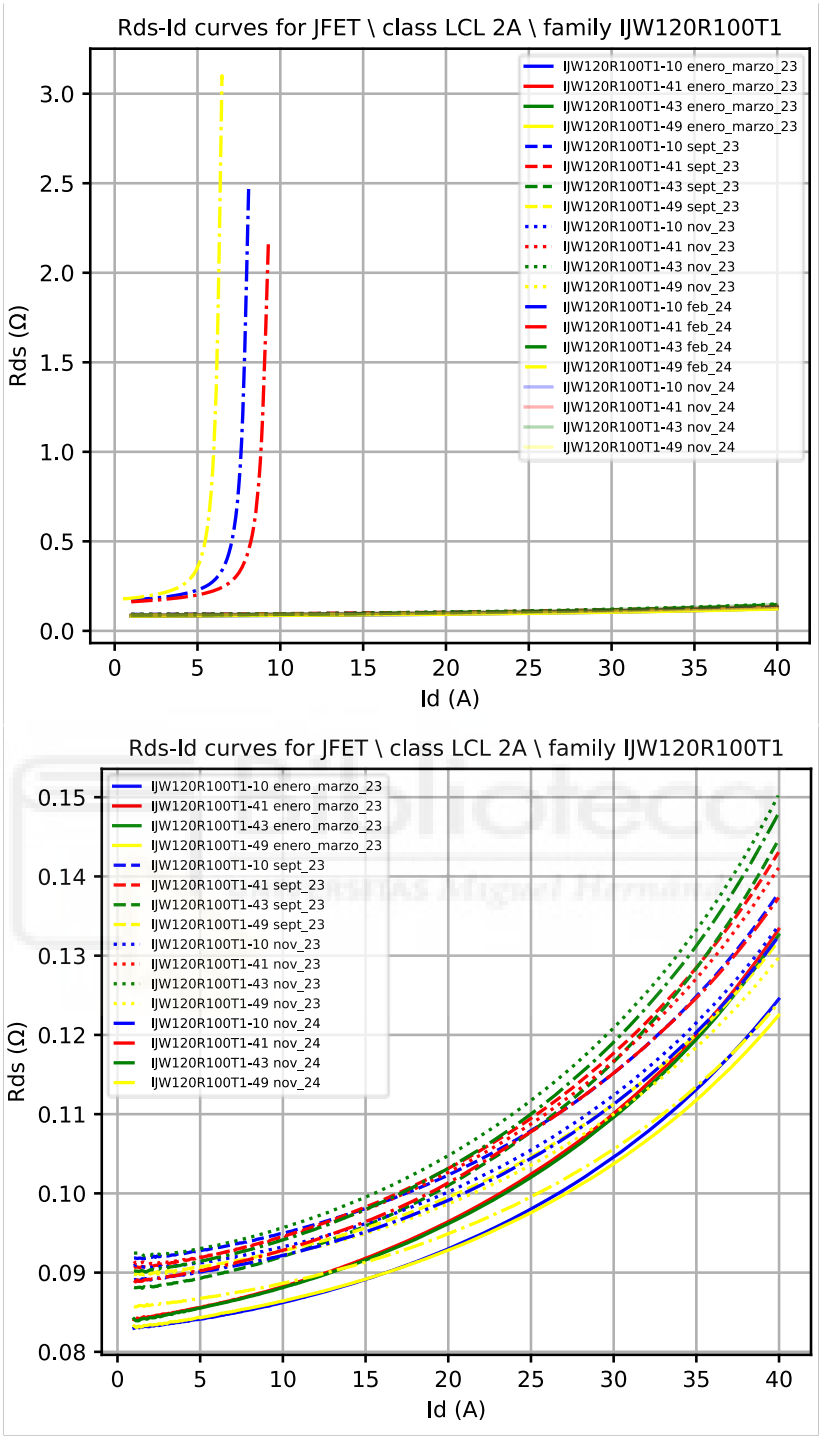


Fig. 56: Gráfica de caracterización de JFET de referencia IJW120R100T1, empleados en grupos de clase 2, de resistencia de canal respecto de la intensidad con comportamiento anómalo de varios miembros en una fecha específica (arriba) y misma gráfica una vez ha sido eliminado la fecha problemática(abajo).

4.2.2.3. Representación gráfica de resultados, segundo enfoque

Como complemento a las representaciones clásicas de las curvas características de los dispositivos, se plantea una forma alternativa de visualizar los cambios de tendencia a lo largo del tiempo mediante el uso de gráficas de violín, agrupadas por fecha de medición. Este tipo de visualización ofrece una perspectiva sintética y estadística del comportamiento colectivo de un grupo de dispositivos. A diferencia de las representaciones convencionales, que permiten distinguir cada dispositivo de forma individual, las gráficas de violín agregan la información estadística de todos los dispositivos de un mismo grupo para cada fecha, lo que impide la identificación específica de cada uno, pero facilita la detección visual de tendencias globales y cambios a lo largo del tiempo.

Para construir estas gráficas, se selecciona un punto representativo de interés en cada curva, dependiente del tipo de medida y del contexto de aplicación del dispositivo. Por ejemplo, en el caso de las curvas de corriente de fuja ($I_{d(off)} - V_{ds}$), se ha elegido el valor de corriente de corte correspondiente a un valor de V_{ds} cercano al límite superior del rango experimental, concretamente el 90% del valor máximo. No obstante, si la aplicación final no requiere tensiones tan elevadas, es posible definir puntos de referencia más bajos y adecuados a las condiciones reales de uso.

Para facilitar la comparación entre fechas, se ha optado por utilizar valores normalizados en lugar de valores absolutos. En concreto, el valor extraído de cada dispositivo se divide entre un valor medio de referencia, calculado como la media de los valores de ese mismo dispositivo durante las primeras fechas (enero y marzo), cuando los dispositivos aún no habían sido sometidos a ningún tipo de prueba adicional más allá de la caracterización inicial. Este valor se considera representativo del estado "base" del dispositivo. De este modo, en las gráficas de violín, el valor normalizado de la primera fecha será igual a 1, y las desviaciones en fechas posteriores indicarán variaciones relativas respecto al comportamiento inicial.

Cada gráfica de violín representa, además de la distribución estimada de los valores, la mediana (línea central) y el rango intercuartílico (área sombreada), siguiendo la convención habitual de este tipo de visualización. Como complemento informativo adicional, se ha incluido un punto rojo en cada violín, que indica el valor medio del grupo para esa fecha. Esta información permite evaluar de forma visual si la distribución está sesgada respecto a su media, ofreciendo una perspectiva más completa de la evolución del parámetro analizado.

Cabe señalar que, en algunos casos —especialmente para los dispositivos de clase 2A, donde el número de muestras disponibles es reducido—, la representación de la distribución en forma de violín se ha realizado mediante una aproximación visual. Esta técnica, basada en métodos de estimación de densidad suavizados, permite mantener la forma característica de la gráfica de violín, aunque la cantidad de datos no sea suficiente para generar una densidad estadísticamente robusta. Esta decisión se ha tomado con el objetivo de mejorar la legibilidad y uniformidad visual del conjunto de gráficas.

En la Fig. 57 se representan las curvas de resistencia de canal correspondientes a un grupo de dispositivos LCL —combinación de JFET y PMOS en configuración de cascodo empleados en el LCL—, obtenidas en distintas fechas de caracterización. A pesar de contener toda la información experimental original, esta representación tradicional

presenta una importante limitación visual ya que la superposición de múltiples curvas dificulta enormemente la identificación de patrones o variaciones progresivas en el comportamiento del grupo. Las posibles tendencias de aumento o dispersión de la resistencia de conducción quedan enmascaradas por el entrelazado de las trayectorias individuales.

Por el contrario, la Fig. 58 muestra las correspondientes gráficas de violín generadas para el mismo grupo de dispositivos comentado, tomando como punto de análisis el valor de resistencia de canal al 90% del valor máximo de I_d en cada curva. Además, los valores representados han sido normalizados respecto a su valor medio en la fecha de referencia inicial (enero/marzo), tal como se ha descrito en el apartado anterior. Esta estrategia de normalización permite comparar valores relativos entre fechas, eliminando el sesgo causado por variaciones absolutas entre dispositivos.

Gracias a esta representación alternativa, se observa con mayor claridad una tendencia al aumento progresivo de la resistencia de canal a medida que los dispositivos son sometidos a sucesivos ciclos de estrés en diferentes fechas. Esta evolución se aprecia no solo en el desplazamiento hacia valores superiores del centro de la distribución (mediana y media), sino también en un aumento de la dispersión de los datos, reflejado por una mayor contracción lateral de las gráficas de violín en las fechas posteriores. Este efecto indica que los dispositivos comienzan a mostrar comportamientos más heterogéneos, probablemente como consecuencia de mecanismos de degradación diferenciados dentro del mismo grupo.

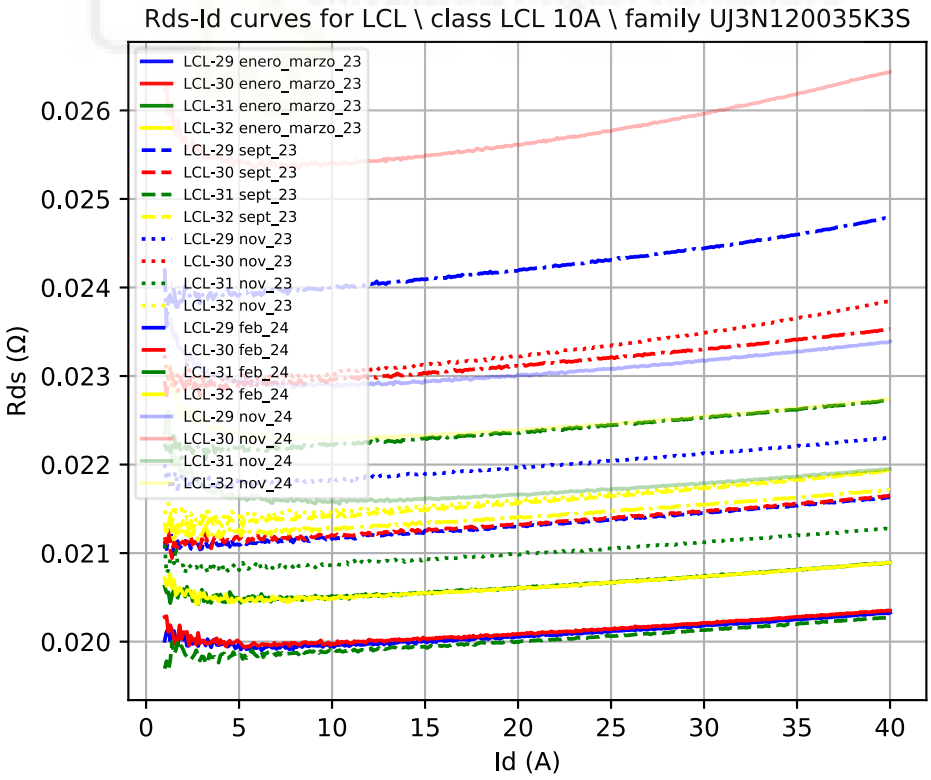


Fig. 57: Gráfica de caracterización de grupo de LCL de clase 10, de resistencia de canal respecto de la intensidad en diferentes fechas.

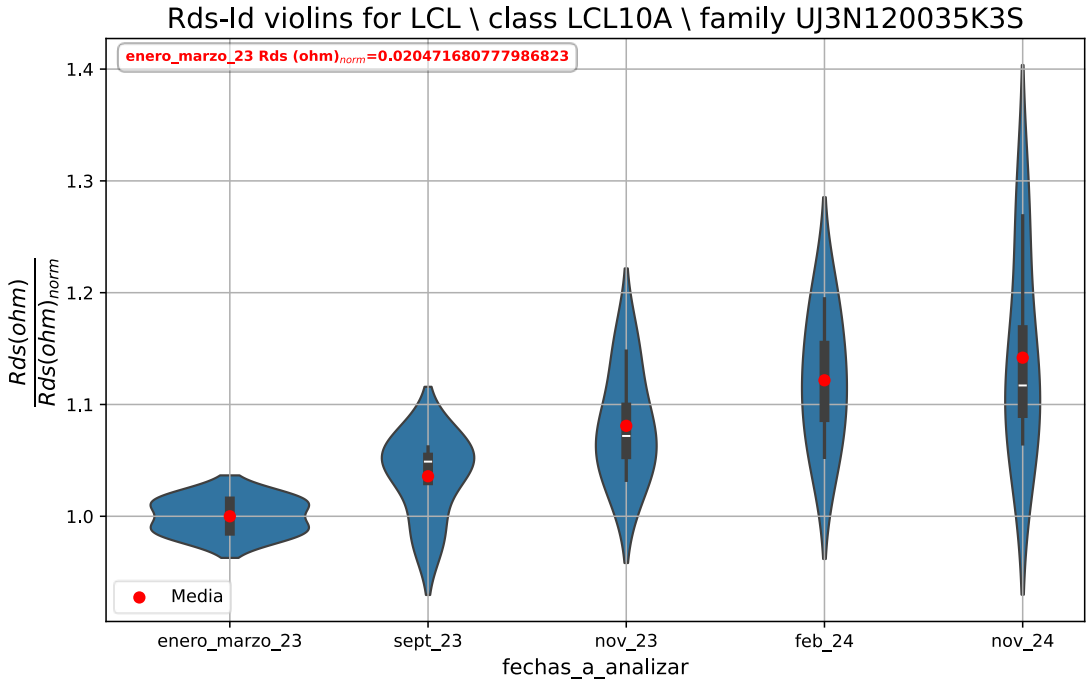


Fig. 58: Gráfica de violín equivalente de la representación de la Fig. 57.

4.2.2.4. Representación gráfica de transconductancia y obtención de voltaje umbral

Además de las representaciones directas de las curvas características, en ciertos casos resulta necesario extraer información derivada de estas curvas para poder llevar a cabo otros tipos de análisis. Este es precisamente el caso de las curvas $I_d - V_{gs}$ (corriente de drenador frente a tensión entre puerta y surtidor), en las que el cálculo del voltaje umbral (V_{th}) es fundamental para la caracterización eléctrica de dispositivos JFET y PMOS, como los analizados en este trabajo.

Tal como se describió en el apartado 2.2, uno de los métodos más utilizados para la determinación del voltaje umbral es el de la extrapolación lineal de la curva $I_d - V_{gs}$. No obstante, la aplicación directa de este método ha presentado dos grandes dificultades en los datos experimentales obtenidos.

La **primera dificultad** reside en la propia morfología de algunas de las curvas obtenidas. En numerosos casos específicos de las curvas d JFET, estas curvas presentan comportamientos no estándar, particularmente en la zona previa al límite superior de corriente impuesto durante la prueba. En estos tramos, se observa un aumento repentino en la pendiente de la curva, lo que genera un pico en la curva de transconductancia ($gm = \frac{dId}{dVgs}$), tal como se muestra en la Fig. 59, la Fig. 60 y la Fig. 61 .Este pico representa un cambio abrupto en la respuesta del dispositivo y no sigue la tendencia usual que caracteriza a la región activa del FET, por lo que no puede ser utilizado como referencia fiable para la construcción de una recta tangente.

Ante esta limitación, se ha desarrollado e implementado un método alternativo basado en el análisis de la curva de transconductancia. En particular, muchas de estas curvas presentan lo que se conoce como mesetas de transconductancia, es decir, tramos en los

que g_m permanece aproximadamente constante a lo largo de un intervalo de V_{gs}, este comportamiento puede apreciarse en la zona resaltada como 'detalle 1' en la Fig. 59. Dado que g_m es la derivada de la curva $I_d - V_{gs}$, una meseta en g_m implica que el tramo correspondiente de la curva original es aproximadamente lineal en esa zona.

El enfoque propuesto consiste en identificar dentro de estas mesetas el segmento con menor variación de valor de transconductancia, lo que equivale a localizar la porción más estable del comportamiento lineal en la curva $I_d - V_{gs}$. Una vez identificado dicho segmento, se construye una recta secante entre sus extremos, y esta se extrapola hasta cortar el eje de abscisas. El punto de intersección se interpreta como una estimación del voltaje umbral del dispositivo.

Para obtener la curva de transconductancia de forma adecuada, se ha optado por aplicar un filtro de Savitz-Golay sobre la curva $I_d - V_{gs}$, derivando numéricamente con un parámetro de tamaño de ventana igual a 21. Esta técnica de suavizado, aplicando este parámetro de ventana, permite preservar la forma general de la señal mientras calcula derivadas más estables que otros métodos, especialmente en presencia de ruido experimental. Aunque el uso de una ventana tan amplia con relación al número total de puntos de las curvas (280) conlleva la pérdida de detalle en zonas con variaciones rápidas, como se evidencia en algunas curvas (véase Fig. 61), se ha considerado que el nivel de suavizado obtenido en la zona de interés —la zona de meseta o aproximadamente lineal— resulta significativamente más adecuado que el obtenido con otras técnicas de derivación o con valores más reducidos de tamaño de ventana, que tienden a no suavizar adecuadamente la zona.

Con la transconductancia suavizada, la función implementada en Python analiza la curva dentro de una zona intermedia de interés, evitando considerar los valores del inicio y del final, los iniciales ya que aún no se entra en la zona casi lineal y los finales debido a que se encuentran en la zona de limitación de corriente forzada por la prueba. En base a esto, se ha seleccionado que el punto inicial por el que se empieza a buscar la meseta es el correspondiente al 20% y el final el 80% de la cantidad total de puntos. El algoritmo recorre la curva punto a punto y, para cada par consecutivo de puntos, calcula la diferencia absoluta de valor. Si esta diferencia es menor que una tolerancia predefinida, se asume que ambos puntos forman parte de un segmento aproximadamente plano. Una forma gráfica de comprender el funcionamiento del programa se aporta mediante la Fig. 62.

El programa almacena todos los segmentos consecutivos que cumplen esta condición y selecciona finalmente el más largo, es decir, el que contiene la mayor cantidad de puntos con una pendiente casi constante. Con los índices del primer y último punto de este segmento, y teniendo en cuenta que el eje abscisas (la tensión V_{gs}) es el mismo tanto para la curva $I_d - V_{gs}$ como para su derivada, se identifican los valores correspondientes en la curva original. A partir de estos, se traza la recta secante y se determina su intersección con el eje de abscisas, lo que permite estimar de forma consistente el voltaje umbral (véase Fig. 63).

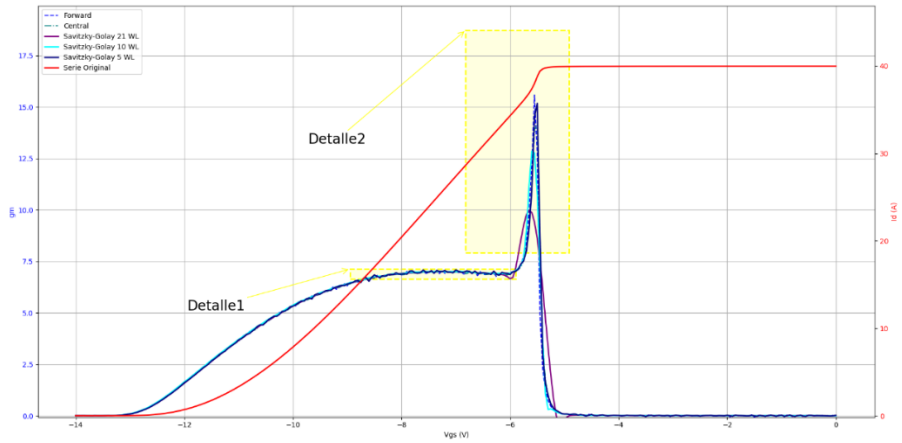


Fig. 59: Gráfica $I_d - V_{gs}$ de JFET con comportamiento anómalo (rojo) y gráficas de transconductancia empleando distintos métodos (azules).

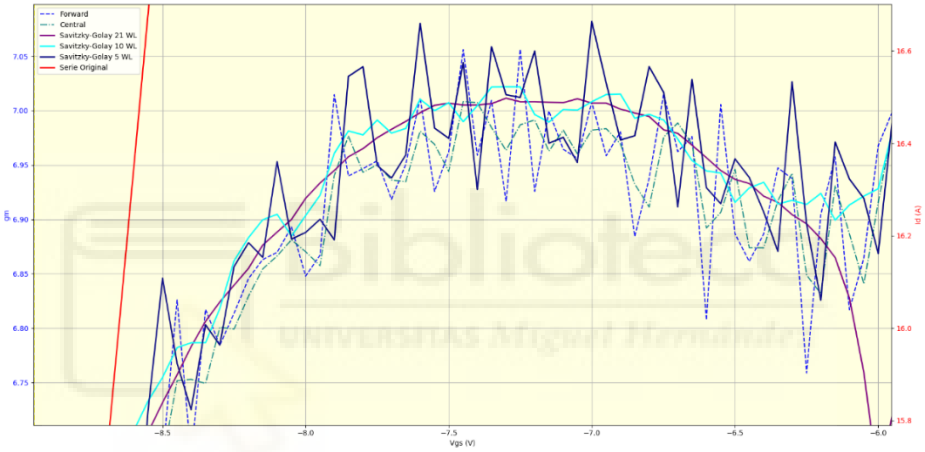


Fig. 60: Detalle 1 de la Fig. 59.

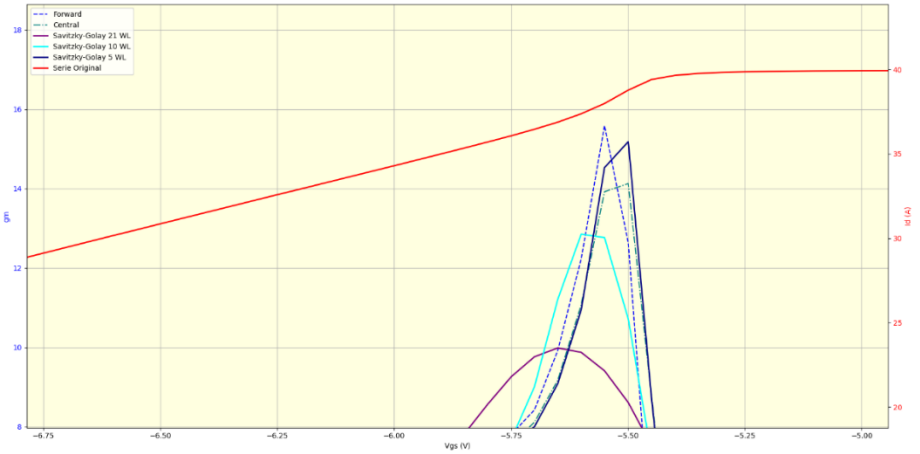


Fig. 61: Detalle 2 de la Fig. 59.

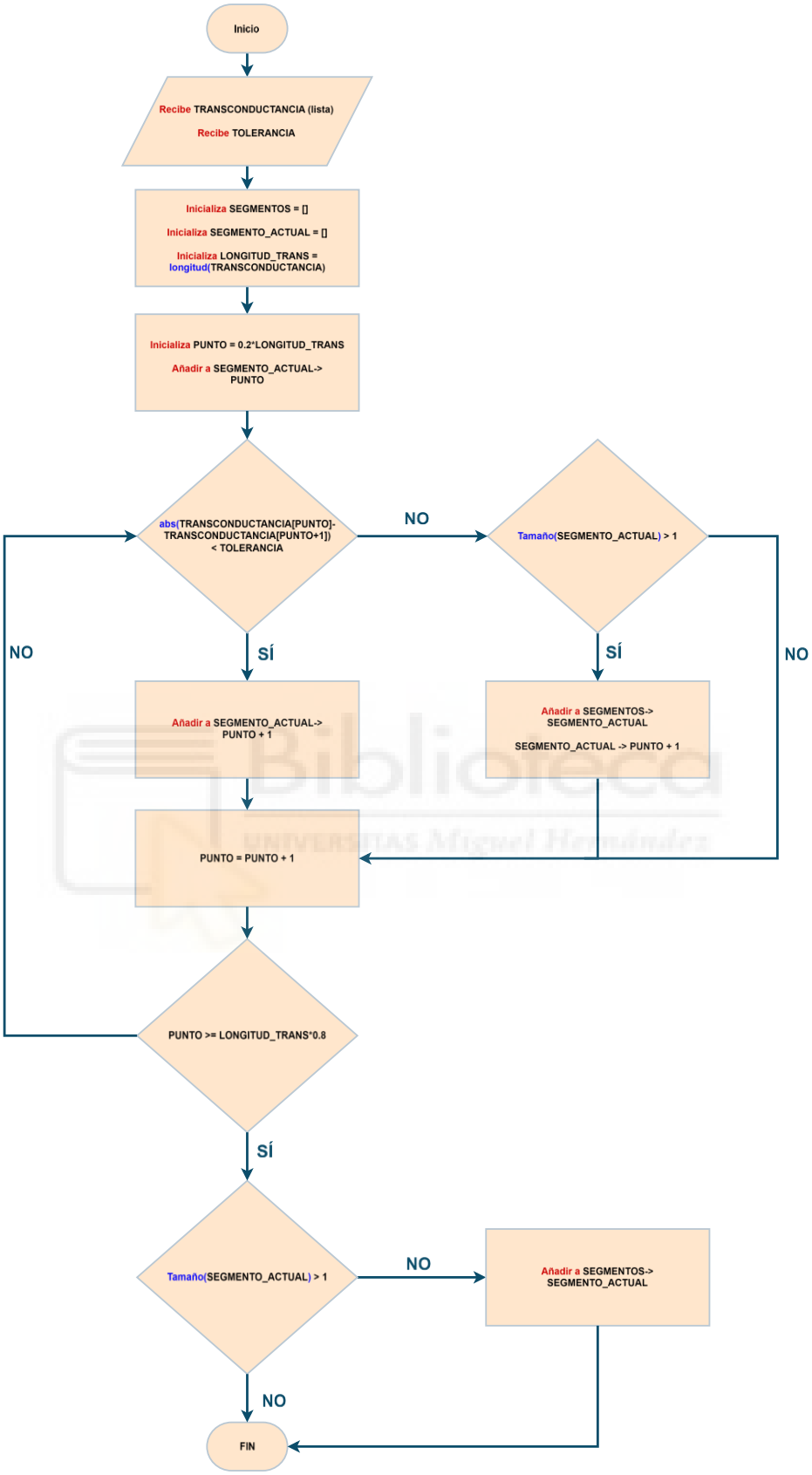


Fig. 62: Diagrama de flujo del programa que encuentra las regiones extrapolables lineales de las curvas de transconductancia con mesetas.

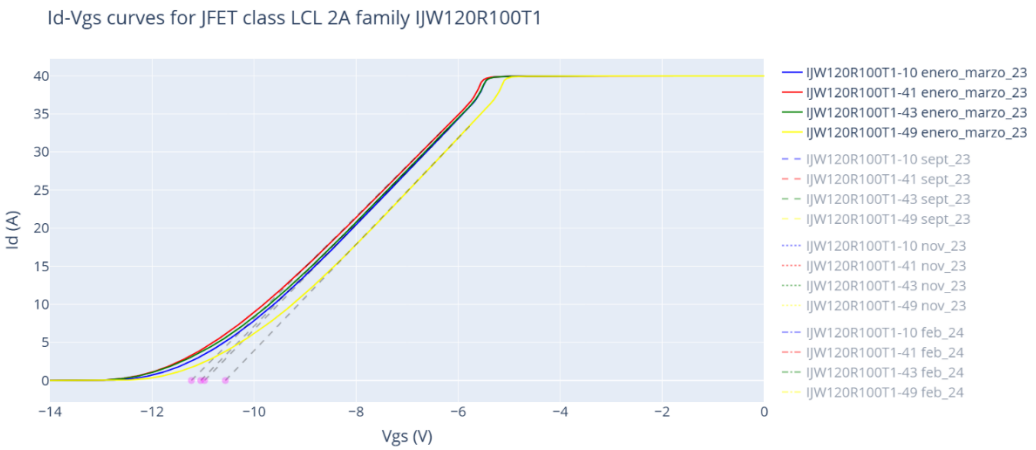


Fig. 63: Curvas de $I_d - V_{gs}$ de JFET de referencia IJW120R100T1 en las fechas de enero/marzojunto con sus rectas de extrapolación lineal y la intersección de estas con el eje de abscisas.

No obstante, este método alternativo basado en mesetas de transconductancia no es aplicable a todos los tipos de dispositivos. En particular, se ha identificado una **segunda dificultad** a la hora de extraer el voltaje umbral en dispositivos cuya curva $I_d - V_{gs}$ y su transconductancia asociada presentan una morfología estándar y continua, sin zonas de meseta evidentes tal como se muestra en la Fig. 64, donde se muestra una curva típica de este estilo. Este es el caso de los dispositivos PMOS y LCL, en los que la curva $g_m - V_{gs}$ muestra un comportamiento suave, pero sin regiones planas claras sobre las que aplicar el criterio descrito anteriormente.

En este tipo de dispositivos, se podría recurrir nuevamente al método clásico de extrapolación lineal a través del punto de máxima transconductancia, sin embargo, aplicar este enfoque no es trivial en la práctica, ya que la identificación precisa del punto de máxima transconductancia se ve fuertemente afectada por el ruido experimental y las irregularidades inherentes a los sistemas de medida. Tal como se pone de manifiesto en el trabajo de Yang e Inokawa [24], incluso en entornos controlados y con dispositivos bien caracterizados, la curva $g_m - V_{gs}$ presenta fluctuaciones locales que pueden alterar la ubicación del máximo, especialmente si no se aplica un filtrado adecuado.

En dicho estudio, los autores proponen una técnica de suavizado diferencial que consiste en aplicar una convolución simétrica sobre la curva Id–Vgs para obtener una derivada más estable. Esta técnica, que se puede considerar conceptualmente equivalente al uso de filtros como el de Savitzky-Golay en nuestro caso, permite atenuar el impacto del ruido. No obstante, los resultados presentados por el trabajo mencionado anteriormente evidencian que el valor y la posición del máximo de transconductancia varían significativamente en función del grado de suavizado aplicado, desplazando el voltaje umbral hacia valores más negativos conforme aumenta el grado de filtrado. Esto implica que el método no es únicamente sensible al ruido, sino también dependiente de los parámetros del algoritmo de suavizado, lo que compromete su fiabilidad como estimador general.

Como se puede observar en la Fig. 64 y la Fig. 65 del presente trabajo, al aplicar distintas técnicas de derivación suavizada sobre las curvas experimentales de los dispositivos PMOS y LCL, se obtienen máximos de transconductancia claramente

distintos, tanto en valor como en ubicación. Esta variabilidad, unida a la dificultad de aplicar un único enfoque homogéneo para todos los tipos de dispositivos analizados (por ejemplo, el caso de las curvas de JFET, cuyas curvas sí presentan mesetas), ha llevado a la conclusión de que el uso del máximo de transconductancia no constituye una técnica fiable ni robusta para este trabajo.

Por todo ello, se ha decidido descartar la estrategia basada en el punto de máxima transconductancia, priorizando así la coherencia metodológica y la estabilidad en la extracción del voltaje umbral en los diferentes dispositivos caracterizados.

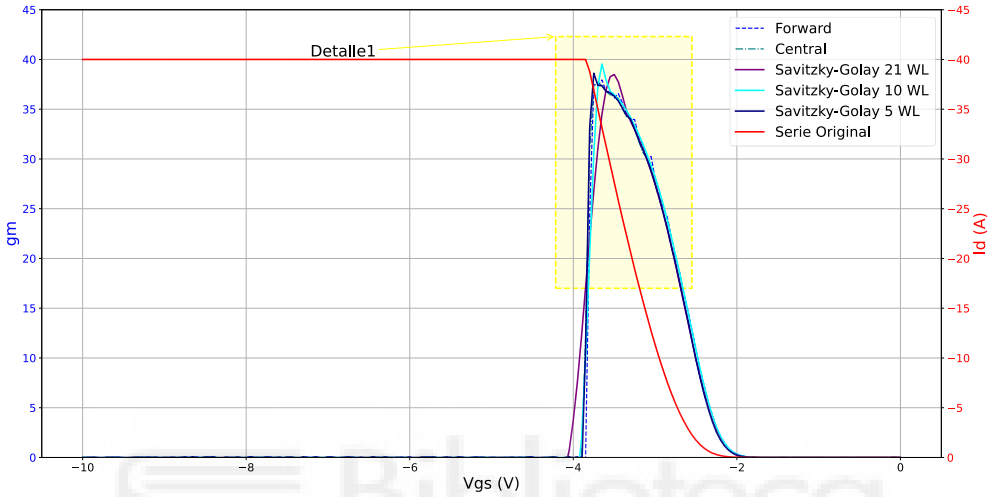


Fig. 64: Gráfica $I_d - V_{gs}$ con comportamiento estándar (rojo) y gráficas de transconductancia empleando distintos métodos (azules).

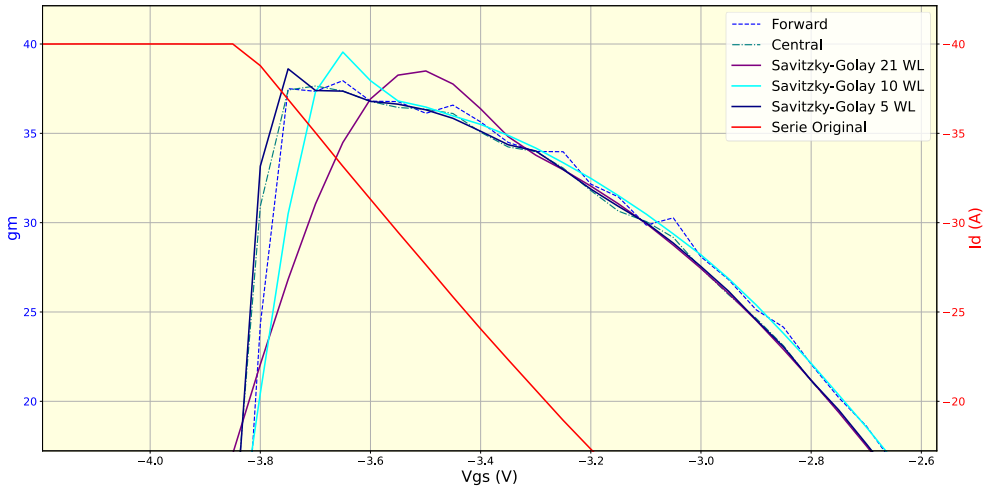


Fig. 65: Detalle de la Fig. 64.

En vista de las dificultades expuestas en los apartados anteriores, tanto en lo referente a la identificación de mesetas en las curvas de transconductancia como en la fiabilidad del método basado en el máximo de transconductancia y la necesidad de consagrar un método unificado, se ha optado finalmente por emplear un método general y aplicable a todos los tipos de dispositivos caracterizados como es el **método de corriente constante**.

El procedimiento en este caso consiste en definir un valor de corriente de referencia, que en este trabajo se ha fijado en 10 mA, y localizar programáticamente en cada curva $I_d - V_{gs}$ el punto donde la corriente de drenador alcanza dicho valor. El valor de tensión de compuerta V_{gs} correspondiente a ese punto se toma como el voltaje umbral estimado. Este método permite una implementación completamente robusta, eliminando la necesidad de derivadas o extrapolaciones que pueden resultar sensibles al ruido o a la estructura local de la curva.

Con el objetivo de acompañar y visualizar los resultados obtenidos a partir de este criterio, se ha optado por generar gráficos de barras agregados, en los que se incluyen todos los valores de voltaje umbral obtenidos a lo largo de las distintas fechas de medida. Esta representación gráfica permite, mediante una inspección visual rápida, identificar posibles variaciones o tendencias en la evolución del voltaje umbral con el tiempo, lo que es de especial interés en estudios de fiabilidad, envejecimiento o variabilidad de proceso. En estas gráficas se ha empelado el mismo código de colores por barra que los utilizados en las representaciones ‘crudas’ de curvas.

En el caso particular de los dispositivos JFET, para los cuales se ha podido aplicar tanto el método de las mesetas como el método de corriente constante, se han generado dos conjuntos de gráficos comparativos. Tal como se observa en la Fig. 66 y la Fig. 67, los valores de voltaje umbral obtenidos mediante el método de las mesetas son significativamente más bajos que los calculados por corriente constante. Esta diferencia pone de manifiesto que, aunque el método de corriente constante es útil y práctico, su resultado depende fuertemente del valor de corriente seleccionado como umbral, lo que introduce un cierto grado de arbitrariedad y puede limitar su precisión en determinados análisis.

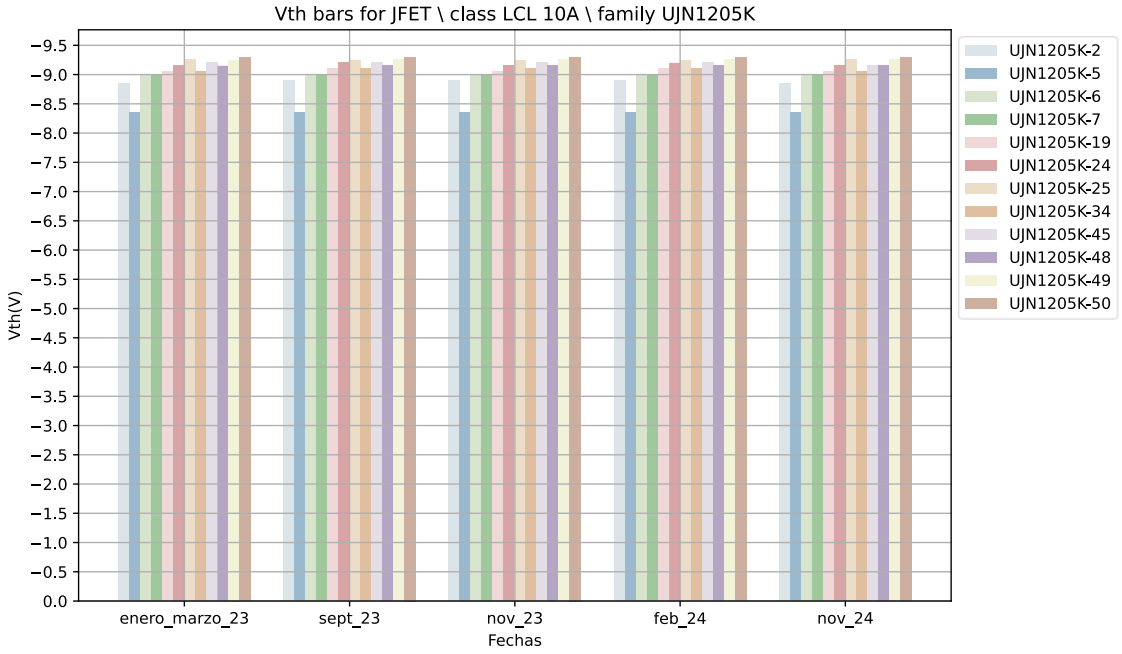


Fig. 66: Gráfica de barras del valor del voltaje umbral en JFET de la familia UJN1205K empleando el método de corriente constante (10mA).

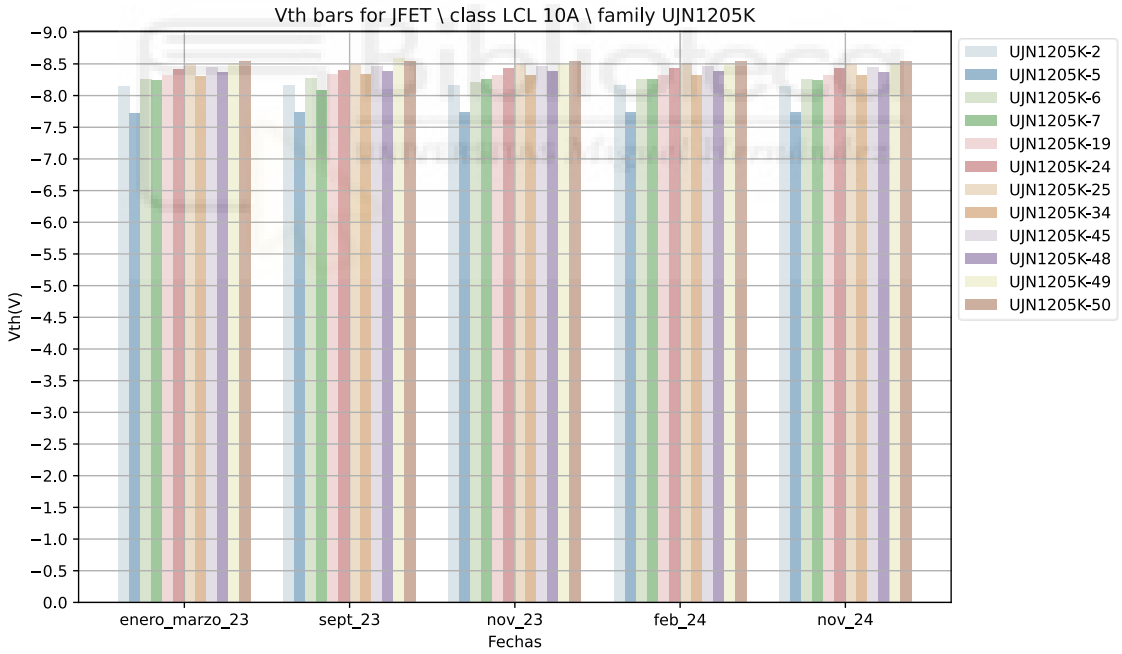


Fig. 67: Gráfica de barras del valor del voltaje umbral en JFET de la familia UJN1205K empleando el método de extrapolación lineal en la región de la meseta —método específico para curvas anómalas obtenidas de varios JFET—.

4.3. Herramienta de cribado y visualización

Para facilitar el desarrollo de software de análisis de curvas y de caracterización es crucial disponer de herramientas que permitan visualizar los datos en crudo de una forma cómoda y eficiente. Visualizar los datos originales sin ningún procesamiento adicional es

fundamental para detectar, de manera temprana, inconsistencias, errores o artefactos en la adquisición de la información. Esta capacidad de inspeccionar los datos en su forma bruta es esencial para validar que el software de análisis está obteniendo resultados consistentes y correctos, ya que permite confirmar que las señales capturadas reflejan fielmente el comportamiento real del sistema. Además, esta visualización directa posibilita realizar ajustes y correcciones inmediatas en los algoritmos de análisis, asegurando que los parámetros de los programas se establecen sobre una base de datos precisa.

4.3.1. Objetivos, Alcance y limitaciones

Con el objetivo de optimizar la visualización de datos y maximizar la eficiencia en el análisis, se desarrollan los objetivos especificados en el apartado 1.5:

- **Funcionalidad versátil y adaptable:** Se pretende que la aplicación sea plenamente funcional para los análisis de sobrecargas repetitivas y los de caracterización. Además, se busca que la herramienta sea lo suficientemente versátil como para adaptarse a otros tipos de datos, promoviendo la reutilización y facilitando su integración en distintos contextos de análisis y experimentación.

- **Personalización y exportación de resultados:** La herramienta deberá ofrecer múltiples opciones de personalización en la visualización de gráficos y datos. Esto incluye la posibilidad de ajustar escalas, títulos y otros parámetros visuales según las necesidades específicas del usuario. Asimismo, se contemplará la funcionalidad de guardar y exportar los resultados en diversos formatos, lo que permitirá su posterior análisis y presentación en otras plataformas.

- **Visualización múltiple para análisis comparativo:** Se implementará la capacidad de visualizar múltiples conjuntos de datos de forma simultánea en forma de distintos gráficos. Esta funcionalidad facilitará la comparación directa entre muestras, permitiendo identificar patrones, tendencias y diferencias de manera clara y eficiente, y aportando un valor añadido en el proceso de toma de decisiones.

- **Operaciones matemáticas integradas:** La herramienta incorporará la posibilidad de realizar operaciones matemáticas básicas sobre los datos, tales como sumas, restas, multiplicaciones y divisiones. Además, se incluirán funciones para el cálculo de parámetros estadísticos relevantes, como promedios y desviaciones, lo que permitirá un análisis cuantitativo más profundo y automatizado de la información.

- **Escalabilidad e integración con otros sistemas:** Se prevé que la arquitectura del programa permita su escalabilidad, facilitando la incorporación de nuevas funcionalidades y la adaptación a futuros requerimientos, lo que posibilitará su integración en flujos de trabajo existentes y su utilización en proyectos de diversa índole.

Dadas las restricciones en cuanto a conocimientos disponibles y el tiempo asignado para el desarrollo del proyecto, la herramienta se concibe principalmente como un prototipo orientado exclusivamente al cumplimiento de los objetivos establecidos en el marco del TFG y en la línea de investigación asociada. En consecuencia, se imponen las siguientes limitaciones:

- **Desarrollo y distribución:** La herramienta no ha sido diseñada ni desarrollada con los estándares propios de un producto comercial o de uso profesional. Su implementación se ha llevado a cabo exclusivamente en el entorno de Python, sin contemplar su distribución o integración en otros entornos o plataformas. Esto implica que su utilización se limita al contexto académico y experimental para el cual ha sido creada.
- **Control de errores y robustez del sistema:** Debido a las restricciones temporales y a la naturaleza experimental del trabajo, el manejo de errores se ha focalizado en la captura y gestión de aquellos fallos críticos que puedan afectar el flujo principal de la aplicación o comprometer el funcionamiento del dispositivo. Las validaciones y controles de errores son, por tanto, parciales y no abarcan todos los posibles escenarios o condiciones de uso, lo que se traduce en una robustez limitada frente a incidencias no previstas.

- **Lectura y manejo de datos:** La herramienta se limitará a la lectura de datos en formato CSV y estructuras de tipo *dataframe* propias de Python, lo que permite un manejo uniforme y simplificado de la información con estas estructuras de datos ampliamente utilizadas. Esta restricción favorece la integración y el procesamiento de datos sin la necesidad de desarrollar adaptadores para otros formatos, asegurando un enfoque eficiente y acorde con los objetivos académicos del trabajo.

- **Interfaz de usuario y funcionalidades avanzadas**: La herramienta no incorpora opciones de personalización de la interfaz ni funcionalidades habituales en aplicaciones de carácter comercial, como la configuración de temas visuales, el guardado de sesiones de trabajo, la exportación automática de gráficos o la integración con sistemas externos de análisis.

4.3.2. Interfaz principal, funcionalidades y módulos

Como se puede observar la Fig. 68, La ventana principal del programa se organiza en módulos claramente diferenciados que facilitan la interacción del usuario y el manejo de datos:

Opciones principales

En la esquina superior izquierda se ubica un conjunto de botones que conforman las **opciones principales**, orientadas a la gestión y procesamiento de la información. Cada botón posee una función específica que, en conjunto, optimiza el flujo de trabajo y permite un manejo intuitivo y ordenado de las tareas.

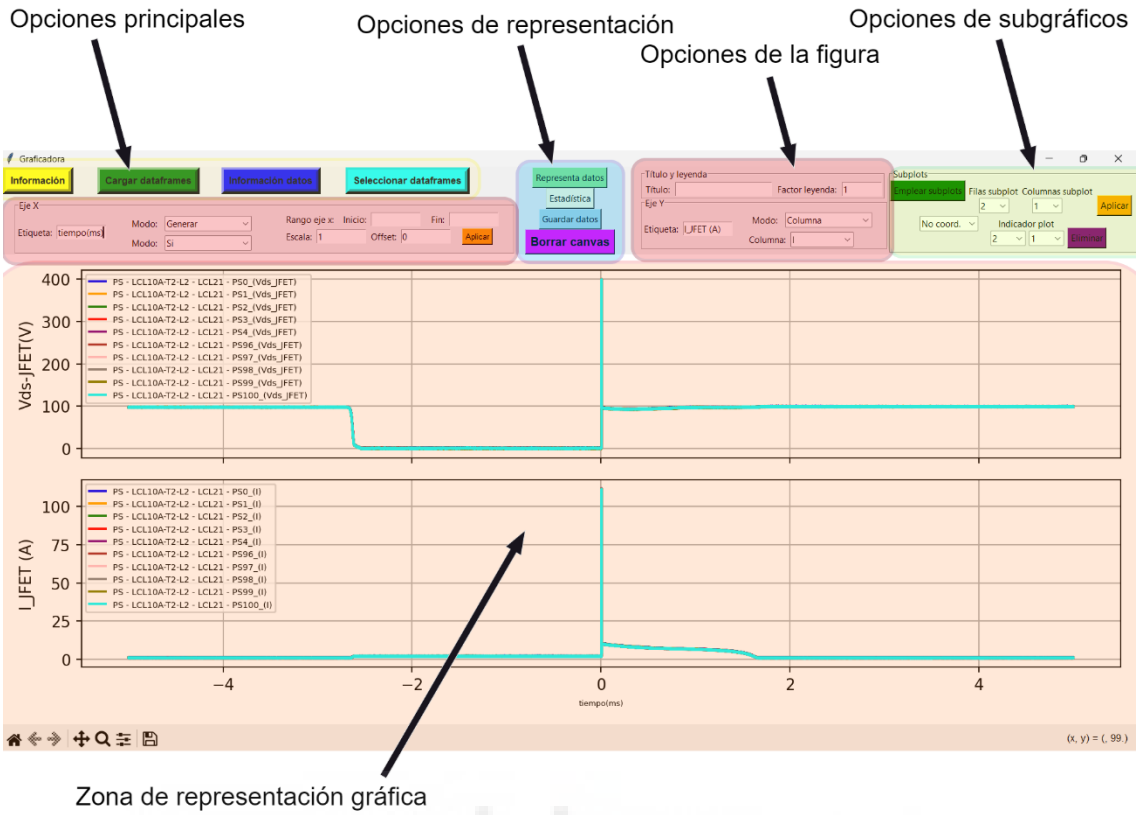


Fig. 68: Pantalla principal de la herramienta de visualización de datos.

El primer botón corresponde a la opción **Información**. Al activarlo, se despliega una ventana secundaria que actúa como una guía introductoria, diseñada para ofrecer al usuario una explicación breve pero precisa sobre las funcionalidades disponibles en la pantalla principal. Esta ventana no solo describe las distintas opciones, sino que también resalta consejos útiles y recomendaciones para aprovechar al máximo la interfaz, haciendo énfasis en los puntos críticos del manejo de datos.

El segundo botón, **Cargar *Dataframes***, es fundamental para el funcionamiento del programa, ya que facilita la importación de datos que se utilizarán en el análisis, almacenando diferentes grupos de *dataframes* para su posterior uso, tal como se muestra en la Fig. 69.

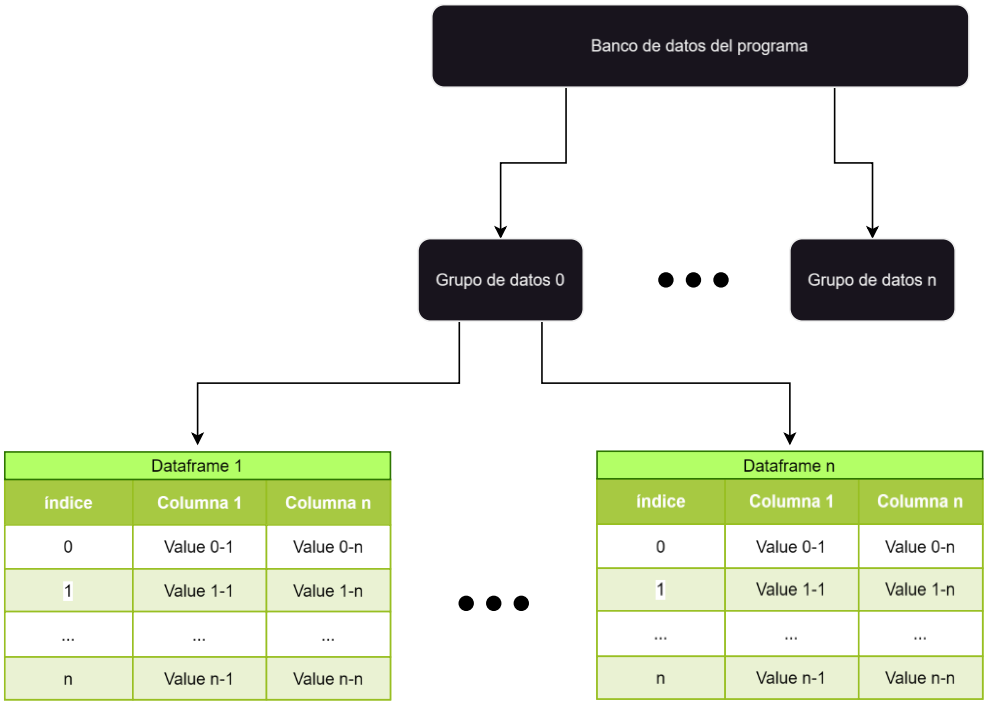


Fig. 69: Diagrama de tratamiento de datos del programa.

La herramienta permite dos métodos distintos de carga, adaptados a diferentes necesidades del trabajo:

- **Cargar Archivos CSV**: Esta modalidad se considera la opción principal, aprovechando la popularidad y estandarización de los archivos CSV para el manejo de datos. El usuario puede explorar su sistema en busca de archivos CSV y seleccionarlos para importarlos. La ventana del programa asociada se muestra en la Fig. 70 y el proceso se realiza de la siguiente manera:
    - Se analiza uno de los archivos seleccionados para identificar la estructura de datos, extrayendo los nombres de las columnas (en caso de estar presentes) y la información contenida en ellas.

    - Posteriormente, el usuario tiene la posibilidad de elegir qué columnas desea importar, renombrarlas según las necesidades del análisis y asignar un identificador único al grupo de datos. En cuanto al identificador de los *dataframes*, se basa en el nombre del archivo, el cual se espera que siga un formato organizado y coherente.

    - Es imprescindible que los archivos seleccionados compartan el mismo formato y estructura para asegurar la correcta integración de la información, lo que evita inconsistencias durante el procesamiento.

- **Cargar Estructuras Jerarquizadas de *dataframes***: Esta opción, aunque de uso menos frecuente, resulta especialmente útil para usuarios que han almacenado datos en archivos con extensión '.spydata', típicos del entorno Spyder. Estos archivos contienen las variables del espacio de trabajo que guardan los diccionarios jerarquizados. La ventana del programa asociada a este tipo de carga se muestra en la Fig. 71 y el proceso para esta modalidad es el siguiente:
    - El usuario asigna un identificador al grupo de datos que se desea cargar.

- Se seleccionan los archivos '.spydata' desde el sistema, y la herramienta los analiza en busca de diccionarios que contengan la información necesaria, si no contienen diccionarios, no permitirá seguir adelante con ese archivo.

- Una vez identificado el diccionario relevante, se abre una nueva ventana que permite navegar a través de la estructura jerarquizada para localizar los *dataframes* (Fig. 72).

- Cada *dataframe* se identifica automáticamente según su ruta en el diccionario, aunque se brinda la posibilidad de simplificar estos nombres eliminando partes redundantes, lo que facilita la identificación y posterior manejo de los datos.

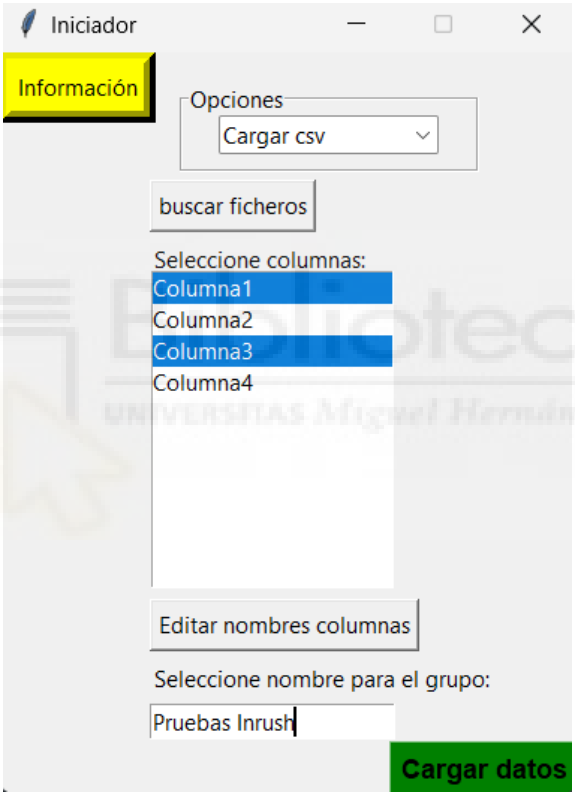


Fig. 70: Ventana de carga de archivos CSV.

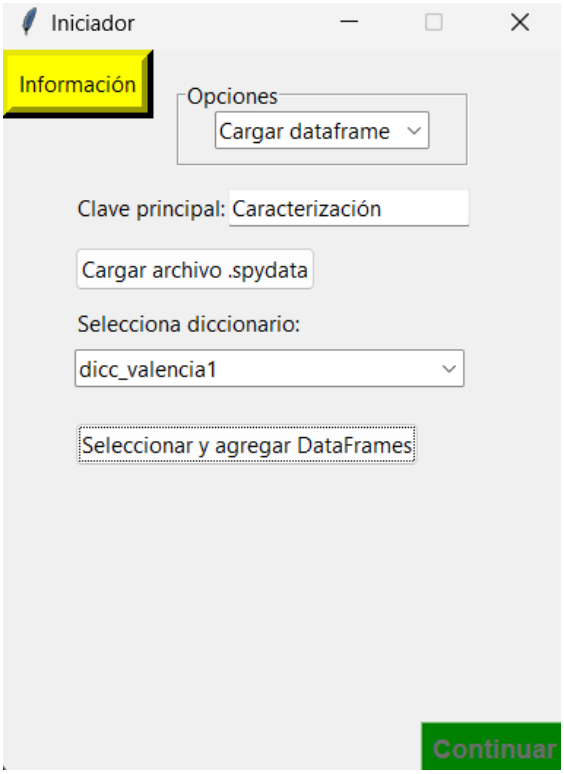


Fig. 71: Ventana de carga de estructuras jerarquizadas de dataframes

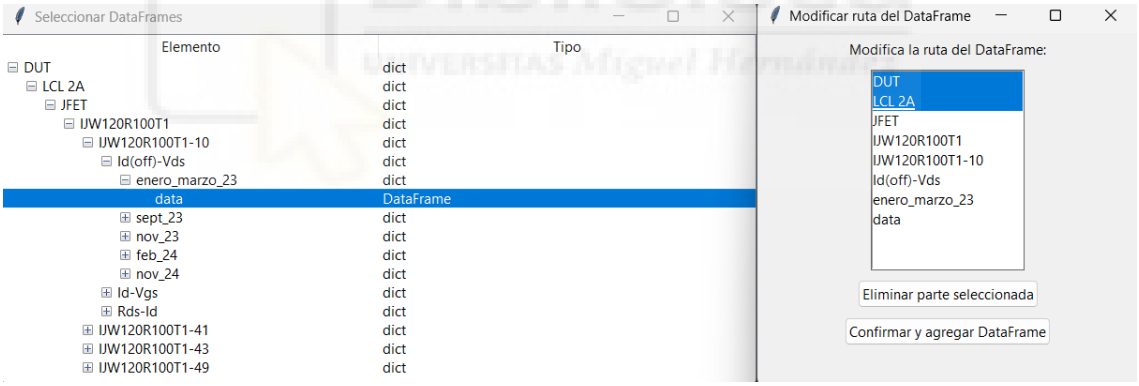


Fig. 72: Ventana de selección de dataframes dentro de la estructura jerarquizada del diccionario

El tercer botón, **Información de Datos**, se activa una vez que se han importado datos al programa. Esta opción permite al usuario visualizar de forma clara y detallada todos los grupos de datos disponibles. Además, ofrece una vista previa del contenido de cada *dataframe* (Fig. 73), lo que es fundamental para:

- Verificar que la carga de datos se haya realizado correctamente.
- Revisar la estructura y consistencia de los datos.
- Identificar rápidamente posibles errores o discrepancias que puedan requerir una revisión o reestructuración antes de proceder con análisis más avanzados.

En este apartado, el usuario dispone además de opciones de eliminación:

- Al seleccionar un *dataframe* dentro de un grupo, puede eliminarlo por completo. Esto resulta muy útil cuando, tras la carga, se detecta que ciertos archivos CSV

contenían información dañada o que la prueba no se realizó adecuadamente, de modo que dichos *dataframes* no aportan valor para el análisis.

- Si lo desea, también puede eliminar un grupo completo de *dataframes*, liberando así memoria RAM cuando ya no vaya a utilizarse.

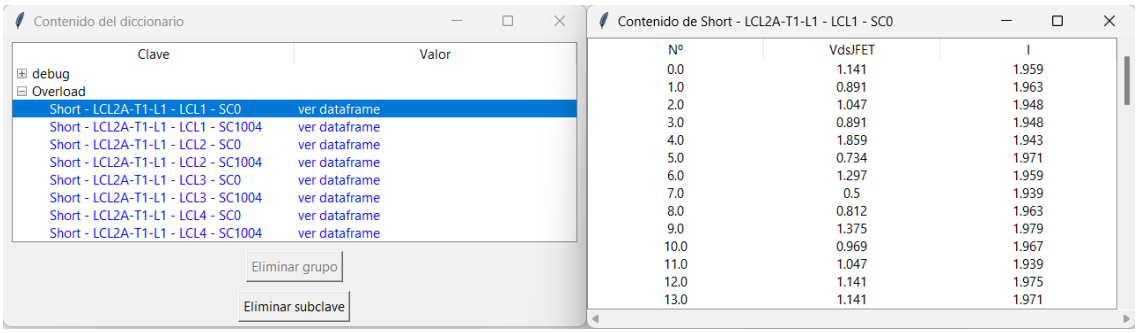


Fig. 73: Ventana de información de datos (izquierda) y ventana de vista previa de dataframe (derecha).

El cuarto botón, **Seleccionar *Dataframes***, está diseñado para afinar el proceso de análisis. A través de este botón se accede a la ventana que se muestra en la Fig. 74, en esta ventana, el usuario puede:

- Determinar de manera precisa sobre qué grupo de datos desea trabajar en cada momento.

- Seleccionar individualmente los archivos o subconjuntos de datos que serán objeto de un análisis más detallado, de esta forma, puede trabajar solo sobre los *dataframes* seleccionados dentro del grupo.

- Organizar y filtrar la información de acuerdo con criterios específicos, lo que permite enfocar los recursos del programa en los conjuntos de datos que resulten más relevantes para el estudio o experimento en curso.

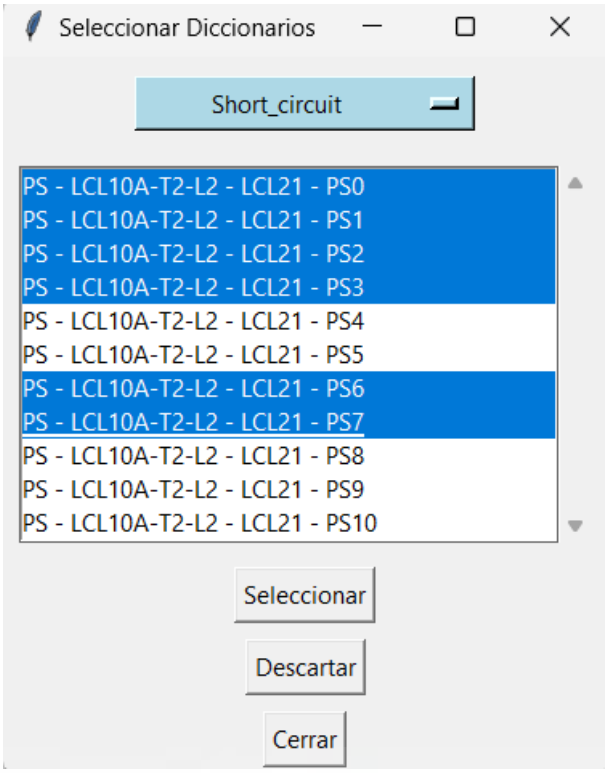


Fig. 74: Ventana de selección de dataframes.

Opciones de subgráficos

Ubicado en la esquina superior derecha de la ventana principal, el módulo de **subgráficos** está diseñado para potenciar la visualización y el análisis comparativo de datos. Este componente permite al usuario configurar una matriz de gráficos en la zona de representación visual, con una disposición de hasta 2 filas por 3 columnas. Gracias a esta funcionalidad, el usuario puede:

- **Seleccionar la ubicación de cada gráfico:** El usuario tiene la libertad de elegir en qué celda de la matriz se realizará la siguiente representación gráfica, facilitando la organización y comparación de diferentes conjuntos de datos.

- **Eliminar gráficos específicos:** En caso de que se requiera, es posible eliminar un gráfico concreto de la matriz, lo que permite ajustar la visualización a las necesidades del análisis o a la relevancia de la información mostrada.

Además, el módulo incorpora una funcionalidad avanzada para coordinar los ejes de los gráficos. Esta opción se puede aplicar de dos maneras:

- **Coordinación global de ejes:** Permite sincronizar los ejes de todas las gráficas dentro de la matriz. Esta coordinación es especialmente útil cuando se desea comparar datos que comparten escalas o rangos similares, asegurando que las variaciones sean comparables visualmente. Se permite compartir ambos ejes o solo el eje x.

- **Coordinación por columna:** También es posible sincronizar únicamente los ejes de las gráficas que se encuentran en la misma columna. Esta funcionalidad facilita la comparación de datos que, aunque se encuentren en diferentes filas, pertenecen a una misma categoría o comparten características similares. De igual forma, se permite compartir ambos ejes o solo el eje x.

Opciones de la figura

Ubicado en la parte superior de la ventana principal y dividido en dos áreas, este módulo permite configurar de manera detallada las opciones de visualización de la gráfica que se va a generar. Se compone de tres secciones claramente diferenciadas:

- **Sección de Título y Leyenda:**
  - *Título de la Gráfica*: Permite definir el título que se mostrará en la próxima gráfica, facilitando la identificación del contenido o el propósito del gráfico.

  - *Ajuste de la Leyenda*: Permite modificar el tamaño de la leyenda mediante un factor multiplicador, adaptándolo a las necesidades específicas de la representación. Esta función es especialmente útil en matrices de gráficos, ya que, al disponer de menor área para cada gráfica individual, el tamaño de la leyenda puede resultar molesto a la hora de visualizarlo, o incluso, ser demasiado pequeño como para poder leer su contenido. Además, el contenido de la leyenda se corresponde automáticamente con el identificador asignado a cada conjunto de datos.

- **Sección del Eje Y:**
  - *Etiqueta del Eje Y*: Permite personalizar la etiqueta que se mostrará en el eje vertical, facilitando la comprensión de la unidad o concepto representado.

  - *Modo de Configuración*: Ofrece dos opciones:
    - **Modo Columna**: Los puntos del eje Y se extraen de una columna específica de los *dataframes* del grupo seleccionado, ideal para representaciones directas sin transformaciones.

    - **Modo Math**: Permite realizar operaciones matemáticas básicas entre las columnas de los *dataframes*, incluso combinando operaciones con escalares. Esta opción es esencial para análisis profundos, como calcular y graficar la potencia disipada por un componente a partir de los datos de corriente y voltaje ya contenidos en el *dataframe*.
- **Sección del Eje X**
  - *Etiqueta y Configuración*: Similar a la del eje Y, pero sin opción para operaciones matemáticas, ya que generalmente no se requiere este tipo de transformación en el eje horizontal.

  - *Generación Automática del Eje X*: Incorpora la opción “Generar”, que permite crear un eje X basado en una escala y un offset definidos por el usuario. Es fundamental que el número de puntos generados coincida con el número de puntos del eje Y para que se pueda efectuar la representación, el número de puntos lo determina automáticamente el programa.

  - *Visibilidad del Eje X*: Permite activar o desactivar la visualización del eje X, lo cual es especialmente útil en matrices de gráficos. Cuando varias gráficas en la misma columna comparten un eje X idéntico, se puede

suprimir la visualización en las filas superiores para evitar redundancias y optimizar el espacio visual.

Opciones de representación

En la parte superior central se ubica el módulo de **representación de datos**, el cual posibilita al usuario visualizar de manera gráfica la información seleccionada, basándose en las configuraciones definidas tanto en las opciones de la figura como en las de subgráficos y realizar diferentes transformaciones, así como la posibilidad de extraer los datos. Además, permite guardar estas representaciones en el sistema. Este módulo se compone de tres partes:

- **Botón de Representar Datos:** Este botón genera la representación gráfica de las columnas seleccionadas de los *dataframes* del grupo elegido, o bien de una operación matemática aplicada entre dichas columnas.

- **Botón de Guardar:** Permite almacenar los datos seleccionados del banco de datos del programa en archivos CSV. Esto es útil para:
  - Aislar *dataframes* específicos de grupos de interés.
  - Guardar los resultados de operaciones matemáticas, como la obtención de la potencia a partir de la multiplicación de datos de tensión y corriente.
  - Almacenar datos estadísticos relevantes.

  El proceso de guardado se desarrolla en varias etapas:
  - Se abre una ventana secundaria con un conjunto de botones organizados en forma matricial (Fig. 75), cuya configuración varía según si se trabaja con subgráficos y las dimensiones de la matriz.
  - Esta ventana detecta y muestra las posiciones de los gráficos existentes en la matriz, indicando su posición y el título del gráfico, ofreciendo al usuario diversos botones distribuidos en forma de matriz, de igual manera que los subgráficos.
  - Una vez seleccionado el gráfico del cual se desea generar el archivo CSV, se despliega una segunda ventana en la que se puede especificar:
    - El nombre de la columna correspondiente al eje X.
    - Los nombres para las columnas del eje Y, cuya cantidad dependerá del número de curvas presentes en el gráfico.

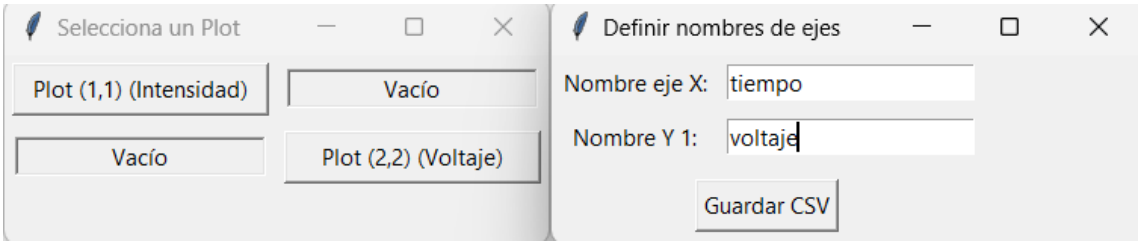


Fig. 75: Ventana de selección de subgráfico (izquierda) y ventana de asignación de nombres de columnas

- **Botón de Análisis/Estadística**: Permite ejecutar distintos procedimientos analíticos sobre los datos actualmente seleccionados en el entorno de trabajo. En conjunción con la opción de guardar datos en archivos CSV, esta herramienta permite al usuario ejecutar diversos análisis y exportarlos para poder cargarlos de nuevo y realizar análisis de otro tipo sobre estos resultados. Este botón da acceso a dos grandes grupos de herramientas:
    - **Análisis de sobrecarga**: Esta sección automatiza la aplicación de los procedimientos descritos en apartados anteriores, específicamente diseñados para evaluar eventos de sobrecarga en dispositivos como JFETs (Fig. 76). Las funciones incluidas son:
        - Búsqueda del valor máximo de corriente o tensión.
        - Detección del intervalo de desconexión o de carga (especialmente útil para el arranque de cargas capacitivas).
        - Cálculo de la energía disipada durante los intervalos críticos.
        - Cálculo de la corriente media durante la fase de limitación (excepto en el caso de cargas capacitivas).

      Estas funciones han sido desarrolladas y optimizadas para operar sobre un conjunto de curvas representativo de comportamientos típicos de sobrecarga. Por ello, su uso debe limitarse exclusivamente a este tipo de señales, ya que aplicar estos análisis sobre otros perfiles puede llevar a interpretaciones erróneas o incluso a largos tiempos de procesamiento. Este módulo resulta esencial para validar la robustez y aplicabilidad de las funciones automáticas a grandes volúmenes de datos, permitiendo al usuario realizar estudios comparativos de forma eficiente y sistemática.

    - **Análisis temporal**: En este apartado se incluyen herramientas orientadas al estudio del comportamiento dinámico de las señales (Fig. 77). Entre ellas se destaca el cálculo de derivadas de las funciones registradas. Dado que las señales analizadas suelen estar afectadas por un nivel significativo de ruido, la derivación directa puede generar resultados no representativos. Por ello, se recurre a aproximaciones mediante filtros suavizantes, en particular el filtro de Savitz-Golay, que ofrece varias ventajas:
        - Mantiene la forma general y las características de la señal original, como los picos y las pendientes.
        - Es especialmente eficaz para preservar la información útil al tiempo que reduce el ruido.
        - Permite obtener derivadas suaves que se aproximan bien a la realidad física del sistema.

      El usuario puede seleccionar parámetros como el orden del polinomio de ajuste y el tamaño de la ventana de suavizado, adaptando así el análisis a las condiciones específicas de la señal a estudiar, puesto que estos parámetros son muy relevantes a la hora de obtener un resultado aceptable.

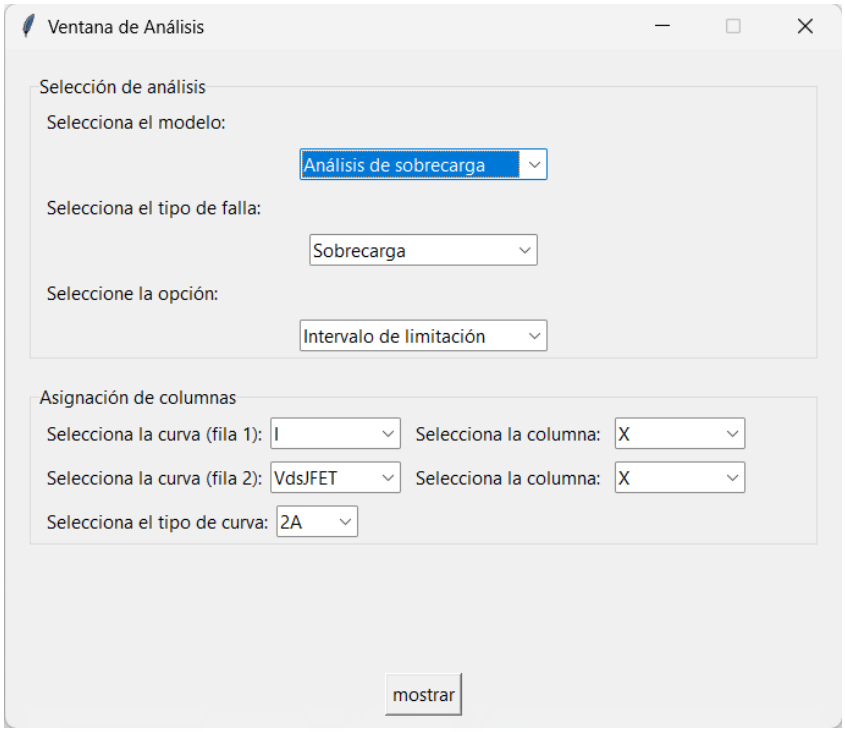


Fig. 76: Ventana de análisis, sección de análisis de sobrecarga.

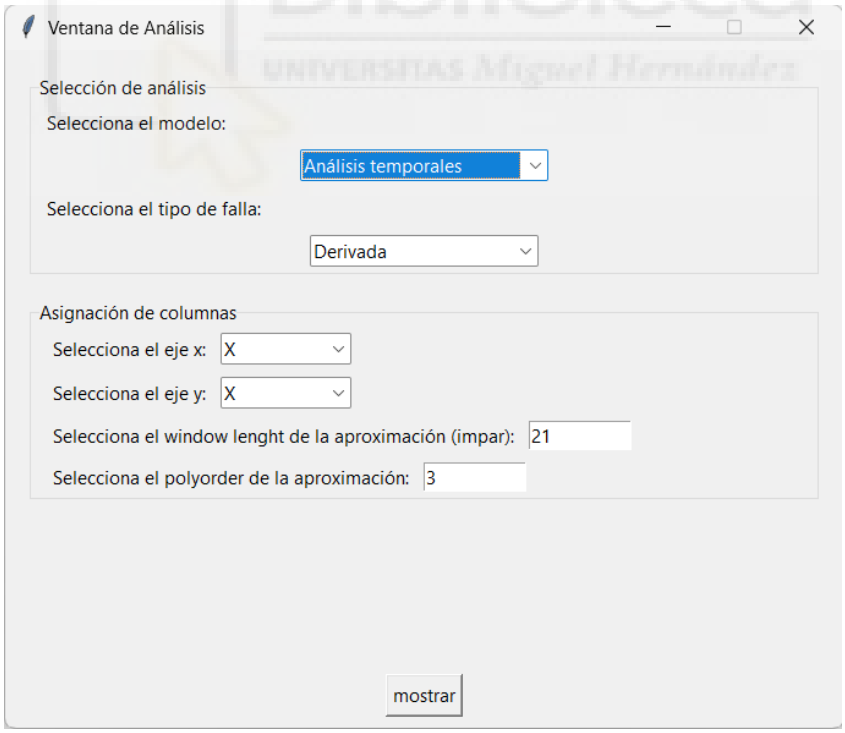


Fig. 77: Ventana de análisis, sección de análisis temporal

Zona de representación gráfica

Finalmente, el resto de la pantalla principal está ocupado por la **zona de representación gráfica**, donde se muestran todas las gráficas generadas a partir de las configuraciones establecidas en los módulos anteriores. Esta área se adapta

dinámicamente para distribuir cada gráfica de manera óptima dentro del espacio disponible, asegurando una visualización clara y ordenada. Además de la simple representación de los datos, esta zona incorpora una **barra de herramientas interactiva** ubicada en la parte inferior, la cual proporciona diversas funcionalidades para mejorar la experiencia del usuario:

- **Navegación y Exploración:** Permite desplazarse a través de las diferentes gráficas generadas, facilitando la inspección detallada de cada representación sin necesidad de regenerarlas.

- **Zoom y Ajustes Dinámicos:** Incluye herramientas de ampliación y reducción, lo que permite al usuario acercar o alejar áreas específicas de las gráficas para analizar detalles con mayor precisión.

- **Exportación de Gráficas:** Ofrece la posibilidad de guardar el conjunto de gráficas representadas en el sistema, permitiendo la exportación en distintos formatos de imagen (PNG, JPG, SVG, entre otros). Esto resulta útil para la documentación de resultados, informes o presentaciones, garantizando que las visualizaciones puedan ser reutilizadas fuera del entorno del programa.

4.3.3. Operaciones internas y control de errores

En cuanto a las **operaciones internas** de la herramienta, su diseño se ha basado en los principios de la **programación orientada a objetos (POO)**, aprovechando sus fortalezas para estructurar el código de manera clara, eficiente y escalable. Además, como se expuso en el apartado 2.5, como variable de almacenamiento de datos se emplearán los *dataframe* de la librería Pandas.

Para ello, se han definido **tres objetos principales**, cada uno con una responsabilidad bien diferenciada, pero interconectados entre sí, lo que facilita la modularidad del programa. Esta estructura no solo mejora la organización y el mantenimiento del código, sino que también permite una mayor flexibilidad para futuras expansiones y la incorporación de nuevas funcionalidades según los requerimientos del trabajo.

Gracias a este enfoque, se garantiza una arquitectura sólida y adaptable, en la que cada módulo puede evolucionar de forma independiente sin comprometer el funcionamiento general de la herramienta. En la Fig. 78 se representa el diagrama de bloques que marca las interacciones entre el usuario y los distintos bloques internos del programa.

Las funciones que integran el programa de cribado y visualización se muestran en el Anexo 8.

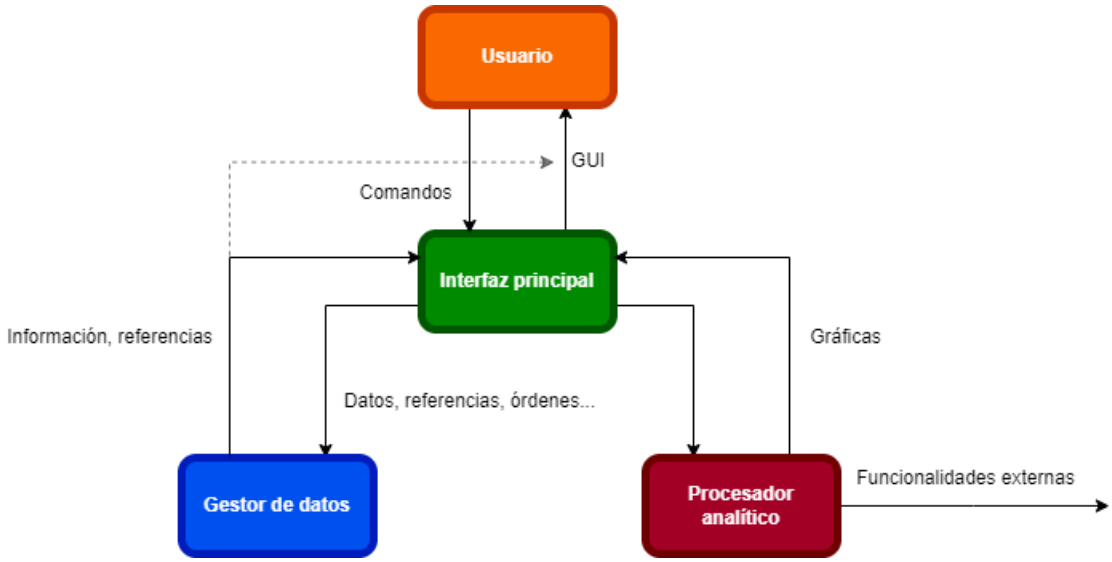


Fig. 78: Diagrama de bloques del funcionamiento del programa.

Interfaz Principal

El primero de los objetos es la **Interfaz Principal**, cuya función es generar la **interfaz gráfica de usuario (GUI)** y coordinar la ventana principal en función de los comandos introducidos por el usuario. Actúa como el núcleo de la aplicación, gestionando la interacción con el usuario y asegurando que las solicitudes se procesen correctamente. Mantiene una comunicación activa y constante con los otros dos módulos principales: el **Gestor de Datos** y el **Procesador Analítico**, además de servir como un puente entre ellos.

En cuanto al manejo de errores, la **Interfaz Principal** desempeña un papel clave en la validación de los comandos introducidos por el usuario, asegurándose de que sean coherentes con las expectativas del sistema. Sin embargo, su función no se limita únicamente a la validación de entradas; también debe comprobar que los datos almacenados en los otros objetos sean adecuados para la operación solicitada. Esto es especialmente crítico en funciones avanzadas como la opción **'Math' del eje Y**, donde una entrada aparentemente válida podría generar errores si los datos del **Gestor de Datos** no cumplen con los requisitos esperados. Para evitar estos problemas, la interfaz implementa mecanismos de control que no solo verifican la sintaxis y estructura de los comandos, sino que también validan la integridad de los datos en uso, minimizando así posibles fallos y garantizando una experiencia de usuario más segura y estable.

Gestor de Datos

El segundo de los objetos es el **Gestor de Datos**, encargado de toda la gestión, validación y organización de la información que se utilizará en el procesamiento y la representación gráfica. Este componente es fundamental para asegurar que los datos que ingresan al sistema sean consistentes, estén correctamente estructurados y se puedan manejar de forma eficiente a lo largo de las distintas etapas del análisis. A continuación, se describen de forma detallada sus principales funcionalidades:

- **Carga de Datos:** La primera responsabilidad del Gestor de Datos es la importación de información desde diferentes fuentes. Esto se lleva a cabo a través de una funcionalidad que se activa mediante botones específicos en la interfaz. Durante este proceso se abordan varios aspectos críticos:

- **Verificación de la Integridad de los Datos:** Al cargar archivos, y en particular los CSV, es común encontrarse con varias líneas iniciales que contienen encabezados descriptivos, metadatos y otra información auxiliar. El objeto debe identificar de manera automática el momento en que finalizan estas líneas y comenzar a procesar la información tabulada. Esta separación es crucial para evitar que datos no estructurados interfieran en el análisis posterior.

- **Gestión de Recursos:** La carga de grandes volúmenes de datos puede llevar a un consumo elevado de memoria RAM. Para prevenir sobrecargas que comprometan la estabilidad del sistema, el Gestor de Datos incorpora mecanismos de monitorización del uso de memoria. En situaciones donde se detecte un uso excesivo, se emite una alerta al usuario, recomendándole acciones correctivas, como la carga parcial de información o la selección de subconjuntos específicos de datos.

- **Estructuración Jerarquizada de la Información:** Una vez validados, los datos se organizan en una estructura de diccionarios jerarquizados simple (Fig. 69). Esta organización facilita el acceso y manejo posterior, ya que permite clasificar y agrupar la información de manera lógica, agilizando el procesamiento por parte del módulo del Procesador Analítico.

- **Visualización de Datos:** Otra funcionalidad esencial del Gestor de Datos es informar al usuario sobre el contenido y la estructura de los datos cargados. Para ello, se despliega una ventana que presenta la información de forma jerárquica, permitiendo navegar a través de los distintos niveles de detalle:
  - **Navegación Estructurada:** La interfaz distingue claramente entre las claves del diccionario anidado y los *dataframes* finales. Esto ayuda a que el usuario comprenda cómo se han organizado los datos y facilite la identificación de aquellos elementos que serán relevantes para el análisis.

  - **Limitación en la Visualización:** Para evitar problemas de rendimiento, especialmente cuando se manejan conjuntos de datos muy extensos, la visualización del contenido de cada *dataframe* se limita a un máximo de 100 filas, si la cantidad de filas es superior a 100, se representan las primeras 50 y las últimas 50 filas. De esta forma, se garantiza una respuesta rápida de la interfaz sin comprometer la capacidad de análisis preliminar de la información.

- **Selección de Datos:** Una vez cargados y visualizados, el siguiente paso consiste en seleccionar aquellos conjuntos de datos que se utilizarán para el análisis y la representación gráfica, guardando una referencia a estos dentro del objeto. Esta funcionalidad se implementa mediante una ventana interactiva que permite al usuario:
  - **Elegir *Dataframes* Específicos:** El usuario puede seleccionar libremente los *dataframes* pertenecientes a un grupo determinado, de manera que solo se procesen y representen aquellos datos que sean relevantes para el análisis en curso.

- **Control del Número de Selecciones:** Es fundamental mantener un control sobre la cantidad de *dataframes* seleccionados, ya que esta cifra determina el número de curvas que se generarán en la visualización gráfica. Una selección excesiva puede no solo alargar innecesariamente los tiempos de carga y procesamiento, sino también provocar sobrecargas en la memoria RAM, afectando la estabilidad y el rendimiento general del programa.

- **Validación Preventiva:** Antes de finalizar el proceso de selección, el Gestor de Datos valida que la calidad de los *dataframes* elegidos sean adecuados para las operaciones posteriores, garantizando que cada conjunto de datos cumpla con los requisitos establecidos para un análisis correcto y sin errores.

Procesador analítico

El tercero de los objetos es el **procesador analítico**, este objeto tiene la peculiaridad de que no todas sus funcionalidades se emplean en el programa de la interfaz gráfica, una parte de las funciones de este se emplean en el análisis de datos de sobrecarga ya explicado en los apartados anteriores, esto se debe a que el análisis de los datos de sobrecarga se concibe como una forma de ver los resultados para todas las repeticiones de una prueba de sobrecarga sobre un dispositivo, debido a que las pruebas de sobrecarga generan 1000 ficheros CSV, esta cantidad de *dataframes* es completamente superior a la recomendada y su procesamiento podría ser muy tedioso. Sin embargo, debido a la modularidad del trabajo, es posible la inserción de estas funcionalidades dentro de esta herramienta de visualización, de forma parcial.

En cuanto a las funcionalidades empleadas en el programa a partir de los comandos y de la información del objeto de gestor de datos aportados por el objeto de interfaz principal, estas se dividen en cuatro:

- **Generación del eje X:** Cuando el usuario opta por generar un eje X artificial en lugar de emplear una de las columnas del *dataframe*, se le solicita introducir dos parámetros: la escala y el *offset*. El número de puntos necesarios para dicho eje se determina de forma automática, tomando como referencia la longitud del primero de los *dataframes* seleccionados para la representación gráfica.

  Aunque este sistema puede parecer poco intuitivo en un primer momento, ha sido diseñado específicamente para emular el comportamiento de adquisición de señales en instrumentos como los osciloscopios digitales. En este contexto, el eje horizontal suele definirse únicamente a partir de una escala temporal y un desplazamiento, sin requerir una serie de tiempo explícita. La aplicación de esta lógica permite una representación coherente y rápida de señales con respecto al tiempo, especialmente en situaciones donde los datos de eje X no se encuentran disponibles como columna independiente en el conjunto de datos.

- **Gestión de recursos:** De igual manera que el **gestor de datos**, este objeto tiene incorporado un control de memoria RAM a lo largo de toda su estructura debido a que, en esencia, se encarga de crear elementos que se alojarán en memoria.

- **Representaciones:** Esta funcionalidad se encarga de generar tres tipos fundamentales de gráficas dentro del sistema:

- **Gráfica en crudo:** Representa directamente las curvas a partir de las columnas seleccionadas de los *dataframes*. En caso de que se haya optado por generar un eje X artificial, este será empleado para la representación.
  Esta modalidad incluye, además, una herramienta de operaciones matemáticas aplicables al eje Y de la gráfica. Como se ha descrito anteriormente, permite realizar tanto operaciones entre columnas como con escalares, lo que añade flexibilidad y potencia visual a las representaciones.

  A nivel interno, esta función incorpora un analizador de operaciones que examina el contenido de la pestaña *Math*. Este analizador valida la sintaxis introducida por el usuario, comprobando, entre otros aspectos, la correcta utilización de operadores matemáticos y la referencia adecuada a los nombres de las columnas implicadas.

- **Gráfica de sobrecarga:** En este caso, el sistema interpreta los datos proporcionados por el usuario a través de la ventana correspondiente, realiza las copias necesarias de los *dataframes* implicados y ejecuta las operaciones requeridas sobre ellos.

  Es importante advertir que esta funcionalidad ha sido desarrollada específicamente para representar curvas de sobrecarga obtenidas en el contexto del presente trabajo. Por tanto, su uso con otros tipos de curvas —o incluso con curvas de sobrecarga procesadas de manera diferente— puede generar incompatibilidades, ralentización del proceso e incluso resultados incorrectos.

  Esto se debe a que el programa incorpora intervalos de búsqueda definidos *a priori*, en los que se ha determinado analíticamente que se encuentran los puntos de interés. Esta optimización evita el procesamiento completo de toda la curva, a costa de depender de la estructura esperada de los datos. En cualquier caso, el sistema lanza un aviso emergente para confirmar que el usuario desea continuar, asumiendo estos condicionantes.

- **Gráfica de análisis:** Se trata de representaciones basadas en operaciones de análisis temporal y estadístico aplicadas al conjunto de datos.
  Este tipo de análisis es transversal y puede aplicarse a cualquier tipo de curva. Tras introducir los parámetros necesarios en la ventana correspondiente, el programa genera una copia temporal de los *dataframes* seleccionados, sobre la cual se ejecutan las operaciones analíticas definidas para la posterior representación gráfica.

- **Subgráficos:** Esta funcionalidad permite al usuario representar múltiples gráficas dentro de una misma figura, organizadas en una matriz de dimensiones configurables, aunque limitadas por cuestiones prácticas de visualización y rendimiento. También se ofrece la opción de representar una única gráfica si así se desea.

  Dentro de este módulo, el usuario puede coordinar los ejes de los distintos subgráficos que componen la matriz. Esta coordinación puede aplicarse a todos

los gráficos de forma simultánea o únicamente a aquellos que comparten la misma columna dentro de la disposición en cuadrícula. Además, es posible especificar si se desea coordinar únicamente el eje X, o tanto el eje X como el Y.

Con el objetivo de garantizar una representación coherente, el sistema incluye comprobaciones automáticas para verificar que las gráficas seleccionadas sean del mismo tipo, es decir, que contengan el mismo número de puntos. Esta verificación es importante, ya que en caso contrario se asume —con elevada probabilidad— que pertenecen a grupos de datos distintos. No obstante, esta comprobación puede ser omitida por el usuario si así lo desea.

Uno de los aspectos más relevantes de esta funcionalidad es la validación de la compatibilidad del eje X entre las gráficas. Por ejemplo, en las representaciones generadas a partir de los análisis de sobrecarga, cada *dataframe* produce un único valor que se representa en el eje X, el cual corresponde al nombre del propio *dataframe*. En estos casos, el eje X no es numérico, lo que impide su coordinación con otros ejes X que sí lo sean. El programa se encarga de advertir al usuario y evitar este tipo de combinaciones incompatibles.

Módulo común de uso de memoria o gestión de recursos

La única funcionalidad compartida entre dos clases es el módulo de control de uso de memoria RAM, esto se debe a que son los únicos que pueden cargar elementos en memoria.

Con el objetivo de evitar problemas relacionados con el consumo excesivo de memoria RAM durante el funcionamiento del programa, se han implementado tres mecanismos de control asociados a las principales modalidades de carga de datos (véase Fig. 79). Todos estos mecanismos se articulan en torno a una variable global que se inicializa al inicio de la ejecución y que representa un umbral de uso de memoria RAM, definido como un porcentaje del total disponible en el sistema. En la presente implementación, dicho umbral se ha fijado en un 60 %.

Esta variable actúa como referencia en los distintos puntos críticos del programa donde se realizan operaciones potencialmente intensivas en memoria. En caso de que el sistema detecte que la carga prevista de datos podría comprometer este umbral, se detiene el proceso automático para consultar al usuario. Según la respuesta recibida, este umbral puede aumentarse progresivamente hasta un máximo del 95 %, permitiendo una gestión flexible y controlada del uso de recursos. A continuación, se describe la primera de las fases de control implementadas:

- **Control directo 1:** Para el proceso de carga de *dataframes* a través de archivos CSV en el **objeto gestor de datos**, para *dataframes* de Pandas, donde los volúmenes de datos pueden crecer rápidamente, una estrategia de muestreo y extrapolación resulta especialmente útil. Aunque existen otras aproximaciones —como estimar el consumo a partir del tamaño del archivo CSV—, se ha demostrado que éstas son poco fiables, tal y como expone Itamar Turner-Trauring en “Don’t bother trying to estimate Pandas memory usage” [25].

  Tal como se indica en el diagrama de flujo de la Fig. 82 Este mecanismo se activa una vez el usuario ha seleccionado un conjunto de archivos CSV para su carga. Bajo el supuesto de que todos los archivos presentan una estructura y un tamaño

similar, se toma como referencia el primer archivo del conjunto seleccionado. Para evitar una carga completa inicial que consuma memoria innecesariamente, se procede a cargar únicamente una fracción parcial del fichero, correspondiente a un número reducido de líneas, redondeado al entero superior.

Una vez cargado este subconjunto, se calcula el peso en memoria de este mediante las funciones internas proporcionadas por la biblioteca de Pandas. A partir de este valor y del número de líneas cargadas, se estima el **peso medio por línea**. Esta relación permite extrapolar el peso aproximado de los archivos restantes, considerando su número total de líneas.

La estimación final del uso total de memoria se calcula multiplicando el peso medio por línea por el total de líneas de todos los archivos seleccionados. A este resultado se le aplica un **porcentaje de mayoración**, cuyo objetivo es introducir un margen de seguridad ante posibles desviaciones debidas a variaciones internas del contenido de los archivos, estructuras de datos intermedias, o sobrecarga temporal del sistema operativo. Este margen adicional permite evitar infravaloraciones en la estimación de recursos necesarios.

Una vez obtenida esta estimación total, se compara con la cantidad de memoria que podría utilizar el sistema sin sobrepasar el umbral de seguridad previamente definido. Si el valor estimado supera dicho límite, el programa detiene la carga y presenta al usuario tres opciones (véase Fig. 80):

- **Continuar con la carga**: Se incrementa el umbral de uso de memoria RAM en un 5 % (hasta un máximo del 95 %) y se continúa el proceso de carga de los *dataframes* restantes.

- **Detener la carga, pero conservar los *dataframes* cargados**: Esta opción solo está disponible si al menos uno de los *dataframes* ha sido cargado previamente. Se interrumpe la operación, pero se mantiene el grupo de datos ya cargado.

- **Abortar la operación**: Se cancela completamente la carga de datos. En caso de que ya se hubieran cargado algunos *dataframes*, estos se eliminan del sistema.

- **Control directo 2:** Este mecanismo está diseñado para situaciones en las que es necesario duplicar estructuras de datos ya cargadas en memoria, como ocurre en dos procesos concretos: la carga de dataframes desde diccionarios contenidos en archivos '.spydata' y la copia de elementos para generar representaciones gráficas en el módulo de análisis.

  En el primer caso, relativo al **objeto gestor de datos**, el procedimiento comienza cuando el usuario carga en memoria un archivo '.spydata', cuyo control de uso de memoria sigue el esquema de control indirecto, descrito en el siguiente apartado. Una vez cargado el archivo, el usuario puede seleccionar ciertos *dataframes* de interés para incorporarlos a un nuevo grupo de trabajo. Esta operación implica la duplicación en memoria de dichos elementos desde la estructura original (el diccionario generado a partir del archivo '.spydata') hacia un grupo propio de

dataframes. Finalizada esta selección, se libera el contenido restante del archivo, reteniéndose exclusivamente los *dataframes* seleccionados por el usuario. De este modo, se evita una ocupación innecesaria de memoria RAM por estructuras no relevantes para el análisis.

En el segundo caso, correspondiente al objeto **procesador analítico**, el programa permite la realización de diversas operaciones sobre los datos ya cargados. Es importante recordar que el sistema gestiona los *dataframes* utilizando estructuras de diccionarios jerárquicos simples (véase Fig. 69). Cuando se ejecuta una operación sobre un elemento de uno de estos diccionarios, el contenido original podría verse modificado, lo cual resulta indeseable. La inmutabilidad de los datos de entrada es una premisa fundamental del programa, dado que estos pueden ser requeridos posteriormente en su forma original por el usuario. Para prevenir cualquier alteración no deseada, el programa genera copias independientes de los elementos sobre los que se va a operar. De este modo, se garantiza que el diccionario principal permanezca inalterado durante el proceso de análisis.
En ambos procesos, se parte del supuesto de que, al tener que copiar elementos ya presentes en memoria, se puede estimar con suficiente precisión el espacio requerido. Para ello, se toma como referencia el tamaño en memoria de uno de los objetos, y se multiplica dicho valor por el número total de elementos a copiar. El resultado se suma a la memoria actualmente en uso y se compara con el umbral máximo definido por el programa. Si la memoria total estimada supera este límite, se solicita la intervención del usuario mediante un aviso, que ofrece dos opciones (véase Fig. 83):

- **Continuar:** se procede con la operación, incrementando el umbral de comparación en un 5 % (hasta un máximo del 95 % del uso de RAM disponible).

- **Cancelar:** se aborta el proceso y no se realiza ninguna copia en memoria.

- **Control indirecto:** El tercer mecanismo de control se aplica en situaciones en las que no es posible anticipar con precisión el impacto en memoria del proceso a realizar. Este tipo de control se implementa en dos contextos específicos del programa: la carga de archivos '.spydata' en el gestor de datos y la generación de gráficas en el procesador analítico.

  En el primer caso, la carga de archivos '.spydata' representa un escenario de alta incertidumbre respecto al consumo de memoria en el **objeto gestor de datos**. Dichos archivos, generados mediante sesiones previas de trabajo en entornos como Spyder, contienen múltiples objetos en memoria, entre ellos *dataframes*, listas y estructuras anidadas cuya dimensión no puede determinarse previamente sin cargarlos. Así, resulta inviable estimar con exactitud su impacto antes de que la operación se lleve a cabo.

  El segundo caso se refiere a la representación gráfica de los datos en el **objeto procesador analítico**. En particular, cuando se generan gráficas complejas —por ejemplo, aquellas que incluyen múltiples subgráficos, anotaciones, o transformaciones como derivadas suavizadas—, se deben almacenar en memoria diversos elementos asociados al objeto gráfico: los datos a representar, las configuraciones de estilo, los ejes, etiquetas, leyendas y demás. Todo esto puede

aumentar considerablemente el uso de memoria en función de la complejidad del gráfico generado.

Ante la imposibilidad de realizar una estimación previa fiable, el programa recurre en estos casos a un mecanismo de control indirecto o pasivo (véase Fig. 81). Este enfoque se basa en asumir que el usuario mantiene un conocimiento razonable del estado actual de la memoria del sistema y realiza un uso consciente de los recursos.

Una vez ejecutado el proceso, el programa comprueba el consumo de memoria del sistema. Si se detecta que este ha superado el umbral definido previamente, se solicita una decisión al usuario mediante un mensaje de advertencia, ofreciéndole dos posibles acciones:

- **Continuar:** se conserva el contenido cargado o generado, y el umbral de control se incrementa en un 5 % (hasta un máximo del 95 %), permitiendo así la continuación del flujo de trabajo.

- **Cancelar:** se deshace el proceso recientemente ejecutado, eliminando de la memoria los datos o gráficos añadidos, y se preserva el umbral actual sin modificaciones.

Finalmente, todo elemento que deba ser desechado del flujo de ejecución del programa es gestionado explícitamente mediante el recolector de basura (*garbage collector*) de Python. Esta medida tiene como objetivo garantizar la liberación efectiva de los recursos ocupados en memoria, evitando acumulaciones innecesarias de objetos no referenciados y contribuyendo a mantener el uso de memoria dentro de los márgenes establecidos. El uso del *garbage collector* se invoca de forma controlada al finalizar cada proceso que implique eliminación de datos, asegurando así un comportamiento eficiente y predecible del sistema en términos de consumo de memoria.

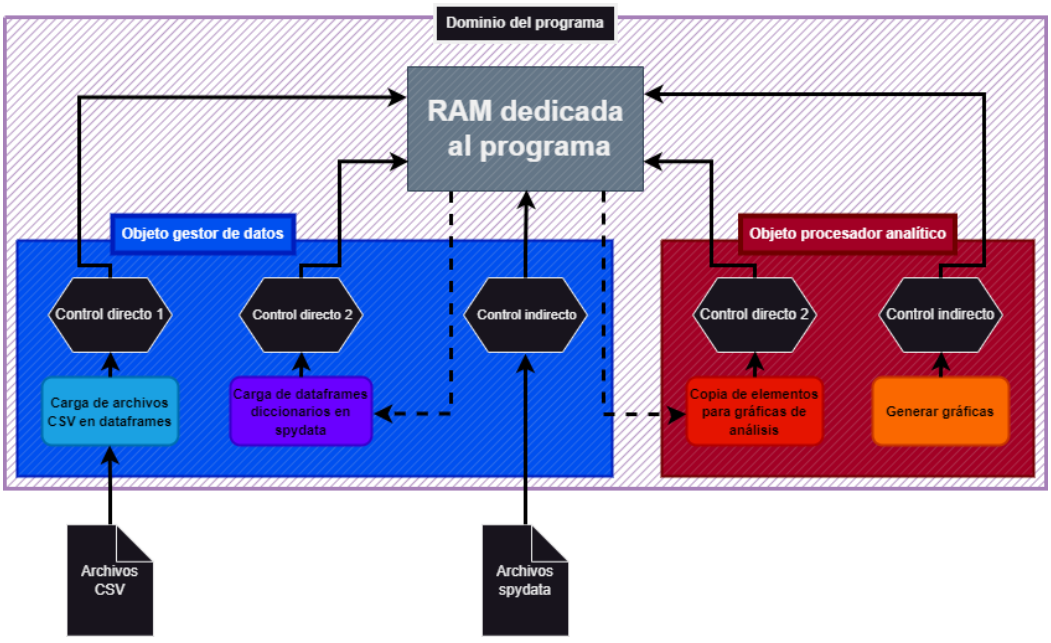


Fig. 79: Diagrama del módulo común de control de memoria.

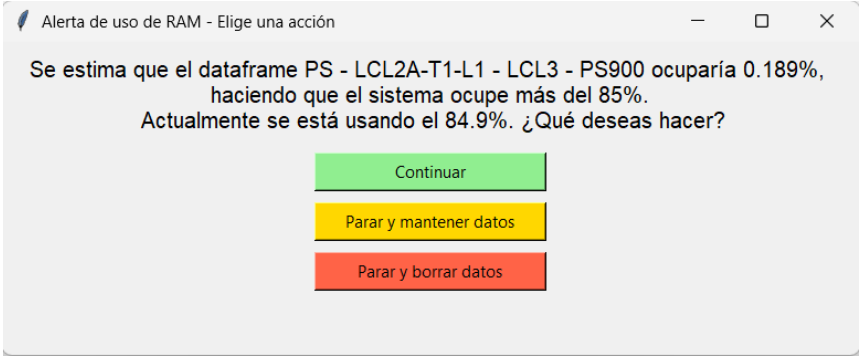


Fig. 80: Ventana de aviso de exceso de RAM en método directo 1.

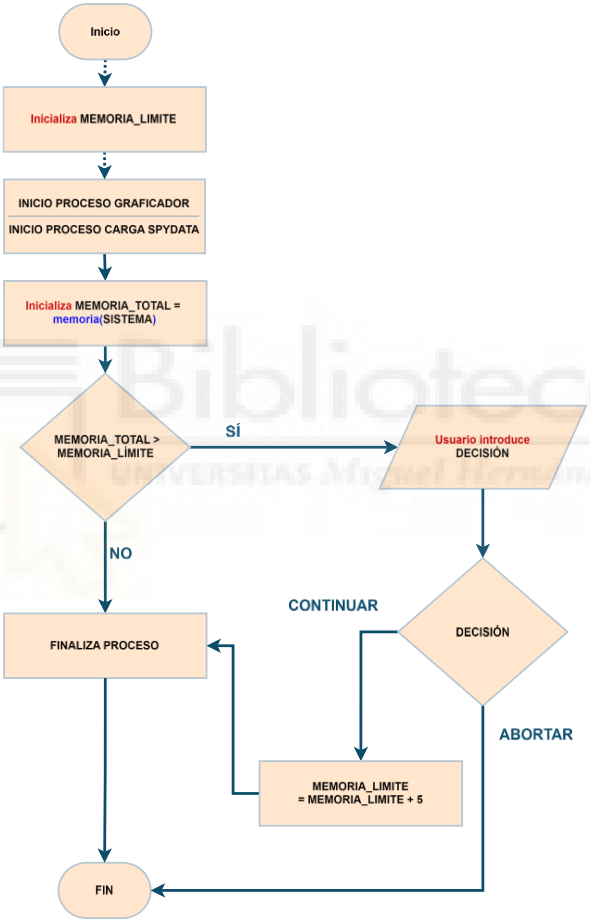


Fig. 81: Diagrama de flujo del control indirecto de memoria.

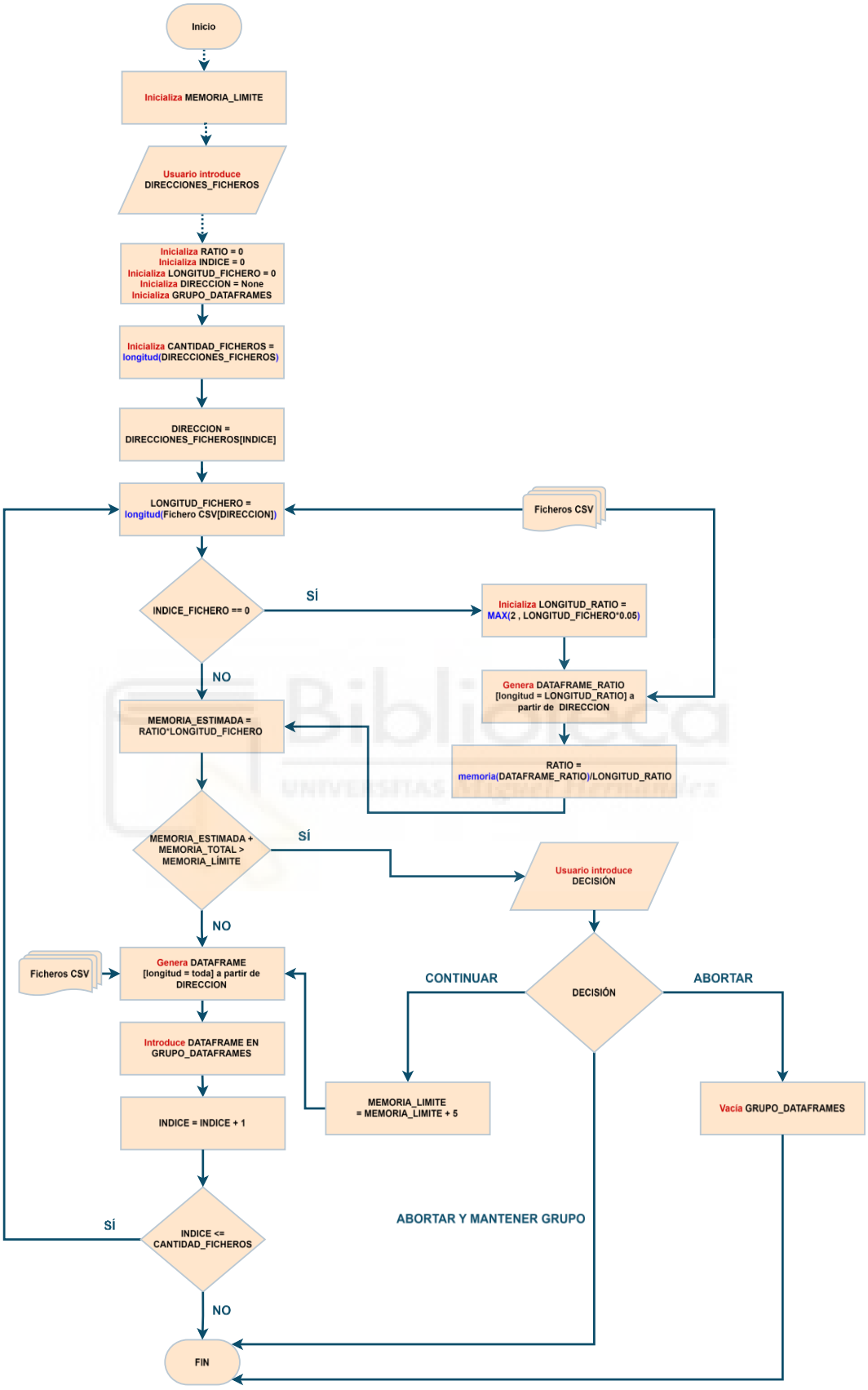


Fig. 82: Diagrama de flujo del control directo de memoria 1.

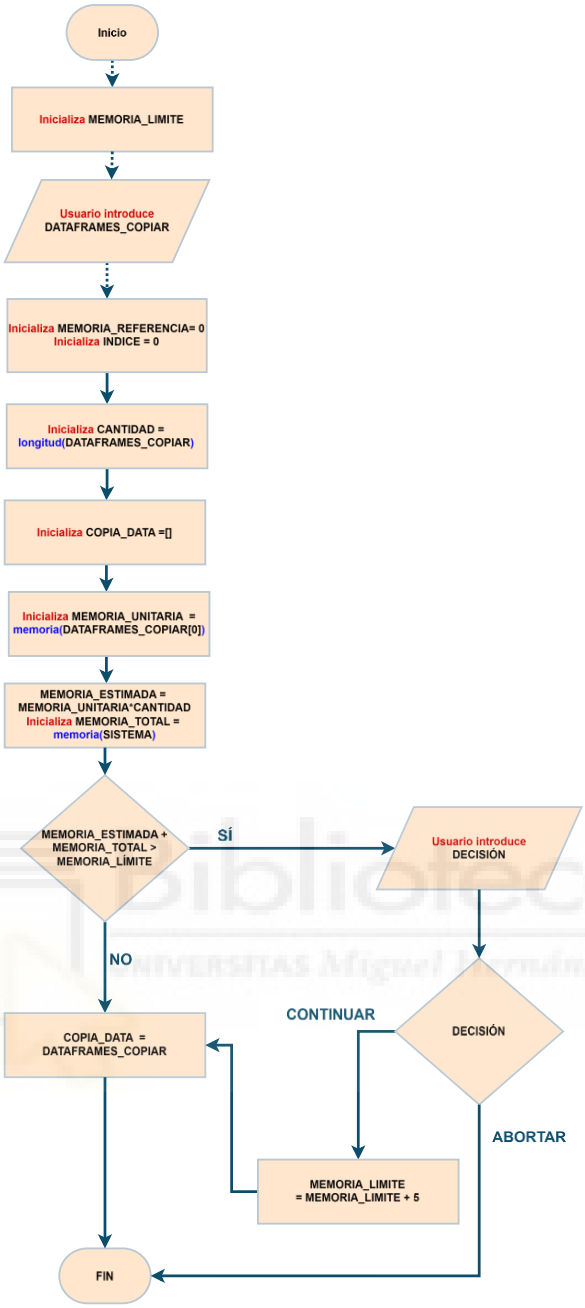


Fig. 83: Diagrama de flujo del control directo de memoria 2.

4.3.4. Ejemplos de uso y casos prácticos

Como muestra inicial, se describe un ejemplo de uso habitual para los eventos de sobrecarga, centrado en la verificación de la fiabilidad de la función de obtención del intervalo de limitación en sobrecargas de tipo cortocircuito.

En primer lugar, se deben cargar los *dataframes* correspondientes. Como no se recomienda cargar una cantidad excesiva de datos simultáneamente, se opta por cargar únicamente 11 *dataframes*, distribuidos equidistantemente a lo largo de las 1000 repeticiones: desde la repetición 0 hasta la 1000, en pasos de 100. Dado que el análisis se limitará a la detección del intervalo de limitación, y para evitar un uso innecesario de

memoria RAM, solo se seleccionarán las columnas del CSV que contienen la tensión del JFET y la corriente (véase Fig. 84).

Los ficheros CSV de repeticiones van acompañados de un archivo de información en formato TXT, que detalla tanto el contenido de las columnas como la escala temporal correspondiente del osciloscopio, dato relevante a la hora de generar el eje x en el programa. Al obtener los *dataframes* desde estos CSV, en el menú de carga se selecciona la opción "Cargar CSV". Tras localizar los archivos deseados en el sistema, estos aparecerán con nombres genéricos como Columna1, Columna2, etc., ya que no incluyen encabezados. A partir del archivo de información, se identifica que los datos relevantes se encuentran en las columnas 1 y 2, por lo que se seleccionan, se renombran para una identificación más clara, y se asigna un nombre al grupo de *dataframes* cargados.

Para esta prueba, se ha elegido el grupo *LCL5,* en concreto, los datos correspondientes a sobrecarga resistiva.

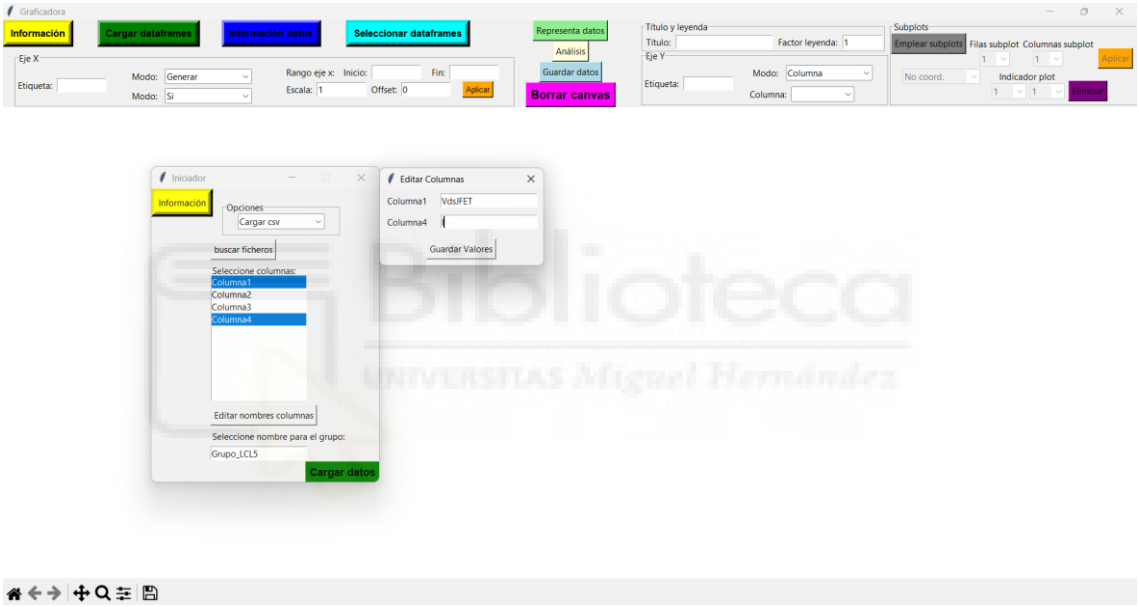


Fig. 84: Proceso de carga de datos y personalización de contenido.

Si la carga se ha realizado correctamente, sin errores ni advertencias por exceso de uso de memoria RAM, el proceso finaliza y se puede continuar con normalidad. A continuación, se comprueba que los archivos requeridos se han cargado correctamente mediante la opción "Información *dataframes*" tal como se ilustra en la Fig. 85. Como medida adicional, se verifica la integridad de los datos de al menos uno de los *dataframes*, asegurándose de que contiene un millón de puntos; esto se hace haciendo doble clic sobre, por ejemplo, sobre el *dataframe* correspondiente a la repetición 0. Seguidamente, se

seleccionan todos los *dataframes* del grupo previamente cargado utilizando el menú "Seleccionar *dataframes*".

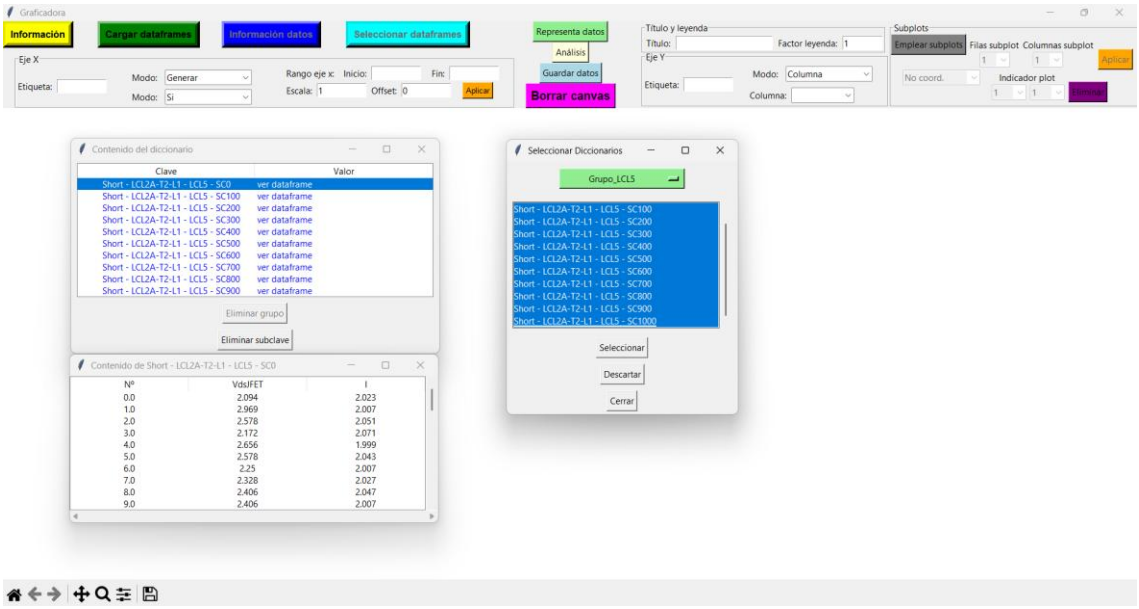


Fig. 85: Proceso de comprobación de la integridad de los datos.

Una vez seleccionados los *dataframes*, se actualizan automáticamente los campos "Columna" del eje X y del eje Y (anteriormente vacíos), permitiendo elegir entre las dos columnas cargadas, identificadas con los nombres asignados previamente. Antes de proceder al análisis del intervalo de limitación, se representa gráficamente la potencia de todos los *dataframes*, ya que esto permite verificar visualmente que las curvas son coherentes y aptas para el análisis.

Para ello, se selecciona el modo "Generar" en las opciones del eje X, aprovechando que se conocen los parámetros del eje horizontal del osciloscopio con el que se tomaron las medidas. Se introducen la escala y el offset adecuados para centrar los datos correctamente. A continuación, en las opciones del eje Y, se elige el modo "Math" para representar el producto de ambas columnas cargadas, es decir, la tensión y la corriente. Una vez configuradas las etiquetas de los ejes y el título, se procede a generar la representación gráfica. El resultado de este proceso se muestra en la Fig. 86.

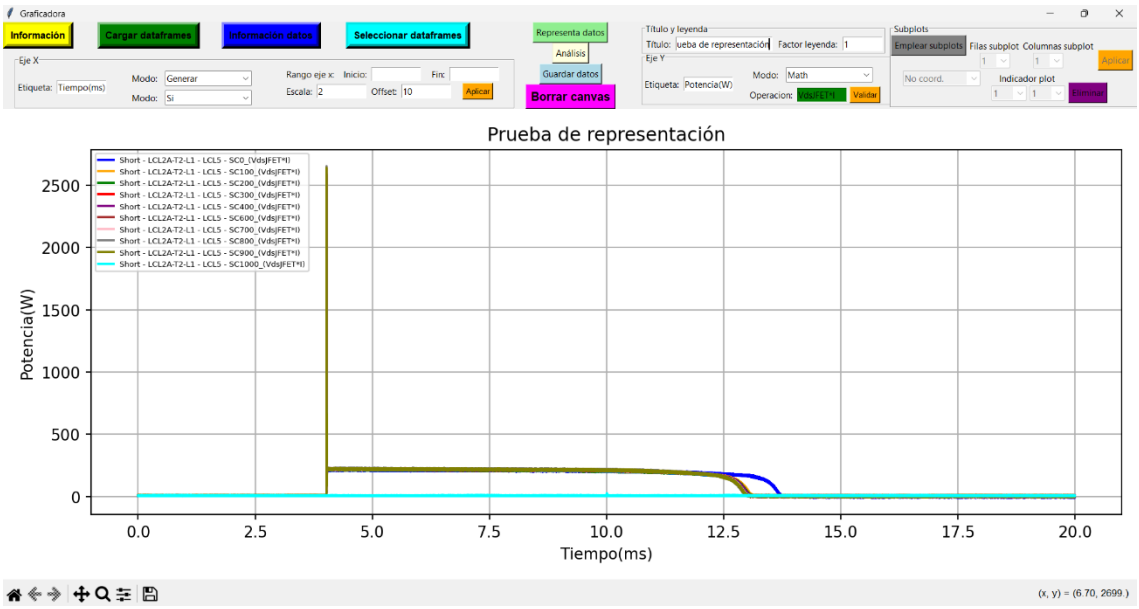


Fig. 86: Proceso de representación de potencia.

En ocasiones, como en este caso, se puede observar que una de las curvas no es válida, probablemente debido a un error en la generación del fichero o a algún incidente durante la campaña de ensayos del grupo correspondiente. Para confirmar esta anomalía, se accede nuevamente al menú "Información datos" (véase Fig. 87), donde se compara la integridad del *dataframe* defectuoso con la de otro que se considere correcto. Una vez verificada la discrepancia, se elimina la repetición afectada mediante la opción "Eliminar subclave", lo que permite continuar con el análisis sin interferencias.

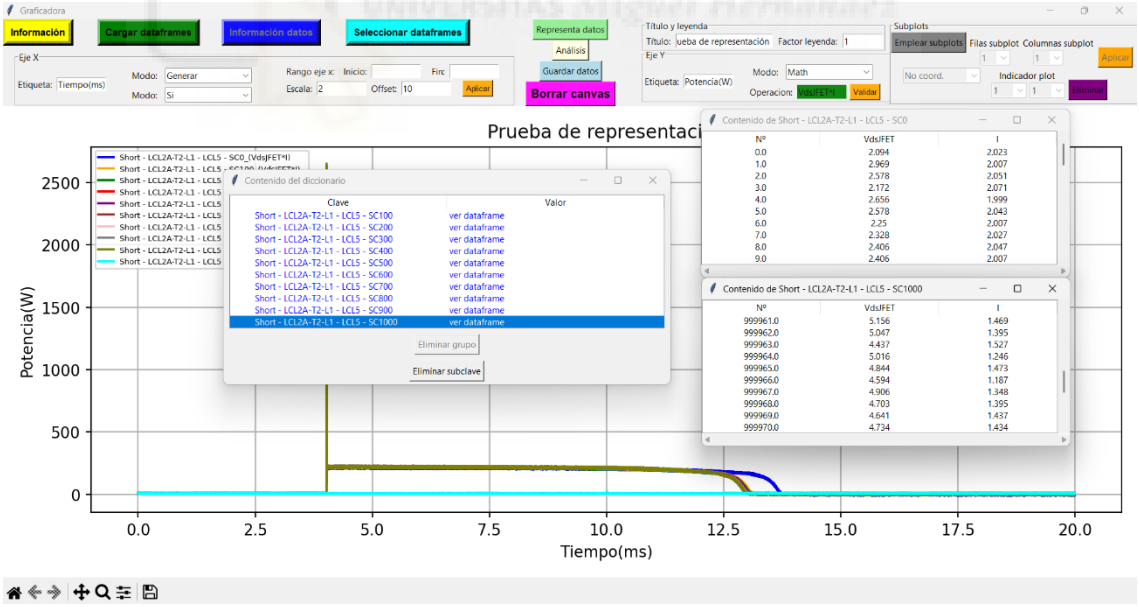


Fig. 87: Proceso de comprobación de curva defectuosa y eliminación.

Solucionado el problema anterior, resulta útil disponer de múltiples gráficas simultáneamente para verificar el correcto funcionamiento del análisis de intervalo. Para ello, se activa la opción "Emplear subplots" y se establece una matriz de dimensiones 2x1, ya que se desea visualizar, por un lado, la representación de la potencia y, por otro, la correspondiente al análisis del intervalo. En caso de que la leyenda interfiera visualmente con la gráfica, su tamaño puede ajustarse desde el mismo menú.

A continuación, se accede al menú “Análisis” y se configuran las opciones pertinentes para el caso, prestando especial atención a la asignación correcta de las columnas de corriente y voltaje, así como a la selección del tipo de curva, que en este ejemplo corresponde a “2A”.

Antes de ejecutar el análisis, se recomienda ajustar las etiquetas de los ejes para reflejar adecuadamente el contenido representado. En este caso concreto, no se asignará título a la gráfica. También es necesario modificar los parámetros de “Indicador plot” dentro del menú “Subplots” para asegurarse de que la nueva gráfica se sitúe debajo de la anterior, evitando que la sobrescriba. La Fig. 88 ilustra el proceso descrito.

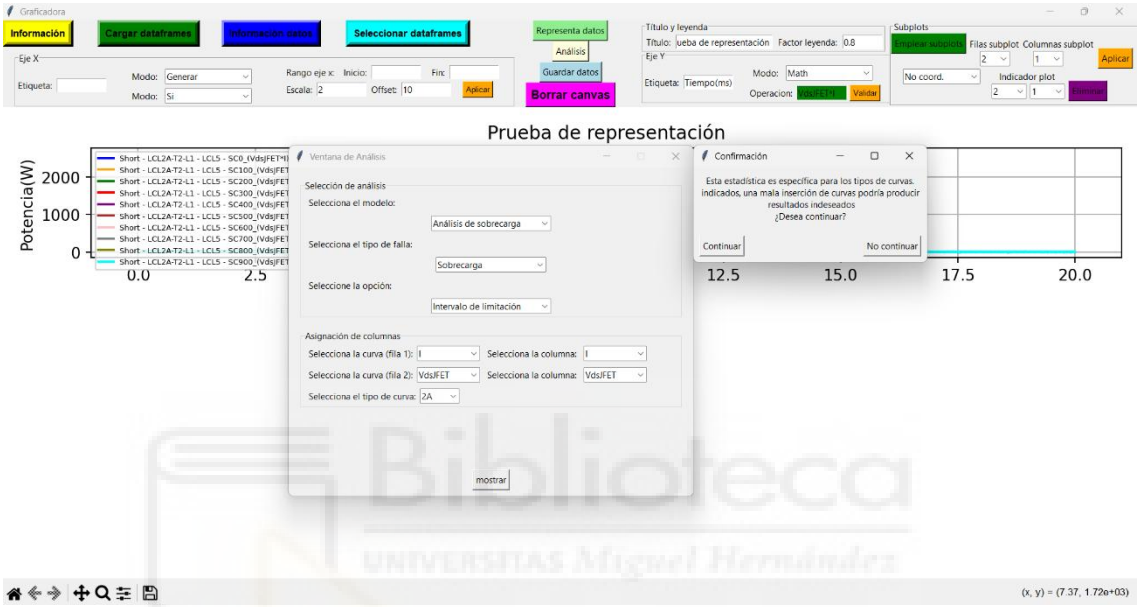


Fig. 88: Proceso de preparación de los datos del análisis de intervalo.

Una vez completada la representación, se visualizan las dos gráficas principales. A simple vista, se aprecia que la repetición 0 presenta un tiempo de limitación superior al del resto en la gráfica superior, y que esta diferencia se refleja correctamente en la gráfica inferior correspondiente al análisis. La representación de las curvas de potencia y la gráfica de análisis de tiempo de desconexión se puede observar en la Fig. 89.

Como buena práctica adicional, es recomendable realizar un *zoom* en la zona de limitación de la gráfica superior. Esto permite observar con mayor precisión los márgenes de cada curva y verificar manualmente el intervalo de limitación, ayudándose del indicador de coordenadas situado en la parte inferior derecha de la barra de herramientas de la gráfica. El resultado de aplicar el *zoom* en la gráfica superior puede observarse en la Fig. 90.

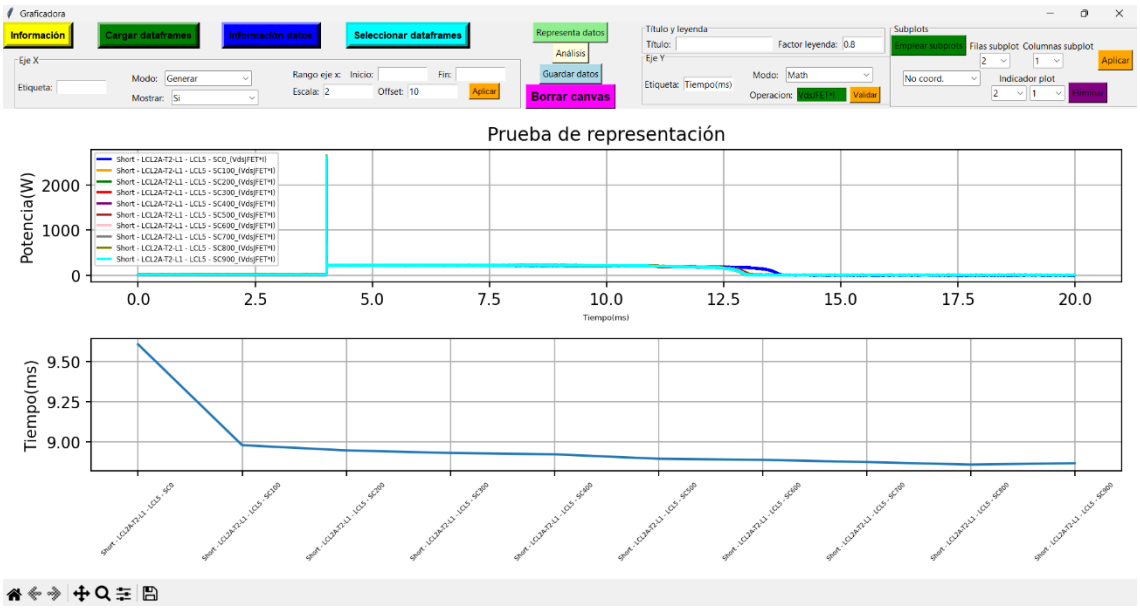


Fig. 89: Resultado de la aplicación de un subgráfico inferior a la representación de las curvas.

Una vez aplicado el zoom, se puede comprobar que, en términos generales, el análisis ha determinado correctamente el intervalo de limitación de las curvas representadas. En caso de detectarse algún error o incoherencia, sería recomendable ajustar los parámetros del programa y repetir el proceso de verificación para asegurar la fiabilidad del análisis.

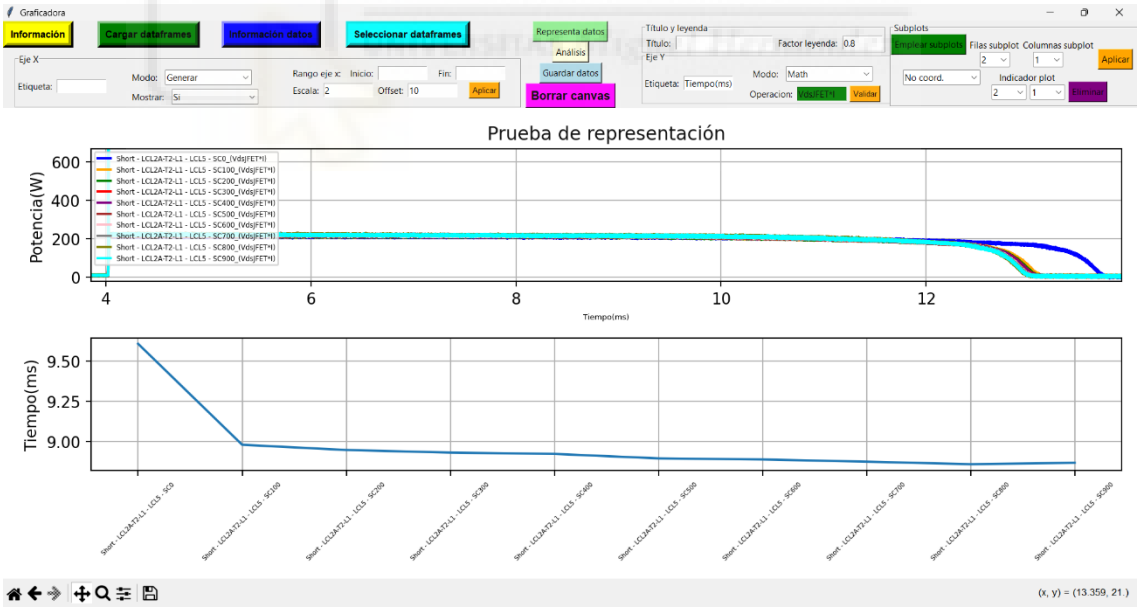


Fig. 90: Aplicación del zoom en la gráfica de las curvas de potencia para la comprobación de la veracidad del análisis de intervalo.

Como segunda muestra, se propone la utilización del programa para el análisis de datos de caracterización. El procedimiento general es análogo al empleado para los eventos de sobrecarga, con la principal diferencia localizada en la fase de carga de datos. En este caso, no se trabaja con ficheros CSV, sino que es necesario acceder a un archivo

con extensión '.spydata', que contiene una sesión completa del entorno de trabajo, incluyendo los diccionarios que contienen los *dataframes* necesarios para el análisis.

Dado que el sistema de protección frente al uso excesivo de memoria RAM en este contexto es pasivo, es decir, no existen mecanismos automáticos que impidan la sobrecarga del sistema, el usuario debe evaluar manualmente el nivel de uso de memoria actual y decidir si procede o no a la carga del archivo. Esto se debe a que no es posible obtener una estimación previa fiable sobre el peso del archivo '.spydata', por lo que la decisión debe basarse en el conocimiento del entorno y en la disponibilidad de recursos.

Una vez aceptada la carga, se navega a través del árbol de directorios que conforma la estructura del archivo. Desde este árbol, el usuario puede seleccionar de forma individual los *dataframes* que desee cargar. En este ejemplo concreto, el grupo de trabajo contiene una función específica: almacenar los datos de caracterización $I_d - V_{gs}$ del dispositivo IJW120R100T1-10, obtenidos en distintas fechas. Por este motivo, se recomienda limpiar la ruta asociada a cada *dataframe* eliminando las ramas de la estructura que no aporten información relevante. Esto es importante porque el nombre que adoptará cada *dataframe* cargado será, por defecto, su ruta completa dentro del árbol del archivo.

Para una mejor organización y claridad durante el análisis, se aconseja conservar únicamente las partes del '*path*' que identifiquen al dispositivo y la fecha correspondiente, eliminando toda información redundante o innecesaria. De este modo, los nombres asignados a los *dataframes* serán más claros, concisos y representativos, lo que facilitará su posterior gestión y análisis. En la Fig. 91 se muestra un resumen del proceso inicial.

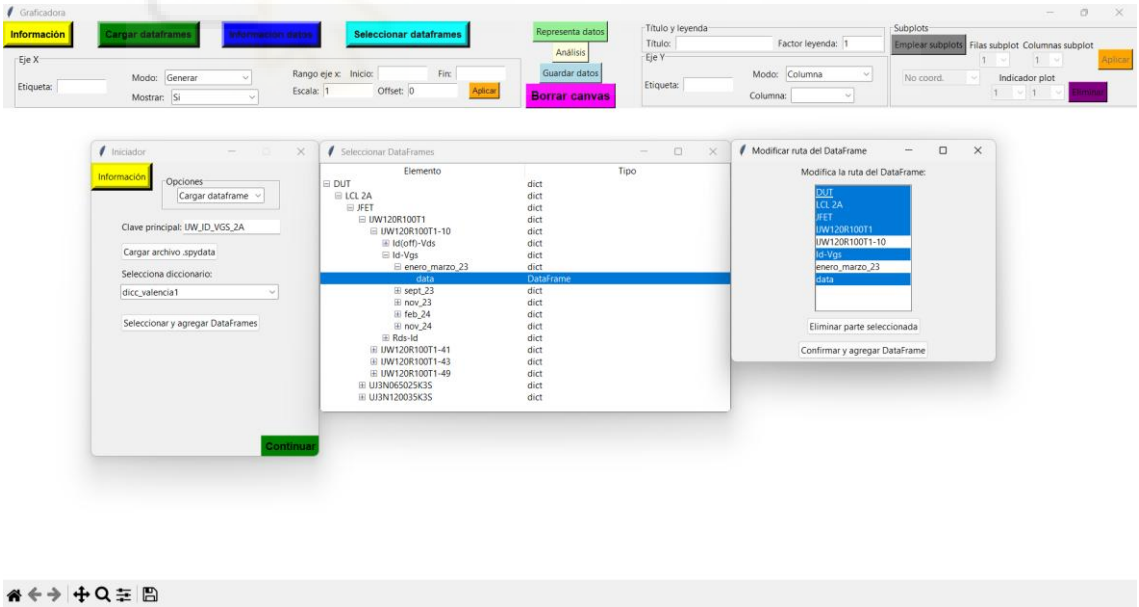


Fig. 91: Proceso de preparación de datos de caracterización extraídos de archivos '.spydata'.

De igual manera que en el caso de los datos de sobrecarga, se recomienda verificar la integridad de los datos cargados utilizando las funcionalidades que proporciona el programa. Esta verificación resulta especialmente útil en el contexto de la caracterización,

ya que los *dataframes* asociados suelen contener múltiples columnas con información diversa. No obstante, para el tipo de análisis que se pretende realizar en este apartado, únicamente son relevantes las dos primeras columnas: la corriente de drenador (I_d) y la tensión entre puerta y surtidor (V_{gs}). Por tanto, resulta conveniente confirmar que dichas columnas se encuentran correctamente cargadas, que contienen el número esperado de puntos de medida, y que sus valores presentan una distribución coherente con las expectativas del ensayo. Esta comprobación inicial permite asegurar la validez del análisis posterior y evitar errores derivados del uso de datos incompletos o mal estructurados. El proceso de verificación de datos y selección de curvas se representa en la Fig. 92.

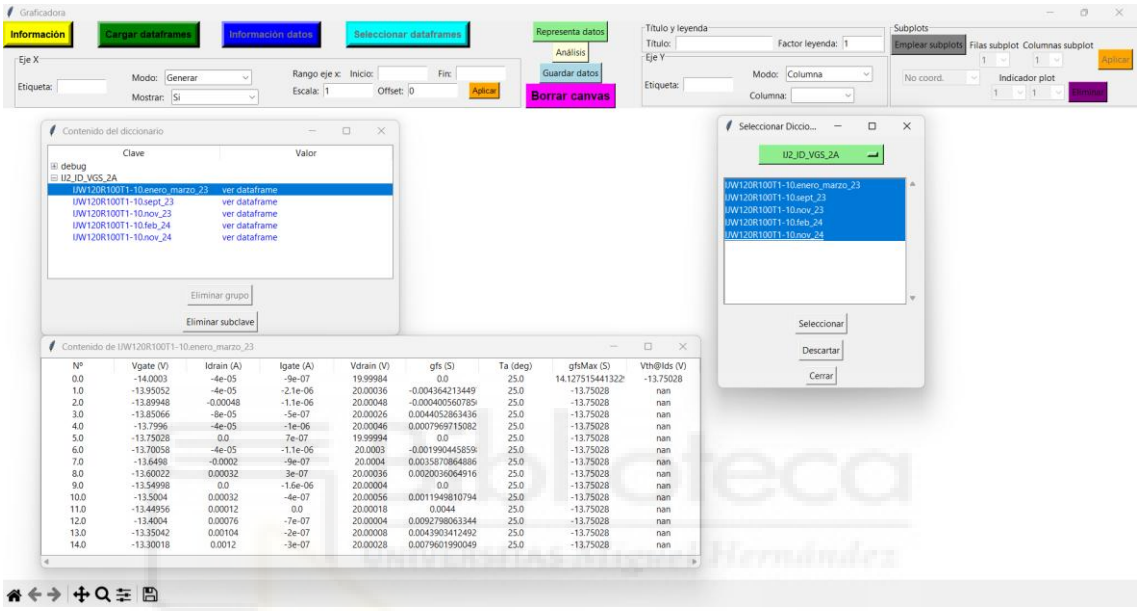


Fig. 92: Proceso de comprobación de la integridad de los datos de caracterización.

Continuando con el proceso, en esta prueba se pretende representar, por un lado, las curvas características $I_d - V_{gs}$ y, por otro, sus correspondientes derivadas, distribuidas en dos subgráficas: la superior para las curvas originales y la inferior para sus derivadas. A diferencia del caso de sobrecarga, no es necesario generar manualmente el eje X, ya que este se encuentra incluido en el propio *dataframe*. Por tanto, basta con seleccionar la columna correspondiente como eje X en el menú de configuración.

Dado que tanto las curvas como sus derivadas comparten el mismo eje horizontal —la tensión puerta-surtidor (V_{gs})—, resulta recomendable activar la opción "No mostrar columna" para el eje X de la gráfica superior, evitando así redundancias visuales. Asimismo, no se asignará etiqueta al eje X de dicha gráfica, ya que dicha información será visible en la gráfica inferior, y su duplicación no aportaría claridad adicional.

Tal como se hizo en el análisis de sobrecargas, se habilita la opción de "Emplear subplots" (véase Fig. 93) para representar ambas gráficas de forma simultánea. Una vez visualizadas las curvas $I_d - V_{gs}$ en la parte superior, se accede al menú "Análisis" para proceder con el cálculo de las derivadas. En este caso, se selecciona la opción "Análisis temporal" y se configuran los parámetros correspondientes al filtro que se aplicará a la derivada, siendo el método elegido el filtro de Savitz-Golay. Este paso es fundamental

para suavizar los datos y obtener una derivada más representativa, minimizando la influencia del ruido experimental.

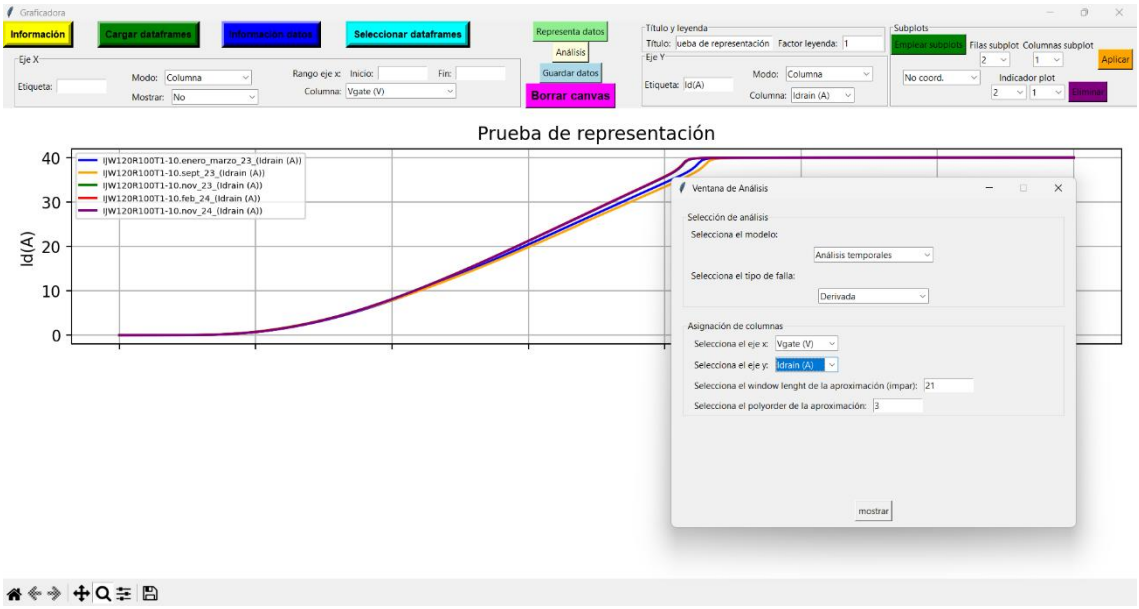


Fig. 93: Proceso de representación de curvas $I_d - V_{gs}$.

Una vez realizada la representación —y, en este caso, asignando una etiqueta al eje X para la gráfica de la derivada—, resulta útil activar la opción de coordinar el eje X de ambos subplots. Esta funcionalidad permite sincronizar el desplazamiento y el zoom horizontal entre las dos gráficas, lo que facilita la comparación visual de zonas específicas de interés, especialmente aquellas donde se presentan transiciones bruscas o comportamientos anómalos. El resultado final se puede observar en la Fig. 94.

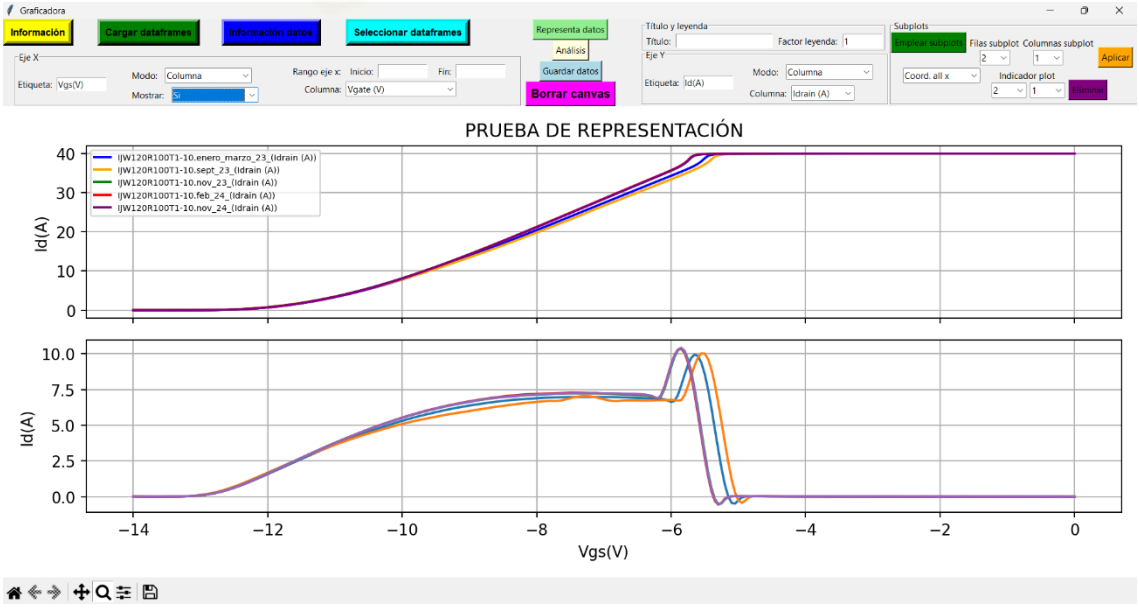


Fig. 94: Representación de curvas y derivadas.

5.CONCLUSIONES Y LÍNEAS FUTURAS

Banco de trabajo

- **Conclusiones:**
  - El banco de trabajo desarrollado ha demostrado un comportamiento sólido y fiable durante todas las pruebas de sobrecarga realizadas (>100k repeticiones), cumpliendo con los requisitos de repetibilidad, seguridad y robustez necesarios en este tipo de ensayos.

  - La integración de los distintos subsistemas —incluyendo los circuitos de conmutación, protecciones, elementos de medida y adquisición— ha funcionado de manera coordinada y estable. La utilización de LabView como entorno de control ha permitido establecer una interconexión eficaz con todos los elementos del sistema, facilitando tanto la automatización de las rutinas como el seguimiento del estado de cada componente.

  - La inclusión en el programa de LabView de un modo de operación específico para la fase de pruebas ha sido especialmente útil. Esta funcionalidad ha permitido realizar todo tipo de comprobaciones de forma sencilla y segura antes de iniciar las rutinas completas de adquisición, evitando errores de configuración o funcionamiento que podrían haber derivado en fallos más serios o en la repetición innecesaria de ensayos.

  - La estrategia de almacenamiento en carpetas jerarquizadas ha resultado muy adecuada para organizar los grandes volúmenes de datos generados, facilitando su posterior análisis. Cada repetición produce archivos CSV de gran tamaño (alrededor de 26 MB por fichero), generando un total aproximado de 26 GB por prueba al realizarse 1000 repeticiones. Esta elevada carga de datos ha obligado a almacenar directamente sobre discos duros externos, lo que ha añadido cierto retardo adicional entre repeticiones debido al tiempo de comunicación entre el sistema de adquisición y el disco. Sin embargo, este aumento de latencia no ha supuesto ningún problema operativo y se ha gestionado adecuadamente con los tiempos de espera definidos.

- **Líneas futuras:**
  - Una posible mejora significativa sería la sustitución del entorno de desarrollo actual basado en LabView por una plataforma de control implementada en Python. Aunque esta alternativa implicaría un mayor esfuerzo de desarrollo inicial, Python, al ser un lenguaje de programación de código abierto, ofrece una gran flexibilidad y cuenta con una comunidad activa y en constante crecimiento. Esto se traduce en una amplia disponibilidad de librerías especializadas, soporte continuado y una mayor capacidad de adaptación a nuevas necesidades futuras sin depender de licencias propietarias. Esta transición permitiría además una integración más fluida con herramientas de análisis de datos, facilitando el desarrollo de un ecosistema unificado de control y *postprocesado*.

  - Otra posible línea futura podría ser el centralizar el sistema de generación de tren de pulsos en un único dispositivo, como una FPGA, en lugar de

depender de la combinación del DAQ y Arduino actuales. Esta centralización reduciría las fuentes de error, mejorando la sincronización y precisión del control de señales, y simplificaría el diseño general del sistema.

- En cuanto al sistema de gestión de errores, se recomienda hacerlo más robusto mediante la activación automática de señales de alarma cuando se detecten valores anómalos de tensión o corriente durante períodos prolongados, tanto en las mediciones del osciloscopio como en la fuente de alimentación. Asimismo, se sugiere la posible inclusión de sensores de temperatura, no solo para un mejor conocimiento y monitorización del proceso de prueba, sino también para activar alarmas ante condiciones térmicas peligrosas. Además, se propone integrar un control directo sobre la fuente de alimentación desde el ordenador, para poder monitorizar y ajustar sus parámetros en tiempo real, aumentando así la seguridad y fiabilidad de las pruebas.

Herramientas de análisis de sobrecarga

- **Conclusiones:**
  - Las herramientas desarrolladas para el análisis de datos en condiciones de sobrecarga han demostrado ser eficaces para procesar el conjunto completo de curvas adquiridas durante los ensayos, permitiendo extraer información relevante de forma estructurada y eficiente. El planteamiento basado en la identificación de zonas clave dentro de las curvas —las cuales se mantienen similares para cada tipo de sobrecarga debido a una configuración uniforme del osciloscopio— ha facilitado en gran medida la automatización del análisis y la reducción del tiempo de procesamiento. Esta estrategia ha permitido aplicar algoritmos de análisis precisos únicamente en los intervalos de mayor interés, optimizando recursos computacionales sin comprometer la calidad de los resultados. Asimismo, la utilización del filtro de Savitz-Golay como técnica de suavizado ha resultado especialmente adecuada para la reducción del ruido presente en las señales, manteniendo la integridad de los datos y conservando las características significativas de las curvas. La selección cuidadosa de los parámetros del filtro en función de la zona específica de la curva ha permitido preservar la información crítica sin introducir distorsiones relevantes.

  - Además, las distintas formas implementadas para representar los resultados, tanto en su forma cruda como agrupadas por conjunto de LCL, han sido fundamentales para facilitar una interpretación clara de los datos. En particular, el uso de diagramas de bigotes ha aportado una visión estadística valiosa sobre la dispersión y tendencia central de los resultados, aunque tan solo se realizaran sobre 4 curvas, permitiendo condensar la información en menos espacio. Esta capacidad de visualizar comparativas de forma directa ha resultado clave para extraer conclusiones cuantitativas sobre el comportamiento de los dispositivos bajo test.

- **Visiones futuras:**
    - Como principales propuestas de mejora para las herramientas de análisis de datos, se plantea la posibilidad de implementar un sistema de almacenamiento basado en bases de datos estructuradas. Este enfoque permitiría organizar y acceder a la información de manera más eficiente y escalable, especialmente útil en caso de que se amplíe el volumen de datos o se busque integrar el análisis con otras herramientas. Existen diversas librerías en Python, como 'sqlite3', 'SQLAlchemy' o incluso las funcionalidades integradas con la librería de uso actual Pandas, con soporte para bases de datos, que facilitarían esta transición y permitirían consultas más dinámicas y estructuradas frente al modelo actual basado únicamente en ficheros.

    - Por otro lado, se considera relevante desarrollar métodos de análisis más robustos y transversales, capaces de adaptarse automáticamente a diferentes tipos de curvas y configuraciones de sobrecarga sin requerir un ajuste fino manual para cada nuevo caso. Esto supondría un importante ahorro de tiempo y evitaría la dependencia de configuraciones personalizadas, además de mejorar la generalización del sistema frente a futuras ampliaciones del banco de pruebas o nuevos escenarios experimentales.

    - En cuanto a la representación gráfica de los resultados, sería conveniente explorar nuevas formas de visualización que permitan condensar la información sin perder claridad, especialmente en situaciones en las que el número de curvas por conjunto es reducido —como en este caso, donde se disponen de apenas 4 curvas por grupo de LCL—. Alternativas como gráficos de radar, heatmaps o representaciones por agrupaciones temporales podrían proporcionar una mayor expresividad y facilitar la comparación cualitativa entre curvas.

Herramientas de análisis de caracterización

- **Conclusiones:**
    - Las herramientas de análisis desarrolladas para la caracterización, aunque han sido las menos evolucionadas dentro del conjunto del trabajo —debido a que las pruebas de caracterización se realizaron de manera externa—, han mostrado una notable eficacia en su función principal: el procesamiento automático de ficheros estructurados en formato no estándar (no CSV), recopilados en directorios, y su conversión en estructuras de datos jerarquizadas mediante diccionarios en Python. Este enfoque ha permitido organizar y acceder a la información de forma ordenada y eficiente, facilitando el análisis posterior.

    - Durante el proceso únicamente se han detectado incidencias menores, relacionadas con ficheros que no seguían el patrón estándar debido a errores en el funcionamiento del instrumento de caracterización. No obstante, estos casos atípicos fueron tratados manualmente sin suponer un impacto relevante en el flujo de trabajo general.

- En cuanto a las herramientas de representación gráfica, estas han cumplido su objetivo de proporcionar una visión clara y general del comportamiento de los diferentes grupos de dispositivos. El análisis simplificado de la tensión umbral y del valor de corriente constante ha sido especialmente útil para identificar tendencias y evaluar posibles degradaciones. Además, el uso de gráficos de violín por fechas ha demostrado ser una herramienta fructífera para visualizar los cambios experimentados por los dispositivos tras cada fase de pruebas, permitiendo observar fácilmente desplazamientos estadísticos y dispersiones dentro de los conjuntos de datos.

- Cabe mencionar que, aunque se exploró el uso del método basado en la meseta de la transconductancia para la estimación de parámetros, este solo pudo aplicarse en ciertos tipos de curvas que no seguían el patrón habitual. Por tanto, si bien ha mostrado resultados prometedores en esos casos concretos, no se considera adecuado como método generalizado dentro del flujo de análisis actual.

- Finalmente, el método de corriente constante, aunque demostró su robustez y su facilidad de aplicación a todos los casos del presente trabajo, también se puso de manifiesto la necesidad de estudiar un criterio adecuado de corriente umbral que pueda aportar valores más fidedignos con la realidad.

- **Líneas futuras:** Como posible mejora se plantea una mayor exploración entre los distintos métodos de obtención de la tensión umbral, con el objetivo de identificar un enfoque más sofisticado y que sea aplicable de forma general a todos los casos.

Herramientas de visualización

- **Conclusiones:**
  - Las herramientas de visualización desarrolladas han resultado ser un componente indispensable dentro del ecosistema de análisis de este trabajo. Su papel ha sido especialmente relevante en el tratamiento de datos de sobrecarga, donde se requiere un ajuste muy fino de los parámetros de análisis, así como en las tareas de caracterización. Gracias a sus múltiples opciones de representación, como la generación de matrices de gráficos con posibilidad de compartir ejes, ha sido posible abordar análisis comparativos complejos de forma clara y estructurada.

  - Uno de los aspectos más destacables de esta herramienta es su carácter transversal. No se encuentra limitada a un único tipo de análisis o conjunto de datos, sino que puede ser utilizada para representar gráficamente cualquier información contenida en archivos CSV o estructuras tipo *dataframe* de pandas. Esta versatilidad ha permitido una integración fluida con el resto de las herramientas desarrolladas en el trabajo.

  - Asimismo, la posibilidad de exportar los datos generados tras la aplicación de distintos análisis añade una funcionalidad clave para el tratamiento posterior de los resultados, ya sea en otros entornos o para su documentación. En este sentido, se ha logrado condensar en una única herramienta todas las funcionalidades de representación gráfica más relevantes para el trabajo, permitiendo al usuario aplicarlas de forma sencilla

sin recurrir a programación manual. De no haber contado con este recurso, la creación de gráficas con distintos parámetros habría supuesto un coste de tiempo considerable.

- Por otro lado, el sistema de control de errores incorporado ha sido especialmente eficaz a la hora de tratar con los archivos CSV generados durante el trabajo. Estos archivos, debido a su elevado peso, han supuesto un desafío técnico considerable. En fases iniciales, antes de la implementación completa de este sistema de control, se produjeron situaciones en las que la interfaz, e incluso el propio equipo, llegaban a bloquearse al alcanzarse casi el 100% del uso de la memoria RAM. La inclusión de este mecanismo ha permitido mitigar por completo dichos problemas, aportando robustez y fiabilidad al entorno de trabajo.

- Finalmente, cabe destacar que la herramienta ha sido programada con una estructura modular, lo que facilita enormemente su mantenimiento y futuras actualizaciones. Esta arquitectura permite adaptar el programa de forma consistente y ordenada ante nuevas necesidades, garantizando su reutilización y ampliación en otros contextos o proyectos similares.

- **Líneas futuras:**
  - Dado que la herramienta de visualización es un programa desarrollado específicamente para este entorno de trabajo, se plantean diversas actualizaciones que permitan ampliar sus funcionalidades y adaptabilidad. En primer lugar, se propone la incorporación de más opciones de personalización gráfica, tales como ajustes avanzados de color, escala, estilos de línea y selección de subconjuntos de datos directamente desde la interfaz, lo cual mejoraría la experiencia de análisis en escenarios más específicos.

  - Además, una línea de mejora especialmente relevante es la personalización de la interfaz de usuario. Se propone permitir al usuario reorganizar los elementos visuales según sus necesidades, activar o desactivar paneles específicos, guardar configuraciones personalizadas de visualización, y crear perfiles de usuario que conserven preferencias entre sesiones. Este tipo de enfoque modular resulta muy útil en contextos donde el tipo de análisis o el perfil del usuario cambia con frecuencia (por ejemplo, entre fases del trabajo o entre diferentes investigadores).

ANEXO 1: TABLAS DE GRUPOS DE DISPOSITIVOS

Tabla 5: Grupos de dispositivos de clase 2.

Id nominal (A)	Ref.	LCL	Componente
2	IJW120R100T1 LCL2A-T1-L1	LCL-1	IJW120R100T1-43
			PMOS-70
		LCL-2	IJW120R100T1-41
			PMOS-35
		LCL-3	IJW120R100T1-49
			PMOS-81
		LCL-4	IJW120R100T1-10
			PMOS-28
2	UJN1205K LCL2A-T2-L1	LCL-5	UJN1205K-39
			PMOS-37
		LCL-6	UJN1205K-27
			PMOS-63
		LCL-7	UJN1205K-12
			PMOS-71
		LCL-8	UJN1205K-17
			PMOS-98
2	UJ3N065025K3S LCL2A-T3-L1	LCL-9	UJ3N065025K3S-9
			PMOS-90
		LCL-10	UJ3N065025K3S-8
			PMOS-74
		LCL-11	UJ3N065025K3S-43
			PMOS-77
		LCL-12	UJ3N065025K3S-10
			PMOS-55
2	UJ3N120035K3S LCL2A-T4-L1	LCL-13	UJ3N120035K3S-44
			PMOS-91
		LCL-14	UJ3N120035K3S-37
			PMOS-49
		LCL-15	UJ3N120035K3S-43
			PMOS-30
		LCL-16	UJ3N120035K3S-26
			PMOS-87

Tabla 6: Grupos de dispositivos de clase 10 (parte 1).

Id nominal (A)	Ref.	LCL	Componente
10	IJW120R100T1 LCL10A-T1-L2	LCL-17	IJW120R100T1-5
			IJW120R100T1-8
			IJW120R100T1-9
			PMOS-75
		LCL-18	IJW120R100T1-6
			IJW120R100T1-4
			IJW120R100T1-46
			PMOS-62
		LCL-19	IJW120R100T1-45
			IJW120R100T1-48
			IJW120R100T1-47
			PMOS-54
		LCL-20	IJW120R100T1-38
			IJW120R100T1-44
			IJW120R100T1-42
			PMOS-50
10	UJN1205K LCL10A-T2-L2	LCL-21	UJN1205K-6
			UJN1205K-49
			UJN1205K-7
			PMOS-53
		LCL-22	UJN1205K-25
			UJN1205K-34
			UJN1205K-50
			PMOS-58
		LCL-23	UJN1205K-2
			UJN1205K-45
			UJN1205K-24
			PMOS-48
		LCL-24	UJN1205K-5
			UJN1205K-19
			UJN1205K-48
			PMOS-45

Tabla 7: Grupos de dispositivos de clase 10 (parte 2).

Id nominal (A)	Ref.	LCL	Componente
10	UJ3N065025K3S LCL10A-T3-L2	LCL-25	UJ3N065025K3S-27
			UJ3N065025K3S-25
			UJ3N065025K3S-20
			PMOS-73
		LCL-26	UJ3N065025K3S-35
			UJ3N065025K3S-44
			UJ3N065025K3S-24
			PMOS-29
		LCL-27	UJ3N065025K3S-36
			UJ3N065025K3S-26
			UJ3N065025K3S-30
			PMOS-32
		LCL-28	UJ3N065025K3S-37
			UJ3N065025K3S-21
			UJ3N065025K3S-45
			PMOS-72
10	UJ3N120035K3S LCL10A-T4-L2	LCL-29	UJ3N120035K3S-47
			UJ3N120035K3S-12
			UJ3N120035K3S-21
			PMOS-46
		LCL-30	UJ3N120035K3S-31
			UJ3N120035K3S-34
			UJ3N120035K3S-33
			PMOS-84
		LCL-31	UJ3N120035K3S-22
			UJ3N120035K3S-46
			UJ3N120035K3S-30
			PMOS-76
		LCL-32	UJ3N120035K3S-32
			UJ3N120035K3S-17
			UJ3N120035K3S-29
			PMOS-34

ANEXO 2: SHIELD PARA ARDUINO

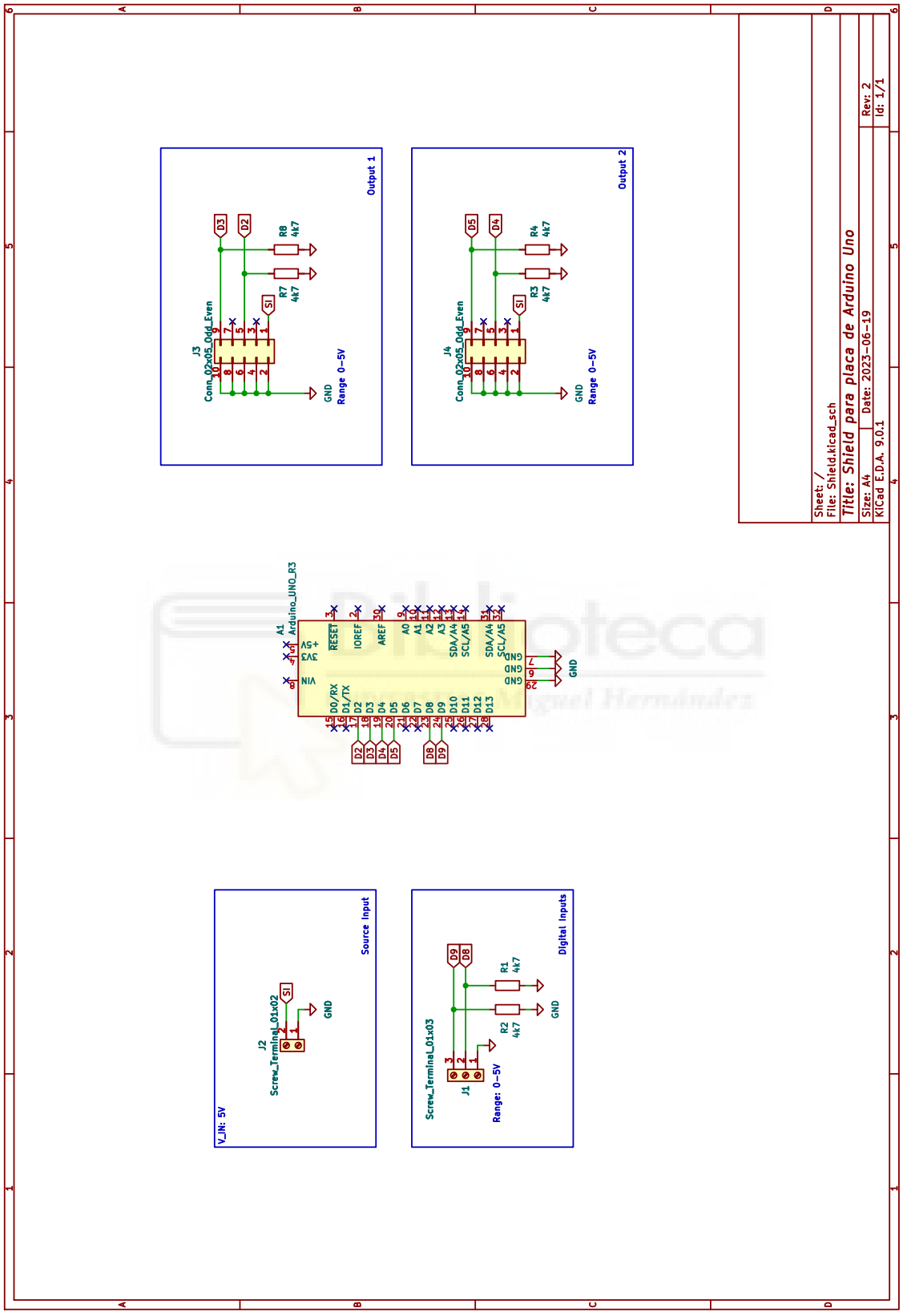


Fig. 95: Esquemático de la placa del shield para la tarjeta de Arduino Uno.

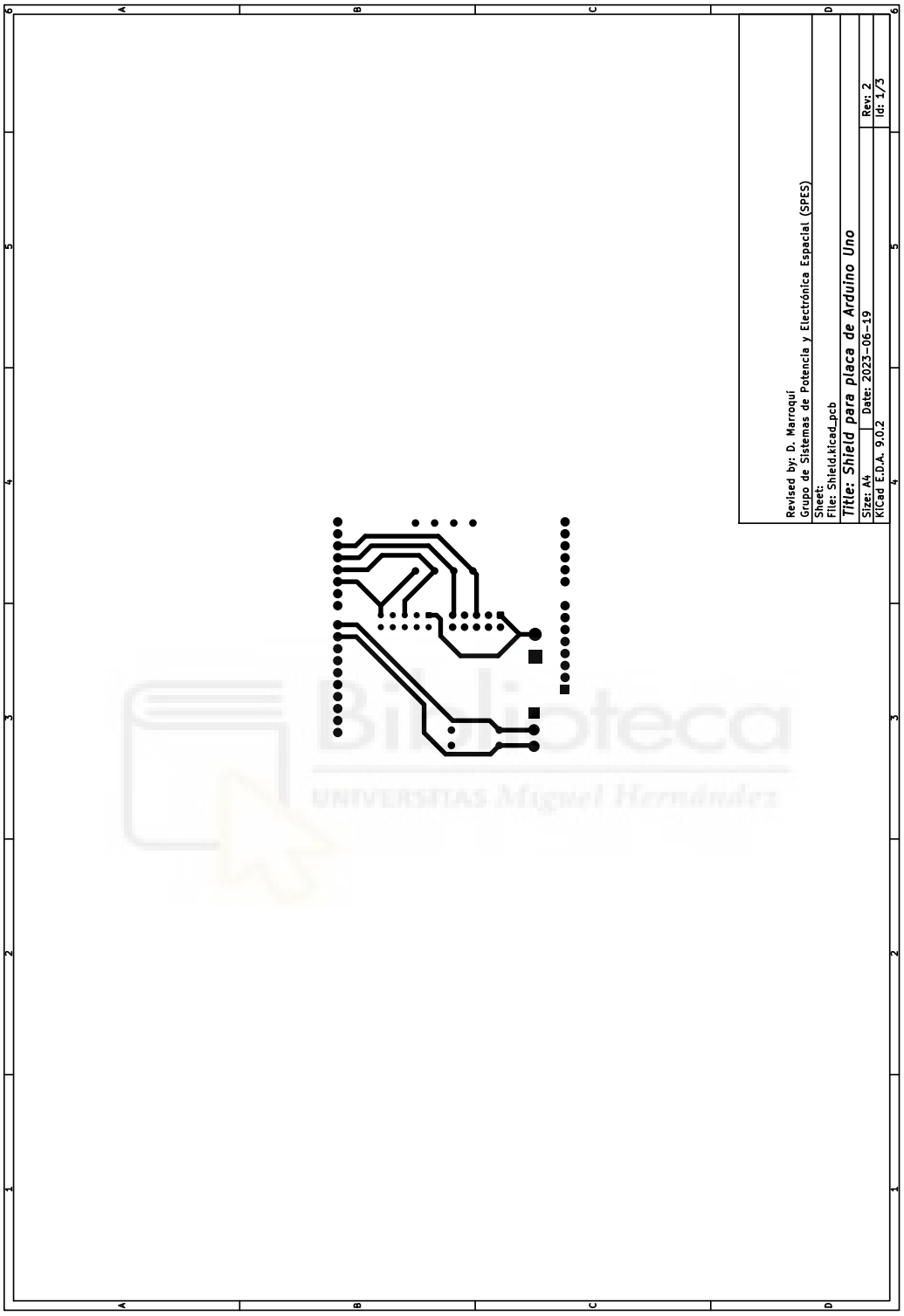


Fig. 96: Capa Top de la placa del shield para la tarjeta de Arduino Uno.

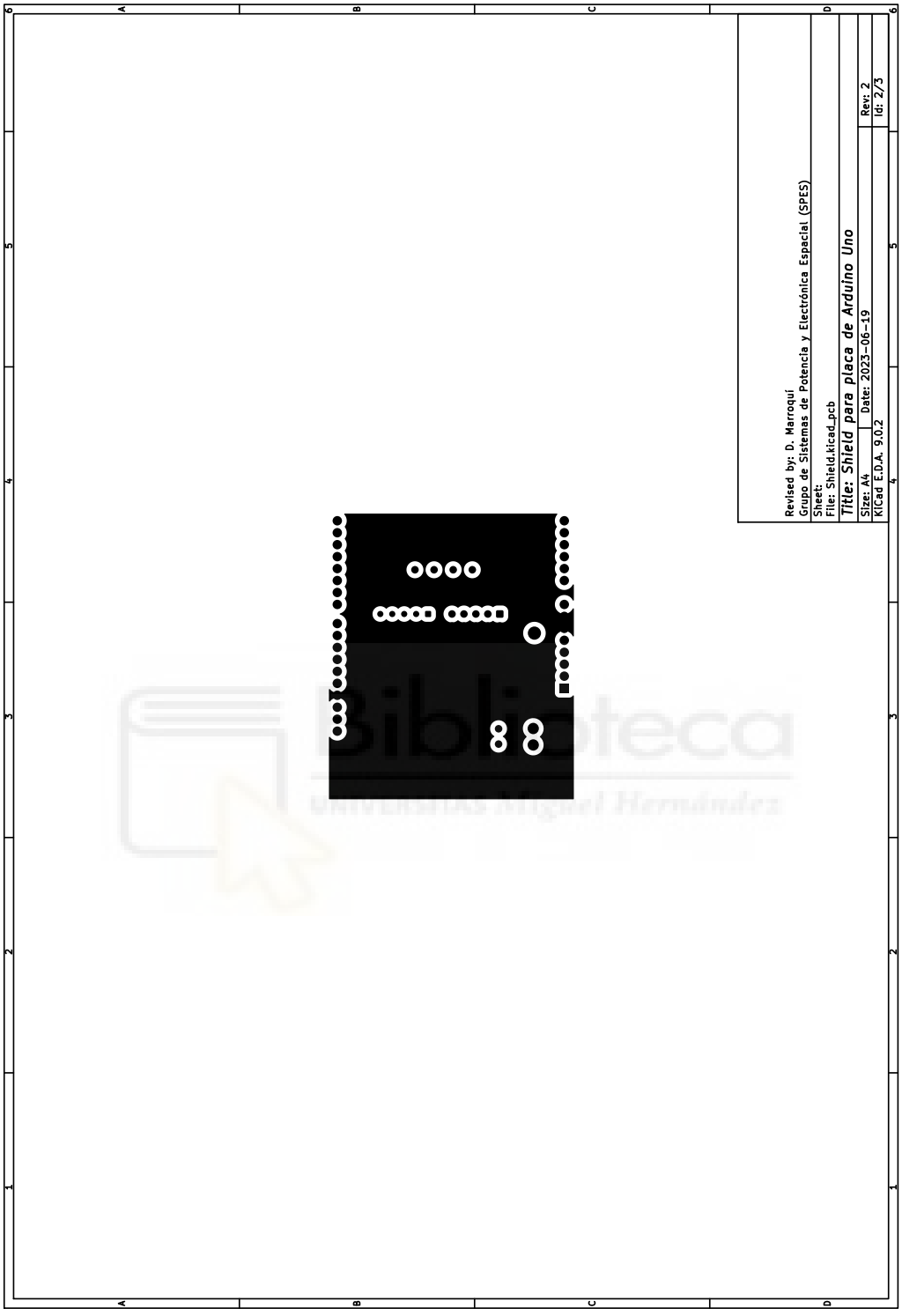


Fig. 97: Capa Bottom de la placa del shield para la tarjeta de Arduino Uno.

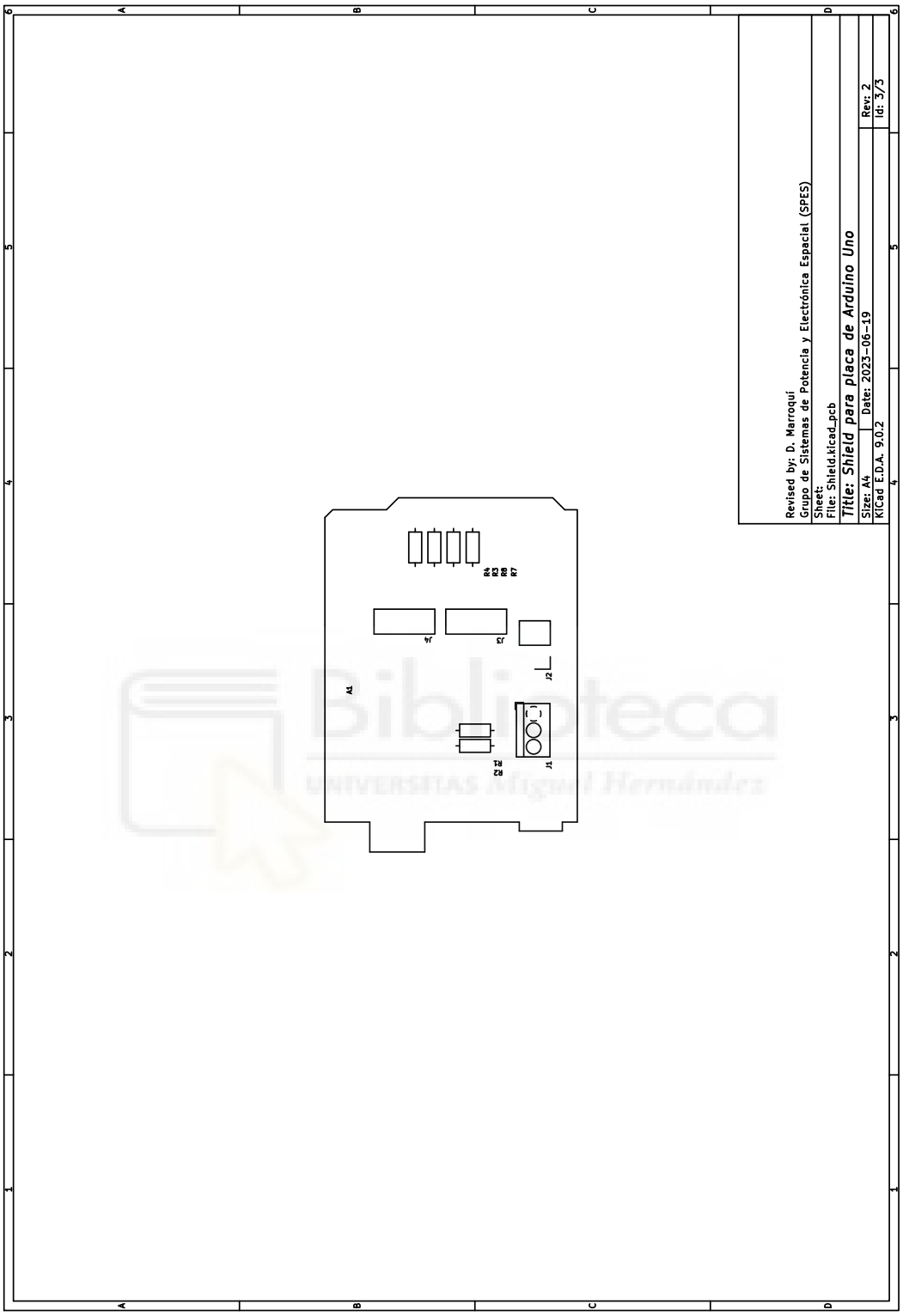


Fig. 98: Capa Silkscreen de la placa del shield para la tarjeta de Arduino Uno.

ANEXO 3: PROGRAMA DE SECUENCIA DE PULSOS DE ARDUINO

```
// Secuencia de doble pulso para sobrecargas para la tarjeta Arduino
//


void setup() {
  Serial.begin(9600);

 //Declaramos pines de entrada
  pinMode(8, INPUT); // Pin para secuencia 1 - Sobrecarga
###IMPORTANTE
  pinMode(9, INPUT); // Pin para secuencia 2 - Arranque capacitivo

  //Declararmos los pines 2 to 5 como salidas
  pinMode(2, OUTPUT); /*pin de ON*/
  pinMode(3, OUTPUT);/*pin IGBT 1*/

  pinMode(4, OUTPUT);
  pinMode(5, OUTPUT);
}

//Ancho del pulso que daremos para activación. En ms (del orden de
0.5s está bien)
int trigger_width=500;

//Definición de los tiempo de la secuencia 1 - Sobrecarga  (Véase
información de gráficos en ONENOTE)
float d_nominal=30; //retraso carga nominal. En ms.
float w_overload=20; //retraso sobrecarga desde el arranque nominal.
En ms.

void loop() {

  if(digitalRead(8) ==  HIGH){
        digitalWrite(3, HIGH);
        delay((int)d_nominal);
//Parte entera del retraso
        delayMicroseconds((d_nominal-(int)d_nominal)*1000);
//Parte decimal del retraso

        digitalWrite(5, HIGH);
        delay((int)w_overload);
//Parte entera del retraso
        delayMicroseconds((w_overload-(int)w_overload)*1000);
//Parte decimal del retraso

        //Parte decimal del retraso

        //Ponemos a cero todas las salidas
        digitalWrite(3, LOW);
        digitalWrite(5, LOW);


        delay(trigger_width*2); // Tiempo de reposo mínimo.
  }

      }
```

ANEXO 4: DIAGRAMA DE BLOQUES DEL PROGRAMA DE LABVIEW EN EL BANCO DE TRABAJO

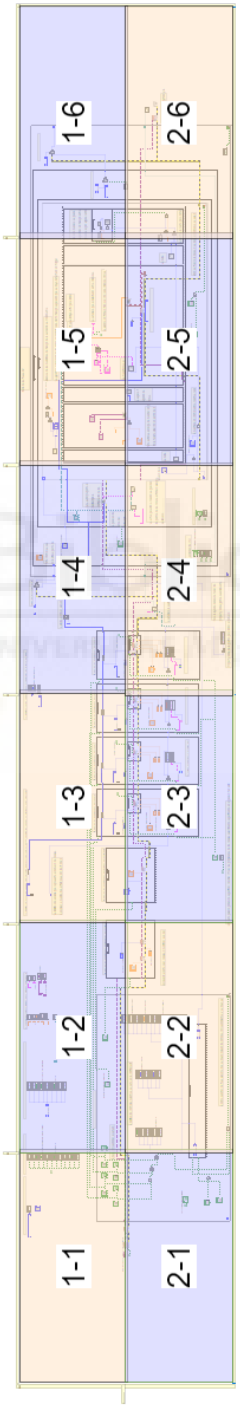


Fig. 99: Diagrama de bloques del programa de LabView en el banco de trabajo (vista general).

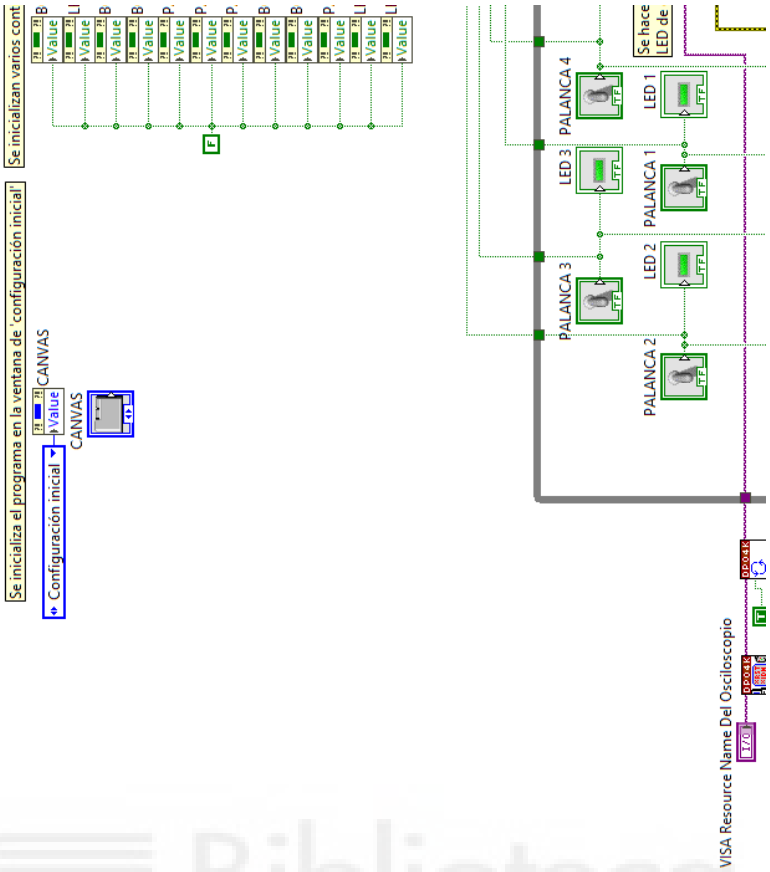


Fig. 100: Diagrama de bloques del programa de LabView en el banco de trabajo (vista 1-1).

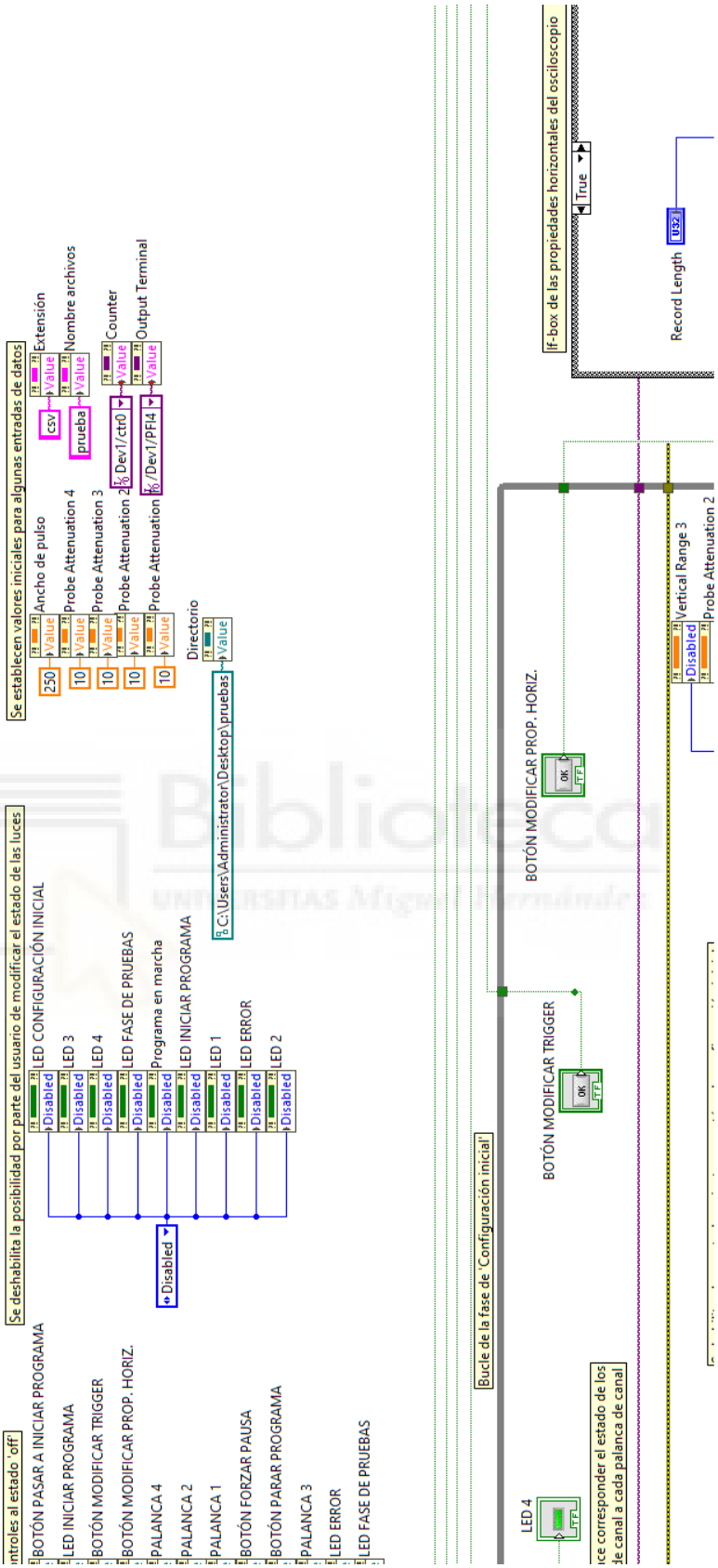


Fig. 101: Diagrama de bloques del programa de LabView en el banco de trabajo (vista 1-2).

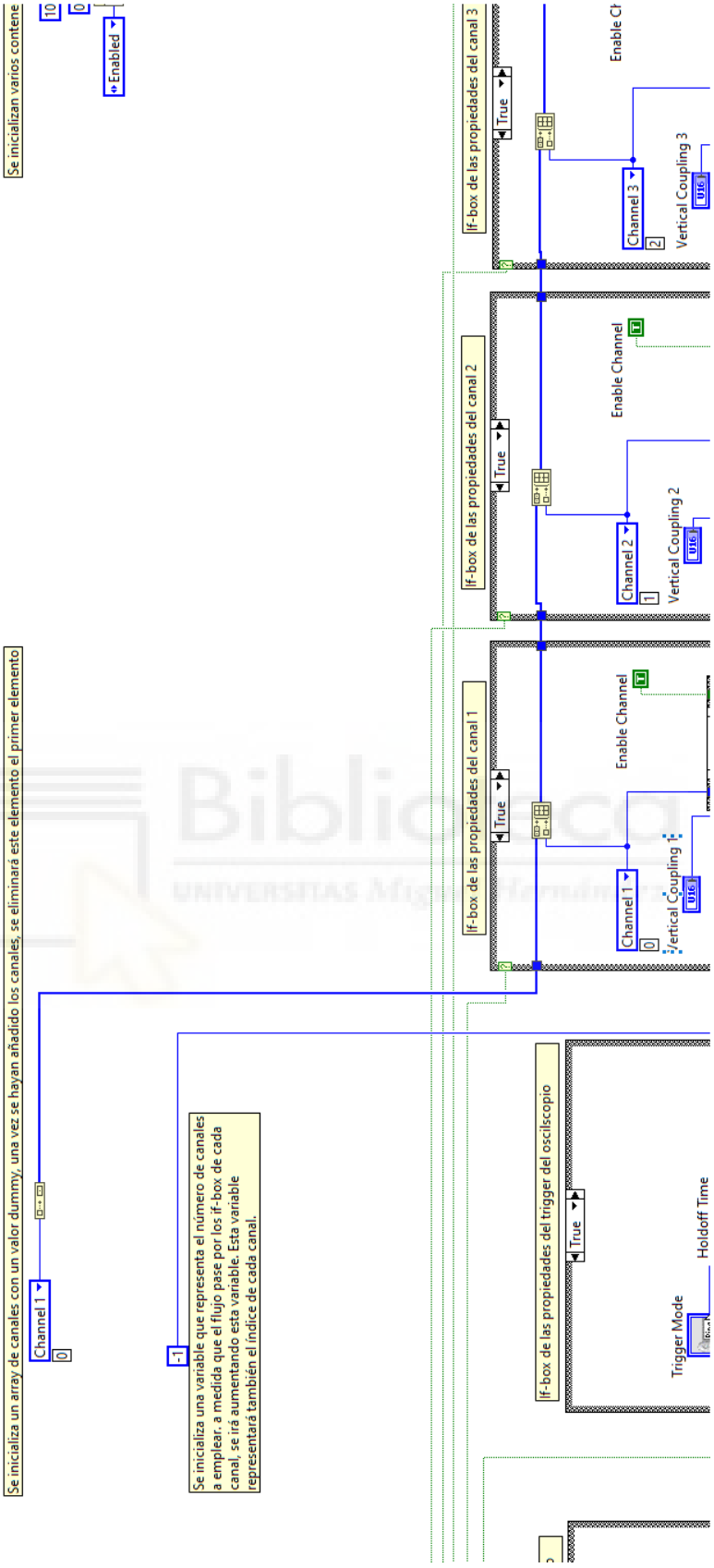


Fig. 102: Diagrama de bloques del programa de LabView en el banco de trabajo (vista 1-3).

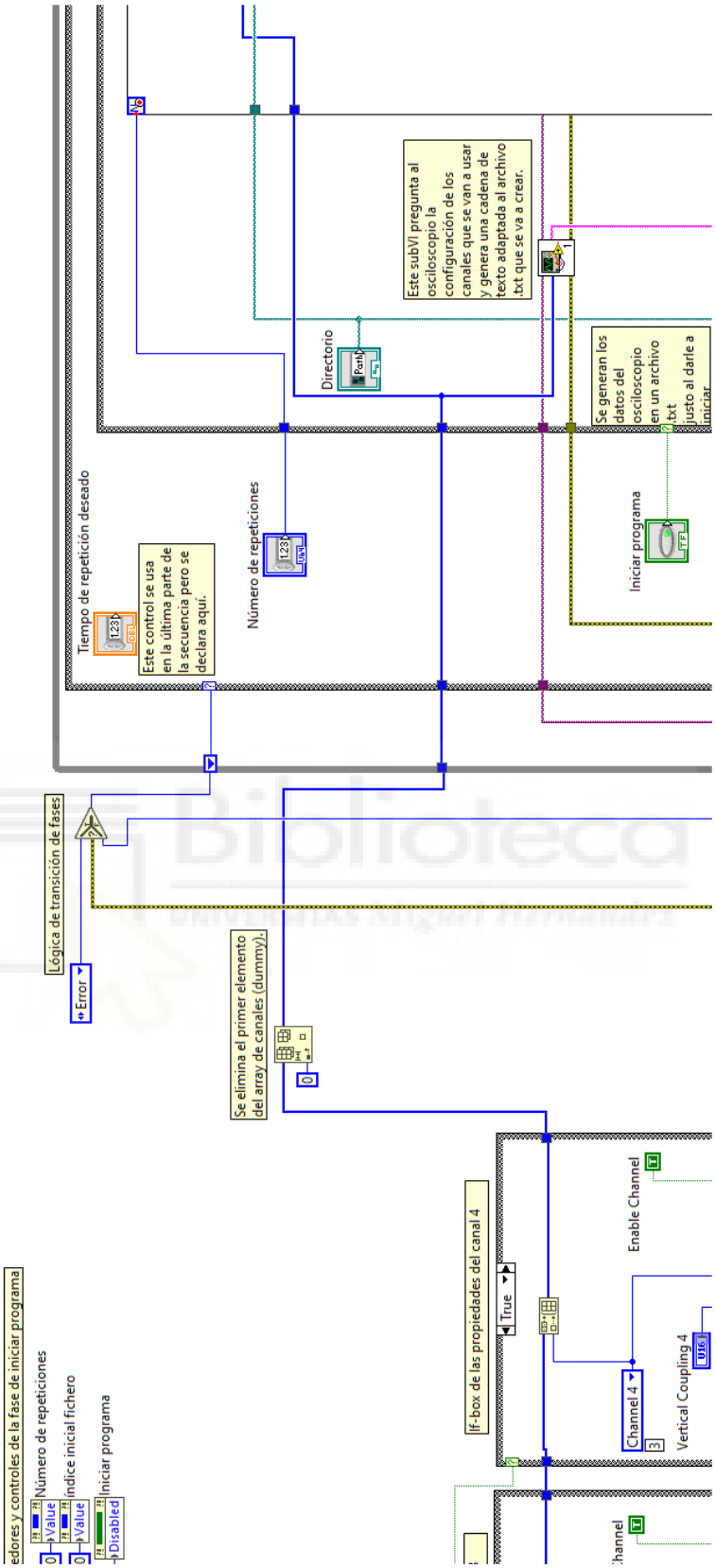


Fig. 103: Diagrama de bloques del programa de LabView en el banco de trabajo (vista 1-4).

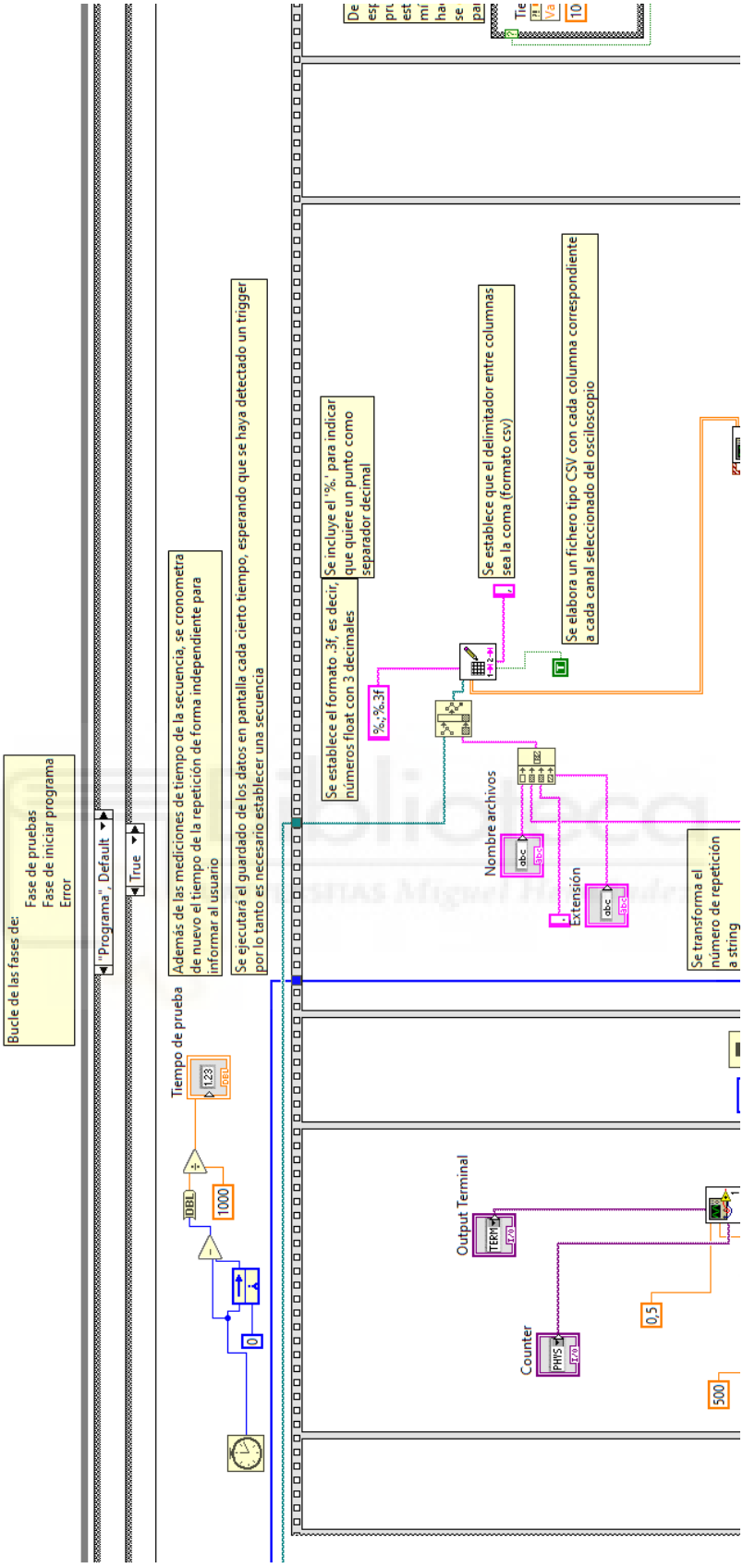


Fig. 104: Diagrama de bloques del programa de LabView en el banco de trabajo (vista 1-5).

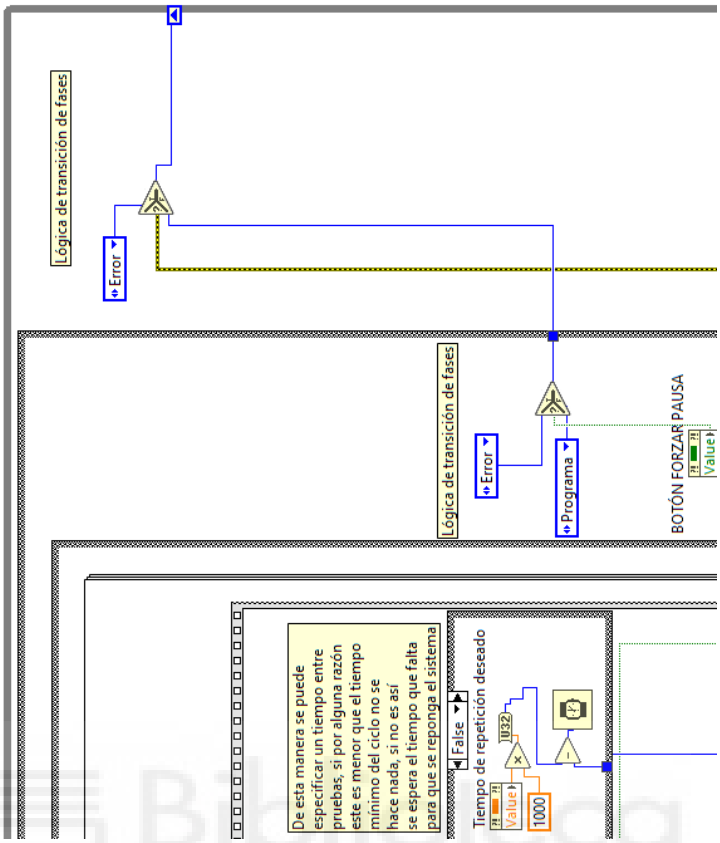


Fig. 105: Diagrama de bloques del programa de LabView en el banco de trabajo (vista 1-6).

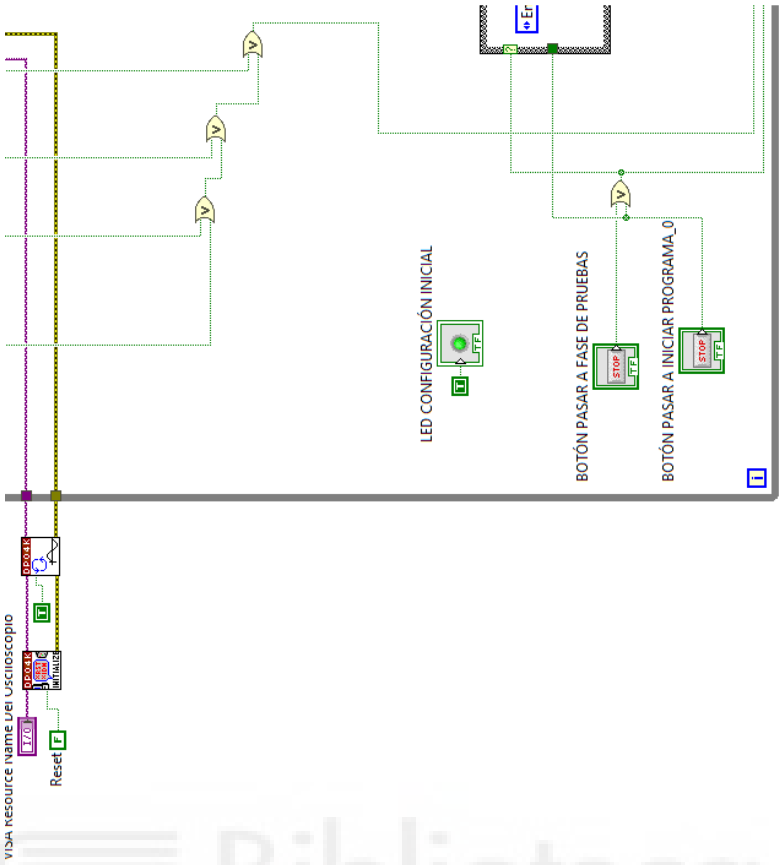


Fig. 106: Diagrama de bloques del programa de LabView en el banco de trabajo (vista 2-1).

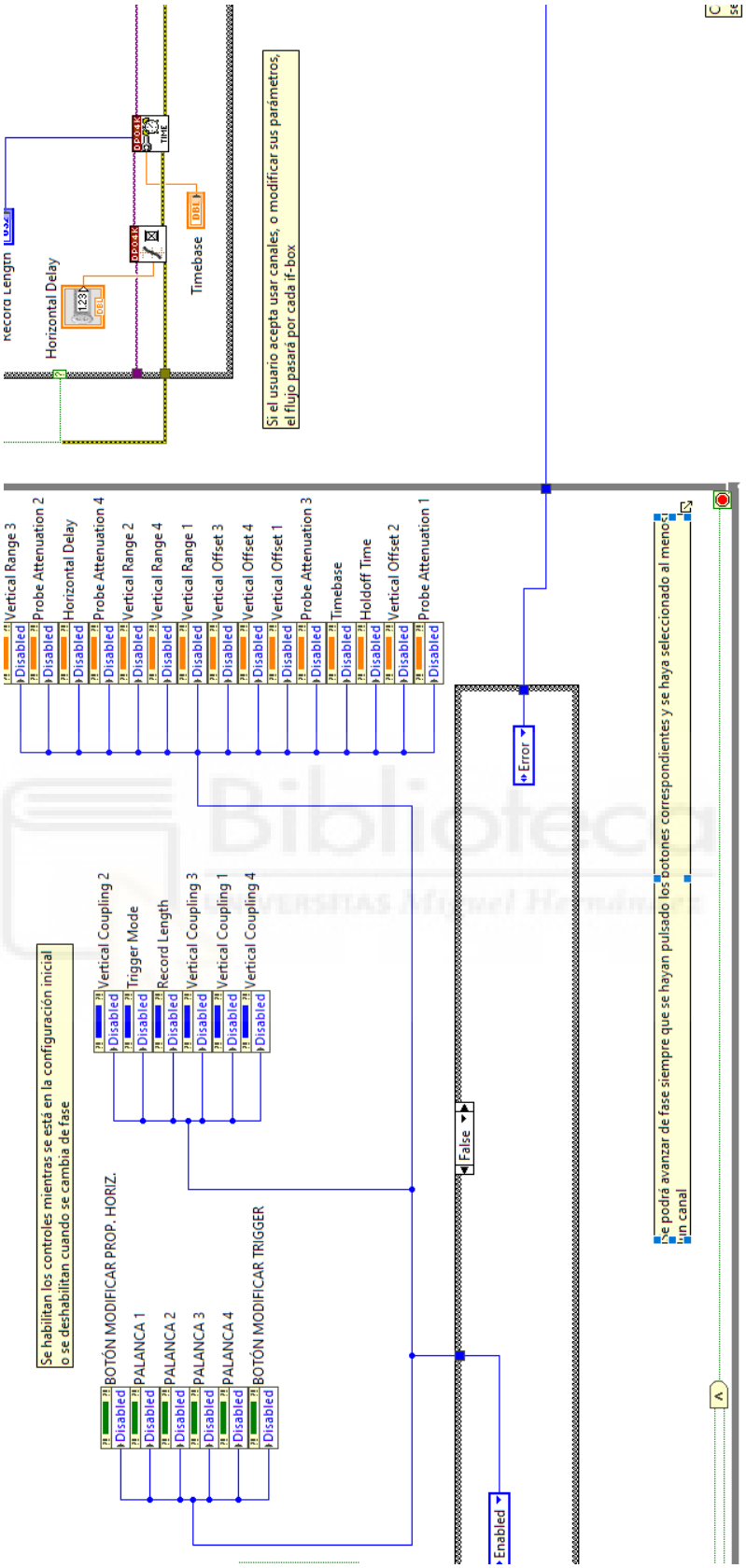


Fig. 107: Diagrama de bloques del programa de LabView en el banco de trabajo (vista 2-2).

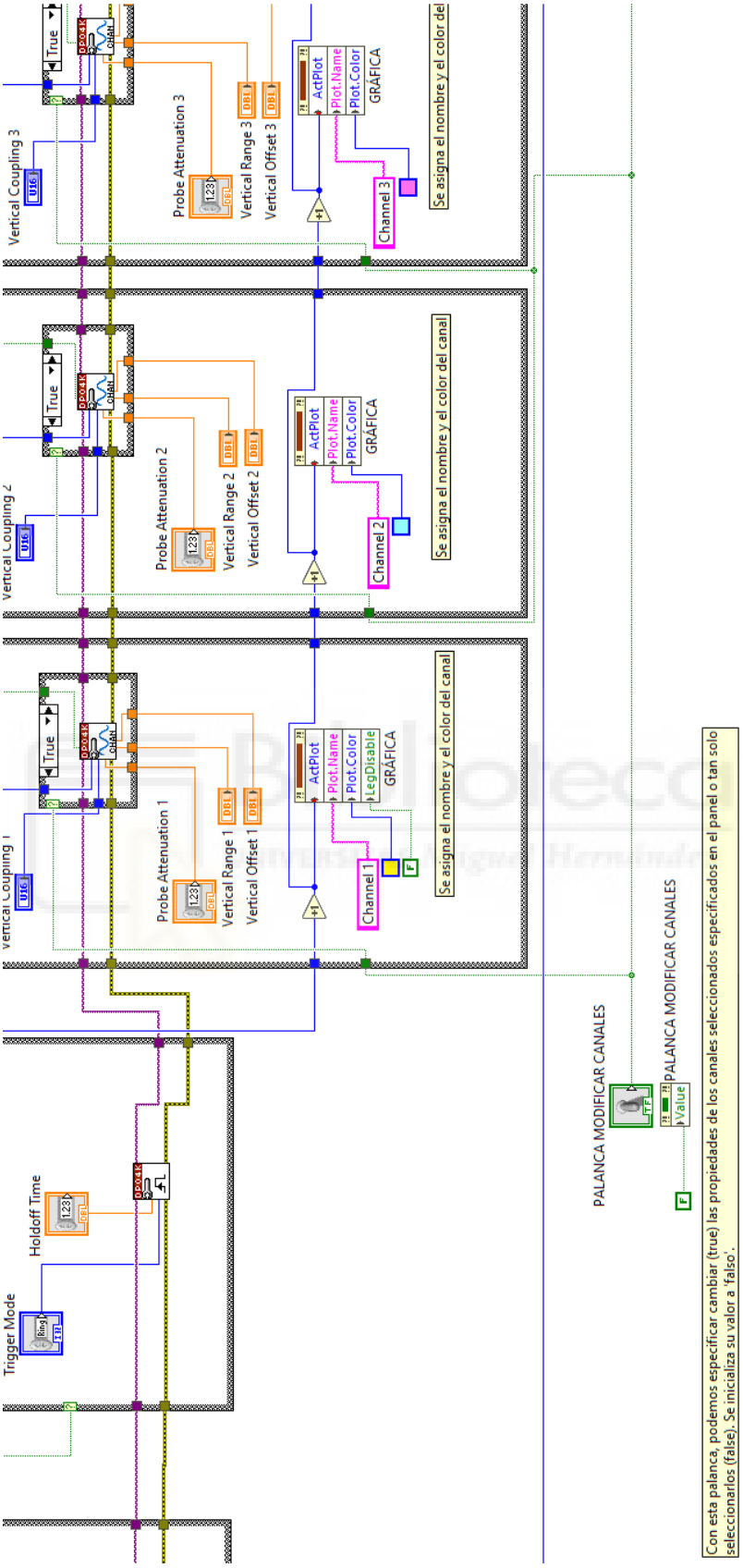


Fig. 108: Diagrama de bloques del programa de LabView en el banco de trabajo (vista 2-3).

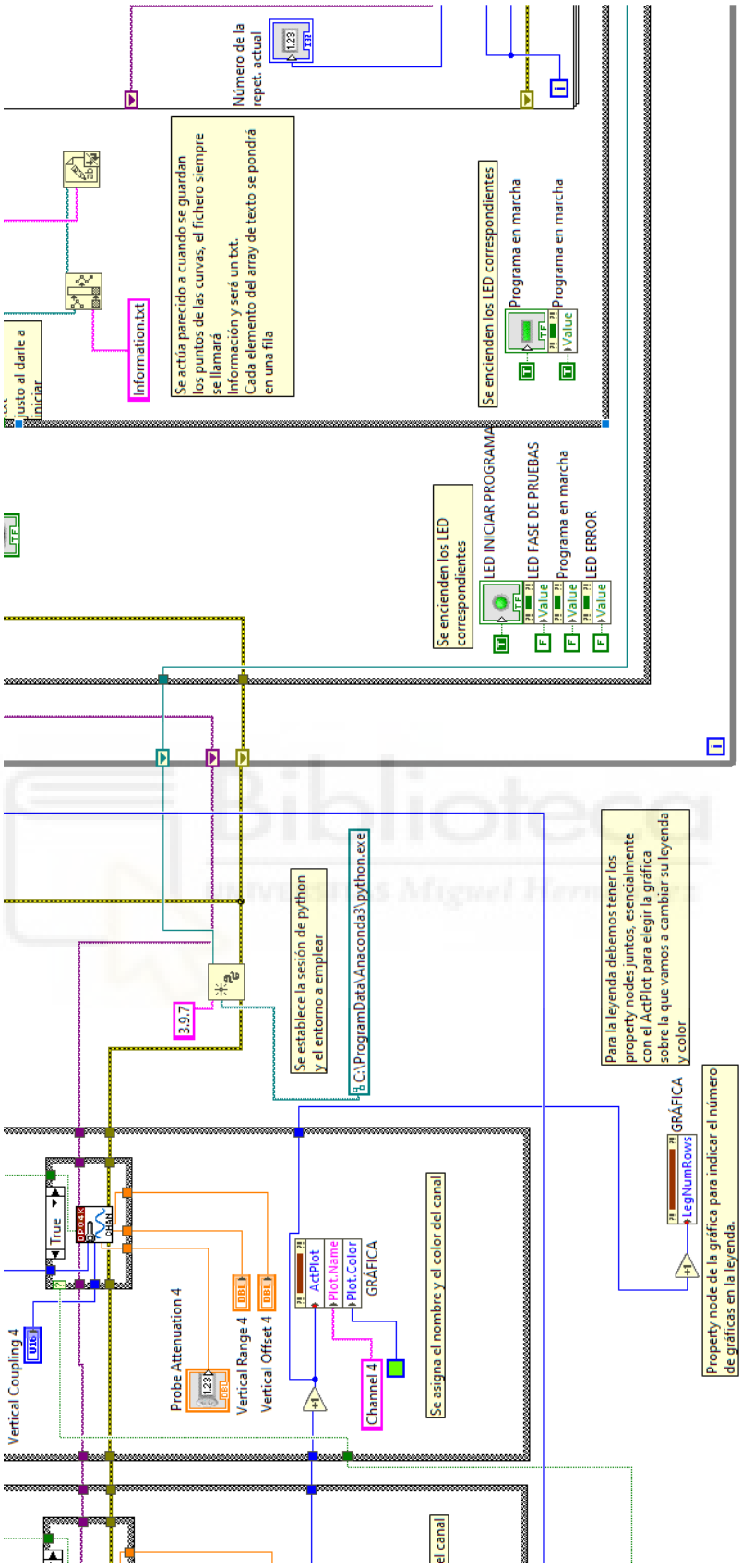


Fig. 109: Diagrama de bloques del programa de LabView en el banco de trabajo (vista 2-4).

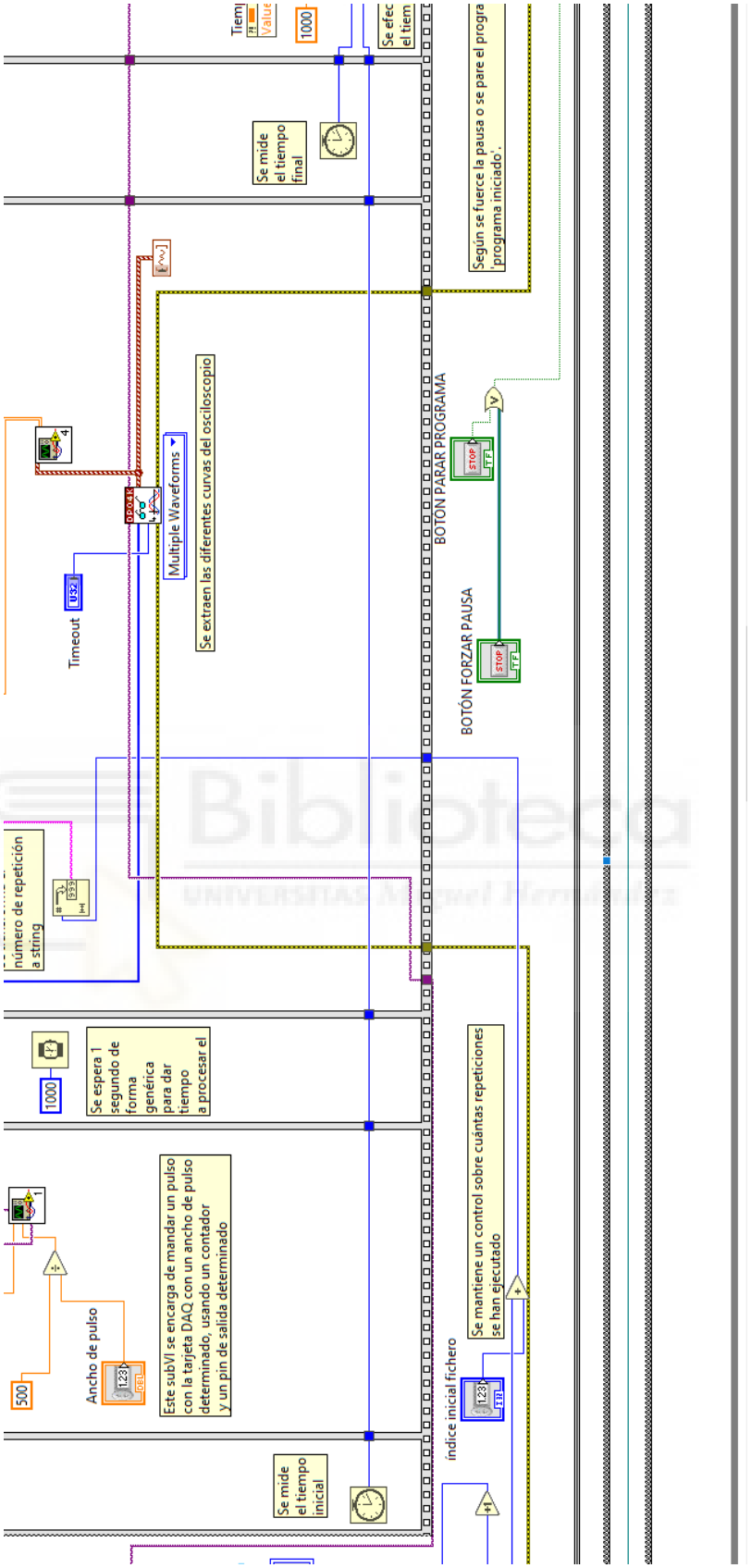


Fig. 110: Diagrama de bloques del programa de LabView en el banco de trabajo (vista 2-5).

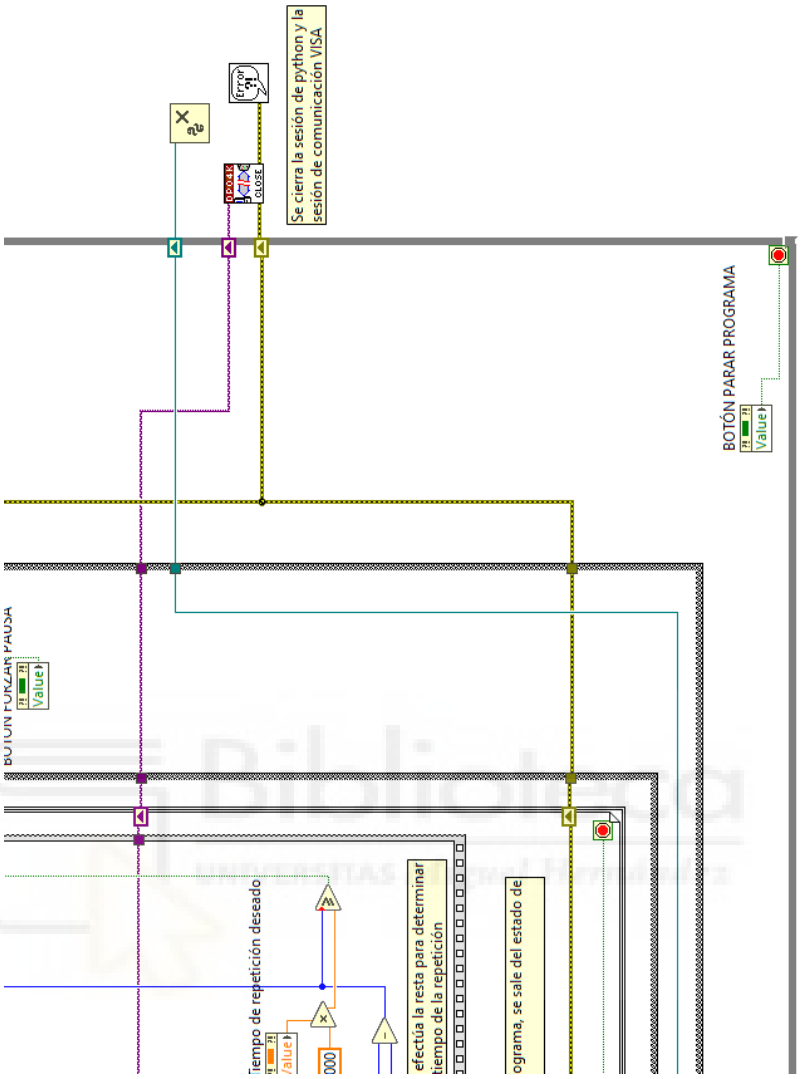


Fig. 111: Diagrama de bloques del programa de LabView en el banco de trabajo (vista 2-6).

ANEXO 5: PUBLICACIONES RELACIONADAS

Ageing and degradation study by thermal, electrical and radiation stress in SiC-JFET/Si-pMOS cascode switches

David Marroquí
Space Power and Electronic Systems, Miguel Hernandez University
Elche, Spain
dmarroqui@umh.es

Ausiàs Garrigós
Space Power and Electronic Systems Miguel Hernandez University
Elche, Spain
augarsir@umh.es

Enrique Maset
Instrumentation and Industrial Electronics Laboratory University of Valencia
Valencia, Spain
enrique.maset@uv.es

David Alcaraz
Space Power and Electronic Systems, Miguel Hernandez University
Elche, Spain
d.alcaraz@umh.es

Guillermo Terol
Instrumentation and Industrial Electronics Laboratory University of Valencia
Valencia, Spain
guillermo.terol@uv.es

Jose M. Blanes
Space Power and Electronic Systems Miguel Hernandez University
Elche, Spain
jmblanes@umh.es

Pablo Casado
Space Power and Electronic Systems Miguel Hernandez University
Elche, Spain
pcasado@umh.es

2024 IEEE 11th Workshop on Wide Bandgap Power Devices & Applications (WiPDA) | 979-8-3503-7560-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/WiPDA62103.2024.10773233

***Abstract*— SiC transistors, particularly JFETs, due to their characteristics, can be an interesting solution for the development of current limiters in the space domain; however, the lack of previous results in this kind of application calls for a detailed study of their reliability.**

In this work, a test campaign is proposed and carried out to corroborate their correct operation and to detect potential problems. In this way, it will be possible to understand which are the most stressful conditions and what is its degradation process. High-temperature conduction tests and several electrical overload campaigns have been carried out on different transistor families and manufacturers. The results reveal, pending further conclusions, a promising future for this type of transistor technology for use in space current limiting applications.

Keywords— SiC, JFET, TID, SEE, COTS, cascode.

I. Introduction

The power distribution and control units (PCDUs) are responsible of distributing power to the different parts of a satellite. They also have the function of protecting the loads and the power bus against fails such as excessive current demands, overvoltage, etc.

In the European environment, PCDUs are typically implemented with Current Limiters (CL) as fundamental elements and their development is covered by the guidelines of the European Space Agency (ESA). [1].

Historically, according to [1, Tbls. 3–1], CLs can distribute voltages up to 52V in non-regulated buses. However, the current industry trend is to increase the voltage of distribution buses. In [2, Sec. 5.7.2.g] ESA establishes bus voltage levels up to 120V for platforms up to 20kW and there are many studies, proposals and ideas to increase the voltage of space platform power buses higher [3], [4], [5].

This work is based on the need for PCDUs and CLs able of handling higher class voltages and power.

According with [1], conventional CLs are implemented with pMOS, allowing a simple driver implementation. This is, nevertheless, a heavy drawback from the point of view of the maximum blocking voltage. The best-in-class radiation tolerant pMOS, the IRHNA597260, allows operation (without considering derating) up to 200V but has 100mOhm R_{DSon}, limiting its use in high power applications. On the other hand, although the development of CLs with nMOS is feasible [6], [7], for the sake of not increasing the complexity of CLs as well as their failure modes, their use in space applications tends to be avoided [8].

The authors propose the use of a switch structure called a p-type cascode to keep the simplicity of a p-type driving and the performance of a high-power transistor. The p-type cascode, previously evaluated as a CL in 100V [9] and 300V [10] applications, is composed of a silicon pMOS low voltage transistor and a silicon carbide (SiC) JFET, see Fig 1.

In order to develop an CL for high-power applications in radiation environments with the p-type cascode structure (Fig 1), radiation-hardened elements are required. In the case of the low-voltage pMOS, there is a commercial offer, but this is not the case for the SiC JFET. According to literature, SiC devices (both MOS and JFET) have demonstrated high electrical reliability and are nowadays widely used in industrial and automotive applications. [11], [12], [13].

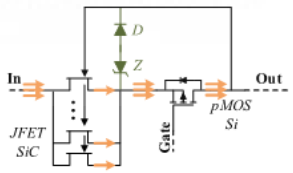


Fig. 1. *P*-type cascode scheme for CL

The reliability of SiC transistors in radiation environments is being studied from different angles. From a Total Dose (TID) view, SiC JFETs present a robust performance, rather similar to SiC diodes (more tested in radiation environments) where only slight increases in leakage currents have been reported in older generations of devices [14]. Overall, it has been observed that they can withstand radiation doses up to 1000krad without relevant changes of their parameters [15]. Their operation in heavy ion environments can generate single events destroying the

device, but this is greatly mitigated applying fail-safe margins on their maximum blocking voltaje [16] in a similar way to SiC and Si MOSFETs. All in all, the SiC JFET can be considered the most reliable of all SiC transistors for use in radiation environments [14] and there are many lines of research to make them more reliable in such environments [17].

Therefore, this work proposes to implement an CL based on a p-type cascode (Fig.1) and to perform an ageing and reliability study in TID and Heavy Ions environments.

II. Test Plan

This work will use an CL design already validated by the authors based on the use of the p-type cascode [9]. Representative components from different manufacturers have been selected to implement class 2 (1 pMOS & 1JFET) and class 10 (1pMOS & 3JFETs in parallel) CLs for 100V buses. The components selected and their main electrical characteristics are shown in Table I.

TABLE I: SELECTED TRANSISTORS

Reference	Tipo	R_{DSon}	$\|V_{DSmax}\|$
UJN1205K	JFET SiC (Gen 1)	35 mΩ	1200 V
IJW120R100T1	JFET SiC (Gen 1)	100 mΩ	1200 V
UJ3N065025K3S	JFET SiC (Gen 3)	25 mΩ	650 V
UJ3N120035K3S	JFET SiC (Gen 3)	35 mΩ	1200 V
SQP50P03-07	pMOS Si	7 mΩ	30 V

There are 32 transistor groups, with which 16 class 2 CLs and 16 class 10 CLs are implemented. Within each class there are four groups for each reference and class (n=4). Each reference corresponds to each JFET family indicated in Table I. The driving transistor (SQP50P03-07) is the same reference for each group. In order to estimate the ageing process, the following parameters are proposed to be obtained, both for the complete cascode and for the JFETs and pMOS individually: Leakage currents vs Blocking voltage, Channel resistance vs Channel current and output curve.

a) Temperature and conduction test

The test was carried out in accordance with JESD22-A108 [18] and the deratings established by ECSS-Q-ST-30_11_0140135 [19] were applied. This consisted in a 1000h test at a junction temperature between 100 and 120°C and at 75% of the maximum current allowed by the whole device. They have been carried out at the Laboratory of Instrumentation and Industrial Electronics of the University of Valencia using a laboratory bench as in Fig. 2(a).

b) Electrical stress test

Under this test, the CL is tested, that is, the JFET limits the power supplied to a load through its operation in a linear region dissipating high power. A campaign of 1000 resistive overloads and a campaign of 1000 start-up of capacitive loads is proposed, both adjusted for the class of the CL used. For this purpose, prototypes have been developed as shown in Fig2(b).

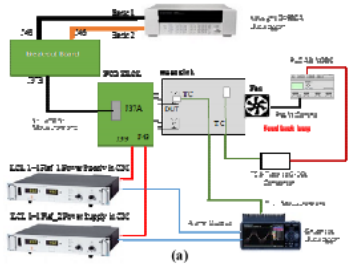
c) Radiation test

A TID radiation campaign is planned at the RADLAB, at the CNA laboratory in Seville, Spain. A Gammabeam X200 from Best Theratronics is available there. The conditions of the planned campaign are detailed in Table II.

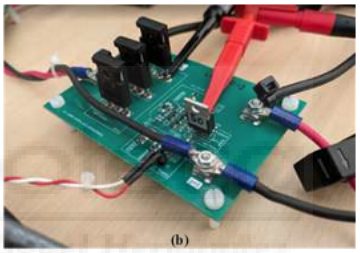
Finally, at the time of writing this abstract, a heavy ion irradiation campaign is also under negotiation. This information together with the final characteristics will be incorporated in the final paper.

TABLE II: TID CAMPAIGN PROPOSAL

Total Dose level	200 krad(Si).
Low Dose rate	220 rad(Si)/h.
Radiation steps	50, 100, 150, 200. Up to 200 krad(Si).
Samples	4 + 1 (reference sample) per part type
Annealing:	24h @RT, 168h @ 100°C



(a)



(b)

Fig.2 (a) Diagram of the test bench for temperature and conduction tests. (b) CL board developed and used for electrical stress tests.

III. Experimental Results

a) Temperature and conduction test

The result obtained from the test bench in Fig.2(a) reveals that the selected components are able to operate during this period without relevant voltage drop variations. The standard deviation in temperature regulation is less than 3% for the whole test, while the standard deviation in voltage drop variation is less than 1%, which demonstrates the consistency of the test and the robustness of the devices under high conduction and temperature conditions.

The differences observed between components of the same reference may be due to different manufacturing batches.

Electrical stress test

The overload test results reveal minimal variation in the parameters measured (peak current, energy dissipated by JFET, energy dissipated by pMOS, maximum blocking voltage). The same applies to class 2 and class 10 CLs as well as to the resistive overload and capacitive start-up tests. The data show that the components are adequately robust to withstand at least a thousand repetitions of each type of overload without significant changes in their behaviour.

Fig 3 shows, as representative cases, the maximum voltages withstood by the pMOS and the energies dissipated by the JFETs of two references in class 2 configuration for the resistive overload tests. Fig 8 shows the results for class 10 configuration.

It can be seen how, in general, there is a phase of about 200 repetitions where all the parameters represented have a transitory phase until it stabilizes at its nominal value. This is due to a thermal stabilization phase of the test.

b) Component characterization

The transistor ageing analysis was performed by characterising all the transistors. For this purpose, the B1506A - Keysight tracer has been used with a custom-developed fixture to characterise the P-type Cascode. In addition, all JFETs and pMOS have been characterised independently.

As a representative example, Fig 5 shows the results of the leakage currents, channel resistance and output curve of the P-type Cascode implemented with the IJW120R100T1 and UJ3N065025K3S transistors used in a class 2 CLs. The initial measurement, the post-test measurement of temperature and current and the post-test measurement of resistive overload have been superimposed. The equivalent data for the class 10 CLs are shown in Figure 10.

The results clearly show that there is no relevant degradation in any of the analysed parameters after the tests.

On the other hand, it is remarkable the excellent channel resistances achieved with the class 10 CL configuration, staying below 22mΩ for the UJ3N065025K3S above 40A.

c) Radiation

The TID campaign specified in table II was carried out at 75% of the maximum current supported by the transistors. During the campaign, the voltages on both the pMOS and the JFETS of the different cascodes, class 2 and class 10, were measured.

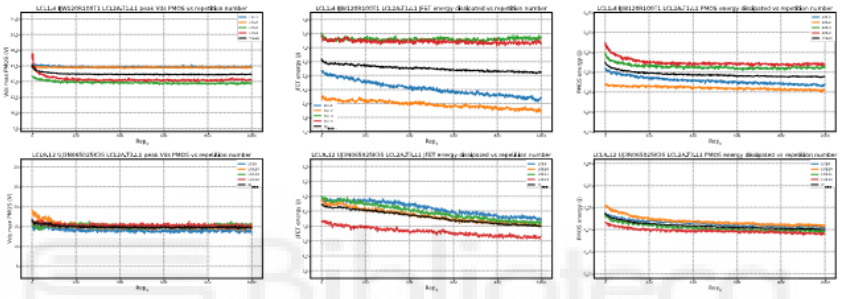


Fig.3 Resistive overload test results on IJW120R100T1 (top row) and UJ3N065025K3S (bottom row) transistors configured as class 2 CLs. Maximum voltage across the pMOS (left column), Power dissipated in the JFET (middle column) and power dissipated in the pMOS (right column) are shown for each of the 1000 repetitions and each of the 4 units tested.

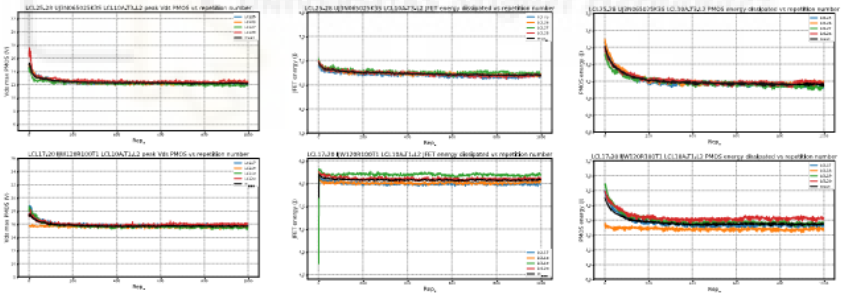


Fig.4 Resistive overload test results on the IJW120R100T1 (top row) and UJ3N065025K3S (bottom row) transistors configured as class 10 CL. Maximum voltage across the pMOS (left column), dissipated energy in the sum of the three JFETs (middle column) and dissipated energy in the pMOS (right column) are shown for each of the 1000 repetitions and each of the 4 units tested.

In the course of the campaign, slight variations in component voltage drops have been observed, without in any case compromising the functionality of the device. The different voltage drops can be indicative of variations in component temperatures.

It is important to note that all the devices evaluated withstood the radiation campaign without suffering any catastrophic damage.

IV. Conclusions

In this work, a test procedure has been proposed and carried out to consider the P-type cascode structure proposed by the authors suitable for use in space platform power distribution systems.

The results obtained confirmed that from the point of view of operating conditions, the proposed devices are capable of withstanding, under the design of a class 2 and class 10 CL, at least 1000 hours of operation under limiting temperature conditions. Moreover, experimental tests have corroborated that the CLs are able to withstand at least 1000 resistive overloads involving dissipations of more than 2J of energy per JFET individually (class 2) and more than 6J with three in parallel (class 10).

An exhaustive characterisation of all the components has been carried out between each test with the aim of understanding the degradation process and although small differences in the characterisations are observed, the devices maintain their initial characteristics practically intact, demonstrating their robustness in this type of situation.

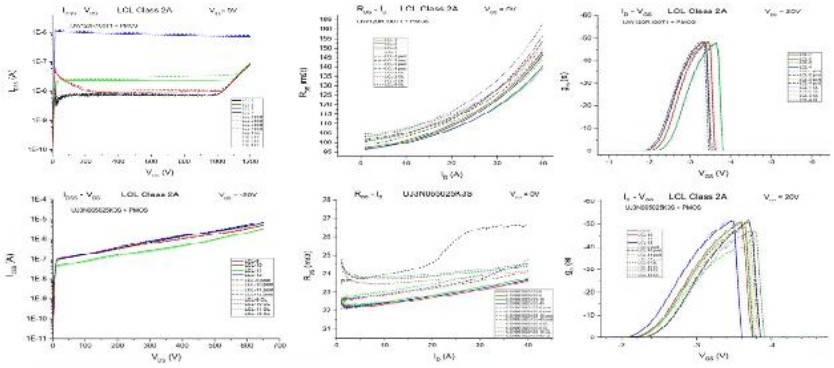


Fig.5 Results IJWW120R100T1 (top row) and UJ3N065025K3S (bottom row) transistors configured as class 2 CL. The leakage currents for the whole p-type cascode (left column), channel resistance for the whole p-type cascode (middle column) and cascode output curve (right column) are shown in their initial characterisation (solid line), after the current in temperature test (dashed line) and after the resistive overload test (dotted line), for each of the 4 units tested..

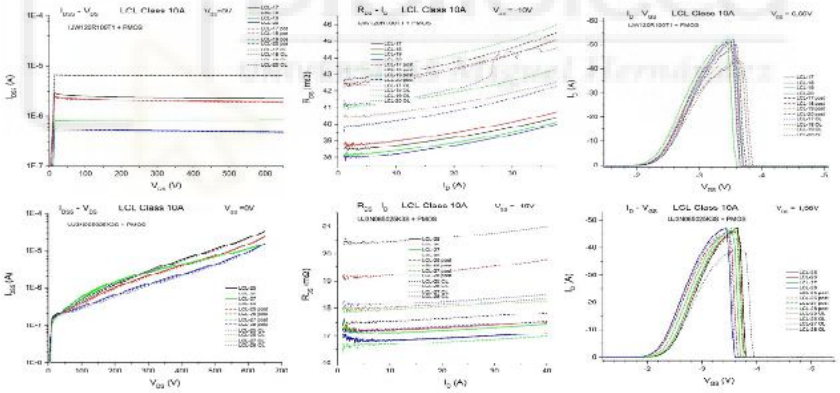


Fig.6 Results IJWW120R100T1 (top row) and UJ3N065025K3S (bottom row) transistors configured as class 10 CL. The leakage currents for the whole p-type cascode (left column), channel resistance for the whole p-type cascode (middle column) and cascode output curve (right column) are shown in their initial characterisation (solid line), after the current in temperature test (dashed line) and after the resistive overload test (dotted line), for each of the 4 units tested.

ACKNOWLEDGMENT

This work is being funded by the project TED2021-129562B-C31, under MCIN/AEI /10.13039/501100011033 and NextGenerationEU/ PRTR.

REFERENCES

[1] European Coorporation for Space Standarization, 'ECSS-E-ST-20-20C - Electrical design and interface requirements for power supply', ESA Requirements and Standards Division, ECSS-E-ST-20-20C, 2016.

[2] European Coorporation for Space Standarization, 'ECSS-E-ST-20C - Electrical and electronic', ESA Requirements and Standards Division, Rev.1, Feb. 2019.

[3] C. Orts, A. Garrigós, D. Marroquí, and A. Franke, 'Sequential switching shunt regulation using DC transformers for solar array power processing in high voltage satellites', IEEE Trans. Aerosp. Electron. Syst., pp. 1–11, 2023, doi: 10.1109/TAES.2023.3325314.

[4] J. D. Boissieu et al., 'High voltage electrical power system architecture optimized for electrical propulsion & high power payload', in European Space Power Conference 2019, 2019, pp. 1–7.

[5] M. Fu, D. Zhang, and T. Li, 'New Electrical Power Supply System for All-Electric Propulsion Spacecraft', IEEE Trans. Aerosp. Electron. Syst., vol. 53, no. 5, pp. 2157–2166, Oct. 2017, doi: 10.1109/TAES.2017.2683638.

[6] D. Marroqui, J. M. Blanes, A. Garrigos, and R. Gutierrez, 'Self-Powered 380 V DC SiC Solid-State Circuit Breaker and Fault Current Limiter', IEEE Trans. Power Electron., vol. 34, no. 10, pp. 9600–9608, Oct. 2019, doi: 10.1109/TPEL.2019.2893104.

[7] D. Marroqui, A. Garrigós, and J. M. Blanes, 'LVDC SiC MOSFET Analog Electronic Fuse With Self-Adjusting Tripping Time Depending on Overcurrent Condition', IEEE Trans. Ind. Electron., vol. 69, no. 8, pp. 8472–8480, Aug. 2022, doi: 10.1109/TIE.2021.3104606.

[8] D. Marroquí et al., 'Towards Higher Current and Voltage LCLs', in 2023 13th European Space Power Conference (ESPC), Oct. 2023, pp. 1–6. doi: 10.1109/ESPC59009.2023.10298130.

[9] A. Garrigós, D. Marroquí, C. Orts, C. Torres, and J. M. Blanes, 'Latching Current Limiter for Space Platform Power Distribution Using a Low-Voltage p-MOSFET and a Normally-ON SiC JFET', IEEE J. Emerg. Sel. Top. Power Electron., vol. 10, no. 5, pp. 5464–5473, Oct. 2022, doi: 10.1109/JESTPE.2022.3165430.

[10] A. Garrigós, D. Marroquí, J. M. Blanes, C. Torres, C. Orts, and P. Casado, 'SiC JFET/P-MOSFET cascode for SSCB and inrush current limiter in 300V DC power systems', in 2023 IEEE 32nd International Symposium on Industrial Electronics (ISIE), Jun. 2023, pp. 1–6. doi: 10.1109/ISIE51358.2023.10228112.

[11] L. F. S. Alves et al., 'SIC power devices in power electronics: An overview', in 2017 Brazilian Power Electronics Conference (COBEP), IEEE, Nov. 2017, pp. 1–8. doi: 10.1109/COBEP.2017.8257396.

[12] H. Lee, V. Smet, and R. Tummala, 'A Review of SiC Power Module Packaging Technologies: Challenges, Advances, and Emerging Issues', IEEE J. Emerg. Sel. Top. Power Electron., pp. 1–1, 2019, doi: 10.1109/JESTPE.2019.2951801.

[13] J. Homberger, A. B. Lostetter, K. J. Olejniczak, T. McNutt, S. M. Lal, and A. Mantooth, 'Silicon-carbide (SiC) semiconductor power electronics for extreme high-temperature environments', in 2004 IEEE Aerospace Conference Proceedings (IEEE Cat. No.04TH8720), IEEE, 2004, pp. 2538–2555. doi: 10.1109/AERO.2004.1368048.

[14] P. Godignon et al., 'SiC Power Switches Evaluation for Space Applications Requirements', Mater. Sci. Forum, vol. 858, pp. 852–855, 2016, doi: 10.4028/www.scientific.net/MSF.858.852.

[15] M. Steffens, 'TN5.12 - Survey of Total Ionising Dose Tolerance of Power Bipolar Transistors and Silicon Carbide Devices for JUICE', TN5.12 TID Test Report for SiC JFET IJW120R100T1, 2018.

[16] M. Steffens, 'TN6.6 - Survey of Total Ionising Dose Tolerance of Power Bipolar Transistors and Silicon Carbide Devices for JUICE', TN6.6 SEE Test Report for SiC JFET IJW120R100T1, 2018.

[17] Y. Qi, M. Antoniou, G. W. Clarke Baker, A. B. Renz, L. Zhang, and P. M. Gammon, 'Optimization of SiC device topologies for Single Event Immunity', in 2022 IEEE Workshop on Wide Bandgap Power Devices and Applications in Europe (WiPDA Europe), Sep. 2022, pp. 1–4. doi: 10.1109/WiPDAEurope55971.2022.9936145.

[18] 'Temperature, Bias, And Operating Life', Global Standards for the Microelectronics Industry, JESD22-A108, Nov. 2022.

[19] European Coorporation for Space Standarization, 'ECSS-Q-ST-30-11C - Derating - EEE components', ESA Requirements and Standards Division, rev 1, 2011

Evaluation of SiC JFET/Si-pMOS switch under high temperature operating life test and repetitive current limitation conditions for space applications

A. Garrigós
SPES research group
Miguel Hernandez University of Elche
Elche, Spain
augarsir@umh.es

D. Marroquí
SPES research group
Miguel Hernandez University of Elche
Elche, Spain
dmarroqui@umh.es

E. Maset
LIEI laboratory
University of Valencia
Burjassot, Spain
enrique.maset@uv.es

D. Alcaraz
SPES research group
Miguel Hernandez University of Elche
Elche, Spain
d.alcaraz@umh.es

G. Terol
LIEI laboratory
University of Valencia
Burjassot, Spain
guillermo.terol@uv.es

J. M. Blanes
SPES research group
Miguel Hernandez University of Elche
Elche, Spain
jmblanes@umh.es

IECON 2024 - 50th Annual Conference of the IEEE Industrial Electronics Society | 978-1-6654-6454-3/24/$31.00 ©2024 IEEE | DOI: 10.1109/IECON55916.2024.10905788

***Abstract*—This work evaluates the performance of a composite switch (SiC JFETs - Si pMOS) under high temperature and high electrical stress conditions for use as overcurrent protection device in DC distribution systems. Four different SiC JFET references have been analyzed. Experimental results demonstrate robustness in all conditions.**

Keywords—SiC JFET, SSCB, current limiter, p-channel MOSFET, DC distribution

I. Introduction

Interest in DC distribution has grown rapidly in recent years [1]. The use of native DC sources, DC links of power electronics converters and the increasing number of DC loads are the main reasons for this interest. Solid-state circuit breakers (SSCBs) and current limiters (CLs) for DC distribution are commonly preferred over electromechanical switches and the SiC JFET has been considered a good candidate for such applications. SiC JFETs offer very low on-resistance and high blocking voltage capability. However, being a normally-on device, SiC JFET requires the cascode configuration to become a normally-off device. A variant of the common cascode intended for satellite power systems has been introduced in [2-4], in which the low-voltage N-MOSFET is replaced by a low-voltage P-MOSFET, see Figure 1. The main benefit of this concept is that simple gate driving circuit referred to the input voltage permits self-powering and avoids the use of high-side drivers. Definition of solid-state current protection devices for European satellites are detailed in [5] and the guidelines for practical application in [6].

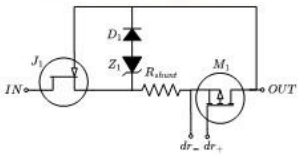


Fig. 1. SiC JFET/Si-pMOS cascode, as proposed by the authors in [2]. Rshunt is used for current measurement purposes.

This work is being funded by the project TED2021- 129562B-C31, under MCIN/AEI /10.13039/501100011033 and NextGenerationEU/PRTR.

Following previous work [2-4], this paper deals with endurance tests: a) High Temperature Operating Life (HTOL) test in conduction mode; b) Repetitive overload test; and c) Repetitive inrush current test. Four different SiC JFET references have been selected and tested, please refer to Figure 2. Characterization of power devices have been done in the initial and intermediate steps to observe any potential degradation. Figure 3 shows the diagram of endurance tests performed so far.

	Manuf. Reference (family)	Manuf.	I_D (@100ºC)	V_{DS}
J1	IJW120R100T1	Infineon	10A	1200V
J2	UJN1205K	Qorvo	23A	1200V
J3	UJ3N120035K3S	Qorvo	46A	1200V
J4	UJ3N065025K3S	Qorvo	62A	650V
P	SQP50P0307GE3	Vishay	50A	30V

Fig. 2. SiC JFET/Si-pMOS references used.

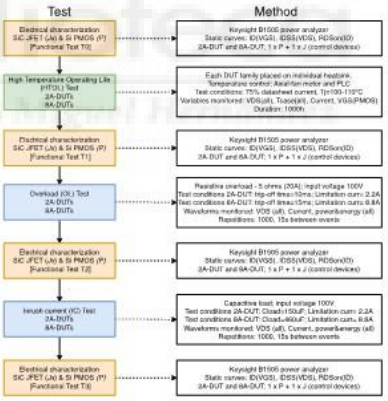


Fig. 3. SiC JFET/Si-pMOS endurance test plan.

II. Part Reference Selection and Latching Current Limiter (LCL) Implementation

Four SiC JFET references from three different device generations have been considered to cover the evolution of the technology. Infineon IJW120R100T1 (now obsolete) belongs to the the first generation of SiC JFETs. UJN1205K (now obsolete) belongs to the second generation of UnitedSiC (now Qorvo) devices. UJ3N120035K3S belongs to the third generation of Qorvo devices. These three devices are 1200V rated and they have doubled the current capacity in each generation. The latest reference, Qorvo's UJ3N065025K3S, is a 650V device but with lower on-resistance. The Si-pMOS device, SQP50P0307GE3, is a 30V automotive (high temperature operation) device with electrical characteristics similar to current space-qualified devices.

Two types of composite devices have been tested, one version employs one SiC JFET and one p-channel MOSFET, the second version includes three paralleled SiC JFETs and one p-channel MOSFET. Considering their uses in satellite applications, these composite devices have been applied for Latching Current Limiters [5] (a kind of SSCB with current limitation), see Figure 4. The single SiC JFET version has been configured as class-2 LCL and the three paralleled SiC JFET version as class-10 LCL [5]. In both cases, the input bus voltage is 100V. The detailed circuit description and operation of the proposed LCL can be found in [2]. The number of DUTs is summarized in Figure 5.

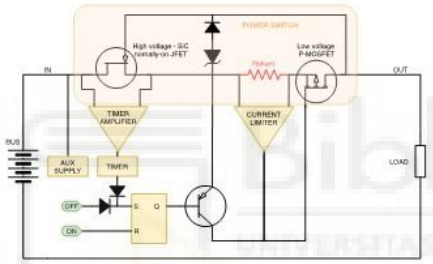


Fig. 4. Block diagram of the LCL using SiC JFET/Si-pMOS composite switch.

DUT type	Nb of DUTS	Nb of PMOS	Nb of JFET
Control devices	-	1	1
Class 2 LCL	4	4	4
Class 10 LCL	4	4	12
TOTAL		9	17

Fig. 5. Number of DUTs and power devices (per reference)

III. Electrical Characterization of Power Devices

Static electrical characterization of the power devices (functional test) has been performed at room temperature during different points of the test campaign to observe any degradation in the main parameters, please refer to Figure 3.

A Keysight B1505 power analyzer has been used to measure:

- Input characteristic (Drain current vs Gate-Source voltage): to monitor the evolution of the transconductance.
- Output characteristic (Drain current vs Drain-Source voltage): to monitor the evolution of the on-resistance.
- Leakage current (Drain leakage current vs Drain-Source voltage).

All power devices have been individually measured according to the limits established in their datasheets.

IV. High Temperature Operating Life (HTOL): Setup

HTOL, defined in JESD22-A108, is an accelerated test used to determine the reliability of a device under high temperature and bias conditions over time. In this case, the main requirements of the HTOL test are: 1000h, controlled junction temperature between 100ºC-110ºC, 75% of nominal current specified in the datasheet. The drain-source voltage (V_{DS}), drain current (I_D) and, case temperature (T_C) of both JFET and PMOS were monitored. In addition, the input-output voltage (V_{LCL}) and supply current (I_{PS}) were also recorded. PMOS driving signal for each composite device is obtained from a breakout board that includes independent, isolated and adjustable power supplies that are also monitored. Temperature control of the power devices is achieved using independent heatsinks for each JFET family with axial-fan motors controlled by an industrial PLC, see Figure 6 for the HTOL test block diagram and Figure 7 for the experimental setup.

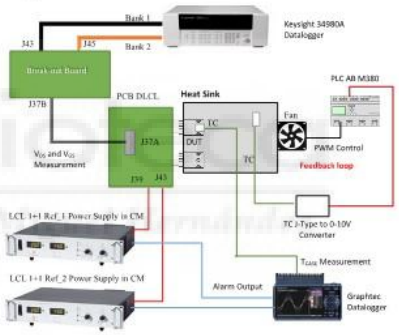


Fig. 6. HTOL test block diagram

V. Overload (OL) and Inrush Current (IC) Tests: Setup

The OL and IC tests share the same experimental setup, with differences in load configuration and command signal patterns. Specific test PCBs have been implemented with complete LCL circuitry, see Figure 8. The test PCBs are reconfigurable (minor changes in resistor values) for class-2 and class-10 LCLs. The ON and OFF control signals, please refer to Figure 4, are obtained from an FPGA control board with isolated outputs.

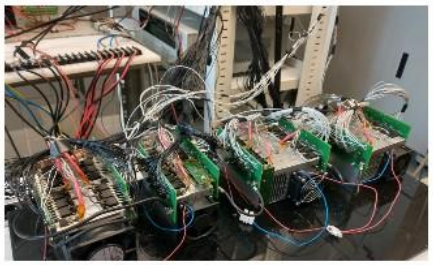


Fig. 7. HTOL experimental setup

The OL and IC experimental setup, please see Figure 9, consists of a power supply feeding an input capacitor bank used to provide inrush and overload current. The DUT waveforms are captured with a mixed-signal oscilloscope, whose triggering is controlled by an external FPGA board (1000 repetitions with 15 seconds between events). The captured waveforms are stored on the control PC for off-line analysis. An IGBT in series with a 5Ω resistor is used for overload tests and a 660µF capacitor is used for inrush current test.

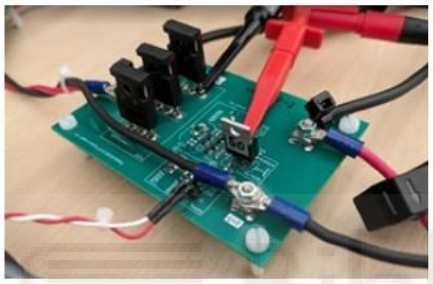


Fig. 8. Test LCL board for overload and inrush current tests

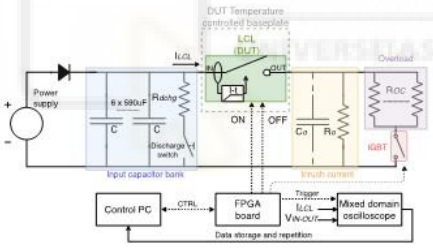


Fig. 9. OL and IC tests block diagram

The off-line analysis allows the computation of the most relevant parameters of the captured waveforms, peak voltage and current, time response, limitation time and dissipated energy for all power semiconductors.

VI. Experimental results and discussion

After the test campaign, the results indicate that the composite device maintains electrical characteristics within datasheet specifications in all families, and with low scatter after each test. Figures 10 and 11 show the electrical characterization for the lowest R_{DSon} device (UJ3N065025K3S) in class-10 LCL configuration. It can be seen that the total on-resistance (3 paralleled SiC JFET in series with Si-PMOS) remains below 21mΩ after all tests, demonstrating very good performance for conduction with minimal losses in LCL applications. It is also important to note that the total leakage current remains below 50uA (@650V) after all tests, providing very good isolation in case the LCL is in the off state.

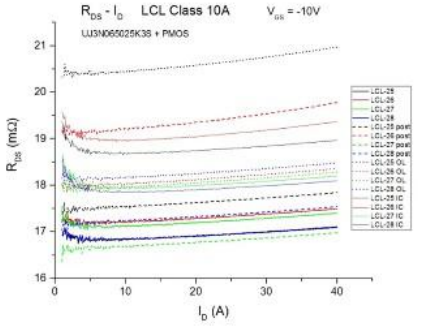


Fig. 10. Class-10 LCL (3 paralleled SiC JFET UJ3N065025K3S and series Si-pMOS SQP50P0307GE3) total on-resistance after HTOL, OL and IC tests.

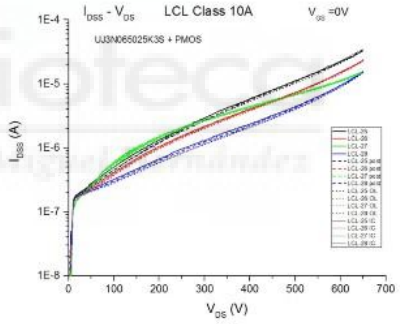


Fig. 11. Class-10 LCL (3 paralleled SiC JFET UJ3N065025K3S and Si-pMOS SQP50P0307GE3) total leakage current after HTOL, OL and IC tests.

The overload and inrush current tests also demonstrate a high level of repeatability across all families. Experimental results after inrush current tests are shown for the same composite device, class-10 LCL (UJ3N065025K3S) in Figures 12, 13, 14, 15 and 16.

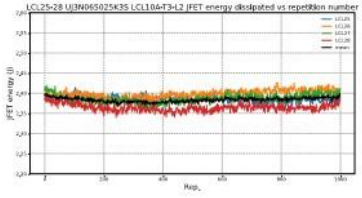


Fig. 12. Class-10 LCL (3 paralleled SiC JFET UJ3N065025K3S and Si-pMOS SQP50P0307GE3) total JFET energy dissipation vs. repetition number during IC test (each SiC JFET dissipates one third of the energy shown in the graph, 0.8J approximately).

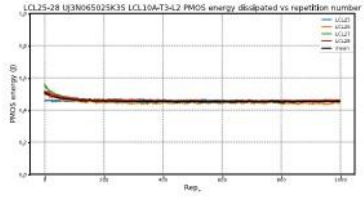


Fig. 13. Class-10 LCL (3 paralleled SiC JFET UJ3N065025K3S and Si-pMOS SQP50P0307GE3) PMOS energy dissipation vs. repetition number during IC test.

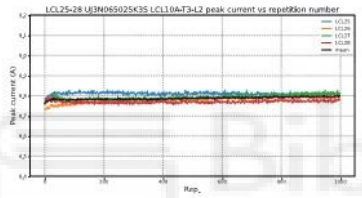


Fig. 14. Class-10 LCL (3 paralleled SiC JFET UJ3N065025K3S and Si-pMOS SQP50P0307GE3) Current limitation vs. repetition number during IC test.

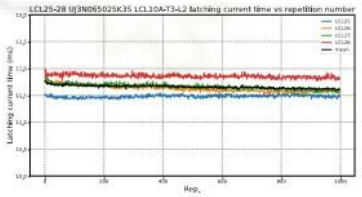


Fig. 15. Class-10 LCL (3 paralleled SiC JFET UJ3N065025K3S and Si-pMOS SQP50P0307GE3) trip-off time vs. repetition number during IC test.

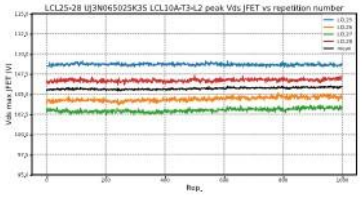


Fig. 16. Class-10 LCL (3 paralleled SiC JFET UJ3N065025K3S and Si-pMOS SQP50P0307GE3) peak drain-source voltage SiC JFET vs. repetition number during IC test (measured three SiC JFET in parallel).

The energy dissipation of the SiC JFET and Si-pMOS during current limitation (inrush current), Figures 12 and 13, remains stable after 1000 repetitions in the last test (HTOL and OL performed previously), showing that the LCL behavior is robust. The current limitation performance is also remarkable, see Figure 14, and is within specifications stated in [5]. As can be seen in Figure 16, the overvoltage of the SiC JFET remains low, demonstrating that capacitive load switching remains smooth in the aged parts.

Since the OL test is more stressful in terms of overvoltage and overcurrent (transient before entering in current limitation mode), Figures 17 and 18 show the results for the same device, class-10 LCL (UJ3N065025K3S), during overload tests. It is clearly seen that the overvoltage and overcurrent are well below the rated voltage and do not require additional devices, i.e. energy-absorbing circuits in parallel.

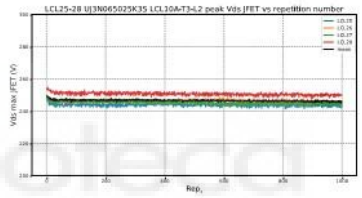


Fig. 17. Class-10 LCL (3 paralleled SiC JFET UJ3N065025K3S and Si-pMOS SQP50P0307GE3) peak drain-source voltage SiC JFET vs. repetition number during OL test (measured three SiC JFET in parallel).

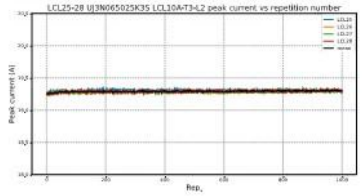


Fig. 18. Class-10 LCL (3 paralleled SiC JFET UJ3N065025K3S and Si-pMOS SQP50P0307GE3) peak LCL current vs. repetition number during OL test.

VII. Conclusions

After the endurance test campaign, the SiC JFET/Si-pMOS composite device shows good performance in all tested devices. The electrical parameters are within datasheet specifications and the repeatability of the results on the LCL circuit parameters is demonstrated. The next step is to perform a repetitive short circuit campaign. Finally, to support this technology in space applications, radiation tests (Total Ionizing Dose and Single Event Effects) will be conducted. So far, experimental results reveal that SiC JFET/Si p-MOS is a good candidate for the next generation of solid-state protection devices in high-performance DC distribution systems for space applications and a natural replacement for existing Si-pMOS LCLs used in low-voltage satellite platforms.

References

[1] R. Rodrigues, Y. Du, A. Antoniazzi, P. Cairoli, "A review of Solid-State Circuit Breakers," IEEE Transactions on Power Electronics, Vol 36, No 1, Jan 2021, pp. 364-377.

[2] A. Garrigós, D. Marroquí, C. Orts, C. Torres, and J. M. Blanes, 'Latching Current Limiter for Space Platform Power Distribution Using a Low-Voltage p-MOSFET and a Normally-ON SiC JFET', *IEEE J. Emerg. Sel. Top. Power Electron.*, vol. 10, no. 5, pp. 5464–5473, Oct. 2022, doi: 10.1109/JESTPE.2022.3165430.

[3] A. Garrigós *et al.*, 'SiC JFET/P-MOSFET cascode for SSCB and inrush current limiter in 300V DC power systems', in *2023 IEEE 32nd International Symposium on Industrial Electronics (ISIE)*, Jun. 2023, pp. 1–6. doi: 10.1109/ISIE51358.2023.10228112.

[4] D. Marroquí *et al.*, 'Towards Higher Current and Voltage LCLs', in *2023 13th European Space Power Conference (ESPC)*, Oct. 2023, pp. 1–6. doi: 10.1109/ESPC59009.2023.10298130.

[5] *Space Engineering—Electrical Design Interface Requirements for Power Supply*, Standard ECSS-E-ST-20-20C, European Cooperation for Space Standardization, ESA-ESTEC, Requirements & Standards Division, Noordwijk, The Netherlands, Apr. 2016, pp. 1–61.

[6] *Space Engineering—Guidelines for Electrical Design Interface Requirements for Power Supply*, Handbook ECSS-E-HB-20-20A, European Cooperation for Space Standardization, ESA-ESTEC, Requirements & Standards Division, Noordwijk, The Netherlands, Apr. 2016, pp. 1–78.

ANEXO 6: CÓDIGO PARA LA HERRAMIENTA DE VISUALIZACIÓN DE DATOS

Funciones de análisis de eventos de sobrecarga

Estas funciones se encuentran contenidas en la clase “Procesador_Analítico” del Anexo 8.

Visualización de datos procesados

```
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import pandas as pd
import os

'''

Estas funciones se encarga de generar gráficas determinada a partir de
datos guardados en diccionarios jerarquizados con la siguiente forma:

    Primera Capa LCL -> LCL_2A | LCL_10A ; Nos hace referencia a los
dispositivos de clase 2 o clase 10
        Capa LCL_2A -> JFET | PMOS ; Donde se indica si vamos con las
familias de JFET o de PMOS
                Capa IJW120R100T1 -> IJW120R100T1 | UJN1205K |
UJ3N065025K3S | UJ3N120035K3S; AquÃ- estarÃ¡n las referencias
concretas de la familia de clase 2A, p. ej. IJW120R100T1-10
                    Capa (referencia concreta) -> Short | Inrush | PS
; Aquí encontramos las distintas pruebas que se realizaron
                        capa (prueba concreta) -> Aquí encontramos el
dataframe de datos y la información de cómo se obtuvieron
                            capa data ->Dataframe que contiene los
puntos de las curvas (generalmente son importantes las dos primeras
columnas)
                            capa info ->Cadena de caracteres
referentes a la información de la caracterización

        Capa LCL_10A -> Totalmente similar, donde se encuentran las
familias de referencias que serÃ¡n de clase 10


'''

'''Funciones preliminares'''
"""
    Conecta un evento de scroll de ratón que hace zoom solo en el eje
y para el objeto Axes dado.

    Parámetros:
      ax: objeto Axes de Matplotlib donde se aplicará el zoom.
      scale_factor_up: Factor para acercar (zoom in) al hacer scroll
hacia arriba.
      scale_factor_down: Factor para alejar (zoom out) al hacer scroll
hacia abajo.
"""

"""
    Añade funcionalidad de zoom interactivo en el eje Y de una gráfica
matplotlib
    mediante la rueda del ratón.

    Parámetros:
        ax (matplotlib.axes.Axes): Objeto Axes donde se aplicará el
zoom en el eje Y.
        scale_factor_up (float, opcional): Factor para acercar (zoom
in) al hacer scroll hacia arriba. Default 0.9.
        scale_factor_down (float, opcional): Factor para alejar (zoom
out) al hacer scroll hacia abajo. Default 1.1.
```

```
    Descripción:
        La función conecta un evento de scroll que ajusta
dinámicamente los límites del eje Y
        para acercar o alejar la vista, centrando el zoom en la
posición vertical del cursor del ratón.
"""
def add_y_zoom(ax, scale_factor_up=0.9, scale_factor_down=1.1):

    def on_scroll(event):
        # Ignora eventos fuera del eje especificado
        if event.inaxes != ax:
            return

        ymin, ymax = ax.get_ylim()
        ydata = event.ydata
        if ydata is None:
            return

        if event.button == 'up':  # Zoom in
            scale = scale_factor_up
        elif event.button == 'down':  # Zoom out
            scale = scale_factor_down
        else:
            scale = 1  # Sin cambio si otro botón

        new_ymin = ydata - (ydata - ymin) * scale
        new_ymax = ydata + (ymax - ydata) * scale

        ax.set_ylim(new_ymin, new_ymax)
        ax.figure.canvas.draw_idle()

    ax.figure.canvas.mpl_connect('scroll_event', on_scroll)


"""
    Extrae listas de nombres y datos correspondientes a una clase,
familia y prueba específica
    desde un diccionario jerarquizado, excluyendo dispositivos
indicados en una lista de bloqueo.
    Opcionalmente calcula la longitud mínima común para análisis
promedio.

    Parámetros:
        clase (str): Clase del dispositivo a analizar.
        familia (str): Familia dentro de la clase.
        prueba (str): Nombre de la prueba a extraer.
        dato (str): Nombre de la columna o dato en el DataFrame.
        diccionario_datos (dict): Diccionario jerarquizado que
contiene los datos experimentales.
        individuos_ban_list (list o None): Lista de dispositivos a
excluir. Si None, no se excluye ninguno.
        media (bool, opcional): Si es True, calcula la longitud mínima
común para análisis estadístico. Default False.

    Retorna:
        Si media es False:
            - nombre_individuos_sub (list): Lista de dispositivos
incluidos.
            - datos_individuos_sub (list): Lista con datos extraídos
para cada dispositivo.
```

```
        Si media es True:
            - nombre_individuos_sub (list)
            - datos_individuos_sub (list)
            - longitudes_sub (list): Longitudes de los datos por
dispositivo.
            - punto_final_media (int): Longitud mínima común para
análisis.
"""
def lista_df_longitud(clase, familia, prueba, dato, diccionario_datos,
individuos_ban_list, media=False):


    nombre_individuos_sub = []
    datos_individuos_sub = []

    if individuos_ban_list is None:
        individuos_ban_list = []

    try:
        print(list(diccionario_datos[clase][familia].keys()))
        for individuos_dataframe in
list(diccionario_datos[clase][familia].keys()):
            if individuos_dataframe not in individuos_ban_list:
                nombre_individuos_sub.append(individuos_dataframe)

        for nombre in nombre_individuos_sub:

datos_individuos_sub.append(diccionario_datos[clase][familia][nombre][
prueba]["data"][dato])

        print(nombre_individuos_sub)
        print(datos_individuos_sub)

    except Exception as e:
        print("Ocurrió un error: " + str(e))
        return

    if media:
        try:
            longitudes_sub = [datos.shape[0] for datos in
datos_individuos_sub]
            punto_final_media = np.min(longitudes_sub)
            print("Para el análisis de la media, se calculará hasta el
punto " + str(punto_final_media))

        except:
            print("Hubo un problema a la hora de detectar el número
mínimo de datos")
            return

        return nombre_individuos_sub, datos_individuos_sub,
longitudes_sub, punto_final_media

    return nombre_individuos_sub, datos_individuos_sub


"""
    Obtiene las etiquetas del eje Y y el título para un gráfico, según
el dato a representar.

    Parámetros:
```

```
        dato (str): Identificador del dato a graficar.

    Retorna:
        tuple:
            - nombre_ejey (str): Etiqueta del eje Y.
            - final_titulo (str): Título descriptivo para la gráfica.

    En caso de dato no reconocido, retorna (None, None) y muestra
mensaje de error.
"""
def parametros_grafico(dato):

    datos_mapeo = {
        "EnJFET":      ('JFET energy (J)', 'JFET energy dissipated'),
        "IMax":        ('Peak current (A)', 'peak current'),
        "Ilim_mean":   ('Latching current mean (A)', 'latching current
time'),
        "VdsMaxJFET":  ('Vds max JFET (V)', 'peak Vds JFET'),
        "VdsMaxPMOS":  ('Vds max JPMOS (V)', 'peak Vds JPMOS'),
        "EnPMOS":      ('PMOS energy (J)', 'PMOS energy dissipated'),
        "TripOffTime": ('Latching current time (ms)', 'latching
current time'),
    }

    resultado_dato = datos_mapeo.get(dato)

    if resultado_dato:
        nombre_ejey, final_titulo = resultado_dato
    else:
        print("Dato incorrecto")
        return None, None

    return nombre_ejey, final_titulo

"""
    Genera gráficos de estadísticas crudas para grupos de LCL,
combinando datos de varias clases,
    con opción de excluir ciertos individuos y guardar la gráfica
resultante.

    Parámetros:
        prueba (str): Nombre de la prueba a analizar.
        familia (str): Familia dentro de la clase para filtrar datos.
        clases (list): Lista de clases cuyos datos se quieren
graficar.
        dato (str): Nombre del dato específico a graficar (ej.
"TripOffTime", "IMax").
        diccionario_datos (dict): Diccionario jerarquizado con los
datos experimentales.
        punto_inicio (int, opcional): Índice desde donde iniciar el
gráfico (default=0). Si es inválido, se usará 0.
        media (bool, opcional): Si True, se calcula y grafica la media
entre individuos (default=True).
        individuos_ban_list (list o None, opcional): Lista de
individuos a excluir. Default None (no se excluye ninguno).
        direccion_guardado (str o None, opcional): Ruta donde guardar
la figura. Si None, no se guarda (default None).

    Descripción:
        - Configura etiquetas y títulos del gráfico según el dato a
mostrar.
```

```
        - Recolecta datos filtrando por clases y familia, ignorando
individuos en la ban list.
        - Aplica transformación especial para "TripOffTime"
(milisegundos).
        - Grafica datos individuales y, si se especifica, la media.
        - Añade zoom vertical interactivo con la rueda del ratón.
        - Muestra la figura y, si se indica ruta, la guarda en formato
SVG.
"""
def graficas_estadisticas(prueba, familia, clases, dato,
diccionario_datos, punto_inicio=0, media=True,
individuos_ban_list=None, direccion_guardado=None):


    # Configuración de etiquetas y título
    nombre_ejex = "Rep."
    nombre_ejey, final_titulo = parametros_grafico(dato)
    titulo = f"{'|'.join(clase for clase in clases)} {familia}
{final_titulo} vs repetition number"

    # Creación de figura y eje
    fig, ax = plt.subplots()
    plt.ioff()  # Desactivar modo interactivo para evitar dibujado
automático

    nombre_individuos = []
    datos_individuos = []
    longitudes = []

    for clase in clases:
        try:
            nombre_individuos_sub, datos_individuos_sub,
longitudes_sub, punto_final_media = lista_df_longitud(
                clase, familia, prueba, dato, diccionario_datos,
individuos_ban_list, True
            )
        except:
            return

        nombre_individuos.extend(nombre_individuos_sub)
        datos_individuos.extend(datos_individuos_sub)
        longitudes.extend(longitudes_sub)

        print("Nombre:")
        print(nombre_individuos_sub)

        # Validación del punto de inicio
        if punto_inicio < 0 or punto_inicio >= punto_final_media or
not isinstance(punto_inicio, int):
            punto_inicio = 0
            print("Punto de inicio no correcto, se selecciona 0")

        # Transformación especial para TripOffTime (ms)
        if dato == "TripOffTime":
            for i, serie in enumerate(datos_individuos_sub):
                datos_individuos_sub[i] = pd.to_numeric(serie,
errors='coerce') * 1000

        # Graficar datos individuales
        for indice_datos, datos in enumerate(datos_individuos_sub):
```

```
            ax.plot(datos[punto_inicio:],
label=nombre_individuos_sub[indice_datos])

    # Graficar media si corresponde
    if media:
        media_datos = [datos[0:punto_final_media] for datos in
datos_individuos]
        if dato == "TripOffTime":
            media_val = np.mean(media_datos, axis=0) * 1000
        else:
            media_val = np.mean(media_datos, axis=0)
        ax.plot(media_val, label='mean', color='black')

    # Configuración de etiquetas, título, leyenda y cuadrícula
    ax.set_xlabel(nombre_ejex, size=10)
    ax.set_ylabel(nombre_ejey, size=10)
    ax.set_title(titulo, size=10)
    ax.legend(loc='best')
    ax.grid(linestyle='--', linewidth=1)

    # Añadir zoom interactivo solo en eje Y
    add_y_zoom(ax)

    plt.show(block=True)

    # Guardar figura si se especifica ruta
    if direccion_guardado:
        if not os.path.exists(direccion_guardado):
            try:
                os.makedirs(direccion_guardado, exist_ok=True)
            except:
                print("Hubo un problema a la hora de crear el
directorio")
                return

        nombre_archivo = f"{'|'.join(clase for clase in
clases)}{familia}{final_titulo}.svg"
        ruta_guardado = os.path.join(direccion_guardado,
nombre_archivo)

        try:
            fig.savefig(ruta_guardado, format="svg", dpi=300)
            print(f"Gráfica {nombre_archivo} guardada con éxito")
        except:
            print("Error al guardar la gráfica")

"""
    Genera gráficos de violín a partir de una o dos listas de Series
de pandas extraídas de los datos,
    con opción a normalizar cada serie en una ventana definida y
guardar el gráfico.

    Parámetros:
        prueba (str): Nombre de la prueba para filtrar los datos.
        familia (str): Familia dentro de la clase para filtrar datos.
        clases (list): Lista con una o dos clases cuyos datos se
quieren graficar.
        dato (str): Nombre del dato específico a graficar.
        diccionario_datos (dict): Diccionario jerarquizado con los
datos experimentales.
```

```
        individuos_ban_list (list o None, opcional): Lista de
individuos a excluir (default None).
        normalizar (bool, opcional): Si True, normaliza cada serie
dividiéndola por la media de una ventana alrededor de 'punto' (default
False).
        punto (int o None, opcional): Índice central de la ventana
para normalizar (requerido si normalizar=True).
        rango (int o None, opcional): Tamaño de la ventana a cada lado
del punto para normalizar (requerido si normalizar=True).
        direccion_guardado (str o None, opcional): Ruta para guardar
la figura. Si None, no se guarda (default None).

    Descripción:
        - Soporta graficar violines para una o dos clases comparadas.
        - Normaliza las series dividiendo por la media en una ventana
si se solicita.
        - Valida que ambas clases tengan la misma cantidad de datos.
        - Aplica conversión a milisegundos para "TripOffTime".
        - Genera el gráfico con seaborn violinplot.
        - Guarda la figura en formato SVG si se indica ruta de
guardado.
"""
def graficas_violin(prueba, familia, clases, dato, diccionario_datos,
individuos_ban_list=None, normalizar=False, punto=None, rango=None,
direccion_guardado=None):


    # Validar normalización
    if normalizar and (punto is None or rango is None):
        print("Si se desea normalizar, debe especificar 'punto' y
'rango'.")
        return

    # Configurar etiquetas y título
    nombre_ejey, final_titulo = parametros_grafico(dato)
    titulo = f"{'|'.join(clase for clase in clases)} {familia}
{final_titulo} vs repetition number violinplot"

    if normalizar:
        nombre_ejey = r"$\frac{" + nombre_ejey + r"}{" + nombre_ejey +
r"_{\text{mean}\ ( " + str(punto - rango) + r"-" + str(punto + rango)
+ r" )}}$"

    # Obtener datos para cada clase
    if len(clases) == 1:
        clase = clases[0]
        try:
            nombre_individuos_sub1, lista_df1 =
lista_df_longitud(clase, familia, prueba, dato, diccionario_datos,
individuos_ban_list, False)
        except:
            return
        lista_df2 = None

    elif len(clases) == 2:
        clase = clases[0]
        try:
            nombre_individuos_sub1, lista_df1 =
lista_df_longitud(clase, familia, prueba, dato, diccionario_datos,
individuos_ban_list, False)
        except:
```

```
            return

        clase = clases[1]
        try:
            nombre_individuos_sub2, lista_df2 =
lista_df_longitud(clase, familia, prueba, dato, diccionario_datos,
individuos_ban_list, False)
        except:
            return

        if len(lista_df1) != len(lista_df2):
            print("No existe el mismo número de datos para ambas
clases")
            return
    else:
        print("Se han aportado más de dos clases")
        return

    # Convertir a milisegundos si es TripOffTime
    if dato == "TripOffTime":
        for i, serie in enumerate(lista_df1):
            lista_df1[i] = pd.to_numeric(serie, errors='coerce') *
1000
        if lista_df2 is not None:
            for i, serie in enumerate(lista_df2):
                lista_df2[i] = pd.to_numeric(serie, errors='coerce') *
1000

    # Función interna para normalizar una Serie
    def normalizar_serie(serie, punto, rango):
        inicio = max(0, punto - rango)
        fin = min(len(serie), punto + rango + 1)
        media_ref = serie.iloc[inicio:fin].mean()
        if media_ref == 0:
            raise ValueError("La media calculada es 0; no se puede
normalizar.")
        return serie / media_ref

    # Preparar datos para gráfico
    datos = []
    for i, serie in enumerate(lista_df1):
        if normalizar:
            serie = normalizar_serie(serie, punto, rango)
        df_temp = pd.DataFrame({'Valor': serie, 'Grupo':
f'{nombre_individuos_sub1[i]}'})

        if lista_df2 is not None:
            serie2 = lista_df2[i]
            if normalizar:
                serie2 = normalizar_serie(serie2, punto, rango)
            df_temp2 = pd.DataFrame({'Valor': serie2, 'Grupo':
f'{nombre_individuos_sub2[i]}'})
            df_temp['Comparación'] = clases[0]
            df_temp2['Comparación'] = clases[1]
            df_temp = pd.concat([df_temp, df_temp2])
        else:
            df_temp['Comparación'] = 'Datos'
        datos.append(df_temp)

    datos = pd.concat(datos, ignore_index=True)
```

```
    # Crear gráfico
    plt.figure(figsize=(10, 6))
    if lista_df2 is not None:
        sns.violinplot(x='Grupo', y='Valor', hue='Comparación',
data=datos, split=True)
    else:
        sns.violinplot(x='Grupo', y='Valor', data=datos)
    plt.title(titulo)
    plt.ylabel(nombre_ejey, fontsize=15)
    plt.tight_layout()
    plt.show()

    # Guardar figura si se indica
    if direccion_guardado:
        if not os.path.exists(direccion_guardado):
            try:
                os.makedirs(direccion_guardado, exist_ok=True)
            except:
                print("Hubo un problema a la hora de crear el
directorio")
                return

        nombre_archivo = f"{'|'.join(clase for clase in
clases)}{familia}{final_titulo}.svg"
        ruta_guardado = os.path.join(direccion_guardado,
nombre_archivo)

        try:
            plt.savefig(ruta_guardado, format="svg", dpi=300)
            print(f"Gráfica {nombre_archivo} guardada con éxito")
        except:
            print("Error al guardar la gráfica")


import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

"""
GRAFICAR GRAFICAS DE BIGOTES

Parámetros:
  - series_lists: lista de listas de pandas Series.
  - start: índice de inicio para tomar muestras (por defecto 0).
  - step: intervalo entre muestras.
  - use_range: si es True, en cada muestra se usa una ventana de datos
de tamaño 'window'
               (desde max(start, i-window+1) hasta i) para cada Series
en lugar del valor puntual.
  - window: tamaño de la ventana (si use_range es True).
  - ext_line_width: longitud horizontal para las líneas horizontales
en cada muestra.
  - ref_sample: si se especifica (entre start y la longitud mínima), a
partir de este punto se
               normalizan (dividiéndose entre el valor obtenido en
ese punto). Por ejemplo, si ref_sample
               es igual a start, el valor en ese punto pasará a ser
1.
En cada sublista válida se crea un subplot en el que:
  - Se muestrea cada 'step' puntos desde 'start'.
```

```
  - Para cada muestra se calcula la media, el máximo y el mínimo de
los valores (o promedios en ventana)
    de todas las Series.
  - Si se especifica ref_sample, para cada muestra con índice >=
ref_sample se normalizan los tres
    valores dividiéndolos entre el valor de referencia.
  - Se grafica la línea de la media (con marcadores).
  - En cada muestra se dibujan líneas verticales (dashed) desde la
media hasta el máximo (rojo)
    y desde el mínimo hasta la media (azul).
  - Además, en cada muestra se dibuja una línea horizontal corta en el
nivel del máximo y otra en el nivel del mínimo.
"""
def plot_series_from_lists(prueba, familia, clase, dato,
diccionario_datos,
                           familias_ban_list=None,
individuos_ban_list=None,
                           start=0, step=100, use_range=False,
window=1,
                           ext_line_width=5, ref_sample=None,
direccion_guardado=None):

    colores = ["red", "green", "blue", "orange"]

    # Configuración del nombre de los ejes y título según familia y
tipo de dato
    nombre_ejey, final_titulo = parametros_grafico(dato)
    titulo = f"{clase} class {final_titulo} vs repetition number"

    # Ajuste del nombre del eje y según parámetros de ventana y
normalización
    if use_range:
        if ref_sample is not None:
            nombre_ejey = (r"$\frac{"
                           r"\mathrm{" + nombre_ejey +
r"}_{\text{mean}\left( (i - " + str(window) + r") - i \right)}}{"
                           r"\mathrm{" + nombre_ejey +
r"}_{\text{mean}\left( " + str(ref_sample - window) + r" - " +
str(ref_sample) + r" \right)}}$")
        else:
            nombre_ejey = (r"$\mathrm{" + nombre_ejey +
r"}_{\text{mean}\left( (i - " + str(window) + r") - i \right)}$")
    else:
        if ref_sample is not None:
            nombre_ejey = (r"$\frac{"
                           r"\mathrm{" + nombre_ejey +
r"}_{\text{mean}\left( (i - " + str(window) + r") - i \right)}}{"
                           r"\mathrm{" + nombre_ejey +
r"}_{\text{mean}\left( " + str(ref_sample - window) + r" - " +
str(ref_sample) + r" \right)}}$")
        else:
            nombre_ejey = (r"$\mathrm{" + nombre_ejey +
r"}_{\text{mean}\left( (i - " + str(window) + r") - i \right)}$")

    # Obtención de las listas de Series para graficar
    series_lists = []
    if familias_ban_list is None:
        familias_ban_list = []

    nombres_familias = ["IJW120R100T1", "UJN1205K", "UJ3N065025K3S",
"UJ3N120035K3S"]
```

```
    for fam in nombres_familias:
        if fam not in familias_ban_list:
            try:
                nombre_individuos_sub1, series_lists_sub =
lista_df_longitud(clase, fam, prueba, dato, diccionario_datos,
individuos_ban_list, False)
                series_lists.append(series_lists_sub)
            except Exception as e:
                print(f"Error al obtener series para familia {fam}:
{e}")
                return

    # Filtrar listas con al menos 3 Series
    valid_series_lists = [lst for lst in series_lists if len(lst) >=
3]
    if not valid_series_lists:
        print("No hay listas válidas (cada lista debe tener al menos 3
Series)")
        return

    n_plots = len(valid_series_lists)

    fig, axes = plt.subplots(n_plots, 1, figsize=(5, 5 * n_plots),
sharex=True, sharey=True, constrained_layout=True)
    if n_plots == 1:
        axes = [axes]

    for iterador, (ax, series_list) in enumerate(zip(axes,
valid_series_lists)):
        min_len = min(len(s) for s in series_list)
        if start >= min_len:
            print("El punto de inicio es mayor o igual a la longitud
mínima de las Series.")
            continue

        sample_indices = list(range(start, min_len, step))
        if (min_len - 1) not in sample_indices:
            sample_indices.append(min_len - 1)

        global_means, global_maxs, global_mins = [], [], []

        # Cálculo de estadísticas para cada índice de muestra
        for i in sample_indices:
            values = []
            for s in series_list:
                if use_range:
                    rng_start = max(start, i - window + 1)
                    segment = s.iloc[rng_start : i + 1]
                    val = segment.mean()
                else:
                    val = s.iloc[i]
                values.append(val)
            values = np.array(values)
            global_means.append(np.mean(values))
            global_maxs.append(np.max(values))
            global_mins.append(np.min(values))

        # Normalización si ref_sample está definido
        if ref_sample is not None:
            ref_val = None
```

```
            for j, idx in enumerate(sample_indices):
                if idx >= ref_sample:
                    ref_val = global_means[j]
                    break
            if ref_val is None or ref_val == 0:
                print("El punto de referencia no está en el rango o su
valor es 0; no se aplica normalización.")
            else:
                for j in range(len(sample_indices)):
                    if sample_indices[j] >= ref_sample:
                        global_means[j] /= ref_val
                        global_maxs[j] /= ref_val
                        global_mins[j] /= ref_val

        # Graficar la línea de la media
        ax.plot(sample_indices, global_means, linestyle='--',
alpha=0.5, marker='o',
                label=nombres_familias[iterador],
color=colores[iterador])

        # Dibujar líneas verticales y horizontales para máximos y
mínimos
        for x, mean_val, max_val, min_val in zip(sample_indices,
global_means, global_maxs, global_mins):
            # Línea vertical: media hasta máximo (color rojo, dashed)
            ax.vlines(x, mean_val, max_val, colors=colores[iterador],
linestyles='dashed')
            # Línea vertical: mínimo hasta media (color azul, dashed)
            ax.vlines(x, min_val, mean_val, colors=colores[iterador],
linestyles='dashed')
            # Líneas horizontales cortas en máximo y mínimo
            ax.hlines(max_val, x - ext_line_width/2, x +
ext_line_width/2, colors=colores[iterador],
                      linestyles='solid', label="Val. extremos" if x
== sample_indices[0] else "")
            ax.hlines(min_val, x - ext_line_width/2, x +
ext_line_width/2, colors=colores[iterador], linestyles='solid')

        if iterador == 0:
            ax.set_title(titulo, fontsize=20)

        # Configurar ejes
        if iterador != n_plots - 1:
            ax.tick_params(axis='x', which='both', bottom=False,
top=False, labelbottom=False)
        else:
            ax.set_xlabel("Rep.", fontsize=20)

        fig.text(0.04, 0.5, nombre_ejey, va='center',
rotation='vertical', fontsize=35)
        ax.tick_params(axis='both', which='major', labelsize=14)
        ax.grid(True)
        ax.legend(fontsize=17)

        # Si tienes una función para zoom en eje y, actívala aquí:
        # add_y_zoom(ax)  # Si tienes definida esta función

    plt.show()

    # Guardar la figura si se especifica ruta
    if direccion_guardado:
```

```
        if not os.path.exists(direccion_guardado):
            try:
                os.makedirs(direccion_guardado, exist_ok=True)
            except Exception as e:
                print(f"Hubo un problema a la hora de crear el
directorio: {e}")
                return

        # Construir nombre archivo
        nombre_archivo = f"{clase}_{familia}_{final_titulo}.svg"
        direccion_completa = os.path.join(direccion_guardado,
nombre_archivo)

        try:
            fig.savefig(direccion_completa, format="svg", dpi=300)
            print(f"Gráfica {nombre_archivo} guardada con éxito")
        except Exception as e:
            print(f"Error al guardar la gráfica: {e}")
```

ANEXO 7: PROGRAMAS DE ANÁLISIS DE DATOS DE CARACTERIZACIÓN

Organización de datos

```
'''
Este programa se centra en crear una base de datos en forma de
diccionarios anidados, a partir de datos almacenados en ficheros .prn
distribuidos en una estructura de directorios organizada.

Ha sido diseñado como un script principal, donde la mayor parte de la
configuración y los datos de control (listas, rutas, filtros, etc.) se
gestionan mediante variables globales, accesibles desde todas las
funciones sin necesidad de pasarlas como argumentos.

Los datos son caracterizaciones de tres tipos de elementos, JFETs,
PMOS, y LCL (JFET y PMOS juntos), estos están jerarquizados por
carpetas :

1-Principalmente tenemos 5 carpetas que corresponden a las fechas en
las que se hicieron (0-Campanya EneroMarzo 2023, 3-Campanya Febrero
2024 post IC...)

    2-Dentro de estas(1), tendremos dos carpetas, llamadas 'DUTs
individuales'(2.1) y 'LCL'(2.2), que corresponden a los datos para
dispositivos individuales o para el LCL entero

        2.1(DUTs individuales)-Dentro de esta carpeta encontramos dos
subcarpetas, 'LCL 2A'(2.1.1) y 'LCL 10A'(2.1.2), que corresponden a la
clase (nivel de corriente que sufrirán)

            2.1.1(LCL 2A)-Dentro de esta carpeta encontramos,
generalmente, 5 carpetas, 4 de ellas corresponden a las familias de
JFETs(2.1.1.a) y otra corresponde a PMOS(2.1.1.b)

                2.1.1.a-En esta carpeta se encuentran los JFET de la
familia numerados(2.1.1.a.x)

         --->       2.1.1.a.x-Aquí se encuentran los datos, habrá,
generalmente, 6 ficheros, 3 de ellos son gráficas, y los otros 3
corresponden a archivos .prn donde se encuentran estructurados los
datos, cuyo nombre siempre suele empezar por el nombre de la prueba
(Rds-Id, Is-Vgs, Id(off)-Vds)

                2.1.1.b-En esta carpeta se encuentran los PMOS
numerados (2.1.1.b.x)

                    2.1.1.b.x-...

            2.1.2(LCL 10A)-...

        2.2(LCL)- La estructura es similar, excepto que en el punto
2.1.1, solo habrá 4 carpetas, de cada familia de LCL (iguales a los
nombres de las familias de JFET)


Los ficheros de datos se encuentran estructurados de la siguiente
manera:
```

```
                                  |    Setup title "Id(off)-Vds medida
Ig UV"
                                  |    Application test name "Id(off)-
Vds medida Ig UV"
                                  |    Test date 25-Jan-23
10 filas de información inicial   |    Test time 11:22:15
                                  |    Device ID IJW120R100T1-10
                                  |    Count 10
                                  |    Name Memo HoldTime DelayTime
IntegTime Gate Vd@Idss Vg IgLimit IgMinRange Source Drain VdStart
VdStop VdStep IdLimit IdZero IdMinRange
                                  |    Value  0 0 MEDIUM "SMU1:MC
      MCSMU" 200 -19.5 0.001 10uA "GNDU:GND    GNDU" "SMU5:HV
      HVSMU" 0 1200 2 0.001 1E-09 10nA
                                  |    Name Polarity Temp
                                  |    Value 1 25
            1 fila de variables\    Vdrain Idrain Vgate Igate BVdss
Idss Ta
             1 fila de unidades|    V A V A  A deg
                                  \    0 1.265E-10 -19.5 -1.1774E-05
2.87230609695152E-11 25
                                  \    1.9995 2.286E-10 -19.5 -1.1636E-
05  2.87230609695152E-11 25
                                  \    4.0005 1.194E-10 -19.5 -1.15479E-
05  2.87230609695152E-11 25
          Varias filas de datos\    6 8.6E-11 -19.5 -1.14767E-05
2.87230609695152E-11 25
                                  \    7.9995 5.8E-11 -19.5 -1.14163E-05
2.87230609695152E-11 25
                                  \    10.0005 9.06E-11 -19.5 -1.13634E-
05  2.87230609695152E-11 25
                                  \    12 3.05E-11 -19.5 -1.13135E-05
2.87230609695152E-11 25

La estructura de los diccionarios será la siguiente:

    Primera capa -> DUT | LCL ; Nos indica si queremos acceder a los
datos realizados a los dispositivos individuales o a LCL enteros
        Capa DUT -> LCL 2A | LCL 10A ; Nos hace referencia a los
dispositivos de clase 2 o clase 10
            Capa LCL 2A -> JFET | PMOS ; Donde se indica si vamos con
las familias de JFET o de PMOS
                Capa JFET -> IJW120R100T1 | UJN1205K | UJ3N065025K3S |
UJ3N120035K3S; Donde encontramos los diferentes dispositivos de la
familia
                    Capa IJW120R100T1 -> (referencias concretas) ;
AquÃ- estarÃ¡n las referencias concretas de la familia de clase 2A, p.
ej. IJW120R100T1-10
                        Capa (referencia concreta) -> Id(off)-Vds |
Id-Vgs | Rds-Id ; AquÃ- se contienen los 3 tipos de pruebas que se han
realizado
                            Capa Id(off)-Vds -> enero_marzo_23 |
feb_24 | nov_23 | nov_24 | sept_23 ; La fecha en la que se realizaron,
puede que alguna no se encuentre, debido a que no hay datos para esta
referencia concreta, de esa prueba especÃ-fica en esa fecha
                                Capa ener0_marzo_23 ->  data | info ;
AquÃ- ttendremos
                                    capa data ->Dataframe que contiene
los puntos de las curvas (generalmente son importantes las dos
primeras columnas)
```

```
                                        capa info ->Cadena de caracteres
referentes a la información de la caracterización
                    Capa PMOS; es totalmente similar
                Capa LCL 10A -> Totalmente similar, donde se encuentran
las familias de referencias que serÃ¡n de clase 10A

            Capa LCL -> LCL 2A | LCL 10A
                Capa LCL 2A

                Capa LCL 10A

*Durante el programa deberemos tener en cuenta que, como en el ejemplo
anterior, la fila de unidades tiene doble espacio entre dos unidades
en medio

*Durante el programa deberemos darnos cuenta de que puede haber
ficheros faltantes

*Durante el programa deberemos darnos cuenta si el fichero de datos no
tiene la misma estructura y, por lo tanto, no se puede obtener su
información como con los otros

'''

import os
import pandas as pd


#----------------------VARIABLES GLOBALES
#---------------------------------------

# Diccionarios globales que almacenan los datos procesados para cada
conjunto
dicc_valencia1 = {}
dicc_valencia2 = {}
dicc_valencia3 = {}

# Diccionarios para almacenar mensajes de error clasificados por
campaña (fecha)
error1 = {}  # Diccionario para errores del primer conjunto de datos
error1["enero_marzo_23"] = []  # Lista de errores para campaña Enero-
Marzo 2023
error1["sept_23"] = []          # Lista de errores para campaña
Septiembre 2023
error1["nov_23"] = []           # Lista de errores para campaña
Noviembre 2023
error1["feb_24"] = []           # Lista de errores para campaña
Febrero 2024
error1["nov_24"] = []           # Lista de errores para campaña
Noviembre 2024
error1["mar_25"] = []           # Lista de errores para campaña Marzo
2025

error2 = {}  # Diccionario para errores del segundo conjunto de datos
error2["enero_marzo_23"] = []  # Lista de errores para campaña Enero-
Marzo 2023
error2["jun_24"] = []           # Lista de errores para campaña Junio
2024

error3 = []  # Lista para almacenar errores del tercer conjunto de
datos (no clasificados por fecha)
```

```
# Ruta base global donde se encuentran las carpetas con los datos
direc_global = r"D:\TESTS_UV_TED2021"

# Lista con nombres de las subcarpetas correspondientes a cada campaña
dentro del directorio global
direc_camp = [
    "0-Campanya EneroMarzo 2023",
    "1-Campanya Septiembre 2023 post HTOL",
    "2-Campaya Noviembre 23 post OL",
    "3-Campanya Febrero 2024 post IC",
    "4-Campanya Nov 2024 post CC + post test acumulado",
    "7-Campanya Marzo 2025"
]

# Lista de nombres usados para identificar cada campaña en los
diccionarios de datos
nom_camp = [
    "enero_marzo_23",
    "sept_23",
    "nov_23",
    "feb_24",
    "nov_24",
    "mar_25"
]

# Lista con nombres de carpetas de campañas que se deben ignorar (ban
list)
direc_camp_ban = []

# Lista con nombres de subcarpetas comunes dentro de las carpetas de
campaña (DUTs y LCLs)
direc_camp_sub = ["DUTs individuales", "LCLs"]

# Nombres que se asignarán a las subcarpetas en los diccionarios para
facilitar el acceso
nom_direc_camp_sub = ["DUT", "LCL"]

# Subapartados específicos dentro de la carpeta 'DUT' (generalmente
JFET o PMOS)
nom_camp_sub_DUT = ["JFET", "PMOS"]

# Nombres de subcarpetas dentro de "DUTs individuales" (pueden variar,
ej. "Class 2A" o "Class 10A")
# También serán claves en los diccionarios para DUT y LCL
direc_camp_sub_DUT_LCL = ["LCL 2A", "LCL 10A"]

# Listas con nombres de bloques DUT que coinciden con subcarpetas
dentro de 'JFET' o 'PMOS'
# Estos nombres serán usados como claves en los diccionarios
DUT2A = ["IJW120R100T1", "UJ3N065025K3S", "UJ3N120035K3S", "UJN1205K"]
DUT10A = ["IJW120R100T1", "UJ3N065025K3S", "UJ3N120035K3S",
"UJN1205K"]

# Lista para nombres de DUT que se deben excluir (ban list)
DUT_ban = []

# Listas con grupos de DUT individuales, identificados por sus nombres
específicos para organización
IJW120R100T1_1 = ["IJW120R100T1-10", "IJW120R100T1-41", "IJW120R100T1-
43", "IJW120R100T1-49"]
```

```
UJ3N065025K3S_1 = ["UJ3N065025K3S-8", "UJ3N065025K3S-9",
"UJ3N065025K3S-10", "UJ3N065025K3S-43"]
UJ3N120035K3S_1 = ["UJ3N120035K3S-26", "UJ3N120035K3S-37",
"UJ3N120035K3S-43", "UJ3N120035K3S-44"]
UJN1205K_1 = ["UJN1205K-12", "UJN1205K-17", "UJN1205K-27", "UJN1205K-
39"]

# Grupos equivalentes para subcarpeta PMOS (también puede llamarse
SQP50P03)
IJW120R100T1_1_PMOS = ["PMOS-70", "PMOS-35", "PMOS-81", "PMOS-28"]
UJ3N065025K3S_1_PMOS = ["PMOS-90", "PMOS-74", "PMOS-77", "PMOS-55"]
UJ3N120035K3S_1_PMOS = ["PMOS-91", "PMOS-49", "PMOS-30", "PMOS-87"]
UJN1205K_1_PMOS = ["PMOS-37", "PMOS-63", "PMOS-71", "PMOS-98"]

# Listas que agrupan todos los DUT2A y sus equivalentes PMOS para
facilitar iteraciones
DUT2A_complete = [IJW120R100T1_1, UJ3N065025K3S_1, UJ3N120035K3S_1,
UJN1205K_1]
DUT2A_complete_PMOS = [IJW120R100T1_1_PMOS, UJ3N065025K3S_1_PMOS,
UJ3N120035K3S_1_PMOS, UJN1205K_1_PMOS]

# Grupos DUT 10A para cada tipo, con listas de nombres específicos de
dispositivos
IJW120R100T1_2 = [
    "IJW120R100T1-4", "IJW120R100T1-5", "IJW120R100T1-6",
"IJW120R100T1-8", "IJW120R100T1-9",
    "IJW120R100T1-38", "IJW120R100T1-42", "IJW120R100T1-44",
"IJW120R100T1-45", "IJW120R100T1-46",
    "IJW120R100T1-47", "IJW120R100T1-48"
]

UJ3N065025K3S_2 = [
    "UJ3N065025K3S-20", "UJ3N065025K3S-21", "UJ3N065025K3S-24",
"UJ3N065025K3S-25", "UJ3N065025K3S-26",
    "UJ3N065025K3S-27", "UJ3N065025K3S-30", "UJ3N065025K3S-35",
"UJ3N065025K3S-36", "UJ3N065025K3S-37",
    "UJ3N065025K3S-44", "UJ3N065025K3S-45"
]

UJ3N120035K3S_2 = [
    "UJ3N120035K3S-12", "UJ3N120035K3S-17", "UJ3N120035K3S-21",
"UJ3N120035K3S-22", "UJ3N120035K3S-29",
    "UJ3N120035K3S-30", "UJ3N120035K3S-31", "UJ3N120035K3S-32",
"UJ3N120035K3S-33", "UJ3N120035K3S-34",
    "UJ3N120035K3S-46", "UJ3N120035K3S-47"
]

UJN1205K_2 = [
    "UJN1205K-2", "UJN1205K-5", "UJN1205K-6", "UJN1205K-7", "UJN1205K-
19", "UJN1205K-24", "UJN1205K-25",
    "UJN1205K-34", "UJN1205K-45", "UJN1205K-48", "UJN1205K-49",
"UJN1205K-50"
]

# Listas de dispositivos PMOS correspondientes al grupo DUT 10A
IJW120R100T1_2_PMOS = ["PMOS-75", "PMOS-62", "PMOS-54", "PMOS-50"]
UJ3N065025K3S_2_PMOS = ["PMOS-73", "PMOS-29", "PMOS-32", "PMOS-72"]
UJ3N120035K3S_2_PMOS = ["PMOS-46", "PMOS-84", "PMOS-76", "PMOS-34"]
UJN1205K_2_PMOS = ["PMOS-53", "PMOS-58", "PMOS-48", "PMOS-45"]

# Agrupación completa de DUT 10A para facilitar iteraciones
```

```
DUT10A_complete = [IJW120R100T1_2, UJ3N065025K3S_2, UJ3N120035K3S_2,
UJN1205K_2]
DUT10A_complete_PMOS = [IJW120R100T1_2_PMOS, UJ3N065025K3S_2_PMOS,
UJ3N120035K3S_2_PMOS, UJN1205K_2_PMOS]

# Ahora definimos estructuras similares para LCL

# Subcarpetas LCL 2A y 10A (nombres de carpetas en campañas)
direc_camp_sub_LCL_LCL = ["LCL 2A", "LCL 10A"]

# Listas con nombres base de LCL para 2A y 10A, coinciden con nombres
DUT base
LCL2A = ["IJW120R100T1", "UJ3N065025K3S", "UJ3N120035K3S", "UJN1205K"]
LCL10A = ["IJW120R100T1", "UJ3N065025K3S", "UJ3N120035K3S",
"UJN1205K"]

# Lista para nombres LCL que se deben excluir (ban list)
LCL_ban = []

# Listas con grupos individuales de LCL 2A para cada tipo
LCL_IJW120R100T1_1 = ["LCL-1", "LCL-2", "LCL-3", "LCL-4"]
LCL_UJ3N065025K3S_1 = ["LCL-9", "LCL-10", "LCL-11", "LCL-12"]
LCL_UJ3N120035K3S_1 = ["LCL-13", "LCL-14", "LCL-15", "LCL-16"]
LCL_UJN1205K_1 = ["LCL-5", "LCL-6", "LCL-7", "LCL-8"]

# Agrupación completa para LCL 2A
LCL2A_complete = [LCL_IJW120R100T1_1, LCL_UJ3N065025K3S_1,
LCL_UJ3N120035K3S_1, LCL_UJN1205K_1]

# Listas con grupos individuales de LCL 10A para cada tipo
LCL_IJW120R100T1_2 = ["LCL-17", "LCL-18", "LCL-19", "LCL-20"]
LCL_UJ3N065025K3S_2 = ["LCL-25", "LCL-26", "LCL-27", "LCL-28"]
LCL_UJ3N120035K3S_2 = ["LCL-29", "LCL-30", "LCL-31", "LCL-32"]
LCL_UJN1205K_2 = ["LCL-21", "LCL-22", "LCL-23", "LCL-24"]

# Agrupación completa para LCL 10A
LCL10A_complete = [LCL_IJW120R100T1_2, LCL_UJ3N065025K3S_2,
LCL_UJ3N120035K3S_2, LCL_UJN1205K_2]

# Array con nombres base de las variables medidas en los ficheros de
análisis
# También serán las claves para los diccionarios que contengan esos
datos
nom_var = ["Id(off)-Vds", "Id-Vgs", "Rds-Id"]

# Lista para contener palabras clave relacionadas con cada campaña,
posición corresponde a direc_camp
palabra_clave = []


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

#variable que me va a contabilizar cuÃ¡ntos errores han ocurrido
"""
    Extrae y organiza los datos de los dispositivos DUT (Device Under
Test) según clase, tipo y estructura de carpetas,
    cargando información en un diccionario jerárquico.

    Parámetros:
      clase: Clase del DUT, por ejemplo "LCL 2A" o "LCL 10A".
```

```
      sujeto: Identificador del sujeto, generalmente "DUT".
      dicc_valencia: Diccionario principal donde se almacenan los
datos extraídos.
      errores: Diccionario donde se registran errores como archivos o
directorios inexistentes.
"""
def clase_DUT(clase, sujeto, dicc_valencia, errores):

    for indice_dut, dut in enumerate(nom_camp_sub_DUT):

        # Dependiendo de la clase y tipo de DUT, se selecciona la
estructura de datos adecuada
        if clase == "LCL 2A":
            if dut == "JFET":
                DUT_complete_PMOS_JFET = DUT2A_complete
            else:
                DUT_complete_PMOS_JFET = DUT2A_complete_PMOS
        else:
            if dut == "JFET":
                DUT_complete_PMOS_JFET = DUT10A_complete
            else:
                DUT_complete_PMOS_JFET = DUT10A_complete_PMOS

        # Se crea una entrada en el diccionario para este DUT
        dicc_valencia[sujeto][clase][dut] = {}

        for indice_subgrupo, subgrupo in enumerate(DUT2A):

            if subgrupo in DUT_ban:
                print("No se usará el subgrupo " + subgrupo)
                continue

            print("--Usando el subgrupo " + subgrupo)

            # Entrada para el subgrupo del DUT
            dicc_valencia[sujeto][clase][dut][subgrupo] = {}

            for sub_subgrupo in
DUT_complete_PMOS_JFET[indice_subgrupo]:
                print("")
                print("-usando el sub-subgrupo " + sub_subgrupo)
                print("")

                # Entrada para el sub-subgrupo

dicc_valencia[sujeto][clase][dut][subgrupo][sub_subgrupo] = {}

                for var in nom_var:
                    print("-Para la variable " + var)

                    # Entrada para la variable que se va a analizar

dicc_valencia[sujeto][clase][dut][subgrupo][sub_subgrupo][var] = {}

                    # Iteración por campañas (direc_camp contiene
nombres de campañas tipo "Camp1", "Camp2", etc.)
                    for indice_campana, direc_campana in
enumerate(direc_camp):

                        # Si la campaña está en la lista de campañas
prohibidas, se salta
```

```
                        if direc_campana in direc_camp_ban:
                            continue

                        # Construcción de la ruta esperada donde se
encuentran los datos
                        direc_comprobar = os.path.join(
                            direc_global, direc_campana,
direc_camp_sub[0],
                            clase, dut, subgrupo, sub_subgrupo
                        )

                        # Si el directorio no existe, se registra el
error y se omite
                        if not os.path.isdir(direc_comprobar):
                            print("No existe " + direc_comprobar)

                            if "No existe " + direc_comprobar not in
errores[nom_camp[indice_campana]]:

errores[nom_camp[indice_campana]].append(
                                    clase + sub_subgrupo + "-No existe
" + direc_comprobar
                                )
                            continue

                        print("Buscando en " + direc_comprobar)

                        # Se crea la entrada para esta campaña

dicc_valencia[sujeto][clase][dut][subgrupo][sub_subgrupo][var][nom_cam
p[indice_campana]] = {}

                        # Cambia de directorio para operar localmente
                        os.chdir(direc_comprobar)

                        # Lista de archivos .prn que coincidan con la
variable actual
                        filtered_files = [
                            f for f in os.listdir(direc_comprobar)
                            if f.lower().startswith(var.lower())
                            and f.lower().endswith(".prn")
                            and
os.path.isfile(os.path.join(direc_comprobar, f))
                        ]

                        # Si no se encuentra ningún archivo válido, se
registra error
                        if filtered_files == []:
                            print("************No se ha encontrado
ningún fichero")
                            errores[nom_camp[indice_campana]].append(
                                clase + sub_subgrupo + "-No se
encuentra un fichero con extensión prn" +
                                "EN DIRECCIÓN " + direc_comprobar + "
USANDO LA VARIABLE " + var
                            )
                            continue

                        # Lectura del contenido del archivo
                        with open(filtered_files[0], "r") as file:
                            lineas = file.readlines()
```

```
                        # Las primeras 10 líneas contienen información
descriptiva o metadatos
                        informacion_inicial = "".join(lineas[:10])

                        # Línea 11: nombres de columnas | Línea 12:
unidades físicas
                        nombres_columnas = lineas[10].strip().split()
                        unidades = lineas[11].strip().split()

                        # Caso especial para esta variable: una
columna extra debe ser eliminada
                        if var == "Id(off)-Vds":
                            nombres_columnas.pop(4)
                            unidades.pop(4)
                            unidades = ["V", "A", "V", "A", "A",
"deg"]

                        # Se guarda la información inicial como texto

dicc_valencia[sujeto][clase][dut][subgrupo][sub_subgrupo][var][nom_cam
p[indice_campana]]["info"] = informacion_inicial

                        # Se emparejan nombres de columnas con sus
unidades: ej. "Vgs (V)"
                        nombres_columnas_con_unidades = [
                            f"{var} ({unidad})" for var, unidad in
zip(nombres_columnas, unidades)
                        ]
                        print(nombres_columnas_con_unidades)


                        # A partir de línea 13 están los datos
tabulados
                        datos = [linea.strip().split() for linea in
lineas[12:]]

                        # Se intenta crear un DataFrame con los datos
                        try:
                            df = pd.DataFrame(datos,
columns=nombres_columnas_con_unidades)
                        except:
                            errores[nom_camp[indice_campana]].append(
                                clase + sub_subgrupo + "-OCURRE UN
PROBLEMA A LA HORA DE GENERAR EL DATAFRAME" +
                                "EN DIRECCIÓN " + direc_comprobar + "
USANDO LA VARIABLE " + var
                            )
                            continue

                        # Se intenta convertir cada columna del
DataFrame a datos numéricos (float, int, etc.)
                        df = df.apply(pd.to_numeric, errors='coerce')

                        # Se guarda el DataFrame en el diccionario

dicc_valencia[sujeto][clase][dut][subgrupo][sub_subgrupo][var][nom_cam
p[indice_campana]]["data"] = df

"""
    Llama a la función clase_DUT según la clase del DUT (2A o 10A).
```

```
    Parámetros:
      clase: Clase del DUT ("LCL 2A" o "LCL 10A").
      sujeto: Identificador del sujeto (debe ser "DUT").
      dicc_valencia: Diccionario donde se almacenan los datos
organizados.
      errores: Diccionario donde se almacenan errores ocurridos
durante la extracción.
"""
def DUT(clase, sujeto, dicc_valencia, errores):

    if clase == "LCL 2A":  # si estamos con la clase 2A
        clase_DUT(clase, sujeto, dicc_valencia, errores)

    elif clase == "LCL 10A":  # si estamos con la clase 10A
        clase_DUT(clase, sujeto, dicc_valencia, errores)


"""
    Recorre los subgrupos de una clase LCL (2A o 10A) para un sujeto
determinado,
    y organiza los datos en un diccionario anidado según estructura
jerárquica:
    clase → subgrupo → sub-subgrupo → variable → campaña.

    Busca archivos .prn, extrae la información de cabecera y los datos
tabulares,
    y los guarda en formato DataFrame.

    Parámetros:
      clase: Cadena que indica la clase de LCL (por ejemplo, "LCL
2A").
      sujeto: Identificador del sujeto para el que se están procesando
los datos.
      dicc_valencia: Diccionario donde se guardarán los datos
procesados.
      errores: Diccionario donde se registrarán los errores ocurridos
durante el proceso.
"""
def clase_LCL(clase, sujeto, dicc_valencia, errores):

    for indice_subgrupo, subgrupo in enumerate(LCL2A):

        if subgrupo in DUT_ban:
            print("No se usará el subgrupo " + subgrupo)
            continue

        print("--Usando el subgrupo " + subgrupo)

        # Selecciona la lista completa correspondiente según la clase
        if clase == "LCL 2A":
            LCL_complete = LCL2A_complete
        else:
            LCL_complete = LCL10A_complete

        # Se crea la entrada para el subgrupo
        dicc_valencia[sujeto][clase][subgrupo] = {}

        for sub_subgrupo in LCL_complete[indice_subgrupo]:
            print("")
            print("-Usando el sub-subgrupo " + sub_subgrupo)
            print("")
```

```
            dicc_valencia[sujeto][clase][subgrupo][sub_subgrupo] = {}

            for var in nom_var:
                print("-Para la variable " + var)

dicc_valencia[sujeto][clase][subgrupo][sub_subgrupo][var] = {}

                for indice_campana, direc_campana in
enumerate(direc_camp):
                    # Se omiten campañas prohibidas
                    if direc_campana in direc_camp_ban:
                        continue

                    # Se construye el path donde deberían estar los
datos
                    direc_comprobar = os.path.join(direc_global,
direc_campana, direc_camp_sub[1], clase, subgrupo, sub_subgrupo)

                    # Si el directorio no existe, se omite
                    if not os.path.isdir(direc_comprobar):
                        print("No existe " + direc_comprobar)
                        continue

                    print("Buscando en " + direc_comprobar)

                    # Se inicializa el diccionario para esa campaña

dicc_valencia[sujeto][clase][subgrupo][sub_subgrupo][var][nom_camp[ind
ice_campana]] = {}

                    os.chdir(direc_comprobar)  # Cambia al directorio
de trabajo actual

                    # Busca el archivo .prn que empiece con el nombre
de la variable
                    filtered_files = [
                        f for f in os.listdir(direc_comprobar)
                        if f.lower().startswith(var.lower()) and
f.lower().endswith(".prn") and
os.path.isfile(os.path.join(direc_comprobar, f))
                    ]

                    if filtered_files == []:
                        print("************No se ha encontrado ningún
fichero")
                        errores[nom_camp[indice_campana]].append(
                            "No se encuentra un fichero con extensión
.prn EN DIRECCIÓN " +
                            direc_comprobar + " USANDO LA VARIABLE " +
var
                        )
                        continue

                    # Lectura del archivo de datos
                    with open(filtered_files[0], "r") as file:
                        lineas = file.readlines()

                    informacion_inicial = "".join(lineas[:10])  #
Cabecera como string

                    nombres_columnas = lineas[10].strip().split()
```

```
                    unidades = lineas[11].strip().split()

                    # Ajuste especial para "Id(off)-Vds" que tiene una
columna malformada
                    if var == "Id(off)-Vds":
                        nombres_columnas.pop(4)
                        unidades.pop(4)
                        unidades = ["V", "A", "V", "A", "A", "deg"]

                    # Guardamos la info de cabecera

dicc_valencia[sujeto][clase][subgrupo][sub_subgrupo][var][nom_camp[ind
ice_campana]]["info"] = informacion_inicial

                    # Unificamos nombre y unidad de cada columna
                    nombres_columnas_con_unidades = [f"{var}
({unidad})" for var, unidad in zip(nombres_columnas, unidades)]
                    print(nombres_columnas_con_unidades)

                    # Procesamos los datos tabulares desde la línea 13
                    datos = [linea.strip().split() for linea in
lineas[12:]]

                    # Intentamos crear el DataFrame
                    try:
                        df = pd.DataFrame(datos,
columns=nombres_columnas_con_unidades)
                    except:
                        errores[nom_camp[indice_campana]].append(
                            "OCURRE UN PROBLEMA A LA HORA DE GENERAR
EL DATAFRAME EN DIRECCIÓN " +
                            direc_comprobar + " USANDO LA VARIABLE " +
var
                        )
                        continue

                    # Convertimos a tipo numérico, los valores
inválidos serán NaN
                    df = df.apply(pd.to_numeric, errors='coerce')

                    # Guardamos el DataFrame en el diccionario

dicc_valencia[sujeto][clase][subgrupo][sub_subgrupo][var][nom_camp[ind
ice_campana]]["data"] = df

"""
    Llama a la función `clase_LCL` según la clase LCL correspondiente.

    Parámetros:
      clase: Cadena con el nombre de la clase ("LCL 2A" o "LCL 10A").
      sujeto: Identificador del sujeto a procesar.
      dicc_valencia: Diccionario donde se almacenarán los datos
resultantes.
      errores: Diccionario donde se acumularán mensajes de error.
"""
def LCL(clase, sujeto, dicc_valencia, errores):

    if clase == "LCL 2A":
        clase_LCL(clase, sujeto, dicc_valencia, errores)

    elif clase == "LCL 10A":
```

```
        clase_LCL(clase, sujeto, dicc_valencia, errores)


"""
    Inicializa la estructura principal del diccionario `dicc_valencia`
recorriendo cada sujeto ("DUT" y "LCL"),
    y delega el procesamiento a funciones específicas dependiendo del
tipo de sujeto.

    Parámetros:
      dicc_valencia: Diccionario donde se almacenarán todos los datos
procesados.
      errores: Diccionario donde se registrarán los errores
encontrados durante el proceso.
"""
# El diccionario global empezará teniendo dos entradas principales:
"DUT" y "LCL"
def Extraccion(dicc_valencia, errores):

    for sujeto in nom_direc_camp_sub:
        dicc_valencia[sujeto] = {}  # Creamos una entrada por sujeto
("DUT" o "LCL")
        print("-----------Para el sujeto " + sujeto)

        for clase in direc_camp_sub_DUT_LCL:  # Para cada clase ("LCL
2A" o "LCL 10A")
            print("-------En la clase " + clase)
            dicc_valencia[sujeto][clase] = {}  # Inicializamos el
diccionario para esa clase

            if sujeto == "DUT":
                # Si el sujeto es DUT, se llama a la función
específica para DUT
                DUT(clase, sujeto, dicc_valencia, errores)

            elif sujeto == "LCL":
                # Si el sujeto es LCL, se llama a la función
específica para LCL
                LCL(clase, sujeto, dicc_valencia, errores)
```

Representación gráfica

```
import pandas as pd
import plotly.graph_objects as go
import matplotlib.pyplot as plt
from matplotlib.widgets import CheckButtons
import numpy as np
import seaborn as sns
import os


"""

 IMPORTANTE: Se asume que en el workspace ya tienes definido el
diccionario
 dicc_valencia_sub con la siguiente estructura:

 dicc_valencia_sub = {
     familia_a_usar: {
         "Dispositivo1": {
             prueba_a_analizar: {
                 "enero_marzo_23": {"data": DataFrame},
                 "sept_23": {"data": DataFrame},
                 ...
             }
         },
         "Dispositivo2": { ... },
         ...
     }
 }

 Asegúrate de que esta variable esté definida antes de ejecutar el
código.
"""

"""
    Convierte un color en formato Matplotlib a un string RGBA para
usar con Plotly.

    Parámetros:
      color: Nombre del color o código que entiende Matplotlib.
      alpha: Valor de transparencia (0.0 totalmente transparente, 1.0
opaco).

    Retorna:
      String en formato 'rgba(r, g, b, alpha)'.
"""

def hacer_transparente_plotly(color, alpha=0.3):

    rgba = plt.cm.colors.to_rgba(color)
    rgba_transparente = f"rgba({int(rgba[0] * 255)}, {int(rgba[1] *
255)}, {int(rgba[2] * 255)}, {alpha})"
    return rgba_transparente


"""
    Convierte un color en formato Matplotlib a RGBA con transparencia
ajustada.

    Parámetros:
      color: Nombre del color o código compatible con Matplotlib.
```

```
      alpha: Nivel de transparencia a aplicar.

    Retorna:
      Tupla RGBA con el nuevo nivel de transparencia.
"""
def hacer_transparente_matplotlib(color, alpha=0.3):

    rgba = plt.cm.colors.to_rgba(color)
    rgba_transparente = rgba[:3] + (alpha,)
    return rgba_transparente


"""
    Crea un gráfico interactivo en Plotly con curvas obtenidas de un
diccionario de dataframes.

    Parámetros:
      dicc_datos: Diccionario anidado con estructura tipo DUT/LCL →
clase → familia → dispositivo → prueba → fecha → 'data': DataFrame
      fechas_a_analizar: Lista de fechas (claves) que se desean
incluir en la gráfica.
      artefacto_a_analizar: Nombre del artefacto a graficar (por
ejemplo, 'JFET', 'PMOS', 'LCL').
      prueba_a_analizar: Tipo de prueba (por ejemplo, 'Id(off)-Vds',
'Id-Vgs', etc).
      clase_a_analizar: Clase del experimento o medición.
      familia_a_analizar: Familia de dispositivos a analizar.
      dispositivos_ban_list: Lista de dispositivos a excluir del
gráfico.
      errores: Diccionario donde se registran mensajes de error o
ausencia de datos.
      tipo_trazado_por_fecha: Lista de estilos de línea (por ejemplo,
'solid', 'dot', etc.) para cada fecha.
      color_por_disp: Lista de colores asignados a los dispositivos.
      nombre_archivo: Nombre base del archivo HTML de salida
(opcional).

    El gráfico incluye botones para:
      - Mostrar todas las curvas agrupadas por fecha.
      - Mostrar todas las curvas individualmente.
      - Mostrar solo las curvas correspondientes a una fecha
específica.
"""
def genera_curvas_plotly(dicc_datos, fechas_a_analizar,
artefacto_a_analizar, prueba_a_analizar, clase_a_analizar,
familia_a_analizar, dispositivos_ban_list, errores,
tipo_trazado_por_fecha, color_por_disp, nombre_archivo=None):

    trazas = []
    trace_meta = []

    # Definir ejes y título en función de la prueba
    if prueba_a_analizar == "Id(off)-Vds":
        xlabel = "Vds (V)"
        ylabel = "Id(off) (A)"
    elif prueba_a_analizar == "Rds-Id":
        xlabel = "Id (A)"
        ylabel = "Rds (Ω)"
    elif prueba_a_analizar == "Id-Vgs":
        xlabel = "Vgs (V)"
        ylabel = "Id (A)"
```

```
    else:
        xlabel = ""
        ylabel = ""

    title = prueba_a_analizar + " curves for " + artefacto_a_analizar
+ " \ class " + clase_a_analizar + " \ family " + familia_a_analizar

    # Selección de diccionario en función del tipo de artefacto
    if artefacto_a_analizar == "JFET" or artefacto_a_analizar ==
"PMOS":
        dicc_valencia_sub =
dicc_datos["DUT"][clase_a_analizar][artefacto_a_analizar]
    elif artefacto_a_analizar == "LCL":
        dicc_valencia_sub = dicc_datos["LCL"][clase_a_analizar]
    else:
        dicc_valencia_sub = dicc_datos["LCL"][clase_a_analizar]
        print("Se ha introducido incorrectamente la variable
artefacto_a_utilizar, se usará 'LCL' ")

    labelfechas = []

    for indice_fecha, fecha in enumerate(fechas_a_analizar):
        i = 0
        labelfechas.append(fecha)
        for clavedisp, valordisp in
dicc_valencia_sub[familia_a_analizar].items():

            if fecha not in
dicc_valencia_sub[familia_a_analizar][clavedisp][prueba_a_analizar]:
                i += 1
                errores["fecha"].append(f"No existen datos para el
dispositivo {clavedisp} para la prueba {prueba_a_analizar} en la fecha
{fecha}")
                continue

            if "data" not in
dicc_valencia_sub[familia_a_analizar][clavedisp][prueba_a_analizar][fe
cha]:
                i += 1
                errores["datos"].append(f"No existen datos para el
dispositivo {clavedisp} para la prueba {prueba_a_analizar} en la fecha
{fecha}")
                continue

            if clavedisp in dispositivos_ban_list:
                i += 1
                continue

            df =
dicc_valencia_sub[familia_a_analizar][clavedisp][prueba_a_analizar][fe
cha]["data"]
            xdata = df.iloc[:, 0]
            ydata = df.iloc[:, 1]

            # Aumentar transparencia si la fecha está a partir del
índice 4 (por ejemplo, nov_24)
            if indice_fecha >= 4:
                color = hacer_transparente_plotly(color_por_disp[i],
0.7)
            else:
                color = color_por_disp[i % len(color_por_disp)]
```

```
            traza = go.Scatter(
                x=xdata,
                y=ydata,
                mode='lines',
                name=f"{clavedisp} {fecha}",
                line=dict(
                    dash=tipo_trazado_por_fecha[indice_fecha],
                    color=color)
            )
            trazas.append(traza)
            trace_meta.append((indice_fecha, i))
            i += 1

    total_trazas = len(trazas)

    def get_mode_arrays(mode, lista_index=None):
        """
        Genera listas de visibilidad y grupos de leyenda según el modo
de visualización.

        Parámetros:
          mode: 'todo' para todas las listas, 'lista' para una lista
específica.
          lista_index: Índice de la lista específica si mode es
'lista'.

        Retorna:
          visible: Lista booleana de visibilidad para cada traza.
          legendgroups: Lista de nombres de grupo de leyenda para cada
traza.
        """
        visible = []
        legendgroups = []
        for meta in trace_meta:
            li, di = meta
            if mode == 'todo':
                visible.append(True)
                legendgroups.append(f'Lista {li+1}')
            elif mode == 'lista':
                if li == lista_index:
                    visible.append(True)
                    legendgroups.append(f'Lista {li+1} DF {di+1}')
                else:
                    visible.append(False)
                    legendgroups.append('')
        return visible, legendgroups

    visible_todo, legendgroups_todo = get_mode_arrays('todo')

    mode_arrays_listas = {}
    num_fechas = len(fechas_a_analizar)
    for i in range(num_fechas):
        visible_lista, legendgroups_lista = get_mode_arrays('lista',
lista_index=i)
        mode_arrays_listas[i] = (visible_lista, legendgroups_lista)

    individual_legendgroups = []
    individual_legendgrouptitles = []
    first_in_list = {}
    for meta in trace_meta:
```

```
        li, di = meta
        individual_legendgroups.append(f"trace_{li}_{di}")
        if li not in first_in_list:
            first_in_list[li] = True

individual_legendgrouptitles.append(dict(text=labelfechas[li]))
        else:
            individual_legendgrouptitles.append(dict(text=""))

    visible_individuales = visible_todo[:]

    fig = go.Figure(data=trazas)

    for idx, trace in enumerate(fig.data):
        trace.legendgroup = legendgroups_todo[idx]

    fig.update_layout(
        legend=dict(
            itemclick='toggle',
            itemdoubleclick='toggleothers'
        )
    )

    botones = []

    botones.append(dict(
        label='Mostrar Todo agrupados',
        method='update',
        args=[
            {'visible': visible_todo, 'legendgroup':
legendgroups_todo},
            {'title': 'Todas las Listas',
             'legend': {'itemclick': 'toggle', 'itemdoubleclick':
'toggleothers'}}
        ]
    ))

    botones.append(dict(
        label='Mostrar Todo individuales',
        method='update',
        args=[
            {'visible': visible_individuales,
             'legendgroup': individual_legendgroups,
             'legendgrouptitle': individual_legendgrouptitles},
            {'title': 'Individuales',
             'legend': {'itemclick': 'toggle', 'itemdoubleclick':
'toggleothers'}}
        ]
    ))

    for i in range(num_fechas):
        visible_lista, legendgroups_lista = mode_arrays_listas[i]
        botones.append(dict(
            label=labelfechas[i],
            method='update',
            args=[
                {'visible': visible_lista, 'legendgroup':
legendgroups_lista},
                {'title': f'Ver Lista {i+1}',
                 'legend': {'itemclick': 'toggle', 'itemdoubleclick':
'toggleothers'}}
```

```
                ]
            ))

        fig.update_layout(
            updatemenus=[dict(
                active=0,
                buttons=botones,
                direction="down",
                showactive=True,
                x=0.1,
                y=1.15,
                xanchor='left',
                yanchor='top'
            )],
            title={
                "text": title,
                "x": 0.5,
                "xanchor": "center",
                "yanchor": "top"
            },
            xaxis_title=xlabel,
            yaxis_title=ylabel
        )

        if nombre_archivo is None:
            incluir_eliminados = "_sin_" + '_'.join(dispositivos_ban_list)
if dispositivos_ban_list else ""
            nombre_archivo =
f"{clase_a_analizar}_{familia_a_analizar}_{prueba_a_analizar}_{artefac
to_a_analizar}{incluir_eliminados}"

        fig.write_html(nombre_archivo + ".html")
        print("Gráfica guardada con nombre: " + nombre_archivo)


"""
    Representa curvas experimentales para distintos dispositivos en
función de varios parámetros de filtrado.

    Parámetros:
        dicc_datos (dict): Diccionario de datos experimentales
estructurado jerárquicamente.
        fechas_a_analizar (list): Fechas específicas para las cuales
se grafican los datos.
        artefacto_a_analizar (str): Tipo de artefacto (e.g., 'JFET',
'PMOS', 'LCL').
        prueba_a_analizar (str): Tipo de prueba a representar
('Id(off)-Vds', 'Id-Vgs', 'Rds-Id').
        clase_a_analizar (str): Clase del dispositivo dentro de los
datos.
        familia_a_analizar (str): Familia específica de dispositivos a
representar.
        dispositivos_ban_list (list): Lista de dispositivos a excluir
del gráfico.
        errores (dict): Diccionario para registrar errores de lectura
de datos.
        tipo_trazado_por_fecha (list): Estilos de línea por fecha.
        color_por_disp (list): Colores por dispositivo.
        tamano_leyenda (int): Tamaño de fuente para la leyenda.
        leyenda_loc (int): Ubicación de la leyenda (1 = fuera de la
gráfica).
```

```
        nombre_archivo (str, opcional): Nombre del archivo a guardar
(sin extensión).
        direccion_guardado (str, opcional): Ruta donde guardar el
archivo resultante (.svg).
"""
def representa_curvas(
    dicc_datos,
    fechas_a_analizar,
    artefacto_a_analizar,
    prueba_a_analizar,
    clase_a_analizar,
    familia_a_analizar,
    dispositivos_ban_list,
    errores,
    tipo_trazado_por_fecha,
    color_por_disp,
    tamano_leyenda,
    leyenda_loc,
    nombre_archivo=None,
    direccion_guardado=None
):

    # Crear figura principal
    fig, ax = plt.subplots()

    # Etiquetas de los ejes según tipo de prueba
    if prueba_a_analizar == "Id(off)-Vds":
        xlabel, ylabel = "Vds (V)", "Id(off) (A)"
    elif prueba_a_analizar == "Rds-Id":
        xlabel, ylabel = "Id (A)", "Rds (Ω)"
    elif prueba_a_analizar == "Id-Vgs":
        xlabel, ylabel = "Vgs (V)", "Id (A)"
    else:
        xlabel, ylabel = "", ""

    # Título del gráfico
    title = f"{prueba_a_analizar} curves for {artefacto_a_analizar} \\
class {clase_a_analizar} \\ family {familia_a_analizar}"

    curvas_por_fecha = []  # Para el control interactivo

    # Redirigir correctamente el subdiccionario a usar
    if artefacto_a_analizar in ["JFET", "PMOS"]:
        dicc_valencia_sub =
dicc_datos["DUT"][clase_a_analizar][artefacto_a_analizar]
    else:
        dicc_valencia_sub = dicc_datos["LCL"][clase_a_analizar]

    for indice_fecha, fecha in enumerate(fechas_a_analizar):
        curvas_fecha = []
        i = 0  # Índice de color
        for clavedisp, valordisp in
dicc_valencia_sub[familia_a_analizar].items():
            # Validaciones
            if fecha not in valordisp[prueba_a_analizar]:
                errores["fecha"].append(f"Falta {clavedisp}
{prueba_a_analizar} {fecha}")
                i += 1
                continue
            if "data" not in valordisp[prueba_a_analizar][fecha]:
```

```
                errores["datos"].append(f"Sin datos {clavedisp}
{prueba_a_analizar} {fecha}")
                i += 1
                continue
            if clavedisp in dispositivos_ban_list:
                i += 1
                continue

            # Obtener datos y graficar
            df = valordisp[prueba_a_analizar][fecha]["data"]
            eje_x, eje_y = df.iloc[:, 0], df.iloc[:, 1]
            color = hacer_transparente_matplotlib(color_por_disp[i],
0.7) if indice_fecha >= 4 else color_por_disp[i]
            label = f"{clavedisp} {fecha}"
            curva, = ax.plot(eje_x, eje_y,
linestyle=tipo_trazado_por_fecha[indice_fecha],
                             color=color, label=label)
            curvas_fecha.append(curva)
            i += 1

        curvas_por_fecha.append(curvas_fecha)

    # Controles interactivos (checkbuttons para activar/desactivar
curvas por fecha)
    rax = plt.axes([0.05, 0.75, 0.1, 0.15])
    visibility = [True] * len(fechas_a_analizar)
    check = CheckButtons(rax, fechas_a_analizar, visibility)

    def func(label):
        index = fechas_a_analizar.index(label)
        for curva in curvas_por_fecha[index]:
            curva.set_visible(not visibility[index])
        visibility[index] = not visibility[index]
        fig.canvas.draw()

    check.on_clicked(func)

    # Ajustes de presentación
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    ax.set_title(title)
    ax.legend(fontsize=tamano_leyenda, loc="best" if leyenda_loc != 1
else "best", bbox_to_anchor=(1, 1) if leyenda_loc == 1 else None)
    ax.grid()
    plt.show(block=True)

    # Guardar figura como SVG si se indica
    if direccion_guardado:
        os.makedirs(direccion_guardado, exist_ok=True)
        ylim_guardado = ax.get_ylim()

        # Segunda figura para guardar (sin interacción)
        fig2, ax2 = plt.subplots()
        plt.subplots_adjust(left=0.2)
        for curvas_fecha in curvas_por_fecha:
            for curva in curvas_fecha:
                ax2.plot(curva.get_xdata(), curva.get_ydata(),
linestyle=curva.get_linestyle(),
                         color=curva.get_color(),
label=curva.get_label())
```

```
        ax2.set_xlabel(xlabel)
        ax2.set_ylabel(ylabel)
        ax2.set_ylim(ylim_guardado)
        ax2.set_title(title, fontsize=10)
        ax2.grid()
        ax2.legend(fontsize=tamano_leyenda, loc="best" if leyenda_loc
!= 1 else "best", bbox_to_anchor=(1, 1) if leyenda_loc == 1 else None)

        if not nombre_archivo:
            incluir_eliminados =
f"_sin_{'_'.join(dispositivos_ban_list)}" if dispositivos_ban_list
else ""
            nombre_archivo =
f"{clase_a_analizar}_{familia_a_analizar}_{prueba_a_analizar}_{artefac
to_a_analizar}{incluir_eliminados}.svg"

        ruta = os.path.join(direccion_guardado, nombre_archivo)
        fig2.savefig(ruta, bbox_inches='tight', dpi=300)
        plt.close(fig2)
        print(f"Gráfico guardado como {nombre_archivo}")



"""
    Genera y prepara los datos para la representación de gráficos tipo
violín normalizados,
    basados en dispositivos agrupados por clase y familia.

    Parámetros:
        dicc_datos (dict): Diccionario con los datos estructurados por
clases, familias y dispositivos.
        fechas_a_analizar (list): Lista de fechas para analizar.
        artefacto_a_analizar (str): Tipo de artefacto ('JFET', 'PMOS'
o 'LCL').
        prueba_a_analizar (str): Prueba específica a representar.
        clases_a_analizar (list): Lista de clases a incluir en la
comparación.
        familia_a_analizar (str): Familia objetivo dentro de cada
clase.
        dispositivos_ban_list (list): Dispositivos que deben excluirse
del análisis.
        errores (dict): Diccionario donde se recogen errores durante
la extracción.
        mostrar_media (bool): Si se deben mostrar líneas de media en
los gráficos.
        percent_norm (float): Porcentaje para el cálculo del valor de
normalización.
        percent_globalpercent_ancho (float): Porcentaje usado para el
ancho del gráfico.
        estructura_datos_percent (dict): Diccionario donde se
almacenan las posiciones y datos del percentil.
"""

def representa_curvas_violin_normaliado_general(dicc_datos,
fechas_a_analizar, artefacto_a_analizar, prueba_a_analizar,
clases_a_analizar, familia_a_analizar, dispositivos_ban_list, errores,
mostrar_media, percent_norm, percent_globalpercent_ancho ,
estructura_datos_percent):
```

```
    #como se ha comentado anteriormente en el apartado ***_4 debemos
tener una variable en la que guardemos el valor máximo del eje x del
primer dispositivo, para usarlo como referencia
    valor_max = 0
    valor_min = 0

    #tendremos dos variables que indicarán la posición para el cual
tenemos el porcentaje del eje x para el normalizador y para los datos
generales
    #se obtendrá para un dispositivo concreto en una fecha concreta y
si los demás están correctos, será la misma posición
    pos_norm = 0

    pos_glo = 0

    #esta variable será una lista que contenga el nombre de los
dispositivos que componen la familia, habrá un primer bucle que
recorra las entradas de clase y familia y lea las claves que hay, de
forma que obtendrá los nombres de los dispositivos dinámicamente
    dispositivos = {  "LCL 2A" : [] , "LCL 10A" : []  }

    #en estas variables guardaremos los nombres de los ejes p. ej. eje
x= Idrain (A)
    ejex = ""
    ejey = ""

    #en esta variable tendremos un diccionario donde cada entrada será
una fecha y el contenido de cada entrada son los datos obtenidos que
se representarán
    datos_fecha = {}
    #esta variable es igual pero guardando los datos del porcentaje
normalizador
    datos_norm_fecha = {}

    #este será el diccioanrio definitivo, formado por los elementos de
los datos_fecha divididos entre la media de datos_norm_fecha para cada
fecha específica
    datos_definitivo = {}

    title = prueba_a_analizar + " violins for " + artefacto_a_analizar
+ " \ class " + f"{'_ '.join( clase.replace(' ', '') for clase in
clases_a_analizar )}" + " \ family " + familia_a_analizar

    #esto será una lista que contendrá texto en cada elemnto,
correspondiente a cada fecha que se analice, mostrando el valor
normalizador
    texto_normalizador = []



    if artefacto_a_analizar == "JFET" or artefacto_a_analizar ==
"PMOS":
        subgrupo_a_analizar = "DUT"
    else:
        subgrupo_a_analizar = artefacto_a_analizar

    #comenzamos con un bucle mediante el que buscaremos los nombres de
los dispositivos para cada clase
    for  clavegrupo, valorgrupo in dicc_datos.items():
```

```
        if clavegrupo != subgrupo_a_analizar:#si no es el grupo,
pasamos
            print("no " + clavegrupo + "\n")
            continue

        for claveclase, valorclase in
dicc_datos[clavegrupo].items():#dentro del grupo, recorremos las
clases
            if claveclase not in clases_a_analizar:#si la clase no es
una de las que vamos a analizar, pasamos
                print("no " + claveclase)
                continue

            #a partir de aquí comienzan las diferencias entre DUT y
LCL, para LCL ya directamente pasas a las familias, en DUT debes
elegir entre JFET o PMOS
            #entonces debemos hacer una redireccion del diccionario


            if artefacto_a_analizar == "JFET" or artefacto_a_analizar
== "PMOS":
                dicc_valencia1_sub =
dicc_datos[clavegrupo][claveclase][artefacto_a_analizar]
            else:
                dicc_valencia1_sub =
dicc_datos[clavegrupo][claveclase]

            for clavefamilia, valorfamilia in
dicc_valencia1_sub.items():#dentro de las clases recorremos las
familias
                if clavefamilia != familia_a_analizar: #si la familia
no es la que queremos analizar, pasamos
                    print("no " + clavefamilia)
                    continue

                for clavedispositivo, valordispositivo in
dicc_valencia1_sub[clavefamilia].items():

                    if clavedispositivo in dispositivos_ban_list: #si
el dispositivo está en la ban list, no lo incluimos
                        print("no " + clavedispositivo)
                        continue


dispositivos[claveclase].append(clavedispositivo)#en el diccioanrio de
dispositivos, para la clase en cuestión, incluimos el nombre de los
dispositivos




    #ahora obtenemos el valor máximo del primer dispositivo de la
primera clase en la primera fecha para usarlo como referencia
    #de manera similar, debemos hacer una separación dependiendo de si
tratamos con LCL o con DUT (JFET/PMOS)
    if artefacto_a_analizar == "JFET" or artefacto_a_analizar ==
"PMOS":
        df= dicc_datos[ subgrupo_a_analizar ][ clases_a_analizar[0]
][artefacto_a_analizar][ familia_a_analizar ][
```

```
dispositivos[clases_a_analizar[0]][0] ][  prueba_a_analizar
][fechas_a_analizar[0]]["data"]

    else:
        df= dicc_datos[ subgrupo_a_analizar ][ clases_a_analizar[0] ][
familia_a_analizar ][ dispositivos[clases_a_analizar[0]][0] ][
prueba_a_analizar  ][fechas_a_analizar[0]]["data"]



    #el valor máximo del eje x será siempre el de la primera columna
    valor_max = df[df.columns[0]].max()
    valor_min = df[df.columns[0]].min()

    if valor_max + valor_min <= 0:#si la suma del valor maximo y del
minimo es negativo es que estamos ante una secuencia negativa
        percent_global = 100 - percent_global
        percent_norm  = 100 - percent_norm

    primer_valor = df.iloc[0, 0]  # Primera fila, primera columna
    ultimo_valor = df.iloc[-1, 0]  # Última fila, primera columna

    print("\nEl valor máximo para la prueba de " + prueba_a_analizar +
" es " + str(valor_max) + "\n")

    print("\nEl valor mínimo para la prueba de " + prueba_a_analizar +
" es " + str(valor_min) + "\n")

    print("\nEl primer valor para la prueba de " + prueba_a_analizar +
" es " + str(primer_valor) + "\n")

    print("\nEl último valor para la prueba de " + prueba_a_analizar +
" es " + str(ultimo_valor) + "\n")

    ejex = df.columns[0] #nombre de la columna 0
    ejey = df.columns[1] #nombre de la columna 1

    print("Para la prueba de " + prueba_a_analizar + " el label del
eje x es: " + ejex + " y el del eje y: " + ejey)



    norm_sub_ref_inferior = valor_min + ((percent_norm -
percent_ancho/2)/100) * (valor_max-valor_min)

    norm_sub_ref_superior = valor_min + ((percent_norm +
percent_ancho/2)/100) * (valor_max-valor_min)


    #ahora obtenemos la posición del porcentaje del normalizador, p.
ej. si es el 75%, buscamos un dato del eje x que se acerque al 75% del
valor maximo

    if valor_max + valor_min >= 0:

        if primer_valor < ultimo_valor: #si la secuencia empeza en el
0 buscamos el primer indice que cumpla
            pos_norm_inferior = df[ df.iloc[:, 0] >=
norm_sub_ref_inferior ].index[0]
        else:# si no, buscamos el ultimo, ya que el primero será el
índice 0
```

```
            pos_norm_inferior = df[ df.iloc[:, 0] >=
norm_sub_ref_inferior ].index[-1]

        if primer_valor < ultimo_valor: #si la secuencia empeza en el
0 buscamos el primer indice que cumpla
            pos_norm_superior = df[ df.iloc[:, 0] >=
norm_sub_ref_superior ].index[0]
        else:# si no, buscamos el ultimo, ya que el primero será el
índice 0
            pos_norm_superior = df[ df.iloc[:, 0] >=
norm_sub_ref_superior ].index[-1]


    else:

        if primer_valor > ultimo_valor:# si la secuencia empieza en el
0 buscamos el primer indice que cumpla
            pos_norm_inferior = df[ df.iloc[:, 0] <=
norm_sub_ref_inferior ].index[0]
        else:
            pos_norm_inferior = df[ df.iloc[:, 0] <=
norm_sub_ref_inferior ].index[-1]

        if primer_valor > ultimo_valor:# si la secuencia empieza en el
0 buscamos el primer indice que cumpla
            pos_norm_superior = df[ df.iloc[:, 0] <=
norm_sub_ref_superior ].index[0]
        else:
            pos_norm_superior = df[ df.iloc[:, 0] <=
norm_sub_ref_superior ].index[-1]



    print("La posición para la cual se alcanza el " + str(percent_norm
- percent_ancho/2) + "_" + str(percent_norm + percent_ancho/2) + "% de
los datos (" + str(norm_sub_ref_inferior) + "_" +
str(norm_sub_ref_superior) + ") es " + str(pos_norm_inferior)+ "_" +
str(pos_norm_superior)  + " que corresponden al intervalo del valor
normalizador")


    #ahora obtenemos la posición del porcentaje del general, p. ej. si
es el 75%, buscamos un dato del eje x que se acerque al 75% del valor
maximo


    norm_glo_ref_inferior = valor_min + ((percent_global -
percent_ancho/2)/100) * (valor_max-valor_min)

    norm_glo_ref_superior = valor_min + ((percent_global +
percent_ancho/2)/100) * (valor_max-valor_min)


    if valor_max + valor_min >= 0:

        if primer_valor < ultimo_valor: #si la secuencia empeza en el
0 buscamos el primer indice que cumpla
            pos_glo_inferior = df[ df.iloc[:, 0] >=
norm_glo_ref_inferior].index[0]
```

```
        else:# si no, buscamos el ultimo, ya que el primero será el
índice 0
            pos_glo_inferior = df[ df.iloc[:, 0] >=
norm_glo_ref_inferior ].index[-1]

        if primer_valor < ultimo_valor: #si la secuencia empeza en el
0 buscamos el primer indice que cumpla
            pos_glo_superior = df[ df.iloc[:, 0] >=
norm_glo_ref_superior].index[0]
        else:# si no, buscamos el ultimo, ya que el primero será el
índice 0
            pos_glo_superior = df[ df.iloc[:, 0] >=
norm_glo_ref_superior ].index[-1]

    else:

        if primer_valor > ultimo_valor:# si la secuencia empieza en el
0 buscamos el primer indice que cumpla
            pos_glo_inferior = df[ df.iloc[:, 0] <=
norm_glo_ref_inferior ].index[0]
        else:
            pos_glo_inferior = df[ df.iloc[:, 0] <=
norm_glo_ref_inferior ].index[-1]

        if primer_valor > ultimo_valor:# si la secuencia empieza en el
0 buscamos el primer indice que cumpla
            pos_glo_superior = df[ df.iloc[:, 0] <=
norm_glo_ref_superior ].index[0]
        else:
            pos_glo_superior = df[ df.iloc[:, 0] <=
norm_glo_ref_superior ].index[-1]

    print("La posición para la cual se alcanza el " +
str(percent_global - percent_ancho/2) + "_" + str(percent_global +
percent_ancho/2) + "% de los datos (" + str(norm_glo_ref_inferior) +
"_" + str(norm_glo_ref_superior) + ") es " + str(pos_glo_inferior)+
"_" + str(pos_glo_superior)  + " que corresponden al intervalo del
valor global")

    print("\n")

    #Ahora ya lo tenemos todo para coomenzar, iremos de fecha en fecha
    for fecha in fechas_a_analizar:



        datos_fecha[fecha] = []      #creamos la lista para la fecha
indicada
        datos_norm_fecha[fecha] = []

        #Ahora haremos un bucle para recorrer cada dispositivo,
gracias a la variable "dispositivos" , con la que además sabemos su
clase

        for clasedisp, valordisp in dispositivos.items():


            #gracias a "clasedisp" sabremos la clase, y en "valordisp"
tenemos la lista de dispositivos de esa clase

            for disp in valordisp:
```

```
                #de manera similar, debemos hacer una separación
dependiendo de si tratamos con LCL o con DUT (JFET/PMOS)
                if artefacto_a_analizar == "JFET" or
artefacto_a_analizar == "PMOS":
                    dic= dicc_datos[ subgrupo_a_analizar ][ clasedisp
][artefacto_a_analizar][ familia_a_analizar ][ disp ][
prueba_a_analizar  ]

                else:
                    dic= dicc_datos[ subgrupo_a_analizar ][ clasedisp
][ familia_a_analizar ][ disp ][  prueba_a_analizar  ]


                #necesitamos comprobar que el dispositivo tiene datos
para la prueba en la fecha indicada, si no, se pasa y se añade a la
avriable de error
                if fecha not in list(dic.keys() ) or "data" not in
list(dic[fecha].keys() ):
                    errores["fecha"].append("No existe " + disp + " en
fecha " + fecha + " en prueba " + prueba_a_analizar)
                    print("no existe " + disp + " en fecha " + fecha +
" en prueba " + prueba_a_analizar)
                    continue




                #ahora necesitamos comprobar que el valor maximo del
eje x coincide con el obtenido anteriormente
                #de manera similar, debemos hacer una separación
dependiendo de si tratamos con LCL o con DUT (JFET/PMOS)
                if artefacto_a_analizar == "JFET" or
artefacto_a_analizar == "PMOS":
                    df= dicc_datos[ subgrupo_a_analizar ][ clasedisp
][artefacto_a_analizar][ familia_a_analizar ][ disp ][
prueba_a_analizar  ][fecha]["data"]

                else:
                    df= dicc_datos[ subgrupo_a_analizar ][ clasedisp
][ familia_a_analizar ][ disp ][  prueba_a_analizar  ][fecha]["data"]


                disp_max = df[df.columns[0]].max()

                if disp_max < norm_sub_ref_superior or disp_max <
norm_glo_ref_superior:
                    errores["valor_maximo"].append(  "el valor maximo
del dispositivo " + disp + " en la fecha " + fecha + " es inferior a "
+ str(( valor_max - (5*abs(valor_max-valor_min))/100)) + " (valor_max
-5%valor_max) o superior a " + str(( valor_max + (5*abs(valor_max-
valor_min))/100)) )
                    print(  "el valor maximo del dispositivo " + disp
+ " es inferior a " + str(( valor_max - (5*valor_max)/100)) + "
(valor_max -5%valor_max) o superior a " + str(( valor_max +
(5*valor_max)/100)))
                    continue

                norm_sub_inferior = valor_min + ((percent_norm -
percent_ancho/2)/100) * (valor_max-valor_min)
```

```
                norm_sub_superior = valor_min + ((percent_norm +
percent_ancho/2)/100) * (valor_max-valor_min)


                #ahora obtenemos la posición del porcentaje del
normalizador, p. ej. si es el 75%, buscamos un dato del eje x que se
acerque al 75% del valor maximo

                if valor_max + valor_min >= 0:

                    if primer_valor < ultimo_valor: #si la secuencia
empeza en el 0 buscamos el primer indice que cumpla
                        pos_norm_inferior = df[ df.iloc[:, 0] >=
norm_sub_inferior ].index[0]
                    else:# si no, buscamos el ultimo, ya que el
primero será el índice 0
                        pos_norm_inferior = df[ df.iloc[:, 0] >=
norm_sub_inferior ].index[-1]

                    if primer_valor < ultimo_valor: #si la secuencia
empeza en el 0 buscamos el primer indice que cumpla
                        pos_norm_superior = df[ df.iloc[:, 0] >=
norm_sub_superior ].index[0]
                    else:# si no, buscamos el ultimo, ya que el
primero será el índice 0
                        pos_norm_superior = df[ df.iloc[:, 0] >=
norm_sub_superior ].index[-1]


                else:

                    if primer_valor > ultimo_valor:# si la secuencia
empieza en el 0 buscamos el primer indice que cumpla
                        pos_norm_inferior = df[ df.iloc[:, 0] <=
norm_sub_inferior ].index[0]
                    else:
                        pos_norm_inferior = df[ df.iloc[:, 0] <=
norm_sub_inferior ].index[-1]

                    if primer_valor > ultimo_valor:# si la secuencia
empieza en el 0 buscamos el primer indice que cumpla
                        pos_norm_superior = df[ df.iloc[:, 0] <=
norm_sub_superior ].index[0]
                    else:
                        pos_norm_superior = df[ df.iloc[:, 0] <=
norm_sub_superior ].index[-1]


                valor_intervalo_norm =
df.iloc[pos_norm_inferior:pos_norm_superior+1, 1].mean()


                datos_norm_fecha[fecha].append(  valor_intervalo_norm
)


                #ahora obtenemos la posición del porcentaje del
general, p. ej. si es el 75%, buscamos un dato del eje x que se
acerque al 75% del valor maximo
```

```
                norm_glo_inferior = valor_min + ((percent_global -
percent_ancho/2)/100) * (valor_max-valor_min)

                norm_glo_superior = valor_min + ((percent_global +
percent_ancho/2)/100) * (valor_max-valor_min)


                if valor_max + valor_min >= 0:

                    if primer_valor < ultimo_valor: #si la secuencia
empeza en el 0 buscamos el primer indice que cumpla
                        pos_glo_inferior = df[ df.iloc[:, 0] >=
norm_glo_inferior].index[0]
                    else:# si no, buscamos el ultimo, ya que el
primero será el índice 0
                        pos_glo_inferior = df[ df.iloc[:, 0] >=
norm_glo_inferior ].index[-1]

                    if primer_valor < ultimo_valor: #si la secuencia
empeza en el 0 buscamos el primer indice que cumpla
                        pos_glo_superior = df[ df.iloc[:, 0] >=
norm_glo_superior].index[0]
                    else:# si no, buscamos el ultimo, ya que el
primero será el índice 0
                        pos_glo_superior = df[ df.iloc[:, 0] >=
norm_glo_superior ].index[-1]

                else:

                    if primer_valor > ultimo_valor:# si la secuencia
empieza en el 0 buscamos el primer indice que cumpla
                        pos_glo_inferior = df[ df.iloc[:, 0] <=
norm_glo_inferior ].index[0]
                    else:
                        pos_glo_inferior = df[ df.iloc[:, 0] <=
norm_glo_inferior ].index[-1]

                    if primer_valor > ultimo_valor:# si la secuencia
empieza en el 0 buscamos el primer indice que cumpla
                        pos_glo_superior = df[ df.iloc[:, 0] <=
norm_glo_superior ].index[0]
                    else:
                        pos_glo_superior = df[ df.iloc[:, 0] <=
norm_glo_superior ].index[-1]

                valor_intervalo =
df.iloc[pos_glo_inferior:pos_glo_superior+1, 1].mean()



                datos_fecha[fecha].append(  valor_intervalo )

                if artefacto_a_analizar == "JFET" or
artefacto_a_analizar == "PMOS":


                    estructura_datos_percent[ subgrupo_a_analizar ][
clasedisp ][artefacto_a_analizar][ familia_a_analizar ][ disp ][
prueba_a_analizar  ][fecha]=   valor_intervalo

                else:
```

```
                    estructura_datos_percent[ subgrupo_a_analizar ][
clasedisp ][ familia_a_analizar ][ disp ][  prueba_a_analizar
][fecha]=  valor_intervalo



    #ahora pasamos al tema de la gráfica de violín
    #primero obtenemos la variable datos_definitivo, lista con la
misma longitud que la lista de fechas y el mismo orden

    #=========================
    #la principal diferencia está aquí, pues normalizaremos todos los
datos con respecto a la primera fecha
    #=========================
    print("\n")

    if percent_ancho == 0:

        texto_intervalo = "(" + str(percent_global - percent_ancho/2)
+  "-" +str(percent_global + percent_ancho/2) + ")%"

    else:
        texto_intervalo =  "(" + str(percent_global - percent_ancho/2)
+  "-" +str(percent_global + percent_ancho/2) + ")%"

    #ahora pasamos al tema de la gráfica de violín
    #primero obtenemos la variable datos_definitivo
    print("\n")
    for clavefecha, valorfecha in datos_norm_fecha.items():
        media = np.mean(datos_norm_fecha[clavefecha])


        print(datos_norm_fecha[clavefecha])

        #rellenamos los textos de las fechas_a_analizar

        texto_normalizador.append( clavefecha + " "  + ejey +
r'$_{norm}$' + texto_intervalo + "=" + str(media) )

        print(media)
        datos_definitivo[ clavefecha ] = [x / media for x in
datos_fecha[clavefecha]]


    print(datos_definitivo)


    #Transformar el diccionario en un formato adecuado para seaborn
(long-form DataFrame).


    # Crear el DataFrame a partir del diccionario (suponiendo que
datos_definitivo es un diccionario)
    df = pd.DataFrame([(fecha, valor) for fecha, valores in
datos_definitivo.items() for valor in valores],
                      columns=["Fecha", "Valor"])

    # Calcular la media por fecha
```

```
    mean_values = df.groupby("Fecha")["Valor"].mean()

    # Crear la gráfica de violín
    plt.figure(figsize=(10, 6))
    sns.violinplot(x="Fecha", y="Valor", data=df)

    # Agregar la media como puntos rojos

    if mostrar_media == 1:
        for i, fecha in enumerate(df["Fecha"].unique()):
            plt.scatter(i, mean_values[fecha], color='red',
label="Media" if i == 0 else "", zorder=5)

    # Personalización
    plt.title(title + texto_intervalo, fontsize=16)
    plt.xlabel("fechas_a_analizar", fontsize=12)
    plt.ylabel(r'$\frac{ '+ ejey + ' }{' + ejey +'_{norm}}$',
fontsize=18)
    plt.grid(True)

    texto_definitivo = '\n'.join( clase for clase in
texto_normalizador )

    plt.text(0.02, 0.98, texto_definitivo, fontsize=8, ha='left',
va='top',
             color='red', weight='bold',  # Negrita y color rojo
             bbox=dict(facecolor='white', edgecolor='black',
linewidth=0.5, boxstyle='round,pad=0.5', alpha=0.5),
             transform=plt.gca().transAxes)  # Usa coordenadas
relativas


    # Mostrar la leyenda
    plt.legend()

    if len(dispositivos_ban_list) != 0:
        incluir_eliminados= "_sin_" + '_'.join(dispositivos_ban_list)
    else:
        incluir_eliminados = ""

    nombre_archivo = f"{'_'.join( clase.replace(' ', '') for clase in
clases_a_analizar )}" + "_" + familia_a_analizar + "_" +
prueba_a_analizar + "_" + artefacto_a_analizar + "_norm_" +
texto_intervalo + "_" + incluir_eliminados +  "_violinplot.svg"
    plt.savefig(nombre_archivo, bbox_inches='tight', dpi=300)
```

Graficado de derivada de Id-Vgs y obtención de Vth

```
import pandas as pd
import plotly.graph_objects as go
import matplotlib.pyplot as plt
from matplotlib.widgets import CheckButtons
import numpy as np
import seaborn as sns
import os


"""

 IMPORTANTE: Se asume que en el workspace ya tienes definido el
diccionario
 dicc_valencia_sub con la siguiente estructura:

 dicc_valencia_sub = {
     familia_a_usar: {
         "Dispositivo1": {
             prueba_a_analizar: {
                 "enero_marzo_23": {"data": DataFrame},
                 "sept_23": {"data": DataFrame},
                 ...
             }
         },
         "Dispositivo2": { ... },
         ...
     }
 }

 Asegúrate de que esta variable esté definida antes de ejecutar el
código.
"""
import numpy as np                          # Importa NumPy para
cálculos numéricos
import pandas as pd                         # Importa pandas para
manejar DataFrames
import matplotlib.pyplot as plt             # Importa matplotlib para
graficar
from scipy.signal import savgol_filter      # Importa el filtro
Savitzky-Golay desde scipy.signal
from matplotlib.widgets import CheckButtons
import matplotlib.colors as mcolors
import colorsys
import matplotlib.ticker as ticker


import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
import colorsys

"""
    Convierte un color base a formato RGBA compatible con Plotly,
    incorporando un nivel de transparencia especificado.

    Parámetros:
        color (str o tuple): Color base en formato aceptado por
Matplotlib.
        alpha (float): Valor de transparencia (0 completamente
transparente, 1 opaco).
```

```
    Retorna:
        str: Cadena en formato RGBA adecuada para uso con Plotly.
"""
def hacer_transparente_plotly(color, alpha=0.3):

    rgba = plt.cm.colors.to_rgba(color)  # Convertir el color a
formato RGBA
    rgba_transparente = f"rgba({int(rgba[0] * 255)}, {int(rgba[1] *
255)}, {int(rgba[2] * 255)}, {alpha})"
    return rgba_transparente

"""
    Ajusta la transparencia de un color para su uso con Matplotlib.

    Parámetros:
        color (str o tuple): Color base en formato aceptado por
Matplotlib.
        alpha (float): Valor de transparencia (0 completamente
transparente, 1 opaco).

    Retorna:
        tuple: Tupla RGBA con la componente alfa modificada.
"""
def hacer_transparente_matplotlib(color, alpha=0.3):

    rgba = plt.cm.colors.to_rgba(color)  # Obtener RGBA desde el color
original
    rgba_transparente = rgba[:3] + (alpha,)  # Reemplazar la
componente alfa
    return rgba_transparente

"""
    Convierte un color a su versión pastel ajustando su saturación y
luminosidad.

    Parámetros:
        color (str o tuple): Color original en formato RGB o cadena
reconocida.
        factor (float): Factor de atenuación de la saturación (entre 0
y 1).

    Retorna:
        tuple: Color transformado en formato RGB con efecto pastel.
"""
def to_pastel(color, factor=0.5):

    # Convertir a formato RGB normalizado (valores entre 0 y 1)
    rgb = mcolors.to_rgb(color)

    # Transformar de RGB a HLS (Hue, Lightness, Saturation)
    h, l, s = colorsys.rgb_to_hls(*rgb)

    # Aplicar reducción de saturación y aumento de luminosidad
    pastel_rgb = colorsys.hls_to_rgb(h, min(1, l + factor * (1 - l)),
s * factor)

    return pastel_rgb

"""
    Aplica la conversión a pastel a una lista de colores.
```

```
    Parámetros:
        color_list (list): Lista de colores en formato válido para
Matplotlib.
        factor (float): Factor de atenuación de saturación.

    Retorna:
        list: Lista de colores en formato pastel (RGB).
"""
def convert_colors_to_pastel(color_list, factor=0.5):

    return [to_pastel(color, factor) for color in color_list]


"""
    Representa gráficamente las curvas Id-Vgs y sus derivadas para
diferentes dispositivos,
    estimando el Vth (tensión de umbral) mediante el método de la
recta.

    Se generan tres tipos de gráficas:
    - Curvas Id-Vgs con las rectas extendidas e intersecciones (Vth).
    - Derivadas gm-Vgs, destacando el segmento de pendiente constante.
    - Barras comparativas de Vth para cada fecha y dispositivo.

    Parámetros:
        dicc_datos (dict): Diccionario con la estructura jerárquica de
datos extraídos.
        fechas_a_analizar (list): Lista de fechas a visualizar.
        artefacto_a_analizar (str): Artefacto a estudiar ('JFET',
'PMOS', 'LCL').
        clase_a_analizar (str): Clase de corriente a analizar (e.g.,
'2A').
        familia_a_analizar (str): Familia de dispositivos a
representar.
        dispositivos_ban_list (list): Dispositivos a excluir del
análisis.
        errores (dict): Diccionario para almacenar errores
encontrados.
        tipo_trazado_por_fecha (list): Estilos de línea por fecha.
        color_por_disp (list): Colores asociados a cada dispositivo.
        tamano_leyenda (int): Tamaño del texto de la leyenda.
        leyenda_loc (int): Posición de la leyenda (1 para fuera del
gráfico).
        window_length (int): Longitud de ventana para el filtro de
Savitzky-Golay.
        polyorder (int): Orden del polinomio para el filtro de
Savitzky-Golay.
        tol (float): Tolerancia máxima en la variación de la derivada
para detectar tramos constantes.
        min_slope (float): Pendiente mínima aceptable para considerar
el tramo válido.
        invert_yaxis_barra (int): Si es 1, invierte el eje Y en la
gráfica de barras.

"""
def representa_curvas_vth_metodo_recta(dicc_datos, fechas_a_analizar,
artefacto_a_analizar, clase_a_analizar, familia_a_analizar,
dispositivos_ban_list, errores, tipo_trazado_por_fecha,
color_por_disp, tamano_leyenda, leyenda_loc, window_length, polyorder,
tol, min_slope, invert_yaxis_barra):
```

```
    prueba_a_analizar = "Id-Vgs"
    xlabel = "Vgs (V)"
    ylabel = "Id (A)"

    prueba_a_analizar_derivada = "gm-Vgs"
    xlabel_derivada = "Vgs (V)"
    ylabel_derivada = "gm (s)"

    # Crear la figura para graficar las curvas originales (cada una
con su recta extendida e intersección)
    fig_curvas, ax_curvas = plt.subplots(figsize=(12, 8))
    # Crear la figura para graficar las derivadas (con el tramo
constante resaltado)
    fig_derivadas, ax_derivadas = plt.subplots(figsize=(12, 8))



    title = prueba_a_analizar + " curves for " + artefacto_a_analizar
+ " \ class " + clase_a_analizar + " \ family " + familia_a_analizar

    title_derivada = prueba_a_analizar_derivada + " curves for " +
artefacto_a_analizar + " \ class " + clase_a_analizar + " \ family " +
familia_a_analizar

    title_barra = "Vth" + " bars for " + artefacto_a_analizar + " \
class " + clase_a_analizar + " \ family " + familia_a_analizar


    # Lista para almacenar las curvas por fecha
    curvas_por_fecha = []

    #lista de listas para almacenar los puntos de interseccion de las
rectas
    vth = []

    #==PARTE DE OBTENER LAS CRUVAS
    #=======================================


    # --- Aquí se asume que dicc_datos existe y contiene tus datos ---
    if artefacto_a_analizar == "JFET" or artefacto_a_analizar ==
"PMOS":
        dicc_valencia_sub =
dicc_datos["DUT"][clase_a_analizar][artefacto_a_analizar]
    elif artefacto_a_analizar == "LCL":
        dicc_valencia_sub = dicc_datos["LCL"][clase_a_analizar]
    else:
        dicc_valencia_sub = dicc_datos["LCL"][clase_a_analizar]
        print("Se ha introducido incorrectamente la variable
artefacto_a_utilizar, se usará 'LCL' ")

    # Recorrer las fechas y los dispositivos
    for indice_fecha, fecha in enumerate(fechas_a_analizar):
        curvas_fecha = []
        derivadas_fecha = []

        vth_sub = []


        cont = 0
```

```
        for clavedisp, valordisp in
dicc_valencia_sub[familia_a_analizar].items():



            if fecha not in
dicc_valencia_sub[familia_a_analizar][clavedisp][prueba_a_analizar]:
                vth_sub.append(0)
                cont += 1 #de esta forma los colores son consistentes
                errores["fecha"].append("No existen datos para el
dispositivo " + clavedisp + " para la prueba " + prueba_a_analizar + "
en la fecha " + fecha)
                continue

            if "data" not in
dicc_valencia_sub[familia_a_analizar][clavedisp][prueba_a_analizar][fe
cha]:
                vth_sub.append(0)
                cont += 1 #de esta forma los colores son consistentes
                errores["datos"].append("No existen datos para el
dispositivo " + clavedisp + " para la prueba " + prueba_a_analizar + "
en la fecha " + fecha)
                continue

            if clavedisp in dispositivos_ban_list:
                vth_sub.append(0)
                cont += 1 #de esta forma los colores son consistentes
                continue



            if indice_fecha == 4:
                color =
hacer_transparente_matplotlib(color_por_disp[cont], 0.3)
            else:

                color = color_por_disp[cont]

            df =
dicc_valencia_sub[familia_a_analizar][clavedisp][prueba_a_analizar][fe
cha]["data"]
            eje_x = df.iloc[:, 0]
            eje_y = df.iloc[:, 1]


            # Calcular el espaciado promedio entre puntos en X (usado
como 'delta' en el filtro)
            dx = np.mean(np.diff(eje_x))

            # Calcular la derivada de Y usando el filtro Savitzky-
Golay
            dy = savgol_filter(eje_y, window_length, polyorder,
deriv=1, delta=dx)


            # ----------------------------
            # DETECCIÓN DE SEGMENTOS EN LA DERIVADA
            # ----------------------------
```

```
            #Se buscarán segmentos en la derivada suavizada cuya
diferencia entre ellos sea menor a una tolerancia dada, de esta forma,
si el segmento en la derivada
            #es aproximadamente horizontal, sus diferencias no serán
muy notorias

            segmentos = []                      # Lista para almacenar
grupos de índices (segmentos), es decir, será una lista de listas,
donde cada lista contiene los índices de los segmentos
            segmento_actual = [0]               # Lista para almacenar
los índices del segmento actual ,Inicializar el primer segmento con el
índice 0

            # Recorrer la derivada para agrupar índices consecutivos
donde la variación es menor a 'tol'
            for i in range(len(dy) - 1):
                if abs(dy[i+1] - dy[i]) < tol:  # Si la diferencia
entre derivadas consecutivas es menor a 'tol'
                    segmento_actual.append(i+1) # Añadir el siguiente
índice al segmento actual
                else:
                    if len(segmento_actual) > 1:  # Si el segmento
actual tiene más de un punto
                        segmentos.append(segmento_actual)  # Guardarlo
en la lista de segmentos
                    segmento_actual = [i+1]       # Reiniciar el
segmento actual con el siguiente índice
            # Al finalizar el bucle, agregar el último segmento si
tiene más de un punto
            if len(segmento_actual) > 1:
                segmentos.append(segmento_actual)

            # Filtrar los segmentos para eliminar aquellos que
representen una recta horizontal en la curva original
            #es decir, los segmentos horizontales de la derivada
suavizada cuyo valor es cercano a 0 ; pendiente 0 --> recta horizontal
            segmentos_validos = []              # Lista para almacenar
segmentos válidos
            for seg in segmentos:
                mean_slope = np.mean(dy[seg])     # Calcular la
pendiente promedio del segmento
                if abs(mean_slope) >= min_slope:    # Verificar que
sea mayor o igual al umbral 'min_slope'
                    segmentos_validos.append(seg) # Si cumple, se
agrega a la lista de segmentos válidos

            # Seleccionar el segmento válido más largo (si existe)
            if not segmentos_validos:
                print( "Para la fecha " + fecha + " con el dispositivo
"+ clavedisp + " no se encontró un segmento inclinado que cumpla los
criterios.")
                tramo_indices = None
            else:
                tramo_indices = max(segmentos_validos, key=len)  #
Seleccionar el segmento con más puntos


            # ---------------------------
            # GRAFICADO DE LA CURVA ORIGINAL Y LA RECTA EXTENDIDA
            # ---------------------------
```

```
            # En la gráfica de curvas, trazar la curva original
            curva, = ax_curvas.plot(eje_x, eje_y, label=f"{clavedisp}
{fecha}" , linestyle=tipo_trazado_por_fecha[indice_fecha],
                                    color=color)

            # En la gráfica de derivadas, trazar la derivada calculada
            derivada, = ax_derivadas.plot(eje_x, dy, label =
f"{clavedisp} {fecha}" ,
linestyle=tipo_trazado_por_fecha[indice_fecha],
                                    color=color)

            # Si se encontró un segmento válido, se calcula y traza la
recta extendida
            if tramo_indices is not None:
                # Obtener el primer y último índice del segmento
detectado
                idx_inicio = tramo_indices[0]
                idx_fin = tramo_indices[-1]
                # Extraer los puntos correspondientes de la curva
original
                x1, y1 = eje_x[idx_inicio], eje_y[idx_inicio]
                x2, y2 = eje_x[idx_fin], eje_y[idx_fin]

                # Evitar la división por cero si los puntos tienen el
mismo valor en X
                if x2 == x1:
                    print( "Para la fecha " + fecha + " con el
dispositivo "+ clavedisp + " Los extremos tienen el mismo valor en X,
se omite la recta.")
                    continue

                # Calcular la pendiente (m) de la recta que une los
dos extremos
                m = (y2 - y1) / (x2 - x1)
                # Calcular la ordenada al origen (b) usando la
fórmula: y = m*x + b
                b = y1 - m * x1

                # Calcular el punto de intersección de la recta
extendida con el eje X (donde y = 0)
                x_int = -b / m
                # (El punto de intersección es (x_int, 0))

                #guardamos este punto
                vth_sub.append(x_int)

                # Definir el rango en X para trazar la recta
extendida: incluir ambos extremos y el punto de corte
                x_min_line = min(x1, x2, x_int)
                x_max_line = max(x1, x2, x_int)
                x_line = np.linspace(x_min_line, x_max_line, 100)  #
Genera 100 puntos entre el mínimo y el máximo
                y_line = m * x_line + b            # Calcula los
valores de Y correspondientes en la recta

                # En la gráfica de curvas:
                # Trazar la recta extendida en línea discontinua roja
                ax_curvas.plot(x_line, y_line, linestyle='--',
color='black', alpha = 0.3)
                # Marcar los extremos del segmento detectado con
puntos rojos
```

```
                ax_curvas.scatter([x1, x2], [y1, y2], color='black',
alpha = 0.3)
                # Marcar el punto de intersección con el eje X en
color magenta
                ax_curvas.scatter(x_int, 0, color='magenta', s=100,
alpha = 0.3)

                # En la gráfica de derivadas:
                # Resaltar el tramo constante detectado trazándolo en
línea discontinua roja
                ax_derivadas.plot(eje_x[tramo_indices],
dy[tramo_indices], linestyle='--', color='black', alpha = 0.3)




            if leyenda_loc == 1:
                ax_curvas.legend(fontsize=tamano_leyenda, loc =
"best",bbox_to_anchor=(1, 1))
                ax_derivadas.legend(fontsize=tamano_leyenda, loc =
"best",bbox_to_anchor=(1, 1))
            else:
                ax_curvas.legend(fontsize=tamano_leyenda, loc =
"best")
                ax_derivadas.legend(fontsize=tamano_leyenda, loc =
"best")


            curvas_fecha.append(curva)
            derivadas_fecha.append(derivada)

            cont += 1

        #actualziamos la lista de listas
        vth.append(vth_sub)


        curvas_por_fecha.append(curvas_fecha)

    #=======================================
    #=======================================


    #====PARTE DE GRAFICAR TODAS LAS CURVAS OBTENIDAS
    #=======================================


    ax_curvas.set_title(title)
    ax_curvas.set_xlabel(xlabel)
    ax_curvas.set_ylabel(ylabel)

    ax_derivadas.set_title(title_derivada)
    ax_derivadas.set_xlabel(xlabel_derivada)
    ax_derivadas.set_ylabel(ylabel_derivada)

    # Crear los checkbuttons para cada fecha
    rax = fig_curvas.add_axes([0.05, 0.75, 0.1, 0.15])
    labels = fechas_a_analizar

    visibility = [True] * len(fechas_a_analizar)
```

```
    check = CheckButtons(rax, labels, visibility)

    def func(label):
        index = labels.index(label)
        if visibility[index]:
            for curva in curvas_por_fecha[index]:
                curva.set_visible(False)
        else:
            for curva in curvas_por_fecha[index]:
                curva.set_visible(True)
        visibility[index] = not visibility[index]
        fig_curvas.canvas.draw()

    check.on_clicked(func)

    ax_curvas.grid()

    plt.grid(True)

    #la gráfica de derivadas la guardamos directamente
    if len(dispositivos_ban_list) != 0:
        incluir_eliminados= "_sin_" + '_'.join(dispositivos_ban_list)
    else:
        incluir_eliminados = ""
    nombre_archivo = clase_a_analizar + "_" + familia_a_analizar + "_"
+ prueba_a_analizar + "_"  + artefacto_a_analizar + incluir_eliminados
+ "_derivadas" +  ".svg"
    print(nombre_archivo)
    fig_derivadas.savefig(nombre_archivo, bbox_inches='tight',
dpi=300)

    if leyenda_loc == 1:
        ax_derivadas.legend(fontsize=tamano_leyenda, loc =
"best",bbox_to_anchor=(1, 1))
    else:
        ax_derivadas.legend(fontsize=tamano_leyenda, loc = "best")

    # Mostrar la gráfica interactiva (la ejecución se bloqueará hasta
cerrar la ventana)
    plt.show(block=True)

    #=======================================
    #=======================================

    #====PARTE DE VOLVER A GRAFICAR LAS CURVAS NORMALES
    #=======================================

    # Una vez cerrada la ventana, se espera que xlim_guardado y
ylim_guardado tengan los valores modificados

    ylim_guardado = ax_curvas.get_ylim()

    # Ajustar la posición de la gráfica


    # Crear la segunda figura
    fig2, ax2 = plt.subplots()

    plt.subplots_adjust(left=0.2)

    # Copiar las curvas de la figura original
```

```
    for curvas_fecha in curvas_por_fecha:
        for curva in curvas_fecha:
            xdata = curva.get_xdata()
            ydata = curva.get_ydata()
            linestyle = curva.get_linestyle()
            color = curva.get_color()
            label = curva.get_label()
            ax2.plot(xdata, ydata, linestyle=linestyle, color=color,
label=label)

    ax2.set_xlabel(xlabel)
    ax2.set_ylabel(ylabel)
    if leyenda_loc == 1:
        ax2.legend(fontsize=tamano_leyenda, loc =
"best",bbox_to_anchor=(1, 1))
    else:
        ax2.legend(fontsize=tamano_leyenda, loc = "best")

    ax2.set_ylim(ylim_guardado)
    ax2.set_title(title, fontsize=10)
    ax2.grid()

    if len(dispositivos_ban_list) != 0:
        incluir_eliminados= "_sin_" + '_'.join(dispositivos_ban_list)
    else:
        incluir_eliminados = ""

    nombre_archivo = clase_a_analizar + "_" + familia_a_analizar + "_"
+ prueba_a_analizar + "_"  + artefacto_a_analizar + incluir_eliminados
+  ".svg"
    fig2.savefig(nombre_archivo, bbox_inches='tight', dpi=300)
    plt.close(fig2)
    print(f"Gráfico guardado como {nombre_archivo}")

    #=======================================
    #=======================================



    #==============PARTE VTH GRÁFICA DE BARRAS
    #=======================================

    # Definir el número de grupos
    num_grupos = len(vth)

    # Crear la figura y los ejes
    fig3, ax3 = plt.subplots(figsize=(10, 6))

    # Inicializar la posición de las barras en el eje X
    indices = np.arange(num_grupos)

    # Ancho de las barras: lo ajustamos en función de la cantidad de
barras en el grupo más grande
    max_barras = max(len(grupo) for grupo in vth)
    ancho_barras = 0.8 / max_barras

    color_por_disp_pastel = convert_colors_to_pastel(color_por_disp)

    # Graficar las barras de cada grupo
    for i, grupo in enumerate(vth):
        # Calcular el desplazamiento de cada barra dentro de un grupo
```

```
        for j, valor in enumerate(grupo):
            ax3.bar(indices[i] + (j - (len(grupo) - 1) / 2) *
ancho_barras,  # Ajuste dinámico de la posición
                    valor,
                    width=ancho_barras,

label=f'{list(dicc_valencia_sub[familia_a_analizar].keys())[j]}' if i
== 0 else "",  # Solo etiqueta en el primer grupo
                    color=color_por_disp_pastel[j])

    if invert_yaxis_barra == 1:
        ax3.invert_yaxis()
    # Añadir título y etiquetas
    ax3.set_title(title_barra)
    ax3.set_xlabel("Fechas")
    ax3.set_ylabel("Vth(V)")
    ax3.set_xticks(indices)
    ax3.set_xticklabels([f'{fechas_a_analizar[i]}' for i in
range(num_grupos)])

    # Agregar leyenda solo una vez
    ax3.legend(bbox_to_anchor=(1, 1))

    # Mejorar el diseño
    plt.tight_layout()

    plt.grid(True)

    # Ajustar el número de particiones en el eje y a 1
    plt.gca().yaxis.set_major_locator(ticker.MultipleLocator(0.5))


    #esta gráfica la guardamos directamente
    if len(dispositivos_ban_list) != 0:
        incluir_eliminados= "_sin_" + '_'.join(dispositivos_ban_list)
    else:
        incluir_eliminados = ""
    nombre_archivo = clase_a_analizar + "_" + familia_a_analizar + "_"
+ prueba_a_analizar + "_"  + artefacto_a_analizar + incluir_eliminados
+ "_barras_recta" +  ".svg"
    print(nombre_archivo)
    fig3.savefig(nombre_archivo, bbox_inches='tight', dpi=300)

    # Mostrar la gráfica
    plt.show()



    #=======================================
    #=======================================

"""
    Genera una representación gráfica combinada de curvas Id-Vgs y
barras de umbral de tensión (Vth)
    para dispositivos electrónicos, utilizando un umbral de corriente
de 10 mA como criterio de corte.

    A diferencia de otras funciones similares, esta función no grafica
derivadas ni curvas adicionales,
    limitándose exclusivamente a la visualización de las curvas Id-Vgs
y la gráfica de barras correspondientes
```

```
    al valor de Vth estimado para cada dispositivo y fecha analizados.

    Parámetros:
        dicc_datos (dict): Diccionario con los datos experimentales
estructurados por artefacto, clase, familia y dispositivo.
        fechas_a_analizar (list): Lista de fechas para las cuales se
extraen y representan los datos.
        artefacto_a_analizar (str): Tipo de dispositivo o artefacto a
analizar (e.g., "JFET", "PMOS", "LCL").
        clase_a_analizar (str): Clase específica dentro del
diccionario de datos.
        familia_a_analizar (str): Familia del dispositivo a analizar
dentro de la clase.
        dispositivos_ban_list (list): Lista de dispositivos excluidos
del análisis y representación.
        errores (dict): Diccionario donde se almacenan mensajes de
error relacionados con la ausencia de datos.
        tipo_trazado_por_fecha (list): Lista de estilos de línea para
la representación gráfica por cada fecha.
        color_por_disp (list): Lista de colores asignados a cada
dispositivo para la visualización.
        umbral (float): Valor de corriente límite (en Amperios)
utilizado para determinar el umbral Vth.
        precision_grid (float): Paso o precisión para el espaciado de
la cuadrícula en el eje Y de la gráfica de barras.
        tamano_leyenda (int): Tamaño de fuente para la leyenda de la
gráfica.
        leyenda_loc (int): Parámetro que determina la ubicación de la
leyenda (1 para fuera, otro para dentro).
        invert_yaxis_barra (int): Indicador para invertir el eje Y de
la gráfica de barras (1 para invertir, 0 para normal).

    Retorna:
        None

"""
def representa_curvas_vth_metodo_10m( dicc_datos, fechas_a_analizar,
artefacto_a_analizar, clase_a_analizar, familia_a_analizar,
dispositivos_ban_list, errores, tipo_trazado_por_fecha,
color_por_disp,umbral , precision_grid, tamano_leyenda, leyenda_loc ,
invert_yaxis_barra ):
    # Crear la figura y el eje
    fig, ax = plt.subplots()

    prueba_a_analizar = "Id-Vgs"
    xlabel = "Vgs (V)"
    ylabel = "Id (A)"

    title = prueba_a_analizar + " curves for " + artefacto_a_analizar
+ " \ class " + clase_a_analizar + " \ family " + familia_a_analizar

    title_barra = "Vth" + " bars for " + artefacto_a_analizar + " \
class " + clase_a_analizar + " \ family " + familia_a_analizar


    # Lista para almacenar las curvas por fecha
    curvas_por_fecha = []

    #lista de listas para almacenar los puntos de interseccion de las
rectas
    vth = []
```

```
    # --- Aquí se asume que dicc_datos existe y contiene tus datos ---
    if artefacto_a_analizar == "JFET" or artefacto_a_analizar ==
"PMOS":
        dicc_valencia_sub =
dicc_datos["DUT"][clase_a_analizar][artefacto_a_analizar]
    elif artefacto_a_analizar == "LCL":
        dicc_valencia_sub = dicc_datos["LCL"][clase_a_analizar]
    else:
        dicc_valencia_sub = dicc_datos["LCL"][clase_a_analizar]
        print("Se ha introducido incorrectamente la variable
artefacto_a_utilizar, se usará 'LCL' ")

    # Recorrer las fechas y los dispositivos
    for indice_fecha, fecha in enumerate(fechas_a_analizar):
        curvas_fecha = []

        vth_sub = []
        i = 0
        for clavedisp, valordisp in
dicc_valencia_sub[familia_a_analizar].items():

            if fecha not in
dicc_valencia_sub[familia_a_analizar][clavedisp][prueba_a_analizar]:
                vth_sub.append(0)
                i += 1 #de esta forma los colores son consistentes
                errores["fecha"].append("No existen datos para el
dispositivo " + clavedisp + " para la prueba " + prueba_a_analizar + "
en la fecha " + fecha)
                continue

            if "data" not in
dicc_valencia_sub[familia_a_analizar][clavedisp][prueba_a_analizar][fe
cha]:
                vth_sub.append(0)
                i += 1 #de esta forma los colores son consistentes
                errores["datos"].append("No existen datos para el
dispositivo " + clavedisp + " para la prueba " + prueba_a_analizar + "
en la fecha " + fecha)
                continue

            if clavedisp in dispositivos_ban_list:
                vth_sub.append(0)
                i += 1 #de esta forma los colores son consistentes
                continue

            df =
dicc_valencia_sub[familia_a_analizar][clavedisp][prueba_a_analizar][fe
cha]["data"]
            eje_x = df.iloc[:, 0]
            eje_y = df.iloc[:, 1]

            #ahora que tenemos ambas columnas, necesitamos buscar el
primer número de la segunda columna que sea menor a 10mA,
            #pero debemos tener en cuenta que el orden en el que se
ponga el eje en el dataframe y si es negativo o positivo influyen

            #el valor máximo del eje x será siempre el de la segunda
columna
            valor_max = df[df.columns[1]].max()
            valor_min = df[df.columns[1]].min()
```

```
            primer_valor = df.iloc[0, 1]  # Primera fila, segunda
columna
            ultimo_valor = df.iloc[-1, 1]  # Última fila, segunda
columna

            if valor_max + valor_min >= 0:# si estamos ante una
secuencia positiva

                if primer_valor < ultimo_valor: #si la secuencia
empeza en el 0 buscamos el primer indice que cumpla
                    posicion= df[ df.iloc[:, 1] >= umbral ].index[0]
                else:# si no, buscamos el ultimo, ya que el primero
será el índice 0
                    posicion = df[ df.iloc[:, 1] >= umbral ].index[-1]

                if primer_valor < ultimo_valor: #si la secuencia
empeza en el 0 buscamos el primer indice que cumpla
                    posicion = df[ df.iloc[:, 1] >= umbral ].index[0]
                else:# si no, buscamos el ultimo, ya que el primero
será el índice 0
                    posicion = df[ df.iloc[:, 1] >= umbral ].index[-1]


            else:#si estamos ante una secuencia negativa

                if primer_valor > ultimo_valor:# si la secuencia
empieza en el 0 buscamos el primer indice que cumpla
                    posicion = df[ df.iloc[:, 1] <= umbral ].index[0]
                else:
                    posicion = df[ df.iloc[:, 1] <= umbral ].index[-1]

                if primer_valor > ultimo_valor:# si la secuencia
empieza en el 0 buscamos el primer indice que cumpla
                    posicion = df[ df.iloc[:, 1] <= umbral ].index[0]
                else:
                    posicion = df[ df.iloc[:, 1] <= umbral ].index[-1]



            vth_sub.append( df.iloc[posicion,0] )

            print("La posición para la que se alcanza un valor menor a
10mA para el dispositivo " + clavedisp + " en la fecha" + fecha  +  "
es " + str(posicion) + " y el valor vth es " +
str(df.iloc[posicion,0]))



            if indice_fecha == 4:
                color =
hacer_transparente_matplotlib(color_por_disp[i], 0.3)
            else:
                color = color_por_disp[i]



            label = f"{clavedisp} {fecha}"
            curva, = ax.plot(eje_x, eje_y,
linestyle=tipo_trazado_por_fecha[indice_fecha],
                             color=color, label=label)
```

```
            ax.set_xlabel(xlabel)
            ax.set_ylabel(ylabel)

            if leyenda_loc == 1:
                ax.legend(fontsize=tamano_leyenda, loc =
"best",bbox_to_anchor=(1, 1))
            else:
                ax.legend(fontsize=tamano_leyenda, loc = "best")

            ax.set_title(title)
            curvas_fecha.append(curva)
            i += 1

        vth.append(vth_sub)
        curvas_por_fecha.append(curvas_fecha)

    plt.axhline(y=umbral, color='black', linestyle='--', label='Línea
horizontal en Y=10')

    ax.grid()

    # Mostrar la gráfica interactiva (la ejecución se bloqueará hasta
cerrar la ventana)
    plt.show(block=True)

    #==============PARTE VTH GRÁFICA DE BARRAS
    #=========================================

    # Definir el número de grupos
    num_grupos = len(vth)

    # Crear la figura y los ejes
    fig3, ax3 = plt.subplots(figsize=(10, 6))

    # Inicializar la posición de las barras en el eje X
    indices = np.arange(num_grupos)

    # Ancho de las barras: lo ajustamos en función de la cantidad de
barras en el grupo más grande
    max_barras = max(len(grupo) for grupo in vth)
    ancho_barras = 0.8 / max_barras

    color_por_disp_pastel = convert_colors_to_pastel(color_por_disp)

    # Graficar las barras de cada grupo
    for i, grupo in enumerate(vth):
        # Calcular el desplazamiento de cada barra dentro de un grupo
        for j, valor in enumerate(grupo):
            ax3.bar(indices[i] + (j - (len(grupo) - 1) / 2) *
ancho_barras,  # Ajuste dinámico de la posición
                    valor,
                    width=ancho_barras,

label=f'{list(dicc_valencia_sub[familia_a_analizar].keys())[j]}' if i
== 0 else "",  # Solo etiqueta en el primer grupo
                    color=color_por_disp_pastel[j])

    if invert_yaxis_barra == 1:
        ax3.invert_yaxis()
    # Añadir título y etiquetas
    ax3.set_title(title_barra)
```

```
    ax3.set_xlabel("Fechas")
    ax3.set_ylabel("Vth(V)")
    ax3.set_xticks(indices)
    ax3.set_xticklabels([f'{fechas_a_analizar[i]}' for i in
range(num_grupos)])

    # Agregar leyenda solo una vez
    ax3.legend(bbox_to_anchor=(1, 1))

    # Mejorar el diseño
    plt.tight_layout()

    plt.grid(True)

    # Ajustar el número de particiones en el eje y a 1

plt.gca().yaxis.set_major_locator(ticker.MultipleLocator(precision_gri
d))


    #esta gráfica la guardamos directamente
    if len(dispositivos_ban_list) != 0:
        incluir_eliminados= "_sin_" + '_'.join(dispositivos_ban_list)
    else:
        incluir_eliminados = ""
    nombre_archivo = clase_a_analizar + "_" + familia_a_analizar + "_"
+ prueba_a_analizar + "_"  + artefacto_a_analizar + incluir_eliminados
+ "_barras_10m" +  ".svg"
    print(nombre_archivo)
    fig3.savefig(nombre_archivo, bbox_inches='tight', dpi=300)

    # Mostrar la gráfica
    plt.show()
    #=======================================
    #=======================================
```

ANEXO 8: CÓDIGO PARA LA HERRAMIENTA DE VISUALIZACIÓN DE DATOS

Clase “Interfaz principal”

```
try:
    import Tkinter as tk
except:
    import tkinter as tk

try:
    from Tkinter import ttk
except:
    from tkinter import ttk

try:
    from tkinter import filedialog, messagebox
except:
    from Tkinter import filedialog, messagebox

import Procesador_Analítico as pyt

import Gestor_Datos as pd

import PPanda_v2 as pp
import tkinter.font as fnt
import numpy as np

import ast  # Importamos el módulo ast (Abstract Syntax Tree) para
analizar la sintaxis de la expresión.

import os

import csv

import gc

"""
Clase Interfaz_Principal

Esta clase representa la interfaz gráfica principal de la aplicación y
actúa como el núcleo del sistema.
Su función principal es gestionar la ventana principal y coordinar la
interacción con el usuario, procesando
los comandos introducidos. Mantiene una comunicación continua con los
módulos Gestor de Datos y Procesador Analítico,
sirviendo como puente entre ellos.

Además, se encarga de validar no solo la sintaxis y coherencia de las
entradas del usuario, sino también de verificar
la integridad y adecuación de los datos gestionados, especialmente en
funciones avanzadas. Este manejo robusto
de errores minimiza fallos y garantiza una experiencia de usuario
estable y segura.
"""


"""
Funciones de validación de apoyo para la clase "Interfaz_Principal
```

```
"""

"""
    Valida que la cadena P represente un número entero impar mayor que
0.

    Parámetros:
      - P: cadena de texto que se desea validar.

    Comportamiento:
      - No permite cadena vacía.
      - Solo permite cadenas que contengan dígitos numéricos.
      - Convierte la cadena a entero y verifica que sea mayor a 0 y
que sea impar.
      - Devuelve True si cumple todas las condiciones; False en caso
contrario.
"""
def validar_impar_mayor_cero(P):

    if P == "":
        return False
    if P.isdigit():
        numero = int(P)
        if numero > 0 and numero % 2 == 1:
            return True
    return False


"""
    Valida que la cadena P represente un número entero mayor que 0.

    Parámetros:
      - P: cadena de texto que se desea validar.

    Comportamiento:
      - No permite cadena vacía.
      - Solo permite cadenas que contengan dígitos numéricos.
      - Convierte la cadena a entero y verifica que sea mayor a 0.
      - Devuelve True si cumple todas las condiciones; False en caso
contrario.
"""
def validar_mayor_cero(P):

    if P == "":
        return False
    if P.isdigit():
        numero = int(P)
        if numero > 0:
            return True
    return False


"""
    Valida que la cadena P contenga solo dígitos numéricos y que su
longitud máxima sea 5.

    Parámetros:
      - P: cadena de texto que se desea validar.

    Comportamiento:
```

```
      - Permite cadena vacía (True).
      - Solo permite cadenas con dígitos numéricos y longitud menor o
igual a 5.
      - Devuelve True si cumple las condiciones; False en caso
contrario.
"""
def validar_numeros_y_longitud(P):

    if P == "":
        return True
    if P.isdigit() and len(P) <= 5:
        return True
    else:
        return False

"""
    Valida que la cadena P no contenga espacios y que su longitud
máxima sea 10 caracteres.

    Parámetros:
      - P: cadena de texto que se desea validar.

    Comportamiento:
      - No permite espacios en la cadena.
      - Permite cadenas con longitud hasta 10 caracteres.
      - Devuelve True si cumple las condiciones; False en caso
contrario.
"""
def validar_sin_espacios_y_longitud(P):

    if " " in P:
        return False
    if len(P) > 10:
        return False
    return True



class Interfaz_Principal():
    def __init__(self):

        '''Antes de nada, inicializaremos las variables que tienen que
ver con la selección
        que podemos hacer, ya sea de columnas o de índices en el
propio init de la clase principal'''

        self.ram_limite = 85

        "En la clase PPanda explicamos la razón por la que usamos
listas de 1 elemento"

        self.columnaseleccionadas = ["",""]

        self.rango = ["", ""]

        #Tendremos una variable tipo lista que contenga el nombre del
eje y, x y el título de la gráfica


        self.labels = [ "" , "", ""]
```

```
        self.mostrar_eje_x = ""

        self.escala_leyenda = 1

        self.nombres_columnas = []#lista que contendrá los nombres de
las columnas de los dataframes seleccionados

        self.textointroducidoenindice = [":"]

        self.indicesseleccionados = ["No introducidos"]
        #La escala por defecto será 0.02 segundos o 20 ms
        self.escala = [0.02]

        #Por defecto será un offset de 0s
        self.offset = [0]

        #================VARIABLES PARA EL MODO DE ESTADÍSTICA
        #esta variable será un booleano que representará si se va a
continuar o no con la representación estadística
        self.continua_estadistica = False
        # Variables internas que almacenarán la selección básica y los
extrasde estadística
        self.estad = []           # Lista de cadenas (por ejemplo,
["modelo", "tipo"] o ["modelo", "tipo", "opción"])
        self.columnas_estad = []   # Lista de listas (cada sublista
representa los valores de la fila de extras)


        #El rango de datos que analizaremos (expresado en segundos)
        #por defecto se inicializan a 1000 que representa que queremos
todo
        self.textointroducidoenrango = [":"]
        self.rangomin = [1000]
        self.rangomax = [1000]

        #Tendremos una variable de texto destinada a mostrar sobre qué
lista de datos genéricos estamos actuando

        self.listagen = ["1"]

        self.textooperacion = ["max"]

        #Tendremos otra variable que guardará la información de lo que
se ha guardado en los datos genéricos 1 y 2 al pulsar realizar

        self.info1 = ["ninguna operación"]
        self.info2 = ["ninguna operación"]


        #Tendremos una variable que será la información más allá de
los datos de ppanda que se añadirá a los de ppanda

        self.infomas = ""

        #variables para plotmix
        self.op_var= ""      # Variable para almacenar la operación
seleccionada.
        self.num_var= ""      # Variable para almacenar el número
ingresado.
```

```
        #Tendremos una variable que controlará si queremos trabajar
con subplots a la hora de representar gráficas
        #también otra que contendrá el tamaño de la matriz de
subplots, y otra para indicar dónde se posicionaría
        #el subplot raw, genérico o mix, será una lista, donde el
primer elemento es la posición del plot raw
        #En subplots, si tenemos grey, será que no queremos, green que
sí
        self.subplots = ["green"]

        self.subplotmatrixantigua = [ [ "1", "1" ] ]

        self.subplotmatrix = [ [ "1", "1" ] ]

        #esta variable contendrá la opción de coordinar o no los ejes
cuando estamos en modo subplot
        self.coordinar = ["no coord."]

        self.subplotposf = [ "1" ]
        self.subplotposc = [ "1" ]


        #Inicializaremos aquí una instancia tipo DatFrame, a
inicializador debemos pasarle la raíz
        #para que cree la ventana
        #DEBE SITUARSE ANTES DE QUE SE CREE LA NUEVA ROOT PORQUE LA
CLASE GENERA UNA ROOT PROPIA
        #SI NO SE PONE ANTES INTERFERIRÍAN. DE ESTA MANERA, CUANDO SE
CIERRE UNA, SE ABRIRÁ LA SIGUIENTE
        self.DF = pp.DataFrame()

        #self.DF.iniciador()

        #Variable que nos servirá para la representación mixta
        self.columnastotales = self.DF.nombrecols;

        #

        data = {
            "X": np.linspace(0, 10, 100),
            "Seno": np.sin(np.linspace(0, 10, 100)),
            "Coseno": np.cos(np.linspace(0, 10, 100)),
            "Logaritmo": np.log(np.linspace(1, 10, 100)),
            "Exponencial": np.exp(np.linspace(0, 2, 100))
        }

        df = pd.DataFrame(data)

        self.DF.dataframes["debug"] = { "debug" : df }



        self.root = tk.Tk()
        self.totalValue = tk.StringVar()

        # Registrar las funciones de validación
        self.validate_window_length =
self.root.register(validar_impar_mayor_cero)
        self.validate_polyorder =
self.root.register(validar_mayor_cero)
```

```
        self.root.geometry('1950x900')
        self.root.title("Graficadora")

        '''Creamos los botones de la ventana con tk.Button y los
colocamos con .place(x,y) teniendo
        en cuenta que el origen es la esquina superior izquierda'''

        #-------Referente a la información para rellenar los campos



        self.info = tk.Button(self.root,font= fnt.Font(weight="bold",
size = 10),bd=8, bg = 'yellow', text='Información',
command=lambda:pp.informacion(self.root,

"Para añadir adecuadamente la infromación debe presionarse en primer
lugar en los botones grises de guardar y,\n si es necesario, en
segundo lugar los botones naranjas.\n\nSe puede introducir:\n\n-Nombre
del eje y de las gráficas que vayamos a representar\n\n-Números de los
ficheros a representar en la gráfica, siguiendo el siguiente
código:\nEscribir dos puntos : implica que elegimos todo el
rango\nEscribir numero:  implica que elegimos todos los que tenemos
partiendo desde el índice numero\nEscribir :numero implica que
elegimos todos partiendo desde el principio hasta el índice
numero\nEscribir numero1:numero2 implica que elegimos desde los
índices numero1 hasta número2\nEscribir :-numero1,numero2,numero3...
implica que queremos todos menos numero1,numero2,numero3...\nEscribir
:+ numero1,numero2,numero3... implica que SOLO QUEREMOS
numero1,numero2,numero3\n\n-Nombre de la columna (header) de los
ficheros a representar\n\n-Escala de tiempo, offset y rango de tiempo,
este último codificado de la siguiente manera:\nEscribir dos puntos :
implica que elegimos todo el rango\nEscribir numero:  implica que
elegimos todos los que tenemos partiendo desde el tiempo
numero\nEscribir :numero implica que elegimos todos partiendo desde el
principio hasta el tiempo número\nEscribir numero1:numero2 implica que
elegimos desde los índices numero1 hasta número2\n\n-Selector de lista
sobre la que se va a realizar la operación\n\n-Operación a realizar
según códigos de operación, actualmente se contienen: max,"

) )

        self.info.place(x=0,y=0)

        #------Referrente a dar información sobre los dataframes
cargados
        #Constará de un botón de información que al pulsarlo despliega
una ventana que da los datos
        #que aporta la clase DataFrames

        #La función que crea la ventana y aporta la información está
creada en PPanda

        self.infodataframe = tk.Button(self.root,font=
fnt.Font(weight="bold",size = 10),bd=8,bg = 'blue', text='Información
datos', command=lambda:self.actualiza_info())

        self.infodataframe.place(x=370,y=0)

        #------------------Referente a cargar otros dataframes
```

```
        self.cargadatos = tk.Button(self.root,font=
fnt.Font(weight="bold",size = 10),bd=8,bg = 'green', text='Cargar
dataframes', command=lambda:[self.cargar_dataframes() ])

        self.cargadatos.place(x=160,y=0)

        #-----------------Referente a un script que  cargue dataframes
de diferentes carpetas y obtenga sus parámetros
        #para guardarlo todo en un dataframe concatenado de dataframes

        self.btnselec = tk.Button(self.root,font=
fnt.Font(weight="bold",size = 10),bd=8,bg = 'cyan', text='Seleccionar
dataframes', command=lambda:[self.selecciona_dataframes() ])

        self.btnselec.place(x=580,y=0)


        # --- Eje X ---

        #en este caso se utilizarán frames y los métodos pack() en
lugar de place() por facilidad de maniobra
        self.frame_x = tk.LabelFrame(self.root, text="Eje X")
        self.frame_x.place(x = 20, y = 53)

        # Campo para la etiqueta del Eje X

        self.label_x_etiqueta = tk.Label(self.frame_x,
text="Etiqueta:")
        self.label_x_etiqueta.pack(side='left')
        self.entry_x_etiqueta = tk.Entry(self.frame_x, width=10)
        self.entry_x_etiqueta.pack(side='left', padx=5, pady = 20)

        # Combobox para elegir el modo (Columna o Generar)
        self.frame_x_modo_mostrar = tk.Frame(self.frame_x)
        self.frame_x_modo_mostrar.pack(fill='x',side = 'left',
padx=10, pady = 2)

        self.frame_x_modo = tk.Frame(self.frame_x_modo_mostrar)
        self.frame_x_modo.pack(fill='x',side = 'top', padx=20, pady =
5)

        self.label_mode_x = tk.Label(self.frame_x_modo, text="Modo:")
        self.label_mode_x.pack(side='left', padx=5)
        self.combo_mode_x =
ttk.Combobox(self.frame_x_modo,state="readonly" , values=[ "Generar",
"Columna"], width=15)
        self.combo_mode_x.pack(side='left', padx=5)
        self.combo_mode_x.current(0)  # Por defecto: "Columna"

        self.frame_x_mostrar = tk.Frame(self.frame_x_modo_mostrar)
        self.frame_x_mostrar.pack(fill='x',side = 'top', padx=20, pady
= 2)
        self.label_mostrar = tk.Label(self.frame_x_mostrar,
text="Mostrar:")
        self.label_mostrar.pack(side='left', padx=5)
        self.combo_mostrar =
ttk.Combobox(self.frame_x_mostrar,state="readonly" , values=[ "Si",
"No"], width=15)
        self.combo_mostrar.pack(side='left', padx=5)
        self.combo_mostrar.current(0)  # Por defecto: "Columna"
```

```
        # Frame para definir el rango del Eje X (aplica para los tres
modos)
        self.frame_x_range = tk.Frame(self.frame_x)
        self.frame_x_range.pack(fill='x',side = 'top', padx=20)
        self.label_range_x = tk.Label(self.frame_x_range, text="Rango
eje x:")
        self.label_range_x.pack(side='left')
        self.label_from_x = tk.Label(self.frame_x_range,
text="Inicio:")
        self.label_from_x.pack(side='left', padx=(10,0))
        self.entry_from_x = tk.Entry(self.frame_x_range, width=10)
        self.entry_from_x.pack(side='left', padx=5)
        self.label_to_x = tk.Label(self.frame_x_range, text="Fin:")
        self.label_to_x.pack(side='left', padx=(10,0))
        self.entry_to_x = tk.Entry(self.frame_x_range, width=10)
        self.entry_to_x.pack(side='left', padx=5)

        # Frame para la opción "Generar eje x" (escala y offset)
        self.frame_x_generar = tk.Frame(self.frame_x)
        self.frame_x_generar.pack(fill='x',side = 'top', padx=20, pady
= 2)
        self.label_escala = tk.Label(self.frame_x_generar,
text="Escala:")
        self.label_escala.pack(side='left')
        self.entry_escala = tk.Entry(self.frame_x_generar, width=10)
        self.entry_escala.pack(side='left', padx=5)
        self.entry_escala.insert(0, "1")
        self.label_offset = tk.Label(self.frame_x_generar,
text="Offset:")
        self.label_offset.pack(side='left')
        self.entry_offset = tk.Entry(self.frame_x_generar, width=10)
        self.entry_offset.pack(side='left', padx=5)
        self.entry_offset.insert(0,"0")
        self.btnapx = tk.Button(self.frame_x_generar,
text="Aplicar",font = fnt.Font(size = 8),
command=lambda:[self.genera_ejex()], bg = "orange"  )
        self.btnapx.pack()

        # Frame para la opción "Seleccionar columna"
        self.frame_x_columna = tk.Frame(self.frame_x,  padx=20, pady =
2)
        self.frame_x_columna.pack_forget()
        self.label_x_columna = tk.Label(self.frame_x_columna,
text="Columna:")
        self.label_x_columna.pack(side='left')
        self.combo_x =
ttk.Combobox(self.frame_x_columna,state="readonly")
        self.combo_x.pack(side='left', padx=5)


        self.combo_mode_x.bind("<<ComboboxSelected>>", lambda event:
self.actualiza_opciones_eje_x(event))


        # --- Eje Y ---

        #en este caso se utilizarán frames y los métodos pack() en
lugar de place() por facilidad de maniobra
        self.frame_y = tk.LabelFrame(self.root, text="Eje Y")
        self.frame_y.place(x = 1080, y = 49)
```

```
        # Campo para la etiqueta del Eje Y

        self.label_y_etiqueta = tk.Label(self.frame_y,
text="Etiqueta:")
        self.label_y_etiqueta.pack(side='left')
        self.entry_y_etiqueta = tk.Entry(self.frame_y, width=10)
        self.entry_y_etiqueta.pack(side='left', padx=5, pady = 20)


        self.frame_y_modo = tk.Frame(self.frame_y)
        self.frame_y_modo.pack(fill='x',side = 'top', padx=20, pady =
5)
        # Combobox para elegir el modo (Columna o Generar)
        self.label_mode_y = tk.Label(self.frame_y_modo, text="Modo:")
        self.label_mode_y.pack(side='left', anchor = "nw", padx=5)
        self.combo_mode_y =
ttk.Combobox(self.frame_y_modo,state="readonly"  , values=["Columna",
"Math"], width=15)
        self.combo_mode_y.pack(side='left',  anchor = "ne" , padx=5)
        self.combo_mode_y.current(0)  # Por defecto: "Columna"

        # Frame para la opción "Seleccionar columna"
        self.frame_y_columna = tk.Frame(self.frame_y)
        self.frame_y_columna.pack(fill='x',side = 'top', padx=20, pady
= 5)
        self.label_y_columna = tk.Label(self.frame_y_columna,
text="Columna:")
        self.label_y_columna.pack(side='left')
        self.combo_y =
ttk.Combobox(self.frame_y_columna,state="readonly", width = 10)
        self.combo_y.pack(side='left', padx=5)

        # Frame para la opción "Math"
        self.frame_y_math = tk.Frame(self.frame_y)
        # Inicialmente oculto, se muestra cuando se selecciona
"Generar"
        self.frame_y_math.pack_forget()
        self.label_y_operacion = tk.Label(self.frame_y_math,
text="Operacion:")
        self.label_y_operacion.pack(side='left')
        self.entry_operacion_y = tk.Entry(self.frame_y_math, width=10)
        self.entry_operacion_y.pack(side='left', padx=5)
        self.btn_validar_y = tk.Button(self.frame_y_math,
text="Validar",font = fnt.Font(size = 8),
command=lambda:[self.validar(self.entry_operacion_y)], bg = "orange"
)
        self.btn_validar_y.pack()

        self.combo_mode_y.bind("<<ComboboxSelected>>", lambda event:
self.actualiza_opciones_eje_y(event))

        '''self.btnplot_debug = tk.Button(self.root, text="Plot
debug", command=self.plotdebug)
        self.btnplot_debug.place( x=1070, y=10)'''


        # --- Título y leyenda ---

        self.frame_tit = tk.LabelFrame(self.root, text="Título y
leyenda")
        self.frame_tit.place(x = 1080, y = 0)
```

```
        self.labeltit = tk.Label(self.frame_tit, text="Título:")
        self.labeltit.pack(side='left', padx=2.5, pady = 2)
        self.entry_tit = tk.Entry(self.frame_tit, width=20)
        self.entry_tit.pack(side='left', padx=2.5, pady = 2)

        self.labelley = tk.Label(self.frame_tit, text="Factor
leyenda:")
        self.labelley.pack(side='left', padx=2.5, pady = 2)
        self.entry_ley = tk.Entry(self.frame_tit, width=8)
        self.entry_ley.pack(side='left', padx=2.5, pady = 2)
        self.entry_ley.insert(0,"1")

        # --- Subplots ---

        #añadimos un frame para rodear las opciones y caracterizarlas
        self.lf = tk.LabelFrame(self.root, text="Subplots")
        self.lf.place( x=1500, y=0, width=450, height=145)

        self.labelfil = tk.Label(self.lf, text="Filas subplot")
        self.labelfil.place(x=130,y=5)

        self.combofil = ttk.Combobox( self.lf, state =
"disabled",width = 3, values = ["1","2"] )
        self.combofil.place(x=150,y=30)
        self.combofil.set("1")

        self.labelcol = tk.Label(self.lf, text="Columnas subplot")
        self.labelcol.place(x=220,y=5)

        self.combocol = ttk.Combobox( self.lf, state =
"disabled",width = 3, values = ["1","2","3"] )
        self.combocol.place(x=240,y=30)
        self.combocol.set("1")


        self.labelposgen = tk.Label(self.lf, text="Indicador plot")
        self.labelposgen.place(x=180,y=60)

        self.comboposfil = ttk.Combobox( self.lf, state =
"disabled",width = 4, values = ["1","2"] )
        #creamos un evento cuando se cambie de opción, se
guardará,PONEMOS UNA e DE EVENT
        self.comboposfil.bind( "<<ComboboxSelected>>", lambda e:
[self.subplotposf.clear(),self.subplotposf.insert(0,
self.comboposfil.get() ) , print("Posición fila cambiada a " +
str(self.subplotposf[0]) )] )
        self.comboposfil.place(x=170,y=85)

        self.comboposfil.set("1")

        self.comboposcol = ttk.Combobox( self.lf, state =
"disabled",width = 4, values = ["1","2","3"] )
        self.comboposcol.bind( "<<ComboboxSelected>>", lambda e:
[self.subplotposc.clear(),self.subplotposc.insert(0,
self.comboposcol.get() ) , print("Posición columna cambiada a " +
str(self.subplotposc[0]) )] )
        self.comboposcol.place(x=235,y=85)

        self.comboposcol.set("1")
```

```
        self.btnel = tk.Button(self.lf,state = "normal",
text="Eliminar", command=lambda:[self.clearsub() ], bg = "purple"  )
        self.btnel.place( x=300, y=80)


        self.btnap = tk.Button(self.lf,state = "disabled",
text="Aplicar", command=lambda:[ self.subplotmatrixantigua.clear() ,
self.subplotmatrixantigua.insert(0,self.subplotmatrix[0] )
,self.subplotmatrix.clear()  , self.subplotmatrix.insert( 0,
[self.combofil.get(),self.combocol.get() ] ),print("La nueva
configuración de subplots es: "),print(self.subplotmatrix),print("La
antigua era: "),print(self.subplotmatrixantigua) ,
self.compartir_ejes() ,self.clear(1) ], bg = "orange"  )
        self.btnap.place( x=350, y=25)

        #botón en el que indicaremos si queremos suar subplots, si
está en verde significa que si
        #al darle, adjuntará su color ACTUAL a la variable, de forma
que cuando pase a verde, le dará a la variable el color que tenía, el
GRIS
        self.btnsub = tk.Button(self.lf, text="Emplear subplots",
command=lambda:[ self.comboposcol.set("1"), self.comboposfil.set("1")
, self.subplots.clear(), self.subplots.insert(0,
self.btnsub["background"]) ,pp.toggleoption( self.btnsub, ["green" ,
"grey"], [  self.combofil,
self.combocol,self.comboposfil,self.comboposcol,self.combocoord ],[
self.btnap] ), print(self.subplots[0])] , bg = "grey"  )
        self.btnsub.place( x=0, y=0)

        #introduciremos una combobox donde seleccionaremos si queremos
que se coordinen los ejes de los subplots y si queremos que sea solo a
las columnas o a todo

        self.combocoord = ttk.Combobox( self.lf, state =
"disabled",width = 13, values = ["No coord.", "Coord. col. x-y" ,
"Coord. all x-y", "Coord. col. x" , "Coord. all x"] )
        #creamos un evento cuando se cambie de opción, se
guardará,PONEMOS UNA e DE EVENT
        self.combocoord.bind( "<<ComboboxSelected>>", lambda e:
[self.coordinar.clear(),self.coordinar.insert(0, self.combocoord.get()
) , print("Opción de coordinar ejes cambiada a " +
str(self.coordinar[0]) )] )
        self.combocoord.place(x=20,y=60)
        #establecemos un valor inicial
        self.combocoord.set("No coord.")


        #ahora emplearemos una combobox con tres opciones, las cuales
serán no coord


        #----------Referente a los botones de graficar


        self.plotGraphButton = tk.Button(self.root, text='Representa
datos', command=self.plotraw, bg = "lightgreen")

        self.plotGraphButton2 = tk.Button(self.root, text='Análisis',
command=self.plotstat, bg = "lightyellow")
```

```
        self.plotGraphButton3 = tk.Button(self.root, text='Guardar
datos', command=self.open_selection_window, bg = "lightblue")

        #self.ExportButton = tk.Button(self.root, text='Exportar',
command=lambda: [self.exporta_datos()  ] )



        self.clearButton = tk.Button(self.root, text='Borrar
canvas',font= fnt.Font(weight="bold", size = 12),
command=lambda:[self.clear(0)],bg="magenta")



        self.plotGraphButton.pack()
        self.plotGraphButton2.pack()
        self.plotGraphButton3.pack()

        #self.ExportButton.place(x=1425,y=85)


        self.clearButton.pack()

        #Ejecturamos una sola vez esto, para adecuar la interfaz de la
gráfica
        self.root.after(0, lambda: ( self.plotdebug() , self.clear(0)
)  )

        self.root.mainloop()



    #%%Funciones intermedias de comunicación con la clase plotting
    #=====================Funciones intermedias de comunicación con la
clase plotting
    def genera_ejex(self):

        if self.DF.dataframes_selec == {}:#si no hay ningún grupo
seleccionado, no tiene sentido generar ningún eje pues no sabemos la
cantidad de puntos
            messagebox.showerror("Error","No hay ningún grupo de
dataframes seleccionado")
            return
        plot.generaejex( self.entry_escala.get() ,
self.entry_offset.get(), self.DF.dataframes, self.DF.dataframes_selec)

    def plotraw(self):

        if self.DF.dataframes_selec == {}:#si no hay ningún grupo
seleccionado, no tiene sentido generar ningún eje pues no sabemos la
cantidad de puntos
            messagebox.showerror("Error","No hay ningún grupo de
dataframes seleccionado")
            return

        try:
            #actualizamos los label de los ejes y el título


            self.labels[0] = self.entry_x_etiqueta.get()
            self.labels[1] = self.entry_y_etiqueta.get()
```

```
            self.labels[2] = self.entry_tit.get()

            self.mostrar_eje_x = self.combo_mostrar.get()

            try:
                self.escala_leyenda = float(self.entry_ley.get())
            except:
                messagebox.showerror("Error", "Error al asignar la
escala de la leyenda")
                return

            if self.combo_mode_x.get() == "Columna":
                self.columnaseleccionadas[0] = self.combo_x.get()
                self.columnaseleccionadas[1] = self.combo_y.get()
            else:
                self.columnaseleccionadas[0] = ""
                self.columnaseleccionadas[1] = self.combo_y.get()

            self.rango[0] = self.entry_from_x.get()
            self.rango[1] = self.entry_to_x.get()

            node_operation = None
            math_operation = ""
            if self.combo_mode_x.get() == "Math" or
self.combo_mode_y.get() == "Math":

                if self.combo_mode_y.get() == "Math":

                    try:

                        math_operation = self.entry_operacion_y.get()
                        node_operation =
self.validar_expresion(math_operation)

                    except Exception as e:

                        messagebox.showerror("Error con la operación
del eje y 'Math'", str(e))



plot.plotRaw(self.DF.dataframes,self.DF.dataframes_selec,self.rango,se
lf.columnaseleccionadas,self.root, self.labels, self.mostrar_eje_x,
self.escala_leyenda , self.subplots[0],self.subplotmatrix[0], [
self.subplotposf[0], self.subplotposc[0] ], node_operation,
math_operation )
        except Exception as e:
            messagebox.showerror("Error con la generación del plot",
str(e))


    def plotstat(self):


        if self.DF.dataframes_selec == {}:#si no hay ningún grupo
seleccionado, no tiene sentido generar ningún eje pues no sabemos la
cantidad de puntos
            messagebox.showerror("Error","No hay ningún grupo de
dataframes seleccionado")
            return
```

```
        self.abrir_ventana_estadistica()


        if self.continua_estadistica:#si es True, se procede a
determinar el requisito mínimo, que los dataframes tengan al menos
999900 puntos

            grupodataframes = list( self.DF.dataframes_selec.keys()
)[0] #obtenemos el nombre del grupo
            nombresdataframes =
self.DF.dataframes_selec[grupodataframes] #obtenemos los idnividuos
del grupo

            if len(nombresdataframes) > 20:#si tenemos más de 20
seleccionados no se continuará por tiempos de carga excesivos
                messagebox.showerror("Error con la generación del
plot", "Se han seleccionado más de 20 dataframes" )
                return

            if self.estad[0] == "Análisis de sobrecarga":#querremos
recorrer todos los dataframes seleccionados para comprobar que tienen
la cantidad de puntos mínima
                for nombres_individuos in nombresdataframes:#querremos
recorrer todos los dataframes seleccionados para comprobar que tienen
la cantidad de puntos mínima


print(self.DF.dataframes[grupodataframes][nombres_individuos].shape[0]
)

                    if
self.DF.dataframes[grupodataframes][nombres_individuos].shape[0] <
999900:
                        messagebox.showerror("Error con la generación
del plot", "El dataframe de nombre " +  nombres_individuos + " tiene
menos de 999900 puntos")
                        return

            if self.estad[0] == "Análisis temporales":#querremos
recorrer todos los dataframes seleccionados para comprobar que tienen
la cantidad de puntos mínima
                for nombres_individuos in nombresdataframes:#querremos
recorrer todos los dataframes seleccionados para comprobar que tienen
la cantidad de puntos mínima


print(self.DF.dataframes[grupodataframes][nombres_individuos].shape[0]
)

                    if
self.DF.dataframes[grupodataframes][nombres_individuos].shape[0] < 10:
                        messagebox.showerror("Error con la generación
del plot", "El dataframe de nombre " +  nombres_individuos + " tiene
menos de 10 puntos")
                        return

            #try:
            #actualizamos los label de los ejes y el título
            self.mostrar_eje_x = self.combo_mostrar.get()
```

```
            self.labels[0] = self.entry_x_etiqueta.get()
            self.labels[1] = self.entry_y_etiqueta.get()
            self.labels[2] = self.entry_tit.get()


plot.plotGener(self.DF.dataframes,self.DF.dataframes_selec,self.column
aseleccionadas,self.root, self.estad, self.columnas_estad,self.labels,
self.mostrar_eje_x , self.subplots[0],self.subplotmatrix[0], [
self.subplotposf[0], self.subplotposc[0] ] )
            #except Exception as e:
                #messagebox.showerror("Error con la generación del
plot", "Asegúrate de comprobar que las cruvas a analizar son
adecuadas\n error:"str(e))



    def plotmix(self,columnas, fichero, escala, offset,operacion):

        plot.plotMix(self.DF.dataframes, self.DF.nombrecols, columnas,
self.DF.indices, fichero, escala,
offset,operacion,self.root,self.labels,self.subplots[0],self.subplotma
trix[0],[ self.subplotposf[0], self.subplotposc[0] ])

    def plotdebug(self):

        plot.plotDebug(self.root)

    def clear(self,selec):

plot.clearPlotPage(selec,self.subplotmatrix,self.subplotmatrixantigua)

    def clearsub(self):
        plot.clearSubplot(self.subplotposf[0], self.subplotposc[0])

    def compartir_ejes(self):

        plot.compartir_ejes(self.subplots[0],self.subplotmatrix[0],
self.coordinar[0])



    #=========================================================
    #%%Funciones intermedias de comunicación con la clase ppanda

    #====================Funciones intermedias de comunicación con la
clase ppanda

    #Esta función se encargará de mostrar la información que
mostraremos con el botón de información cada vez que se pulse
    def actualiza_info(self):

        self.DF.mostrar_diccionario(self.root, self.infodataframe)

    def cargar_dataframes(self):
        self.ram_limite = self.DF.iniciador(self.root,
self.ram_limite)
        print(self.ram_limite)

    #esta funcion se encargara de mostrar una ventana para seleccioanr
dataframes
    def selecciona_dataframes(self):
```

```
        self.DF.open_selection_window(self.btnselec, self.root)

        #una vez se han seleccionado los dataframes, podemos
actualizar los campos de seleccion de columnas de los ejes
        self.actualiza_selector_columna( )

    #============================================================


    #%%FUNCIONES PARA ACTUALIZAR CAMPOS DE LOS EJES
    #====================Funciones para los botones
    def actualiza_opciones_eje_x(self,event):
        """Actualiza la visibilidad de los frames según el modo
seleccionado."""
        if self.combo_mode_x.get() == "Columna":
            self.frame_x_columna.pack(fill='x', padx=20, pady=2)
            self.frame_x_generar.pack_forget()

        elif self.combo_mode_x.get() == "Generar":
            self.frame_x_columna.pack_forget()

            self.frame_x_generar.pack(fill='x', padx=20, pady=2)


    def actualiza_opciones_eje_y(self,event):
        """Actualiza la visibilidad de los frames según el modo
seleccionado."""
        if self.combo_mode_y.get() == "Columna":
            self.frame_y_columna.pack(fill='x', padx=20, pady=2)
            self.frame_y_math.pack_forget()
        else:
            self.frame_y_columna.pack_forget()
            self.frame_y_math.pack(fill='x', padx=20, pady=2)


    def actualiza_selector_columna(self):
        """
        Actualiza la interfaz de los ejes en función del estado actual
de 'opciones' y 'seleccion'.
        Se extrae el primer DataFrame (si existe) y se actualizan los
combobox de Eje Y y, en modo "Columna",
        también el de Eje X.

        Parámetros:
        - opciones: Diccionario principal con las opciones y sus
subopciones.
        - seleccion: Diccionario con la opción seleccionada y sus
subopciones correspondientes.
        - combo_y: Combobox para seleccionar la columna del eje Y.
        - combo_x: Combobox para seleccionar la columna del eje X.
        """
        # Inicializamos las variables locales para el DataFrame y los
nombres de las columnas
        df_first = None
        column_names = []
        opciones = self.DF.dataframes
        seleccion = self.DF.dataframes_selec

        # Verificamos que 'seleccion' y 'opciones' contengan datos
        if seleccion and opciones:
```

```
            print("diccionario de seleccion")
            print(seleccion)
            # Obtenemos la lista de opciones seleccionadas
            selected_options = list(seleccion.keys())
            print("claves del diccionario de seleccion")
            print(selected_options)
            if selected_options:
                # Seleccionamos la primera opción

                selected_option = selected_options[0]
                print("primera opcion")
                print(selected_option)
                # Obtenemos las subopciones correspondientes a la
opción seleccionada
                selected_suboptions = seleccion.get(selected_option,
[])
                print("subopciones")
                print(selected_suboptions)

                if selected_suboptions:
                    # Seleccionamos la primera subopción
                    first_suboption = selected_suboptions[0]
                    print("primera subopcion")
                    print(first_suboption)
                    # Verificamos que la subopción exista en
'opciones'
                    if first_suboption in
opciones.get(selected_option, {}):
                        # Asignamos el DataFrame correspondiente a la
subopción seleccionada
                        df_first =
opciones[selected_option][first_suboption]
                        print("dataframe")
                        print(df_first)
                        # Extraemos los nombres de las columnas del
DataFrame
                        column_names = list(df_first.columns)
                        print("columnas")
                        print(column_names)

        # Actualizamos el combobox del Eje Y con los nombres de las
columnas

        self.combo_y['values'] = column_names
        # Si hay columnas disponibles, seleccionamos la primera; de lo
contrario, dejamos el combobox vacío
        if column_names:
            self.combo_y.current(0)
        else:
            self.combo_y.set("")

        # Actualizamos el combobox del Eje X (modo "Columna") con los
nombres de las columnas
        self.combo_x['values'] = column_names
        # Si hay columnas disponibles, seleccionamos la primera; de lo
contrario, dejamos el combobox vacío
        if column_names:
            self.combo_x.current(0)
        else:
            self.combo_x.set("")
```

```
        self.nombres_columnas = column_names


    #=============================================================


    #%%FUNCIONES PARA EVALUAR EXPRESIONES EN EL MODO MATH

    # ------------------------------------------------------------------
--------------------
    # ¿Por qué usamos ast?
    # ------------------------------------------------------------------
--------------------
    # El módulo ast nos permite convertir una cadena de texto (que
contiene código Python)
    # en un árbol de sintaxis abstracta (AST). Esto es muy útil para
validar y evaluar
    # expresiones de forma segura, ya que podemos inspeccionar cada
nodo del árbol y
    # asegurarnos de que solo se utilicen las operaciones y
estructuras permitidas.
    #
    # En lugar de usar eval(), que ejecuta código arbitrario y puede
ser peligroso,
    # se utiliza ast.parse para transformar la expresión en una
estructura de datos.
    # Luego, se recorre este árbol y se valida cada parte (operadores,
constantes, nombres)
    # para garantizar que la expresión se limita a operaciones seguras
y permitidas.
    # ------------------------------------------------------------------
--------------------


    def validar_expresion(self, expr):
        """
        Valida la expresión ingresada comprobando que:
          - Se utilicen únicamente los operadores permitidos: suma
(+), resta (-) y multiplicación (*).
          - Los nombres (variables) que representen series se
encuentren en la lista 'nombres_permitidos'.
          - Se permitan constantes numéricas y operadores unarios (+ y
-).
          - No se permita una operación que involucre únicamente
escalares (por ejemplo, 1+1).

        Parámetros:
            expr (str): La expresión a validar.

        Devuelve:
            node: El nodo AST parseado si la expresión es válida.

        Lanza:
            ValueError: Si la expresión es inválida o se utiliza una
operación/nodo no permitido.
        """

        nombres_permitidos = self.nombres_columnas
        print("Nombres:")
        print(nombres_permitidos)
```

```
        # Operadores permitidos: suma, resta y multiplicación.
        allowed_ops = {ast.Add, ast.Sub, ast.Mult}

        try:
            # Parseamos la expresión en modo 'eval', lo que genera un
árbol de sintaxis
            node = ast.parse(expr, mode='eval')
        except Exception as e:
            raise ValueError("Expresión inválida: " + str(e))

        # Usamos una lista mutable para marcar si se ha encontrado al
menos un nombre permitido.
        # Esto nos ayuda a detectar si la expresión opera solo con
escalares.
        found_name = [False]

        def _check_node(node):
            """
            Función interna recursiva que recorre el AST y valida cada
nodo.
            """
            if isinstance(node, ast.Expression):
                # Nodo raíz: se continúa evaluando su cuerpo.
                return _check_node(node.body)
            elif isinstance(node, ast.BinOp):
                # Nodo de operación binaria (ej. a + b)
                if type(node.op) not in allowed_ops:
                    raise ValueError(f"Operador
'{type(node.op).__name__}' no permitido")
                # Se validan recursivamente el operando izquierdo y el
derecho.
                _check_node(node.left)
                _check_node(node.right)
            elif isinstance(node, ast.UnaryOp):
                # Nodo de operación unaria (ej. -a o +a)
                if type(node.op) not in {ast.UAdd, ast.USub}:
                    raise ValueError("Operador unario no permitido")
                _check_node(node.operand)
            elif isinstance(node, ast.Constant):
                # Nodo constante: se permite si es un número (int o
float)
                if not isinstance(node.value, (int, float)):
                    raise ValueError("Constante no permitida")
            elif isinstance(node, ast.Name):
                # Nodo de nombre: se valida que el nombre esté en
nuestra lista permitida.
                if node.id not in nombres_permitidos:
                    raise ValueError(f"Nombre '{node.id}' no
permitido")
                # Se marca que se encontró un nombre permitido.
                found_name[0] = True
            else:
                # Si se encuentra cualquier otro nodo, se rechaza la
expresión.
                raise ValueError(f"Nodo '{type(node).__name__}' no
permitido")

        # Se inicia la validación recorriendo todo el árbol.
        _check_node(node)
```

```
        # Si no se ha encontrado ningún nombre (por ejemplo, solo se
opera con escalares), se rechaza.
        if not found_name[0]:
            raise ValueError("Operaciones solo entre escalares no
están permitidas")

        # Si la validación es exitosa, se retorna el nodo AST para una
evaluación posterior.
        return node

    def validar(self, entry):
        """
        Función que se llama desde la interfaz gráfica (Tkinter) para
validar la expresión.
        Muestra un messagebox informando si la expresión es válida o
si contiene errores.
        """
        expr = entry.get()
        try:
            self.validar_expresion(expr)
            entry.config(bg = "green")
        except Exception as e:
            entry.config(bg = "red")
            messagebox.showerror("Error con la operación 'Math'",
str(e))


    #%%FUNCIONES PARA LA OPCIÓN DE GUARDAR EN ARCHIVO CSV
    #===========FUNCIONES PARA LA OPCIÓN DE GUARDAR EN ARCHIVO CSV


    def check_entries(self,x_var, y_vars, save_button):
        """
        Habilita el botón de guardar solo si el campo del eje X y
TODOS los campos del eje Y tienen texto.

        Parámetros:
          - x_var: StringVar asociado al campo de entrada del nombre
del eje X.
          - y_vars: Lista de StringVar asociados a los campos de
entrada del nombre del eje Y.
          - save_button: Botón de guardar que se habilitará o
deshabilitará.
        """
        # Verifica que el campo de eje X no esté vacío.
        if not x_var.get().strip():
            save_button.config(state=tk.DISABLED)
            return
        # Verifica que cada campo del eje Y tenga texto.
        for y_var in y_vars:
            if not y_var.get().strip():
                save_button.config(state=tk.DISABLED)
                return
        # Si todos los campos tienen contenido, se habilita el botón.
        save_button.config(state=tk.NORMAL)


    """
    Extrae los datos del subplot seleccionado (del primer objeto
'line' en ax para X y
    de cada línea para Y) y guarda un archivo CSV usando:
```

```
      - La primera columna con el nombre del eje X (según x_var)
      - Una columna por cada línea con el nombre correspondiente de
cada entrada en y_vars.

    Parámetros:
      - axis_window: La ventana de diálogo de ejes que se cierra al
guardar.
      - ax: El objeto ax del subplot seleccionado.
      - x_var: StringVar con el nombre asignado al eje X.
      - y_vars: Lista de StringVar con los nombres asignados a cada
eje Y.
    """
    def save_csv(self,axis_window,ax, x_var, y_vars):

        # Se cierra la ventana de diálogo de ejes.
        axis_window.destroy()
        # Se obtienen las líneas graficadas en el subplot.
        lines = ax.get_lines()
        # Si no hay líneas, se muestra un mensaje de error y se
detiene la función.
        if not lines:
            messagebox.showerror("Error", "El plot seleccionado no
contiene datos.")
            return

        # Se asume que la primera línea contiene el vector X común
para todas las series.
        x_data = lines[0].get_xdata()
        # Se obtienen los datos Y de cada línea y se guardan en una
lista.
        y_data_all = [line.get_ydata() for line in lines]

        # Se construye la cabecera del CSV con el nombre del eje X
seguido de los nombres para cada eje Y.
        header = [x_var.get().strip()] + [y_var.get().strip() for
y_var in y_vars]

        # Se abre un diálogo para que el usuario elija la ubicación y
nombre del archivo CSV.
        file_path =
filedialog.asksaveasfilename(defaultextension=".csv",
                                                    filetypes=[("CSV
files", "*.csv")])
        # Si el usuario cancela, se sale de la función.
        if not file_path:
            return

        # Se abre el archivo en modo escritura y se escribe la
cabecera y los datos.
        with open(file_path, mode="w", newline="") as file:
            writer = csv.writer(file)
            writer.writerow(header)  # Escribe la primera fila con los
nombres de columnas.
            # Se recorre cada punto (asumiendo que todos los arrays
tienen la misma longitud).
            for i in range(len(x_data)):
                # Se construye cada fila: el valor de X y los
correspondientes valores Y de cada línea.
                row = [x_data[i]] + [y_data[i] for y_data in
y_data_all]
                writer.writerow(row)
```

```
        # Se muestra un mensaje informativo indicando que el archivo
se guardó correctamente.
        messagebox.showinfo("Guardado", f"Archivo guardado en
{file_path}")

    """
    Abre una ventana para solicitar los nombres de los ejes del
subplot seleccionado.
    Si en ax hay más de una línea, se crean tantos campos para el eje
Y como líneas existan.
    Además, se sugiere el nombre basado en la etiqueta asignada con
'label=' en ax.plot().

    Parámetros:
      - selection_window: La ventana de selección desde la cual se
abre este diálogo.
      - ax: El objeto ax del subplot seleccionado.
    """
    def open_axis_dialog(self, selection_window, ax):

        # Se crea una ventana secundaria para la definición de nombres
de ejes.
        axis_window = tk.Toplevel(selection_window)
        axis_window.title("Definir nombres de ejes")


        #================eje x
        # Se crea un campo de entrada para el nombre del eje X.
        tk.Label(axis_window, text="Nombre eje X:").grid(row=0,
column=0, padx=5, pady=5)
        x_var = tk.StringVar()
        tk.Entry(axis_window, textvariable=x_var).grid(row=0,
column=1, padx=5, pady=5)
        #==================

        #=================eje y
        # Se obtienen las líneas graficadas en el subplot.
        lines = ax.get_lines()
        num_lines = len(lines)

        #Tan solo se pedirá el nombre si son menos de 10
        if num_lines <= 10:
            # Lista que contendrá los StringVar para cada campo del
eje Y.
            y_vars = []
            # Se crea un campo de entrada para cada línea existente.
            for i in range(num_lines):
                # Se obtiene la etiqueta de la línea asignada con
label=; si no existe o es predeterminada, se asigna un nombre
genérico.
                suggested_label = lines[i].get_label()
                if not suggested_label or
suggested_label.startswith('_'):
                    suggested_label = f"Serie {i+1}"
                # Se crea la etiqueta y campo de entrada, sugerido el
valor obtenido.
                tk.Label(axis_window, text=f"Nombre Y
{i+1}:").grid(row=i+1, column=0, padx=5, pady=5)
                y_var = tk.StringVar(value=suggested_label)
                tk.Entry(axis_window,
textvariable=y_var).grid(row=i+1, column=1, padx=5, pady=5)
```

```
                y_vars.append(y_var)
        else:#si son más de 10 curvas, se pondrá el nombre predefinido
(sus labels)
            y_vars = []
            for i in range(num_lines):
                # Se obtiene la etiqueta de la línea asignada con
label=; si no existe o es predeterminada, se asigna un nombre
genérico.
                suggested_label = lines[i].get_label()
                if not suggested_label or
suggested_label.startswith('_'):
                    suggested_label = f"Serie {i+1}"
                y_var = tk.StringVar(value=suggested_label)
                y_vars.append(y_var)
        #===================

        # Se crea el botón "Guardar CSV", inicialmente deshabilitado.
        save_button = tk.Button(axis_window, text="Guardar CSV",
                                command=lambda:
self.save_csv(axis_window,ax, x_var, y_vars),
                                state=tk.DISABLED)
        save_button.grid(row=num_lines+1, column=0, columnspan=2,
pady=10)

        # Función interna para validar que el campo del eje X y todos
los campos de eje Y tengan texto.
        def check_all_entries(*args):
            self.check_entries(x_var, y_vars, save_button)

        # Se añaden trazas a los StringVar para que, al modificarse,
se verifique la validez de los datos.
        x_var.trace("w", check_all_entries)
        for y_var in y_vars:
            y_var.trace("w", check_all_entries)

    """
        Abre una ventana que muestra una grilla de botones para
seleccionar un subplot.
        Cada botón se coloca en la celda de la grilla según la
posición real del subplot, obtenida de:
          - fila: ax.get_subplotspec().rowspan.start
          - columna: ax.get_subplotspec().colspan.start
        El texto de cada botón es "Plot (fila,columna) (Título)".

        Parámetros internos relevantes:
          - root: Ventana principal de tkinter.
          - axes: Lista de objetos ax creados.
          - no_filas, no_columnas: Dimensiones de la grilla en la
ventana de selección.


        """
    def open_selection_window(self):

        # Se crea una ventana secundaria (Toplevel) para la selección
de subplots.
        selection_window = tk.Toplevel(self.root)
        selection_window.title("Selecciona un Plot")

        axes = plot.ax
```

```
        no_filas = int(self.subplotmatrix[0][0])
        no_columnas =  int(self.subplotmatrix[0][1])

        # Se crea un diccionario que mapea las posiciones (fila,
columna) reales a cada ax,
        # usando la información obtenida con get_subplotspec() de cada
subplot.
        ax_mapping = {}
        for ax in axes:
            subspec = ax.get_subplotspec()  # Obtiene el objeto
SubplotSpec del ax.
            r = subspec.rowspan.start    # Índice de la fila donde
inicia el subplot.
            c = subspec.colspan.start    # Índice de la columna donde
inicia el subplot.
            ax_mapping[(r, c)] = ax      # Se guarda el ax en el
diccionario con la clave (r, c).

        # Se recorre cada celda de la grilla definida por no_filas y
no_columnas.
        for r in range(no_filas):
            for c in range(no_columnas):
                # Si existe un ax en la celda (r, c), se crea un botón
para seleccionarlo.
                if (r, c) in ax_mapping:
                    ax = ax_mapping[(r, c)]
                    # El texto del botón incluye la posición (r+1,
c+1) y el título del subplot.
                    btn_text = f"Plot ({r+1},{c+1})
({ax.get_title()})"
                    # Se crea el botón y se asocia una función lambda
para abrir el diálogo de ejes.
                    btn = tk.Button(selection_window, text=btn_text,
width=20,
                                    command=lambda ax=ax:
self.open_axis_dialog(selection_window,ax))
                    # Se posiciona el botón en la grilla.
                    btn.grid(row=r, column=c, padx=5, pady=5)
                else:
                    # Si no hay un ax en la posición, se coloca una
etiqueta indicando "Vacío".
                    lbl = tk.Label(selection_window, text="Vacío",
width=20, relief=tk.SUNKEN)
                    lbl.grid(row=r, column=c, padx=5, pady=5)

        #return selection_window



    #=============================================================


    #===========Funciones para la selección de operaciones en plotmix
    #=============================================================

    # Función que actualiza el campo de entrada numérico según la
operación seleccionada.
    # Parámetros:
    # - frame_num: frame donde se muestra el Entry.
    # - self.op_var: variable que contiene la operación seleccionada.
    # - self.num_var: variable para almacenar el número ingresado.
```

```
    def update_num_entry(self, frame_num):
        # Elimina todos los widgets existentes en el frame para
limpiarlo.
        for widget in frame_num.winfo_children():
            widget.destroy()
        # Si la operación seleccionada es "sumar" o "multiplicar", se
muestra el Entry para ingresar el número.
        if self.op_var.get() in ["sumar", "multiplicar"]:
            label_num = ttk.Label(frame_num, text="Ingresa el
número:")  # Crea una etiqueta para el Entry.
            label_num.pack(side="left", padx=5, pady=5)
# Empaqueta la etiqueta.
            self.entry = ttk.Entry(frame_num,
textvariable=self.num_var)           # Crea un Entry asociado a
self.num_var.
            self.entry.pack(side="left", padx=5, pady=5)
# Empaqueta el Entry.

    # Función que actualiza las opciones de operación según las
columnas seleccionadas en el Listbox.
    # Parámetros:
    # - event: evento de selección en el Listbox.
    # - listbox: Listbox que contiene las columnas.
    # - frame_radio: frame donde se mostrarán los radio buttons para
las operaciones.
    # - frame_num: frame para la entrada numérica.
    # - self.op_var: variable que almacenará la operación
seleccionada.
    # - self.num_var: variable que almacenará el número ingresado.
    def actualizar_opciones(self,event, listbox, frame_radio,
frame_num):
        # Elimina todos los widgets existentes en el frame de radio
buttons.
        for widget in frame_radio.winfo_children():
            widget.destroy()
        # Elimina todos los widgets existentes en el frame del Entry
numérico.
        for widget in frame_num.winfo_children():
            widget.destroy()
        # Obtiene los índices de las columnas seleccionadas en el
Listbox.
        selected_indices = listbox.curselection()
        # Crea una lista con los nombres de las columnas
seleccionadas.
        selected_columns = [listbox.get(i) for i in selected_indices]
        n = len(selected_columns)

        # Si no se ha seleccionado ninguna columna, muestra un
mensaje.
        if n == 0:
            label = ttk.Label(frame_radio, text="No se han
seleccionado columnas")
            label.pack(padx=5, pady=5)
        # Si se selecciona una única columna, muestra las opciones
"Sumar" y "Multiplicar" y un Entry para ingresar un número.
        elif n == 1:
            label = ttk.Label(frame_radio, text="Opciones para 1
columna:")
            label.pack(padx=5, pady=5)
            rb_sumar = ttk.Radiobutton(frame_radio, text="Sumar",
variable=self.op_var, value="sumar",
```

```
                                        command=lambda:
self.update_num_entry(frame_num))
            rb_sumar.pack(anchor="w", padx=10, pady=2)
            rb_multiplicar = ttk.Radiobutton(frame_radio,
text="Multiplicar", variable=self.op_var, value="multiplicar",
                                                command=lambda:
self.update_num_entry(frame_num))
            rb_multiplicar.pack(anchor="w", padx=10, pady=2)
            self.update_num_entry(frame_num)
        # Si se seleccionan varias columnas, muestra las opciones para
operar entre ellas (sin campo numérico).
        else:
            label = ttk.Label(frame_radio, text="Opciones para varias
columnas:")
            label.pack(padx=5, pady=5)
            rb_sumar_entre = ttk.Radiobutton(frame_radio, text="Sumar
entre ellas", variable=self.op_var, value="sumar_entre_columnas")
            rb_sumar_entre.pack(anchor="w", padx=10, pady=2)
            rb_multiplicar_entre = ttk.Radiobutton(frame_radio,
text="Multiplicar entre ellas", variable=self.op_var,
value="multiplicar_entre_columnas")
            rb_multiplicar_entre.pack(anchor="w", padx=10, pady=2)

    # Función principal que crea la ventana modal para seleccionar
operaciones sobre un DataFrame.
    # Parámetros:
    # - diccionario: diccionario jerarquizado que contiene el
DataFrame.
    # - df_path: lista de claves para acceder al DataFrame dentro del
diccionario.
    # - listbox_width: ancho del Listbox para la selección de
columnas.
    # - callback: función que se llamará al finalizar la selección
(puede ser None).
    def abrir_ventana_operaciones(self):
        self.root.attributes('-disabled', True)  # Deshabilita la
ventana principal para hacerla modal.
        ventana = tk.Toplevel(self.root)           # Crea una nueva
ventana modal.
        ventana.title("Operaciones sobre DataFrame")  # Define el
título de la ventana modal.
        ventana.geometry("500x400")           # Define el tamaño de la
ventana modal.

        # Función interna que se ejecuta al cerrar la ventana; se
mantiene anidada para acceder a 'open_button' y 'selection_window'
        def on_close():
            self.root.attributes('-disabled', False)  # Reactiva la
ventana principal.
            ventana.destroy()  # Cierra la subventana

        ventana.protocol("WM_DELETE_WINDOW", on_close)  # Asocia el
evento de cierre de la ventana a 'on_close'

        # Crea un frame para la selección de columnas.
        frame_columnas = ttk.LabelFrame(ventana, text="Selecciona
Columnas")
        frame_columnas.pack(fill="both", expand=True, padx=10,
pady=10)
```

```
        # Navega a través del diccionario para obtener el DataFrame
usando la ruta proporcionada.
        df_selec = self.DF.dataframes_selec
        df = self.DF.dataframes

        if df_selec:#Si se ha seleccionado algún grupo

            #obtenemos el grupo seleccionado y el primer elemento de
este grupo como representante
            grupo = list(df_selec.keys())[0]
            primer_elemento = df_selec[grupo][0]

            columnas = list(df[grupo][primer_elemento].columns)
# Obtiene la lista de nombres de columnas del DataFrame.

        else:
            columnas = []


        # Crea un Listbox para la selección de columnas con modo
extendido y ancho configurable.
        listbox = tk.Listbox(frame_columnas, selectmode=tk.EXTENDED)
        listbox.pack(side=tk.LEFT, fill=tk.BOTH, expand=True, padx=5,
pady=5)

        # Crea un scrollbar para el Listbox.
        scrollbar = ttk.Scrollbar(frame_columnas, orient="vertical",
command=listbox.yview)
        scrollbar.pack(side=tk.RIGHT, fill=tk.Y)
        listbox.config(yscrollcommand=scrollbar.set)

        # Inserta cada columna en el Listbox.
        for col in columnas:
            listbox.insert(tk.END, col)

        # Crea un frame para la selección de operaciones.
        frame_operaciones = ttk.LabelFrame(ventana, text="Selecciona
Operación")
        frame_operaciones.pack(fill="both", expand=True, padx=10,
pady=10)
        frame_radio = ttk.Frame(frame_operaciones)  # Frame para los
radio buttons (opciones de operación).
        frame_radio.pack(fill="x", padx=5, pady=5)
        frame_num = ttk.Frame(frame_operaciones)    # Frame para la
entrada numérica.
        frame_num.pack(fill="x", padx=5, pady=5)

        self.op_var=  tk.StringVar(value="")       # Variable para
almacenar la operación seleccionada.
        self.num_var=  tk.StringVar(value="")

        # Asocia el evento de selección del Listbox para actualizar
las opciones de operación.
        listbox.bind("<<ListboxSelect>>",
                     lambda event: self.actualizar_opciones(
event,listbox, frame_radio, frame_num))
        # Crea un botón que finaliza la selección y llama a la función
cerrar_ventana.
        btn_finalizar = ttk.Button(ventana, text="Finalizar
selección", command = lambda:  print( self.op_var.get() ) )
```

```
        btn_finalizar.pack(pady=10)

        ventana.grab_set()                      # Hace que la ventana
modal bloquee la interacción con la principal



    #=================================================================
    #%%FUNCIONES PARA ANÁLISIS

    '''
    Función que procesa la selección realizada por el usuario.
    Actualiza las variables internas:
      - self.estad: contendrá la lista básica de cadenas (modelo, tipo
y, en caso de sobrecarga, la opción)
      - self.columnas_estad: contendrá la lista de listas con los
valores de cada fila de extras (cada fila es [curva, columna]), en
casos especiales, habrá un último element que indique el tipo de curva
(2A, 10A)
    Finalmente, se imprime el contenido de ambas variables.

    Parámetros:
      - ventana: la ventana actual desde la que se llama la función.
      - combo_principal: combobox de selección principal (modelo).
      - combo_sub: combobox de subopción (tipo).
      - combo_subsub: combobox de sub-subopción (opcional; se utiliza
en estadísticas de sobrecarga).
      - extras: diccionario con la información extra (los pares de
combobox extra), donde las claves indican la fila.
    '''
    def mensaje_continuar(self, ventana, combo_principal, combo_sub,
combo_subsub=None, extras=None):
        opcion = combo_principal.get()  # Obtener la opción principal

        def procesar_seleccion():
            # Actualizar la variable interna self.estad según la
opción seleccionada
            if opcion == "Análisis temporales":
                # Almacenar 2 cadenas: el contenido del modelo y el
del tipo.
                self.estad = [combo_principal.get(), combo_sub.get()]
            elif opcion == "Análisis de sobrecarga":
                # Almacenar 3 cadenas: modelo, tipo y la opción
(sub-subopción).
                self.estad = [combo_principal.get(), combo_sub.get(),
combo_subsub.get()]
            else:
                self.estad = [combo_principal.get(), combo_sub.get()]

            # Procesar los extras (filas de combobox extras)
            self.columnas_estad = []
            if extras:
                # Para cada clave en el diccionario, se obtiene el
valor seleccionado usando .get()
                for key in sorted(extras.keys()):
                    # extras[key] es una tupla: (valor_curva (puede
ser None), combobox_de_columna)
                    # Se obtiene el contenido del combobox extra con
.get()
                    self.columnas_estad.append([extras[key][0] if
extras[key][0] is not None else "", extras[key][1].get()])
```

```
            # Realizar un print del resultado final
            print("Estad.:", self.estad)
            print("Columnas Estád.:", self.columnas_estad)
            self.continua_estadistica = True
            ventana.destroy()


        # Si la opción requiere confirmación, se muestra una ventana
de confirmación
        if opcion in ["Estadísticas de caracterización", "Análisis de
sobrecarga"]:
            conf = tk.Toplevel(ventana)
            conf.title("Confirmación")
            tk.Label(conf, text="Esta estadística es específica para
los tipos de curvas.\nindicados, una mala inserción de curvas podría
producir\n resultados indeseados\n¿Desea continuar?").pack(padx=10,
pady=10)
            tk.Button(conf, text="Continuar",
                      command=lambda: [conf.destroy(),
procesar_seleccion()]).pack(side="left", padx=10, pady=10)
            tk.Button(conf, text="No continuar",
command=conf.destroy).pack(side="right", padx=10, pady=10)
            conf.grab_set()
        else:
            procesar_seleccion()

    '''
    Función que abre una nueva ventana con las opciones de
estadísticas y subopciones anidadas.
    '''
    def abrir_ventana_estadistica(self):
        ventana = tk.Toplevel(self.root)
        ventana.title("Ventana de Análisis")
        ventana.geometry("685x555")
        ventana.resizable(False, False)

        self.continua_estadistica = False

        # Opciones y subopciones definidas
        opciones = ["Análisis de sobrecarga", "Análisis temporales"]
        subopciones = {
            "Análisis de sobrecarga": {
                "Sobrecarga": ["Valor máximo", "Intervalo de
limitación", "Energía", "Corriente media limitación"],
                "Carga capacitiva": ["Valor máximo", "Intervalo de
transición", "Energía"],
                "Cortocircuito": ["Valor máximo", "Intervalo de
limitación", "Energía", "Corriente media limitación"]
            },
            "Análisis temporales": ["Derivada"]
        }

        # Crear frame contenedor general
        frame_contenedor = tk.Frame(ventana)
        frame_contenedor.pack(padx=10, pady=10, fill="both",
expand=True)

        # LabelFrame para selección principal de estadísticas
        frame_seleccion = ttk.LabelFrame(frame_contenedor,
text="Selección de análisis")
```

```
        frame_seleccion.pack(padx=10, pady=10, fill="x")

        # Opción principal (modelo)
        ttk.Label(frame_seleccion, text="Selecciona el
modelo:").pack(padx=10, pady=5, anchor="w")
        combo_principal = ttk.Combobox(frame_seleccion,
values=opciones, state="readonly", width=22)
        combo_principal.pack(padx=10, pady=5)
        combo_principal.current(0)

        # Subopción (tipo)
        ttk.Label(frame_seleccion, text="Selecciona el tipo de
falla:").pack(padx=10, pady=5, anchor="w")
        combo_sub = ttk.Combobox(frame_seleccion, state="readonly")
        combo_sub.pack(padx=10, pady=5)

        # Sub-sub opción (se muestra según corresponda)
        label_subsub = ttk.Label(frame_seleccion, text="Seleccione la
opción:")
        combo_subsub = ttk.Combobox(frame_seleccion, state="readonly",
width=22)

        # LabelFrame para los extras (asignación de columnas y curvas)
        self.frame_extras = ttk.LabelFrame(frame_contenedor,
text="Asignación de columnas")
        self.frame_extras.pack(padx=10, pady=10, fill="x")

        # Diccionario para almacenar las filas de pares de combobox
extras.
        self.extras_seleccion = {}

        '''
        Función para actualizar las subopciones y sub-subopciones al
cambiar la opción principal.
        '''
        def actualizar_subopciones(event):
            opcion = combo_principal.get()
            datos = subopciones.get(opcion, [])
            if isinstance(datos, dict):
                lista_sub = list(datos.keys())
                combo_sub['values'] = lista_sub
                if lista_sub:
                    combo_sub.current(0)
                label_subsub.pack(padx=10, pady=5, anchor="w")
                combo_subsub.pack(padx=10, pady=5)
                actualizar_subsub()
            else:
                combo_sub['values'] = datos
                if datos:
                    combo_sub.current(0)
                label_subsub.pack_forget()
                combo_subsub.pack_forget()
            mostrar_extras()

        '''
        Función para actualizar las sub-subopciones al cambiar la
subopción.
        '''
        def actualizar_subsub(event=None):
            opcion = combo_principal.get()
            datos = subopciones.get(opcion, {})
```

```
            if isinstance(datos, dict):
                sub = combo_sub.get()
                lista_subsub = datos.get(sub, [])
                combo_subsub['values'] = lista_subsub
                if lista_subsub:
                    combo_subsub.current(0)
            mostrar_extras()

        '''
        Función que muestra (o reconstruye) los extras en función de
las selecciones.
        Dependiendo de la selección:
          - Para Análisis generales:
              *Si el detalle es la derivada: se pregunta por el eje x,
el eje y, y los parámetros de la aproximación de Savitz-Golay
          - Para Análisis de sobrecarga:
            * Si el detalle es "Valor máximo": se muestra 1 fila con
3 opciones en el combobox de curva.
            * Si el detalle es "Intervalo de transición": se muestra
1 fila con 1 opción en el combobox de curva.
            * Si el detalle es "Intervalo de limitación", "Energía"
o "Corriente media limitación": se muestran 2 filas.
              En la primera fila, el combobox de curva solo tiene
la opción "I".
              En la segunda fila, el combobox de curva solo tiene
"VdsJFET" y "VdsPMOS".
        '''
        def mostrar_extras(event=None):
            # Limpiar el frame de extras
            for widget in self.frame_extras.winfo_children():
                widget.destroy()
            self.extras_seleccion.clear()

            opcion = combo_principal.get()
            if opcion == "Análisis temporales":
                detalle = combo_sub.get()
                if detalle == "Derivada":
                    # Primera fila: eje x
                    frame_row4 = ttk.Frame(self.frame_extras)
                    frame_row4.pack(padx=10, pady=5, fill="x")
                    label_col4 = ttk.Label(frame_row4,
text="Selecciona el eje x:")
                    label_col4.pack(side="left", padx=5)
                    combo_col4 = ttk.Combobox(frame_row4,
values=self.nombres_columnas, state="readonly", width=10)
                    combo_col4.pack(side="left", padx=5)
                    combo_col4.current(0)
                    self.extras_seleccion["Fila 1"] = ("ejex",
combo_col4)

                    # Segunda fila: eje y
                    frame_row5 = ttk.Frame(self.frame_extras)
                    frame_row5.pack(padx=10, pady=5, fill="x")
                    label_col5 = ttk.Label(frame_row5,
text="Selecciona el eje y:")
                    label_col5.pack(side="left", padx=5)
                    combo_col5 = ttk.Combobox(frame_row5,
values=self.nombres_columnas, state="readonly", width=10)
                    combo_col5.pack(side="left", padx=5)
                    combo_col5.current(0)
```

```
                    self.extras_seleccion["Fila 2"] = ("ejey",
combo_col5)

                    # Tercera fila: window_lenght
                    frame_window_lenght = ttk.Frame(self.frame_extras)
                    frame_window_lenght.pack(padx=10, pady=5,
fill="x")
                    label_window_lenght =
ttk.Label(frame_window_lenght, text="Selecciona el window lenght de la
aproximación (impar):")
                    label_window_lenght.pack(side="left", padx=5)
                    entry_window_lenght =
tk.Entry(frame_window_lenght, width=10,validate="key",
validatecommand=(self.validate_window_length, "%P"))
                    entry_window_lenght.pack(side="left", padx=5)
                    entry_window_lenght.insert(0, "21")
                    self.extras_seleccion["Fila 3"] =
("window_lenght", entry_window_lenght)

                    # Cuarta fila: polyorder
                    frame_poly = ttk.Frame(self.frame_extras)
                    frame_poly.pack(padx=10, pady=5, fill="x")
                    label_poly = ttk.Label(frame_poly,
text="Selecciona el polyorder de la aproximación:")
                    label_poly.pack(side="left", padx=5)
                    entry_poly = tk.Entry(frame_poly,
width=10,validate="key", validatecommand=(self.validate_polyorder,
"%P"))
                    entry_poly.pack(side="left", padx=5)
                    entry_poly.insert(0, "3")
                    self.extras_seleccion["Fila 4"] = ("polyorder",
entry_poly)

                else:
                    pass


            elif opcion == "Análisis de sobrecarga":
                detalle = combo_subsub.get()
                # Para "Valor máximo": 1 fila con 3 opciones
                if detalle == "Valor máximo":
                    frame_row = ttk.Frame(self.frame_extras)
                    frame_row.pack(padx=10, pady=5, fill="x")
                    label_curve = ttk.Label(frame_row,
text="Selecciona la curva:")
                    label_curve.pack(side="left")
                    combo_curve = ttk.Combobox(frame_row, values=["I",
"VdsJFET", "VdsPMOS"], state="readonly", width=10)
                    combo_curve.pack(side="left", padx=5)
                    label_col = ttk.Label(frame_row, text="Selecciona
la columna:")
                    label_col.pack(side="left", padx=5)
                    combo_col = ttk.Combobox(frame_row,
values=self.nombres_columnas, state="readonly", width=10)
                    combo_col.pack(side="left", padx=5)
                    combo_curve.current(0)
                    combo_col.current(0)
                    self.extras_seleccion["Fila 1"] =
(combo_curve.get(), combo_col)
                # Para "Intervalo de transición": 1 fila con 1 opción
(I)
```

```
                elif detalle =="Intervalo de transición":
                    # Primera fila: combobox de curva solo con "I"
                    frame_row3 = ttk.Frame(self.frame_extras)
                    frame_row3.pack(padx=10, pady=5, fill="x")
                    label_curve3 = ttk.Label(frame_row3,
text="Selecciona la curva:")
                    label_curve3.pack(side="left")
                    combo_curve3 = ttk.Combobox(frame_row3,
values=["I"], state="readonly", width=10)
                    combo_curve3.pack(side="left", padx=5)
                    label_col3 = ttk.Label(frame_row3,
text="Selecciona la columna:")
                    label_col3.pack(side="left", padx=5)
                    combo_col3 = ttk.Combobox(frame_row3,
values=self.nombres_columnas, state="readonly", width=10)
                    combo_col3.pack(side="left", padx=5)
                    combo_curve3.current(0)
                    combo_col3.current(0)
                    self.extras_seleccion["Fila 1"] =
(combo_curve3.get(), combo_col3)

                    # Segunda fila: tipo de curva (corriente de
alimentación)
                    frame_type2 = ttk.Frame(self.frame_extras)
                    frame_type2.pack(padx=10, pady=5, fill="x")
                    label_type2 = ttk.Label(frame_type2,
text="Selecciona el tipo de curva:")
                    label_type2.pack(side="left")
                    combo_type2 = ttk.Combobox(frame_type2,
values=["2A", "10A"], state="readonly", width=5)
                    combo_type2.pack(side="left", padx=5)
                    combo_type2.current(0)
                    self.extras_seleccion["Tipo"] = ( "Tipo" ,
combo_type2 )

                # Para "Intervalo de limitación", "Energía" y
"Corriente media limitación": 2 filas
                elif detalle in ["Intervalo de limitación", "Energía",
"Corriente media limitación"]:
                    # Primera fila: combobox de curva solo con "I"
                    frame_row1 = ttk.Frame(self.frame_extras)
                    frame_row1.pack(padx=10, pady=5, fill="x")
                    label_curve1 = ttk.Label(frame_row1,
text="Selecciona la curva (fila 1):")
                    label_curve1.pack(side="left")
                    combo_curve1 = ttk.Combobox(frame_row1,
values=["I"], state="readonly", width=10)
                    combo_curve1.pack(side="left", padx=5)
                    label_col1 = ttk.Label(frame_row1,
text="Selecciona la columna:")
                    label_col1.pack(side="left", padx=5)
                    combo_col1 = ttk.Combobox(frame_row1,
values=self.nombres_columnas, state="readonly", width=10)
                    combo_col1.pack(side="left", padx=5)
                    combo_curve1.current(0)
                    combo_col1.current(0)
                    self.extras_seleccion["Fila 1"] =
(combo_curve1.get(), combo_col1)

                    # Segunda fila: combobox de curva solo con
"VdsJFET" y "VdsPMOS"
```

```
                    frame_row2 = ttk.Frame(self.frame_extras)
                    frame_row2.pack(padx=10, pady=5, fill="x")
                    label_curve2 = ttk.Label(frame_row2,
text="Selecciona la curva (fila 2):")
                    label_curve2.pack(side="left")
                    combo_curve2 = ttk.Combobox(frame_row2,
values=["VdsJFET", "VdsPMOS"], state="readonly", width=10)
                    combo_curve2.pack(side="left", padx=5)
                    label_col2 = ttk.Label(frame_row2,
text="Selecciona la columna:")
                    label_col2.pack(side="left", padx=5)
                    combo_col2 = ttk.Combobox(frame_row2,
values=self.nombres_columnas, state="readonly", width=10)
                    combo_col2.pack(side="left", padx=5)
                    combo_curve2.current(0)
                    combo_col2.current(0)
                    self.extras_seleccion["Fila 2"] =
(combo_curve2.get(), combo_col2)

                    # Tercera fila: tipo de curva (corriente de
alimentación)
                    frame_type2 = ttk.Frame(self.frame_extras)
                    frame_type2.pack(padx=10, pady=5, fill="x")
                    label_type2 = ttk.Label(frame_type2,
text="Selecciona el tipo de curva:")
                    label_type2.pack(side="left")
                    combo_type2 = ttk.Combobox(frame_type2,
values=["2A", "10A"], state="readonly", width=5)
                    combo_type2.pack(side="left", padx=5)
                    combo_type2.current(0)
                    self.extras_seleccion["Tipo"] = ("Tipo",
combo_type2 )

                else:
                    pass
            else:
                pass

        # Vincular eventos para actualizar la selección y extras
        combo_principal.bind("<<ComboboxSelected>>",
actualizar_subopciones)
        combo_sub.bind("<<ComboboxSelected>>", actualizar_subsub)
        combo_subsub.bind("<<ComboboxSelected>>", mostrar_extras)

        actualizar_subopciones(None)  # Inicializar combos

        # Botón para mostrar: se activa el mensaje de confirmación y,
en caso de continuar, se procesa la selección.
        tk.Button(ventana, text="mostrar",
                  command=lambda: self.mensaje_continuar(
                      ventana, combo_principal, combo_sub,
combo_subsub, self.extras_seleccion
                  )).pack(padx=10, pady=10, side="bottom")

        self.root.wait_window(ventana)#para detener la ejecución aquí
hasta que la ventana se destruya
```

Clase “Gestor de datos”

```
import pandas as pd
import os
try:
    import Tkinter as tk
except:
    import tkinter as tk
try:
    from Tkinter import ttk
except:
    from tkinter import ttk

from tkinter import filedialog, messagebox  # Importa filedialog para
diálogos de selección de archivos y carpetas

import tkinter.font as fnt

import re

import csv

import psutil

import math

import gc #El garrbage collector, para deshacernos de la memoria
indeseada activamente con los plots porque a veces spyder no lo hace


try:
    # Intenta importar la función 'load_dictionary' desde el módulo
'spyder_kernels.utils.iofuncs'.
    from spyder_kernels.utils.iofuncs import load_dictionary
except ImportError:
    # Si no se puede importar la función, muestra un mensaje de error
en un cuadro de mensaje.
    messagebox.showerror("Error", "No se pudo importar
'load_dictionary' desde 'spyder_kernels.utils.iofuncs'.\n"
                                  "Asegúrate de estar ejecutando este
código en un entorno de Spyder.")
    raise  # Detiene la ejecución si ocurre un error de importación.


"""
Clase Gestor_datos

Encargada de la gestión, validación y organización de los datos para
su posterior
procesamiento y visualización. Su función principal es asegurar que la
información
ingresada sea consistente, estructurada y eficiente de manejar a lo
largo del análisis.

Principales responsabilidades:
- Carga de datos desde distintas fuentes, incluyendo la detección
automática del
  inicio de los datos tabulados, evitando procesar encabezados o
metadatos.
```

```
- Monitorización del uso de memoria RAM para evitar sobrecargas,
alertando al usuario
  en caso de consumo elevado y sugiriendo acciones correctivas.
- Organización jerarquizada de los datos en estructuras tipo
diccionario, facilitando
  el acceso y manejo eficiente.
- Visualización estructurada de los datos cargados, mostrando la
jerarquía entre
  claves y dataframes, limitando la vista a 100 filas para mantener el
rendimiento.
- Interfaz interactiva para la selección de dataframes relevantes para
análisis,
  controlando la cantidad seleccionada para evitar sobrecargas y
validar la calidad
  de los datos antes de continuar con el procesamiento.

Esta clase es fundamental para garantizar la integridad y usabilidad
de los datos
a lo largo de todo el flujo de trabajo del sistema.
"""


"""
Funciones de validación de apoyo para la clase "Gestor_Datos"

"""

"""
Función que verifica el uso de la memoria RAM del sistema.

Parámetros:
- threshold (int): El umbral de porcentaje de uso de RAM. Si el uso
supera este valor, la función devuelve True.

Retorna:
- bool: True si el uso de RAM es mayor que el umbral, False si no lo
es.
"""
def check_ram(threshold):
    return psutil.virtual_memory().percent > threshold  # Verifica si
el porcentaje de RAM excede el umbral dado


"""
Función de validación: permite solo números y longitud máxima de 5
caracteres.
Permite cadena vacía.
Parámetros:
- P (str): Cadena a validar.

Retorna:
- bool: True si cumple las condiciones, False en caso contrario.
"""
def validar_numeros_y_longitud(P):
    if P == "":  # Permitir que esté vacío
        return True
    if P.isdigit() and len(P) <= 5:  # Solo permitir números y
longitud <= 5
        return True
    else:
        return False
```

```
"""
Función de validación: no permite espacios y longitud máxima de 10
caracteres.
Parámetros:
- P (str): Cadena a validar.

Retorna:
- bool: True si no tiene espacios y longitud <= 10, False en caso
contrario.
"""
def validar_sin_espacios_y_longitud(P):
    if " " in P:  # Verifica si hay espacios en el valor de P
        return False
    if len(P) > 10:  # Verifica si la longitud excede los 10
caracteres
        return False
    return True


"""
Función para verificar si un valor es numérico.
Parámetros:
- valor: Valor a comprobar.

Retorna:
- bool: True si es numérico (puede ser float), False si no.
"""
def es_numero(valor):
    try:
        float(valor)
        return True
    except ValueError:
        return False


"""
Función que deshabilita un widget (botón u otro) y cambia su color de
fondo a gris.
Parámetros:
- x: widget Tkinter a modificar.
"""
def apagar(x):
    x["state"] = tk.DISABLED
    x["bg"] = "grey"


"""
Función para crear una ventana emergente que muestra información.
Parámetros:
- root: ventana raíz de Tkinter.
- informacion (str): texto a mostrar en la ventana.
"""
def informacion(root, informacion):
    newWindow = tk.Toplevel(root)
    newWindow.grab_set()  # Bloquea interacción con otras ventanas
hasta cerrar esta
    newWindow.resizable(False, False)  # No permitir cambiar tamaño
    newWindow.title("Información")
    newWindow.geometry("1000x600")
```

```
    info = tk.Label(newWindow, text=informacion, justify="left",
padx=2)
    info.pack()


"""
Función para cambiar el texto de una etiqueta (label).
Parámetros:
- label: widget Label de Tkinter.
- texto (str): nuevo texto para la etiqueta.
"""
def cambiatexto(label, texto):
    label["text"] = texto


"""
Función para alternar el color de un botón y habilitar/deshabilitar
listas de widgets asociados.
Parámetros:
- boton: widget botón de Tkinter.
- color (tuple): par de colores (color_inicial, color_alternativo).
- listacombobox (list): lista de widgets ComboBox a
habilitar/deshabilitar.
- listabotones (list): lista de widgets botones a
habilitar/deshabilitar.
"""
def toggleoption(boton, color, listacombobox, listabotones):
    if boton["background"] == color[0]:
        boton["background"] = color[1]
        for i in listacombobox:
            i["state"] = "disabled"
        for i in listabotones:
            i["state"] = tk.DISABLED
    elif boton["background"] == color[1]:
        boton["background"] = color[0]
        for i in listacombobox:
            i["state"] = "readonly"
        for i in listabotones:
            i["state"] = tk.NORMAL


"""
Detecta la primera línea de datos numéricos consecutivos en un CSV y
cuenta el total de filas.
Verifica que la línea detectada y las siguientes tres contengan datos
numéricos en las columnas indicadas.
Evita falsos positivos si la cabecera contiene números o está
tabulada.

Parámetros:
- path (str): ruta al archivo CSV.
- use_columns (list[int], opcional): índices de columnas a analizar.
Por defecto analiza todas.

Retorna:
- tuple (int, int): (fila_inicio_datos, total_filas)
    fila_inicio_datos: índice (0-based) de la primera línea de datos
numéricos.
    total_filas: número total de líneas del archivo.
"""
```

```
def detectar_inicio_y_conteo_total(path, use_columns=None):
    if use_columns is not None:
        use_columns = set(use_columns)
    with open(path, 'r', encoding='utf-8') as archivo:
        ventana = []
        for idx, linea in enumerate(archivo):
            ventana.append((idx, linea))
            if len(ventana) > 4:
                ventana.pop(0)
            if len(ventana) == 4:
                todas_numericas = True
                for _, ln in ventana:
                    valores = ln.strip().split(',')
                    if use_columns is not None:
                        valores_considerados = [v for i, v in
enumerate(valores) if i in use_columns]
                    else:
                        valores_considerados = valores
                    if not all(es_numero(v) for v in
valores_considerados):
                        todas_numericas = False
                        break
                if todas_numericas:
                    fila_inicio_datos = ventana[0][0]
                    total_filas = idx + 1 + sum(1 for _ in archivo)
                    break
    return fila_inicio_datos, total_filas



class Gestor_Datos():

    ''' Esta clase contendrá un diccionario, el cual tendrá otro
diccionario anidado por cada grupo de dataframes que se quiera cargar
    La clave de cada uno será la introducida por el usuario'''
    dataframes={}

    ''' Este diccionario contendrá una sola clave que será el grupo de
dataframes que se haya seleccioando, y dentro de la clave
    una lista que contiene todos los nombres de dentro del grupo de
dataframes'''
    dataframes_selec = {}

    ''' De la misma forma, un diccionario para los nombres de ls
columnas de cada grupo '''
    colu_dataframes = {}


    "También querremos saber el número de filas que tienen nuestros
dataframes"
    num_filas = {}

    "Además, tendremos una variable tipo string donde guardaremos la
información de los datos"
    "que se hayan cargado"

    info = ""

    "-------------A partir de aquí tendremos funciones referentes a
los datos, pero cuyos valores no se generan con nada"
    "referente a esta clase----------"
```

```
    "También tendremos una variable que contendrá los índices de los
dataframes seleccionados en la lista ya cargada"
    "ESTO SE HARÁ EXCLUSIVAVMENTE EN LA CLASE APLICACION() , DE
MOMENTO SE INICIALIZARÁ POSTERIORMENTE"
    "COMO TODOS LOS ÍNDICES DE LOS NÚMEROS SELECCIONADOS"
    "Al tener aquí los números seleccionados, accederemos a la
posición de cada número en el array de índices"
    "para saber la posición del dataframe en cuestión"

    "Variables que contienen información útil sobre lo que se ha
cargado"

    seleccion = []

    nombrecols = []

    numdataframes = 0

    num_filas = 0

    #variable intermedia que controlará los cambios en el límite de
uso de memoria ram dentro de esta clase
    ram_dataframe = 0

    #En esta función , que deberemos llamar desde el principio,
prepararemos todo
    #lo relacionado con los data frames a utilizar
    #Usaremos tkinter
    def iniciador(self, root, memoria_ram):

        self.ram_dataframe = memoria_ram

        self.loaded_workspace = {}  # Diccionario vacío que almacenará
el contenido del workspace cargado desde un archivo .spydata.
        #Inicializamos la raíz principal de tkinter
        #self.cargaroot = tk.Tk()

        self.cargaroot = tk.Toplevel(root)

        # Seteamos el nombre de la ventana principal y su tamaño
        self.cargaroot.title("Iniciador")
        self.cargaroot.geometry("385x495")
        self.cargaroot.resizable(False, False)

        # Registrar la función de validación
        self.validate_letras =
self.cargaroot.register(validar_sin_espacios_y_longitud)

        #Creamos un botón de información que al ser pulsado haga
emerger una ventana con el texto informativo
        #Lo hacemos con fondo de color amarillo (bg="yellow")
        #Cambiamos el borde y su tamaño para hacerlo más visible
("bd=8")

        self.btni = tk.Button(self.cargaroot, text="Información",
bg="yellow",bd=8, command=lambda: [informacion(self.cargaroot,

"Se ha de introducir la dirección completa de los ficheros y su nombre
sin la extensión (se asume .csv).\n\nLos modos de funcionamiento
implican cómo se seleccionarán los ficheros a cargar:\n\t-Modo 1. Se
```

```
cargarán TODOS los ficheros en la dirección empezando por el de índice
inicial.\n\t-Modo 2. Se cargarán SOLO los ficheros de índice
seleccioando en una ventana posterior.\n\t-Modo3. Se cargarán TODOS
los ficheros comenzando desde el índice inicial MENOS los
que\n\tseleccionemos en una ventana psoterior\n\nEn cuanto al modo de
header:\n\t-Modo 1. Indicamos que los ficheros ya contienen header
para cada lista de puntos.\n\t-Modo 2. Indicamos que los ficheros NO
contienen header y queremos especificarlo\n\t posteriormente.\n\nEn
cuanto al número inicial de los ficheros, indicamos por qué número
empieza el índice de\nlos ficheros que queremos cargar, generalmente
será el 0, aunque puede cambiarse.\n\nEn cuanto a las columnas a
cargar, implica qué columnas querremos cargar (separadas por
comas)\nde nuestros ficheros, la numeración comienza desde el 0. Por
ejemplo, si nuestros ficheros contienen\n3 columnas y solo queremos
analizar la primera y la segunda, pondremos 0,1."

)])
        #Cambiamos la posición del botón, teniendo en cuenta que el
origen es la esquina
        #superior izquierda
        self.btni.place(x=0,y=0)
        #Lo añadimos a la lista

        # Crear el combobox con dos opciones

        self.opciones = tk.LabelFrame(self.cargaroot, text="Opciones")
        self.opciones.place(x = 120, y = 20, width=200, height=60)

        self.combobox_opciones = ttk.Combobox(self.opciones,
values=["Cargar csv", "Cargar dataframe"], width=15, state="readonly")
        self.combobox_opciones.pack()
        self.combobox_opciones.current(0)  # Seleccionamos la opción
por defecto
        # Asociamos la función de actualización con el combobox
        self.combobox_opciones.bind("<<ComboboxSelected>>", lambda
event: (self.actualizar_visibilidad_principal(event),
self.actualizar_estado_boton_cargar()))



        #El método pack() es un tipo de posicionamiento para los
widgets que
        #ajusta todo los elementos acomodándolos entre sí, para luego
hacer
        #la ventana raíz tan grande para contener todos estos
elementos.
        #Por ejemplo en el código anterior al ejecutar vemos que la
ventana raíz
        #se ajusta al tamaño del frame.

        #---------------------- para la selección de los ficheros

        self.rutas_absolutas = []
        self.nombres_archivos = []
        self.nombregrupo = ""
        self.columnasselec = []
        self.nombrescolumnas = []
        self.coincidencia = 0


        self.frame_columnas = tk.Frame(self.cargaroot)
```

```
        self.label_resultado = tk.Label(self.cargaroot, text="")
        self.label_resultado.pack()

        self.btnd = tk.Button(self.cargaroot, text ="buscar ficheros",
command=lambda: [self.seleccionar_archivos()] )
        #La posición de esto estará en 5 veces menos que en entry pero
en x =300
        self.btnd.place( x=100, y=85)




        #------------------------para columnas

        self.label_columnas = tk.Label(self.cargaroot,
text="Seleccione columnas:")

        self.label_columnas.place(x = 100, y = 125)

        self.listbox_columnas = tk.Listbox(self.cargaroot,
selectmode=tk.MULTIPLE, exportselection=False)
        self.listbox_columnas.place(x = 100, y = 145)

        self.btn_editar_columnas = tk.Button(self.cargaroot,
text="Editar nombres columnas", command=self.abrir_ventana_edicion,
state=tk.DISABLED)
        self.btn_editar_columnas.place(x = 100, y = 365)

        #---------------------para el nombre del conjunto de
dataframes


        self.label_nombre_grupo = tk.Label(self.cargaroot,
text="Seleccione nombre para el grupo:")

        self.label_nombre_grupo.place(x = 100, y = 405)

        self.entry_nombre_grupo =
tk.Entry(self.cargaroot,validate="key",
validatecommand=(self.validate_letras, "%P"))

        self.entry_nombre_grupo.place(x = 100, y = 435)


        # Añadir eventos que se ejecuten cada vez que el Listbox
cambie de selección o el Entry cambie
        self.listbox_columnas.bind('<<ListboxSelect>>', lambda event:
self.actualizar_estado_boton_cargar())
        self.entry_nombre_grupo.bind('<KeyRelease>', lambda event:
self.actualizar_estado_boton_cargar())  # Detecta cuando el usuario
escribe



        #================================

        # Entrada para la clave principal.
```

```
        self.frame_main_key = ttk.Frame(self.cargaroot, width=300,
height=50)  # Crea un marco para la clave principal.
        # Etiqueta para la clave principal.
        self.labelframe = ttk.Label(self.frame_main_key, text="Clave
principal:")  # Etiqueta para la clave principal.

        self.entry_main_key = ttk.Entry(self.frame_main_key)  # Crea
un campo de texto para ingresar la clave principal.
        self.entry_main_key.bind('<KeyRelease>', lambda event:
self.actualizar_estado_boton_cargar())

        # Botón para cargar archivo .spydata.
        self.btncarga = ttk.Button(self.cargaroot, text="Cargar
archivo .spydata", command=self.load_spydata)
        # Combobox para seleccionar diccionario.
        self.labeldic = ttk.Label(self.cargaroot, text="Selecciona
diccionario:")
        self.combobox_dict = ttk.Combobox(self.cargaroot, width=30)  #
Crea un combobox para seleccionar el diccionario.

        self.btnselec = ttk.Button(self.cargaroot, text="Seleccionar y
agregar DataFrames", command=self.open_selection_window_v2)



        #================================

        #Ahora creamos el botón que nos hará finalmente cargar la
información
        #Cambiamos el la fuente a negrita con (font=
fnt.Font(weight="bold"))
        self.cbtn = tk.Button(self.cargaroot,text="Cargar datos",font=
fnt.Font(weight="bold",size =
11),command=lambda:[self.cargadataframes()],bg="green",state=tk.DISABL
ED)

        self.cbtn.place(x=260,y=460)

        self.cbtn1 = tk.Button(self.cargaroot, text="Continuar",font=
fnt.Font(weight="bold",size = 11),command=lambda:[
self.cargaroot.destroy()],bg="green",state=tk.DISABLED)



        "Con el .bind podemos asignar una acción a un widget cuando se
produce un evento sobre ellos"
        "En este caso el evento será presionarlo, es como usar
command, aunque hay muchos más eventos"


        # running the main loop
        #self.cargaroot.mainloop()



        root.wait_window(self.cargaroot)#para detener la ejecución
aquí hasta que la ventana se destruya
        return self.ram_dataframe
```

```
def actualizar_visibilidad_principal(self, event):
    # Dependiendo de la opción seleccionada en el combobox
    if self.combobox_opciones.get() == "Cargar csv":

        self.frame_main_key.place_forget()
        self.labelframe.place_forget()
        self.entry_main_key.place_forget()
        self.btncarga.place_forget()
        self.labeldic.place_forget()
        self.combobox_dict.place_forget()
        self.btnselec.place_forget()
        self.cbtn1.place_forget()

        self.btnd.place( x=100, y=85)

        self.label_columnas.place(x = 100, y = 125)

        self.listbox_columnas.place(x = 100, y = 145)

        self.btn_editar_columnas.place(x = 100, y = 355)

        self.label_nombre_grupo.place(x = 100, y = 405)

        self.entry_nombre_grupo.place(x = 100, y = 435)

        self.cbtn.place(x=255,y=460)

    else:

        self.btnd.place_forget()

        self.label_columnas.place_forget()

        self.listbox_columnas.place_forget()

        self.btn_editar_columnas.place_forget()

        self.label_nombre_grupo.place_forget()

        self.entry_nombre_grupo.place_forget()

        self.cbtn.place_forget()


        self.frame_main_key.place(x=50, y=95)
        self.labelframe.pack(side = 'left')
        self.entry_main_key.pack(side = 'left')
        self.btncarga.place(x=50, y=135)
        self.labeldic.place(x=50, y=175)
        self.combobox_dict.place(x=50, y=205)
        self.btnselec.place(x=50, y=255)
        self.cbtn1.place(x=287,y=460)
```

```
    #====================================FUNCIONES PARA LA OPCION 1
    #=================================================================

    def seleccionar_archivos(self):
        """Abre un cuadro de diálogo para seleccionar archivos y
guarda sus rutas y nombres."""
        archivos = filedialog.askopenfilenames(title="Seleccione uno o
más archivos",
                                               filetypes=(("Archivos
CSV", "*.csv"), ("Todos los archivos", "*.*")))
        if archivos:
            global rutas_absolutas, nombres_archivos, nombre_base
            self.rutas_absolutas = list(archivos)  # Lista con las
rutas absolutas
            self.nombres_archivos = [os.path.basename(f) for f in
archivos]  # Lista con los nombres de los archivos

            print("\nRutas absolutas:", self.rutas_absolutas)
            print("\n\nNombres de archivos:", self.nombres_archivos)

            # Comprobar si los archivos tienen el mismo nombre base
excepto por un número
            patron = re.compile(r"^(.*?)[ _]?\d+\.csv$")
            coincidencias = [patron.match(nombre) for nombre in
self.nombres_archivos]
            bases = {m.group(1) for m in coincidencias if m}
            archivos_validos = [nombre for nombre in
self.nombres_archivos if patron.match(nombre)]

            if len(bases) == 1 and len(archivos_validos) ==
len(self.nombres_archivos):
                nombre_base = bases.pop()
                print("Nombre base detectado:", nombre_base)
                self.coincidencia = 1
            else:
                nombre_base = None
                print("No se encontró un nombre base único o algunos
archivos no siguen el patrón.")
                self.coincidencia = 0

            self.analizar_csv(self.rutas_absolutas[0])


    """
    Detecta automáticamente los nombres de las columnas de un CSV sin
cargar todo el fichero en
    memoria, usando una ventana de filas para encontrar el bloque
estable de datos.

    El método:
    1. Lee el CSV línea a línea con `csv.reader`, respetando comillas
y comas internas.
    2. Mantiene una ventana deslizante de tamaño `window_size`.
    3. Cuando detecta que todas las `window_size` filas tienen el
mismo número de columnas,
       considera la primera de esas filas como línea de encabezado
candidata.
    4. Solo si **todas** las celdas en esa línea candidata son **no
numéricas**, se asume como header.
    5. Devuelve la lista de nombres de columna. Si no hay header
textual, devuelve nombres genéricos
```

```
       ["Columna1", ..., "ColumnaN"], donde N es el número de columnas
detectado.

    Parámetros
    ----------
    path : str
        Ruta al archivo CSV a procesar.
    window_size : int, optional
        Número de filas consecutivas que deben tener el mismo número
de columnas
        para validar un posible header (por defecto 4).


    """
    def analizar_csv(self,path, window_size=4):

        self.columnas_disponibles = []

        header = None
        num_cols = None

        # Primera pasada: detectar bloque estable y posible header
        with open(path, newline='', encoding='utf-8') as f:
            reader = csv.reader(f)
            ventana = []
            for idx, row in enumerate(reader):
                if not row:
                    continue
                ventana.append((idx, row))
                if len(ventana) > window_size:
                    ventana.pop(0)
                if len(ventana) == window_size:
                    longitudes = [len(r) for _, r in ventana]
                    if len(set(longitudes)) == 1:
                        # detectado bloque estable
                        num_cols = longitudes[0]
                        _, header_row = ventana[0]
                        if all(not es_numero(val) for val in
header_row):
                            header = header_row
                        break

        # Si no se detectó encabezado o no hay bloque estable,
generamos genéricos
        if header is None:
            if num_cols is None:
                # Determinar número de columnas leyendo primera fila
útil
                with open(path, newline='', encoding='utf-8') as f2:
                    reader2 = csv.reader(f2)
                    for row in reader2:
                        if row:
                            num_cols = len(row)
                            break
            # Generar nombres genéricos
            header = [f"Columna{i+1}" for i in range(num_cols)]

        self.columnas_disponibles = header
        self.actualizar_selector_columnas()

    def actualizar_selector_columnas(self):
```

```
        """Actualiza el Listbox con las columnas detectadas y habilita
los botones."""
        self.listbox_columnas.delete(0, tk.END)
        for col in self.columnas_disponibles:
            self.listbox_columnas.insert(tk.END, col)


        self.btn_editar_columnas.config(state=tk.NORMAL)



    def obtener_columnas_seleccionadas(self):
        """Obtiene la lista de columnas seleccionadas en el
Listbox."""
        return [self.listbox_columnas.get(i) for i in
self.listbox_columnas.curselection()]

    def abrir_ventana_edicion(self):
        """Abre una nueva ventana con un Entry para cada columna
seleccionada."""
        seleccionadas = self.obtener_columnas_seleccionadas()
        if not seleccionadas:
            return

        ventana_edicion = tk.Toplevel(self.cargaroot)
        ventana_edicion.title("Editar Columnas")

        ventana_edicion.grab_set()          # Dirige todos los eventos
a la subventana
        ventana_edicion.transient(self.cargaroot)    # La subventana
se asocia con la ventana principal


        lista_entradas = []  # Lista para guardar los valores de los
Entry

        # Crear un Entry para cada columna seleccionada
        for i, col in enumerate(seleccionadas):
            label = tk.Label(ventana_edicion, text=col)
            label.grid(row=i, column=0, padx=10, pady=5)
            entry = tk.Entry(ventana_edicion, validate="key",
validatecommand=(self.validate_letras, "%P"))
            entry.grid(row=i, column=1, padx=10, pady=5)
            lista_entradas.append(entry)  # Guardamos la referencia
del Entry en la lista

        def guardar_valores():
            """Guarda los valores de los Entry en una lista."""
            self.nombrescolumnas = [entry.get() for entry in
lista_entradas]  # Obtener el valor de cada Entry
            print("´Nombres de columnas guardados:",
self.nombrescolumnas)  # Mostrar los valores en consola
            ventana_edicion.destroy()
            # Aquí puedes hacer algo con los valores, como
almacenarlos en una variable global o hacer otro procesamiento.

        # Botón para guardar los valores
        btn_guardar = tk.Button(ventana_edicion, text="Guardar
Valores", command=guardar_valores)
        btn_guardar.grid(row=len(seleccionadas), column=0,
columnspan=2, pady=10)
```

```
    def asignar_nombres_col(self):
        self.nombrescolumnas


    def actualizar_estado_boton_cargar(self):
        seleccionados = self.listbox_columnas.curselection()  # 
Obtener si hay algo seleccionado
        texto_nombre_grupo_opcion1 = self.entry_nombre_grupo.get()  # 
Obtener el texto del Entry

        opcion = self.combobox_opciones.get()

        texto_nombre_grupo_opcion2 = self.entry_main_key.get()


        if (seleccionados and texto_nombre_grupo_opcion1.strip() and 
opcion == "Cargar csv") :  # Si hay selección y el Entry no está vacío 
en la opción 1
            self.cbtn.config(state=tk.NORMAL)
        else:  # Si no hay selección o el Entry está vacío
            self.cbtn.config(state=tk.DISABLED)

        if (texto_nombre_grupo_opcion2.strip() and self.dataframes and 
opcion == "Cargar dataframe") :  # comrpobar si se ha puesto nombre al 
grupo y si existen dataframes
            self.cbtn1.config(state=tk.NORMAL)
        else:  # Si no hay selección o el Entry está vacío
            self.cbtn1.config(state=tk.DISABLED)


    #==============================Funciones para el idnicador


    def insertar_items(self, tree, parent, dic, dataframes_items):
        """
        Inserta elementos en el Treeview de forma recursiva.
        - Si el valor es un diccionario, se crea un nodo padre y se 
llama recursivamente.
        - Si el valor es un DataFrame, se agrega un nodo con el texto 
"ver dataframe" en azul.
        """
        for key, value in dic.items():
            if isinstance(value, dict):  # Si el valor es un 
diccionario anidado
                nodo = tree.insert(parent, "end", text=key, 
values=("",))  # Insertamos un nodo
                self.insertar_items(tree, nodo, value, 
dataframes_items)  # Llamamos recursivamente
            elif isinstance(value, pd.DataFrame):  # Si el valor es un 
DataFrame
                nodo = tree.insert(parent, "end", text=key, 
values=("ver dataframe",), tags=("dataframe",))
                dataframes_items[nodo] = value  # Guardamos el nodo en 
el diccionario de DataFrames
            else:  # Si el valor no es un diccionario ni un DataFrame
                tree.insert(parent, "end", text=key, 
values=(str(value),))  # Insertamos el valor como string

    def mostrar_dataframe(self, root, df, title="DataFrame"):
```

```
        """
        Abre una ventana para mostrar un DataFrame con scroll.
        Si el DataFrame tiene más de 100 filas, muestra las primeras
50 y las últimas 50.
        """
        ventana_df = tk.Toplevel(root)  # Creamos una nueva ventana
secundaria
        ventana_df.title(f"Contenido de {title}")  # Asignamos un
título con el nombre de la clave

        # Creamos un frame para contener el Treeview y su scrollbar
        frame = tk.Frame(ventana_df)
        frame.pack(expand=True, fill="both")  # Expandimos el frame
dentro de la ventana

        # Creamos el Treeview para mostrar el DataFrame
        tree_df = ttk.Treeview(frame, show="headings")  # "headings"
oculta la columna de claves
        tree_df.pack(side="left", expand=True, fill="both")  #
Expandimos en la ventana

        # Creamos la barra de desplazamiento vertical
        scrollbar = ttk.Scrollbar(frame, orient="vertical",
command=tree_df.yview)
        scrollbar.pack(side="right", fill="y")  # La colocamos en el
lado derecho
        tree_df.configure(yscrollcommand=scrollbar.set)  # Asociamos
la barra con el Treeview

        # Creamos la barra de desplazamiento horizontal
        h_scrollbar = ttk.Scrollbar(ventana_df, orient="horizontal",
command=tree_df.xview)
        h_scrollbar.pack(side="bottom", fill="x")  # La colocamos en
la parte inferior
        tree_df.configure(xscrollcommand=h_scrollbar.set)  # Asociamos
la barra con el Treeview

        # Si el DataFrame tiene más de 100 filas, mostramos solo las
primeras 50 y las últimas 50
        if len(df) > 100:
            df = pd.concat([df.head(50), df.tail(50)])  # Unimos las
primeras 50 y las últimas 50 filas

        # Agregamos una columna con el número de fila original
        df.insert(0, "N°", df.index)



        # Configuramos las columnas del Treeview según las columnas
del DataFrame
        columnas = list(df.columns)  # Obtenemos los nombres de las
columnas del DataFrame
        tree_df["columns"] = columnas  # Asignamos las columnas al
Treeview
        for col in columnas:
            tree_df.heading(col, text=col)  # Asignamos encabezados a
las columnas
            tree_df.column(col, anchor="center")  # Centramos el texto
en cada columna

        # Insertamos cada fila del DataFrame en el Treeview
```

```
        for _, row in df.iterrows():
            tree_df.insert("", "end", values=list(row))  # Añadimos
cada fila como una nueva entrada



    def on_item_double_click(self, event, root, tree,
dataframes_items):
        """
        Evento que se ejecuta al hacer doble clic en un elemento del
Treeview.
        - Si el nodo tiene un DataFrame asociado, se abre en una nueva
ventana.
        """
        item = tree.focus()  # Obtenemos el elemento seleccionado en
el Treeview
        if item in dataframes_items:  # Si el nodo está en el
diccionario de DataFrames
            df = dataframes_items[item]  # Recuperamos el DataFrame
            self.mostrar_dataframe(root, df, title=tree.item(item,
"text"))  # Mostramos la ventana del DataFrame

    """
    Crea una ventana con un Treeview para visualizar el contenido del
diccionario.
    """

    '''
    ventana_dic = tk.Toplevel(root)  # Creamos una nueva ventana
secundaria
    ventana_dic.title("Contenido del diccionario")  # Asignamos un
título a la ventana

    btn.config(state=tk.DISABLED)  # Deshabilitamos el botón principal
mientras la ventana está abierta

    # Diccionario para almacenar la relación entre nodos y DataFrames
    dataframes_items = {}

    # Creamos el Treeview con dos columnas (clave y valor)
    tree = ttk.Treeview(ventana_dic, columns=("Valor",), show="tree
headings")
    tree.heading("#0", text="Clave")  # Columna de claves
    tree.heading("Valor", text="Valor")  # Columna de valores

    # Configuramos el tag 'dataframe' para que aparezca en azul
    tree.tag_configure("dataframe", foreground="blue")

    # Insertamos los elementos del diccionario en el Treeview
    self.insertar_items(tree, "", self.dataframes, dataframes_items)

    tree.pack(expand=True, fill="both")  # Expandimos el Treeview en
la ventana

    # Asociamos el evento de doble clic con la función
on_item_double_click
    tree.bind("<Double-1>", lambda event:
self.on_item_double_click(event, root, tree, dataframes_items))

    btn_eliminar = tk.Button(ventana_dic, text="Eliminar selección",
command=lambda: self.eliminar_item(tree), state=tk.DISABLED)
```

```
    btn_eliminar.pack(pady=5)

    def on_tree_select(event):
        seleccion = tree.selection()
        if seleccion:
            item = seleccion[0]
            parent = tree.parent(item)
            # Solo activar si es un nodo raíz (clave principal)
            if parent == "":
                btn_eliminar.config(state=tk.NORMAL)
            else:
                btn_eliminar.config(state=tk.DISABLED)
        else:
            btn_eliminar.config(state=tk.DISABLED)

    tree.bind("<<TreeviewSelect>>", on_tree_select)



    # Al cerrar la ventana, se reactiva el botón principal
    ventana_dic.protocol("WM_DELETE_WINDOW", lambda:
(btn.config(state=tk.NORMAL), ventana_dic.destroy()))
    '''

    """
    Crea una ventana con un Treeview para visualizar y eliminar claves
o subclaves del diccionario.

    self.dataframes debe tener esta estructura:
    {
        'grupo1': {
            'subclave1': dataframe1,
            'subclave2': dataframe2,
            ...
        },
        ...
    }

    Parámetros:
    - root: ventana principal (Tk o Toplevel).
    - btn: botón que lanzó esta función (para deshabilitarlo mientras
la ventana está activa).
    """

    def mostrar_diccionario(self, root, btn):


        # Crear una nueva ventana secundaria
        ventana_dic = tk.Toplevel(root)
        ventana_dic.title("Contenido del diccionario")

        # Desactivar el botón que abre esta ventana mientras esté
abierta
        btn.config(state=tk.DISABLED)
        '''
        # Crear el Treeview (vista jerárquica) con dos columnas
        tree = ttk.Treeview(ventana_dic, columns=("Valor",),
show="tree headings")
        tree.heading("#0", text="Clave")        # Columna de la
izquierda (clave)
```

```
        tree.heading("Valor", text="Valor")     # Columna adicional
(puedes personalizarla si lo deseas)
        tree.tag_configure("dataframe", foreground="blue")  # Estilo
para los nodos
        '''
        # Diccionario para almacenar la relación entre nodos y
DataFrames
        dataframes_items = {}

        # Creamos el Treeview con dos columnas (clave y valor)
        tree = ttk.Treeview(ventana_dic, columns=("Valor",),
show="tree headings")
        tree.heading("#0", text="Clave")  # Columna de claves
        tree.heading("Valor", text="Valor")  # Columna de valores

        # Configuramos el tag 'dataframe' para que aparezca en azul
        tree.tag_configure("dataframe", foreground="blue")

        # Insertamos los elementos del diccionario en el Treeview
        self.insertar_items(tree, "", self.dataframes,
dataframes_items)

        tree.pack(expand=True, fill="both")  # Expandimos el Treeview
en la ventana

        # Asociamos el evento de doble clic con la función
on_item_double_click
        tree.bind("<Double-1>", lambda event:
self.on_item_double_click(event, root, tree, dataframes_items))


        '''# Insertar claves y subclaves en el Treeview
        for grupo, subclaves in self.dataframes.items():
            id_grupo = tree.insert("", "end", text=grupo,
values=("",))
            for subclave in subclaves:
                tree.insert(id_grupo, "end", text=subclave,
values=("DataFrame",), tags=("dataframe",))
                '''
        # Empaquetar el Treeview para que ocupe el espacio de la
ventana
        tree.pack(expand=True, fill="both", padx=10, pady=5)

        # Crear botón para eliminar claves principales (grupos),
inicialmente desactivado
        btn_eliminar_grupo = tk.Button(ventana_dic, text="Eliminar
grupo", state=tk.DISABLED)
        btn_eliminar_grupo.pack(pady=5)

        # Crear botón para eliminar subclaves (elementos
individuales), también desactivado
        btn_eliminar_elemento = tk.Button(ventana_dic, text="Eliminar
subclave", state=tk.DISABLED)
        btn_eliminar_elemento.pack(pady=5)

        # === FUNCIÓN: Detectar selección y activar botones
correspondientes ===
        def on_tree_select(event):
            seleccion = tree.selection()
            if seleccion:
                item = seleccion[0]
```

```
                parent = tree.parent(item)
                if parent == "":
                    # Si no tiene padre, es una clave principal
                    btn_eliminar_grupo.config(state=tk.NORMAL)
                    btn_eliminar_elemento.config(state=tk.DISABLED)
                else:
                    # Si tiene padre, es una subclave
                    btn_eliminar_grupo.config(state=tk.DISABLED)
                    btn_eliminar_elemento.config(state=tk.NORMAL)
            else:
                # Si no hay selección, desactivar ambos
                btn_eliminar_grupo.config(state=tk.DISABLED)
                btn_eliminar_elemento.config(state=tk.DISABLED)

        tree.bind("<<TreeviewSelect>>", on_tree_select)

        # === FUNCIÓN: Eliminar grupo completo ===
        def eliminar_grupo():
            seleccion = tree.selection()
            if seleccion:
                item = seleccion[0]
                grupo = tree.item(item, "text")  # Obtener nombre del
grupo (clave)
                if grupo in self.dataframes:
                    del self.dataframes[grupo]   # Eliminar del
diccionario
                tree.delete(item)                # Eliminar del
Treeview
            btn_eliminar_grupo.config(state=tk.DISABLED)  # Desactivar
botón
            gc.collect()

        btn_eliminar_grupo.config(command=eliminar_grupo)

        # === FUNCIÓN: Eliminar subclave individual ===
        def eliminar_elemento():
            seleccion = tree.selection()
            if seleccion:
                item = seleccion[0]
                subclave = tree.item(item, "text")           # Nombre
de la subclave
                parent_item = tree.parent(item)              # ID del
grupo (padre)
                grupo = tree.item(parent_item, "text")       # Nombre
del grupo
                if grupo in self.dataframes and subclave in
self.dataframes[grupo]:
                    del self.dataframes[grupo][subclave]     #
Eliminar subclave
                    tree.delete(item)                        #
Eliminar del Treeview

                    # Si el grupo queda vacío después de borrar,
eliminar también el grupo
                    if not self.dataframes[grupo]:
                        del self.dataframes[grupo]
                        tree.delete(parent_item)
            btn_eliminar_elemento.config(state=tk.DISABLED)
            gc.collect()

        btn_eliminar_elemento.config(command=eliminar_elemento)
```

```
        # Al cerrar la ventana, se reactiva el botón principal y se
destruye la ventana
        ventana_dic.protocol("WM_DELETE_WINDOW", lambda:
(btn.config(state=tk.NORMAL), ventana_dic.destroy()))


    #============================================================

    #==============================Funciones para el seleccionador

    # Función para actualizar el Listbox de subopciones según la
opción seleccionada
    def update_suboptions(self, main_option_var, main_option_menu,
sublistbox, data, resultados, colors):
        main_key = main_option_var.get()  # Obtiene la opción
principal seleccionada
        # Actualiza el color de fondo del OptionMenu según la opción
seleccionada
        main_option_menu.config(bg=colors.get(main_key, "white"))
        sublistbox.delete(0, tk.END)  # Limpia el Listbox
        subkeys = list(data[main_key].keys())  # Obtiene la lista de
subopciones para la opción seleccionada
        for sub_key in subkeys:
            sublistbox.insert(tk.END, sub_key)  # Inserta cada
subopción en el Listbox
        # Si existen resultados guardados para esta opción,
preselecciona las subopciones correspondientes
        if main_key in resultados:

            #necesitamos que la lista del diccionario esté distribuida
de una manera ligeramente diferente
            resultados_adaptado = resultados

            resultados_adaptado[ list(resultados.keys())[0] ] = [ ",
".join(resultados[ list(resultados.keys())[0] ]) ]

            preselected_str = resultados_adaptado[main_key][0]  #
Obtiene el string de subopciones guardado
            preselected_items = [s.strip() for s in
preselected_str.split(",") if s.strip()]  # Separa el string en una
lista
            for idx, sub_key in enumerate(subkeys):
                if sub_key in preselected_items:
                    sublistbox.select_set(idx)  # Preselecciona el
ítem en el Listbox

    # Función para imprimir y guardar la selección de subopciones en
el diccionario 'resultados'
    def print_selection(self, main_option_var, sublistbox,
resultados):

        main_key = main_option_var.get()  # Obtiene la opción
principal seleccionada
        selected_indices = sublistbox.curselection()  # Obtiene los
índices de las subopciones seleccionadas
        selected_items = [sublistbox.get(i) for i in selected_indices]
# Obtiene el texto de cada subopción seleccionada
        resultados.clear()  # Limpia cualquier resultado previo (solo
se guarda una opción a la vez)
```

```
        resultados[main_key] = selected_items  # Guarda la selección
en 'resultados'
        print("Resultados guardados:", resultados)  # Imprime los
resultados guardados en la consola


    # Función para deseleccionar, que borra el contenido de
'resultados' y reinicia la ventana de selección
    def deselect_all(self, main_option_var, main_option_menu,
sublistbox, main_keys, data, colors, resultados):
        resultados.clear()  # Borra el diccionario de resultados
        print("Resultados borrados:", resultados)  # Imprime el
diccionario vacío
        main_option_var.set(main_keys[0])  # Resetea la opción
principal a la primera opción
        main_option_menu.config(bg=colors.get(main_keys[0], "white"))
# Actualiza el color de fondo del OptionMenu
        sublistbox.delete(0, tk.END)  # Limpia el Listbox
        for sub_key in data[main_keys[0]]:
            sublistbox.insert(tk.END, sub_key)  # Inserta las
subopciones de la primera opción en el Listbox
        sublistbox.selection_clear(0, tk.END)  # Limpia cualquier
selección en el Listbox

    # Función para abrir la ventana de selección
    def open_selection_window(self, btn, root):

        # Define una lista de colores para asignar de forma alterna a
las opciones
        color_list = ["lightblue", "lightgreen"]

        # Crea un diccionario 'colors' que asocia cada opción a un
color alternante
        colors = {}
        for index, key in enumerate(self.dataframes.keys()):
            colors[key] = color_list[index % len(color_list)]  #
Alterna colores usando el módulo

        btn.config(state="disabled")  # Desactiva el botón de la
ventana principal para evitar abrir múltiples ventanas
        selection_window = tk.Toplevel(root)  # Crea una nueva ventana
(subventana)
        selection_window.title("Seleccionar Diccionarios")  #
Establece el título de la subventana

        # Función interna que se ejecuta al cerrar la ventana; se
mantiene anidada para acceder a 'open_button' y 'selection_window'
        def on_close():
            btn.config(state="normal")  # Reactiva el botón de la
ventana principal
            selection_window.destroy()  # Cierra la subventana

        selection_window.protocol("WM_DELETE_WINDOW", on_close)  #
Asocia el evento de cierre de la ventana a 'on_close'

        # Crea una variable StringVar para almacenar la opción
principal seleccionada
        main_option_var = tk.StringVar(selection_window)
        main_keys = list(self.dataframes.keys())  # Obtiene la lista
de todas las opciones
```

```
        # Si existen resultados guardados, preselecciona la última
opción almacenada; de lo contrario, usa la primera opción
        if self.dataframes_selec: #si está vacío el diccionario cuenta
como FALSE
            last_option = list(self.dataframes_selec.keys())[-1]
            if last_option in main_keys:
                main_option_var.set(last_option)
            else:
                main_option_var.set(main_keys[0])
        else:
            main_option_var.set(main_keys[0])

        # Crea un OptionMenu para seleccionar la opción principal y
actualiza el Listbox al cambiar la opción
        main_option_menu = tk.OptionMenu(
            selection_window, main_option_var, *main_keys,
            command=lambda *args:
self.update_suboptions(main_option_var, main_option_menu, sublistbox,
self.dataframes, self.dataframes_selec, colors)
        )
        # Configura el ancho y el color de fondo del OptionMenu según
la opción seleccionada
        main_option_menu.config(width=20,
bg=colors.get(main_option_var.get(), "white"))
        main_option_menu.pack(padx=10, pady=10)  # Coloca el
OptionMenu en la ventana con márgenes

        # Configura el fondo de cada entrada del menú desplegable del
OptionMenu (se alternan los colores)
        menu = main_option_menu["menu"]
        for index, key in enumerate(main_keys):
            menu.entryconfigure(index, background=colors.get(key,
"white"))

        # Crea un frame para alojar el Listbox y la barra de
desplazamiento (scrollbar)
        listbox_frame = tk.Frame(selection_window)
        listbox_frame.pack(padx=10, pady=10, fill="both", expand=True)

        # Crea un Listbox para mostrar las subopciones y permite
selección extendida (varios ítems a la vez)
        sublistbox = tk.Listbox(listbox_frame, selectmode="extended",
width=30)
        sublistbox.pack(side="left", fill="both", expand=True)

        # Crea una barra de desplazamiento vertical para el Listbox
        scrollbar = tk.Scrollbar(listbox_frame, orient="vertical",
command=sublistbox.yview)
        scrollbar.pack(side="right", fill="y")
        sublistbox.config(yscrollcommand=scrollbar.set)  # Vincula el
Listbox con la scrollbar

        # Rellena el Listbox con las subopciones correspondientes a la
opción principal seleccionada
        self.update_suboptions(main_option_var, main_option_menu,
sublistbox, self.dataframes, self.dataframes_selec, colors)

        # Crea un botón que al pulsarlo imprime y guarda la selección
de subopciones
        print_button = tk.Button(
            selection_window, text="Seleccionar",
```

```
            command=lambda: self.print_selection(main_option_var,
sublistbox, self.dataframes_selec)
        )
        print_button.pack(padx=10, pady=5)  # Coloca el botón con
márgenes

        # Crea un botón "Deseleccionar" que borra los resultados y
resetea la ventana de selección
        deselect_button = tk.Button(
            selection_window, text="Descartar",
            command=lambda: self.deselect_all(main_option_var,
main_option_menu, sublistbox, main_keys, self.dataframes, colors,
self.dataframes_selec)
        )
        deselect_button.pack(padx=10, pady=5)

        # Crea un botón para cerrar la ventana de selección
        close_button = tk.Button(selection_window, text="Cerrar",
command=on_close)
        close_button.pack(padx=10, pady=5)

        root.wait_window(selection_window)



    #============================================================
    #====================================================================

    #====================================FUNCIONES PARA LA OPCION 2
    #====================================================================

    def load_spydata(self):
        """Carga un archivo .spydata y permite seleccionar un
diccionario para explorar."""
        filename = filedialog.askopenfilename(
            title="Selecciona archivo .spydata",  # Título del cuadro
de diálogo.
            filetypes=[("Spydata files", "*.spydata")]  # Solo muestra
archivos con extensión .spydata.
        )
        if filename:  # Si se seleccionó un archivo
            try:
                workspace = load_dictionary(filename)  # Intenta
cargar el workspace desde el archivo .spydata seleccionado.
                if isinstance(workspace, tuple):
                    workspace = workspace[0]  # Si se devuelve una
tupla, extrae el primer elemento.
                dict_keys = [key for key, value in workspace.items()
if isinstance(value, dict)]  # Filtra diccionarios.
                if dict_keys:  # Si se encontraron diccionarios
                    self.combobox_dict['values'] = dict_keys  # Asigna
los diccionarios encontrados al combobox.
                    self.combobox_dict.current(0)  # Selecciona el
primer diccionario por defecto.
                    global loaded_workspace  # Usa la variable global
loaded_workspace.
                    loaded_workspace = workspace  # Asigna el
workspace cargado a la variable global.
                    messagebox.showinfo("Éxito", f"Archivo
cargado:\n{filename}")  # Muestra un mensaje de éxito.
                else:
```

```
                    messagebox.showwarning("Advertencia", "No se
encontraron diccionarios en el archivo.")  # Advertencia.
            except Exception as e:
                messagebox.showerror("Error", f"Error al cargar el
archivo:\n{e}")  # Muestra un mensaje de error.

    def open_selection_window_v2(self ):
        """Abre una ventana para que el usuario seleccione los
DataFrames deseados."""
        selected_dict_name = self.combobox_dict.get()  # Obtiene el
nombre del diccionario seleccionado.
        if not selected_dict_name:  # Si no se ha seleccionado un
diccionario
            messagebox.showerror("Error", "No se ha seleccionado un
diccionario.")  # Muestra un mensaje de error.
            return

        main_key = self.entry_main_key.get()  # Obtiene la clave
principal ingresada en el campo de texto.
        if not main_key:  # Si no se ha ingresado una clave principal
            messagebox.showerror("Error", "Por favor, ingresa una
clave principal.")  # Muestra un mensaje de error.
            return

        if main_key not in self.dataframes:  # Si la clave principal
no existe en el diccionario global
            self.dataframes[main_key] = {}  # Crea una nueva entrada
para la clave principal.

        selected_dict = loaded_workspace.get(selected_dict_name, {})
# Obtiene el diccionario seleccionado.

        # Crea una nueva ventana para la selección de DataFrames.
        selection_window = tk.Toplevel(self.cargaroot)  # Crea una
ventana secundaria.
        selection_window.title("Seleccionar DataFrames")  # Título de
la ventana.

        # Crea un Treeview para mostrar la estructura del diccionario
(como árbol).
        tree = ttk.Treeview(selection_window, columns=("type",),
selectmode="browse")
        tree.heading("#0", text="Elemento")  # Cabecera para los
nombres de los elementos.
        tree.heading("type", text="Tipo")  # Cabecera para mostrar los
tipos de elementos.
        tree.pack(fill=tk.BOTH, expand=True)  # Muestra el Treeview en
la ventana, expandiéndose en ambas direcciones.

        self.insert_items("", selected_dict, tree)  # Llamamos a la
nueva función insert_items.

        def on_item_selected(event):
            selected_item = tree.selection()  # Obtiene el item
seleccionado.
            if selected_item:  # Si se ha seleccionado un item
                item_id = selected_item[0]  # Obtiene el id del item
seleccionado.
                item_type = tree.item(item_id, "values")[0]  # Obtiene
el tipo de item (DataFrame o dict).
                if item_type == "DataFrame":  # Si es un DataFrame
```

```
                    path = []  # Inicializa la lista de partes de la
ruta.
                    while item_id:  # Recorre el árbol desde el item
seleccionado hasta la raíz.
                        path.insert(0, tree.item(item_id, "text"))  #
Inserta el nombre del item en la ruta.
                        item_id = tree.parent(item_id)  # Se mueve al
nodo padre.
                    full_path = '.'.join(path)  # Une las partes de la
ruta con puntos.
                    self.modify_path_window(full_path, selected_dict,
main_key)  # Abre la ventana para modificar la ruta.

        tree.bind("<<TreeviewSelect>>", on_item_selected)  # Vincula
el evento de selección al Treeview.

    def insert_items(self, parent, dictionary, tree):
        """Inserta los elementos del diccionario en el Treeview de
manera recursiva."""
        for key, value in dictionary.items():  # Itera sobre las
claves y valores del diccionario.
            if isinstance(value, dict):  # Si el valor es un
diccionario
                node_id = tree.insert(parent, "end", text=key,
values=("dict",))  # Inserta el nodo como "dict".
                self.insert_items(node_id, value, tree)  # Llama
recursivamente para insertar los elementos del subdiccionario.
            elif isinstance(value, pd.DataFrame):  # Si el valor es un
DataFrame
                tree.insert(parent, "end", text=key,
values=("DataFrame",))  # Inserta el nodo como "DataFrame".

    def modify_path_window(self, path, base_dict, main_key):
        """Abre una ventana que permite al usuario modificar la ruta
antes de agregar el DataFrame."""
        top = tk.Toplevel(self.cargaroot)  # Crea una ventana
secundaria.
        top.title("Modificar ruta del DataFrame")  # Título de la
ventana.

        ttk.Label(top, text="Modifica la ruta del
DataFrame:").pack(pady=5)  # Etiqueta para la modificación de la ruta.

        path_parts = path.split('.')  # Separa la ruta original en
partes.
        listbox = tk.Listbox(top, selectmode=tk.MULTIPLE)  # Crea un
Listbox con modo de selección múltiple.
        for part in path_parts:  # Inserta las partes de la ruta en el
Listbox.
            listbox.insert(tk.END, part)
        listbox.pack(pady=5)  # Muestra el Listbox.

        ttk.Button(top, text="Eliminar parte seleccionada",
command=lambda: self.remove_part(listbox)).pack(pady=5)  # Botón para
eliminar parte de la ruta.
        ttk.Button(top, text="Confirmar y agregar DataFrame",
command=lambda: self.confirm_selection(listbox, path, base_dict,
main_key, top)).pack(pady=5)  # Botón para confirmar la selección.

    def remove_part(self, listbox):
        """Elimina una parte seleccionada de la ruta en el Listbox."""
```

```
        selected_indices = listbox.curselection()  # Obtiene los
índices de los elementos seleccionados en el Listbox.
        if not selected_indices:  # Si no se ha seleccionado ninguna
parte
            messagebox.showwarning("Advertencia", "No se ha
seleccionado ninguna parte de la ruta.")  # Muestra una advertencia.
            return
        for index in reversed(selected_indices):  # Recorre los
índices seleccionados de atrás hacia adelante.
            listbox.delete(index)  # Elimina las partes seleccionadas
del Listbox.

    def confirm_selection(self, listbox, path, base_dict, main_key,
top):
        """Confirma la selección y agrega el DataFrame con la ruta
modificada si hay espacio en RAM"""

        self.opcion = "continuar"
        modified_path = '.'.join(listbox.get(0, tk.END))  # Obtiene la
ruta modificada desde el Listbox.
        if check_ram(self.ram_dataframe):
            print("RAM alta, mostrando ventana...")
            self.ventana_exceso_ram(modified_path,0)

            if self.opcion == "parar_mantener":
                top.destroy()  # Cierra la ventana.
                return
            elif self.opcion == "parar_borrar":

                #Borramos la entrada del diccionario
                self.dataframes.pop(main_key, None)
                gc.collect()
                top.destroy()  # Cierra la ventana.
                return



        df = self.get_value_from_path(base_dict, path.split('.'))  #
Obtiene el DataFrame usando la ruta original.
        self.add_dataframe_to_global_dict(main_key, modified_path, df)
# Agrega el DataFrame al diccionario global.
        messagebox.showinfo("Éxito", f"DataFrame agregado bajo la ruta
'{modified_path}' en '{main_key}'.")  # Muestra un mensaje de éxito.
        top.destroy()  # Cierra la ventana.



    def get_value_from_path(self, dictionary, path_parts):
        """Obtiene el valor (DataFrame) desde el diccionario dado una
lista de claves que representan la ruta."""
        for part in path_parts:  # Recorre las partes de la ruta.
            dictionary = dictionary[part]  # Accede al siguiente nivel
en el diccionario.
        return dictionary  # Devuelve el valor final (DataFrame).

    def add_dataframe_to_global_dict(self, main_key, path, df):
        """Agrega el DataFrame al diccionario global 'dataframe' en la
ruta especificada bajo la clave principal."""
        if main_key not in self.dataframes:  # Si la clave principal
no existe en el diccionario global.
```

```
            self.dataframes[main_key] = {}  # Crea una nueva entrada.
        self.dataframes[main_key][path] = df  # Agrega el DataFrame en
la ruta especificada.

        self.actualizar_estado_boton_cargar()#se comprueba si se han
añadido datos para permitir continuar


    #============================================================
    #============================================================

    def cargadataframes(self):

        #Debemos saber los índices de las columnas seleccionadas

        self.columnasselec = self.listbox_columnas.curselection()

        #en primer lugar debemos comprobar que el numero de columnas
seleccionadas es igual al nombre de las columnas introducidas  si se
ha dado el caso
        #adem,as también debemos tener en cuenta si no todos los
ficheros seleccioandos tienen la misma estructura
        continuar = [0]
        if self.coincidencia == 0 or len(self.columnasselec) !=
len(self.nombrescolumnas):
            ventana_confirmacion = tk.Toplevel(self.cargaroot)
            ventana_confirmacion.title("Confirmación")

            texto_error = "¿Seguro que quieres cargar los datos?"

            if self.coincidencia == 0:
                texto_error += " No todos los ficheros tienen el mismo
nombre"

            elif len(self.columnasselec) != len(self.nombrescolumnas):
                texto_error += " No se ha seleccionado la misma
cantidad de columnas que de nombres asignados"

            mensaje = tk.Label(ventana_confirmacion, text=texto_error)
            mensaje.pack(padx=10, pady=10)

            boton_cancelar =
tk.Button(ventana_confirmacion,text="Cancelar",font=
fnt.Font(weight="bold",size =
13),command=lambda:[ventana_confirmacion.destroy()],bg="red")

            boton_cancelar.pack()

            boton_aceptar =
tk.Button(ventana_confirmacion,text="aceptar",font=
fnt.Font(weight="bold",size =
13),command=lambda:[ventana_confirmacion.destroy(),
continuar.insert(0, 1)],bg="green")

            boton_aceptar.pack()

            self.cargaroot.wait_window(ventana_confirmacion)#para
detener la ejecución aquí hasta que la ventana se destruya
        else:
            continuar = [1]
```

```
        if continuar[0] == 0:
            return

        self.nombregrupo = self.entry_nombre_grupo.get()

        print(self.nombrescolumnas)

        if self.nombrescolumnas == []:
            for i in self.columnasselec:
                self.nombrescolumnas.append("Columna" + str(i + 1))

        print("Se va a proceder a cargar un grupo de dataframes con
nombre: " + self.nombregrupo + " Usando las columnas de posicion: " +
f"{' '.join(str(i) for i in self.columnasselec) }" + " con nombres: "
+ f"{' '.join(str(i) for i in self.nombrescolumnas) }" )



        #creamos un subdiccionario dentro del diccionario 'dataframes'

        self.dataframes[ self.nombregrupo ] = {}

        peso_muestra_bytes = 0
        relacion_estimacion = 0
        longitud_referencia = 0
        for indicedirec, direcciones in
enumerate(self.rutas_absolutas):


            #Vamos a proceder a detectar en nuestros archivos csv
cuándo empiezan las columnas numéricas, de forma que podemos empezar a
cargar a partir de ahí
            #además detectaremos el número total de filas de este, que
nos servirá en un futuro para aproximar el tamaño total del dataframe
            #además, hay que tener en cuenta que, en principio, todos
los dataframes cargados para un grupo, deben tener la misma estructura
y,
            #al menos, una cantidad parecida de filas

            #---Adaptamos al formato de la función las columnas que no
se quieren cargar
            skip_col = []
            for i in self.columnasselec:
                if i is not None:
                    skip_col.append(i)
            if skip_col == []:
                skip_col =None
            print(skip_col)

            try:

                fila_inicio_datos, total_filas =
detectar_inicio_y_conteo_total(direcciones, skip_col)

                if indicedirec == 0:
                    longitud_referencia = total_filas
            except:
                messagebox.showerror("Error","No se pudo leer
adecuadamente el archivo csv")
                return
```

```
            if fila_inicio_datos == None:
                messagebox.showerror("Error","No se detecta un patrón
adecuado de números en el archivo csv")
                return
            print("Los datos comienzan en la fila numero " +
str(fila_inicio_datos) + " de los ficheros")
            print("El fichero tiene un total de " + str(total_filas) +
" datos")

            #Ahora se realiza el proceso de estimar el peso en memoria
de una porción pequeña del csv, de forma
            #que se podrá extrapolar al peso final del csv completo,
se hará una sola vez, asumiendo que todoos los csv
            #cargados son similares

            if indicedirec == 0:
                # Cálculo del número de filas a cargar (mínimo 2)
                filas_muestra = max(2, math.ceil(total_filas * 0.05))
                # Cargar muestra del CSV
                df_muestra =
pd.read_csv(direcciones,skiprows=fila_inicio_datos,
nrows=filas_muestra, usecols=self.columnasselec,names =
self.nombrescolumnas,header=None)
                # Calcular peso en bytes de la muestra
                peso_muestra_bytes =
df_muestra.memory_usage(deep=True).sum()
                # Estimar el peso total
                peso_estimado_total_bytes = (peso_muestra_bytes /
filas_muestra) * total_filas
                relacion_estimacion = (peso_muestra_bytes /
filas_muestra)
                # Obtener RAM total del sistema
                ram_total_bytes = psutil.virtual_memory().total
                # Calcular porcentaje de RAM
                porcentaje_ram = (peso_estimado_total_bytes /
ram_total_bytes) * 100
                print("\nse estima : " + str(porcentaje_ram) + "% para
el primer dataframe, al aplicar la mayoración se estima: " +
str(porcentaje_ram*1.3))
                #Es importante indicar que esta estimación es
orientativa y es adecuado aplicar un factor de mayoración de alrededor
del 30%

            #Una vez que ya tenemos el peso muestra de los bytes,
imprimiremos por comandos siempre el peso estimado y si este supera
cierto umbral
            #se le pregunta al usuario si se quiere continuar, además,
si el número de filas es muy distinto a la del primer dataframe
cargado
            #se pregunta igualmente
            self.opcion = "continuar"
            ram_total_bytes = psutil.virtual_memory().total
            peso_estimado_total_bytes = relacion_estimacion *
total_filas
            estimacion_ram = (peso_estimado_total_bytes /
ram_total_bytes) * 100
            if psutil.virtual_memory().percent + estimacion_ram>=
self.ram_dataframe:
                self.ventana_exceso_ram(
self.nombres_archivos[indicedirec][:-4], estimacion_ram)
```

```
                print("RAM alta, mostrando ventana...")

                if self.opcion == "parar_mantener":
                    break
                elif self.opcion == "parar_borrar":

                    #Borramos la entrada del diccionario
                    self.dataframes.pop(self.nombregrupo, None)
                    gc.collect()
                    break

            if psutil.virtual_memory().percent + estimacion_ram>=
self.ram_dataframe:

self.ventana_diferencia_linea(self.nombres_archivos[0][:-4],
longitud_referencia, self.nombres_archivos[indicedirec][:-4],
total_filas)

                print("Diferencia en longitudes, mostrando
ventana...")

                if self.opcion == "parar_mantener":
                    break
                elif self.opcion == "parar_borrar":

                    #Borramos la entrada del diccionario
                    self.dataframes.pop(self.nombregrupo, None)
                    gc.collect()
                    break


            #antes = psutil.virtual_memory().percent
            print("Cargando " + self.nombres_archivos[indicedirec][:-
4])

self.dataframes[self.nombregrupo][self.nombres_archivos[indicedirec][:
-4]] = pd.read_csv(direcciones,skiprows=fila_inicio_datos,
usecols=self.columnasselec,names = self.nombrescolumnas,header=None)
            #despues = psutil.virtual_memory().percent
            #print("este dataframe ocupa " + str(despues-antes) + "%")



        print(self.dataframes)



        self.cargaroot.destroy()


    #================================================
    def ventana_exceso_ram(self, nombredf, estimacion_ram):

        """Muestra la ventana con 3 botones"""

        self.opcion = "continuar"
        def continuar():

            self.ram_dataframe += 10
            root.destroy()
```

```
        def parar_mantener():

            self.ram_dataframe += 10
            self.opcion = "parar_mantener"
            root.destroy()


        def parar_borrar():
            self.opcion = "parar_borrar"
            root.destroy()


        root = tk.Toplevel(self.cargaroot)
        root.title("Alerta de uso de RAM - Elige una acción")
        root.geometry("350x180")

        label = tk.Label(root, text="Se estima que el dataframe " +
nombredf + " ocuparía " + str( round(estimacion_ram, 3)) + "%,
\nhaciendo que el sistema ocupe más del " + str(self.ram_dataframe) +
"%.\n" + " Actualmente se está usando el " +
str(psutil.virtual_memory().percent)  +  "%. ¿Qué deseas hacer?",
font=("Arial", 12))
        label.pack(pady=10)

        btn_continuar = tk.Button(root, text="Continuar",
command=continuar, bg="lightgreen", width=25)
        btn_continuar.pack(pady=5)

        btn_parar_mantener = tk.Button(root, text="Parar y mantener
datos", command=parar_mantener, bg="gold", width=25)
        btn_parar_mantener.pack(pady=5)

        btn_parar_borrar = tk.Button(root, text="Parar y borrar
datos", command=parar_borrar, bg="tomato", width=25)
        btn_parar_borrar.pack(pady=5)

        self.cargaroot.wait_window(root)#para detener la ejecución
aquí hasta que la ventana se destruya

    def ventana_diferencia_linea(self,nombre_referencia,
longitud_referencia, nombredf, longitud_df):

        """Muestra la ventana con 3 botones"""

        self.opcion = "continuar"
        def continuar():

            self.ram_dataframe += 10
            root.destroy()


        def parar_mantener():

            self.ram_dataframe += 10
            self.opcion = "parar_mantener"
            root.destroy()


        def parar_borrar():
```

```
            self.opcion = "parar_borrar"
            root.destroy()


        root = tk.Toplevel(self.cargaroot)
        root.title("Alerta de diferencia de longitudes de dataframes -
Elige una acción")
        root.geometry("350x180")

        label = tk.Label(root, text="La longitud del fichero "  +
nombredf + " (" + str(longitud_df) +  ")" + " es diferente a la del
primer fichero cargado, + "  + nombre_referencia +  " (" +
str(longitud_referencia) +  ")" +  " ¿Qué deseas hacer?",
font=("Arial", 12))
        label.pack(pady=10)

        btn_continuar = tk.Button(root, text="Continuar",
command=continuar, bg="lightgreen", width=25)
        btn_continuar.pack(pady=5)

        btn_parar_mantener = tk.Button(root, text="Parar y mantener
datos", command=parar_mantener, bg="gold", width=25)
        btn_parar_mantener.pack(pady=5)

        btn_parar_borrar = tk.Button(root, text="Parar y borrar
datos", command=parar_borrar, bg="tomato", width=25)
        btn_parar_borrar.pack(pady=5)

        self.cargaroot.wait_window(root)#para detener la ejecución
aquí hasta que la ventana se destruya
```

Clase “Procesador analítico”

```
try:
    import Tkinter as tk
except:
    import tkinter as tk

import numpy as np
import matplotlib
#INDICAMOS QUE QUEREMOS QUE MATPLOTLIB USE DE BACKEND Agg
matplotlib.use("Agg")
import matplotlib.pyplot as py
import pandas as pd
import gc #El garrbage collector, para deshacernos de la memoria
indeseada activamente con los plots porque a veces spyder no lo hace

try:
    from tkinter import filedialog, messagebox
except:
    from Tkinter import filedialog, messagebox

from matplotlib.backends.backend_tkagg import (FigureCanvasTkAgg
,NavigationToolbar2Tk)
from matplotlib.figure import Figure #Usaremos Figure en lugar de
pyplot

import ast  # Importamos el módulo ast (Abstract Syntax Tree) para
analizar la sintaxis de la expresión.

from scipy.signal import savgol_filter#Filtro savgol para cierto ruido
indeseado a la hora de detectar cosas en los datoa

import copy

"""
Procesador Analítico

Objeto responsable del procesamiento y análisis avanzado de los datos
para su
representación gráfica. Aunque algunas funciones están destinadas al
análisis
específico de datos de sobrecarga (por ejemplo, análisis de múltiples
ficheros CSV),
muchas funcionalidades se integran modularmente para su uso general en
la interfaz.

Funcionalidades principales:
- Generación del eje X artificial a partir de parámetros de escala y
offset,
  emulando el comportamiento de adquisición de señales en instrumentos
como osciloscopios.
- Gestión de recursos y control de uso de memoria RAM para evitar
sobrecargas.
- Creación de tres tipos de gráficas:
  * Gráfica en crudo: muestra datos directamente, con opción de
operaciones matemáticas
    sobre el eje Y, validando sintaxis de expresiones.
  * Gráfica de sobrecarga: análisis específico para curvas de
sobrecarga, con optimizaciones
    basadas en intervalos predefinidos y avisos al usuario sobre
posibles incompatibilidades.
```

```
  * Gráfica de análisis: aplica operaciones temporales y estadísticas
sobre los datos para análisis avanzado.
- Subgráficos: permite organizar múltiples gráficos en una matriz
configurable, con
  opciones para coordinar ejes X e Y entre gráficos, y validaciones
para asegurar la
  compatibilidad de los datos representados.

En conjunto, este objeto facilita un análisis flexible, eficiente y
coherente de datos,
con especial atención a la gestión de recursos y la adaptabilidad a
diferentes tipos
de representación gráfica.
"""


"""
Funciones de validación de apoyo para la clase "Procesador_Analítico"

"""

"""
Función para manejar la interactividad en gráficos (plots).

Actualiza la anotación mostrada al seleccionar una curva, mostrando
la etiqueta (label) asociada a dicha curva.

Parámetros:
- sel: Objeto de selección de Matplotlib que contiene información
       sobre el artista (curva) seleccionada.
"""
def on_add(sel):
    line = sel.artist
    label = line.get_label()
    sel.annotation.set_text(label)


"""
Verifica si un valor dado puede interpretarse como un número.

Se reemplaza el carácter '−' (menos tipográfico) por el guion normal
'-'
para facilitar la conversión.

Parámetros:
- valor (str): Cadena de texto que se desea verificar.

Retorna:
- bool: True si el valor es numérico, False en caso contrario.
"""
def es_numero(valor):
    try:
        valor = valor.replace('−', '-')  # reemplaza el "signo menos
bonito" por el guion normal
        float(valor)
        return True
    except ValueError:
        print("El valor " + valor + " no es un número")
        return False
```

```
class Procesador_Analítico:
    def __init__(self):

        self.ejex_generado = []

        #Tendremos un control sobre qué subplots hacemos, guardaremos
objetos subplot, los cuales contienen información de dónde
        #están posicionados
        #Esta lista se borrará siempre que se cambie el tamaño de la
matriz de subplots o que se quite la opción de subplots
        #De igual manera que se borrarán todos los subplots existentes

        #estos objetos tipo subplot tienen todas sus propiedades,
incluso su posición en el canvas

        self.ax = []

        self.line = []

        self.listagener = []


        #Esta lista contendrá el tiempo  (tmax - tmin) sobre el que se
calculó la energía en el modo de detección de valles
        self.tiempoenergia = []

        #Esta lista es complementaria a la anterior, contendrá la
posición (en array) desde la que inicia el intervalo y el final
        self.intervaloenergia = []

        #a continuación se crean una serie de listas para datos de
interés para las pruebas para las que fue concebido el programa

        #para el tiempo que tarda la corriente en alcanzar su máximo
absoluto cuando comienza la subida
        self.tiempoimax = []

        #es la corriente a la que el LCL limita a la carga (viene
después del pico)
        self.ilimite = []

        #es el tiempo que dura el pico de corriente, desde su subida
hasta su bajada al valor límite
        self.tiempototmax = []


        self.indicemin = 0
        self.indicemax = 999999



        self.canvas = None
        self.fig = Figure(figsize=(12, 10), dpi=150)

    '''
    Función con el objetivo de realizar una gráfica cualquiera en el
canvas con propósito de hacer debugging y corrección de fallos

    Parámetros:
```

```
    container:

    '''
    def plotDebug( self , container):

        ax = self.fig.add_subplot( 1 , 1 ,1)

        ax.plot([0,1,2] , [2,1,0], label = "debug")


        if not self.canvas:
            self.canvas = FigureCanvasTkAgg(self.fig , container)
            self.toolbar = NavigationToolbar2Tk(self.canvas,
container)

            #py.show()
            #toolbar.update()
            self.canvas.get_tk_widget().pack(side="bottom",
fill="both", expand=True)

        self.toolbar.update()
        self.canvas.draw_idle()

    #Con esta función representaremos las gráficas de la columna
seleccionada y con los dataframes seleccionados con indices
    #IMPORTANTE TENER EN CUENTA QUE LO QUE TENEMOS EN INDICES ES EL
NÚMERO DEL FICHERO, NO LOS ÍNDICES EN LA LISTA!!

    #se manda columnasseleccionadas, pero solo nos inetersa la del
ejey
    def plotRaw(self, dataframes,dataframes_selec,rango, columnas,
container, labels, mostrar_x, escala_leyenda
,subplot,subplotmat,subplotpos, node_operation, operation):



        #Comprobaremos que se ha seleccionado la opción de subplots
        if subplot == "grey":

            if self.canvas:#se borrara si había un canvas, si no lo
habia es que directamente no había plots
                self.clearSubplot(subplotpos[0], subplotpos[1])

            #ahora comprobaremos la coherencia de los datos, es decir,
que se hayan seleccionado tanto filas como
            #columnas y que la posición asociada el raw data está
dentro de los límites, si no, no representaremos nada
            if 1 <= int(subplotpos[0]) <=   int(subplotmat[0]) and 1
<= int(subplotpos[1]) <=   int(subplotmat[1]):
                #El int(subplotmat[1])* es debido a que hemos puesto
que como máximo sean 3 columnas, si se cambiara, debería cambiarse
esto también
                ax = self.fig.add_subplot( int(subplotmat[0]),
int(subplotmat[1]), int(subplotmat[1])*( int(subplotpos[0]) - 1 ) +
int(subplotpos[1]) )
            else:
                messagebox.showerror("Error","No se ha seleccionado
una posición de subplot adecuada")

                return
        else:
```

```
            if self.canvas:#se borrara si había un canvas, si no lo
habia es que directamente no había plots
                self.clearSubplot(1, 1)

            ax = self.fig.add_subplot( 1 , 1 ,1)



        ax.set_xlabel(labels[0])
        ax.set_ylabel(labels[1])
        ax.set_title(labels[2])

        if mostrar_x == "No":
            ax.set_xticklabels([])



        grupodataframes = list( dataframes_selec.keys() )[0]
        nombresdataframes = dataframes_selec[grupodataframes]

        #creamos una lista donde guardaremos los objetos 'line' cada
vez que represenetemos una curva
        lines = [ ]



        #creamos unos 10 colores bien distinguibles, que serán los más
representativos para la leyenda
        colores =
['blue','orange','green','red','purple','brown','pink','gray','olive',
'cyan']
        indice_colores = 0
        #Se realizan los plots, si se le han pasado operaciones de
Math se actuará en consecuencia para cada eje
        for indice, clave in enumerate(nombresdataframes):#recorremos
los nombres de los dataframes



            #------------EJE X
            if columnas[0] != "":#si no hay eje x generado es que
cogeremos columna o Math

                df = dataframes[grupodataframes][clave]

                if rango[0] != "" and rango[1] != "":#si se ha escrito
algo en el rango


                    ejex = df[(df[columnas[0]] >= float(rango[0]) ) &
(df[columnas[0]] <= float(rango[1]) )]
                else:

                    ejex = df[columnas[0]]

                indices = ejex.index.tolist()
                indiceminy = indices[0]
                indicemaxy = indices[-1]
                print(indiceminy)
                print(indicemaxy)
```

```
            else:#de lo contrario, cogeremos el eje x generado


                if rango[0] != "" and rango[1] != "": #si se ha
escrito algo en el rango

                    indices = [i for i, x in
enumerate(self.ejex_generado) if float(rango[0]) <= x <=
float(rango[1])]

                    ejex = self.ejex_generado[  indices[0] : indices[-
1] + 1 ]

                else:# si no, se genera automáticamente

                    ejex = self.ejex_generado
                    indices = [ 0, len(ejex) - 1 ]


                indiceminy = indices[0]
                indicemaxy = indices[-1]
                print(indiceminy)
                print(indicemaxy)


            print(ejex)
            print(dataframes[grupodataframes][clave].loc[
indiceminy:indicemaxy ][columnas[1]])

            #---------------EJE Y

            label_sub  = clave

            if node_operation is not None:
                df = dataframes[grupodataframes][clave]


                ejey = self.realizar_operacion(node_operation, df)

                ejey = ejey[  indiceminy:indicemaxy + 1 ]

                label_sub += "_(" + operation + ")"
            else:
                ejey = dataframes[grupodataframes][clave].loc[
indiceminy:indicemaxy  ][columnas[1]]
                label_sub += "_(" + columnas[1] + ")"



            #-----------------PLOT

            if len(ejey) != len(ejex):
                messagebox.showerror("Error","La longitud del eje x ("
+ str(len(ejex)) + ") es distinta a la del eje y (" + str(len(ejey)) +
")")
                print(ejex)
                print(ejey)
```

```
                self.ax.append( ax )
                self.line.append([])
                return
            if indice < 5 or indice >= (len(nombresdataframes) - 5) :
                line, = ax.plot(ejex , ejey, label = label_sub, color
= colores[indice_colores])
                indice_colores += 1
                lines.append(line)
            else:
                line, = ax.plot(ejex , ejey, label = label_sub)
                lines.append(line)



            #Cambiamos el tamaño de fuente de los ejes, el default es
10, lo dividiremos según las filas o columnas que tengamos
            '''for tick in ax.xaxis.get_major_ticks():
                tick.label.set_fontsize( (10/(int( subplotmat[0] ) ) )
)

            for tick in ax.yaxis.get_major_ticks():
                tick.label.set_fontsize( (10/(int( subplotmat[1] )  ))
)'''

            #Cambiamos el tamaño de las xlabel e ylabel si estamos con
subplots
            if subplot == "grey":
                ax.xaxis.label.set_size( 10/(int( subplotmat[0] ) )  )
                ax.yaxis.label.set_size( 10/(int( subplotmat[1] )  ) )

            print(clave)
            print("\n")



        #Si tenemos más de 10 curvas querremos manejar adecuadamente
la leyenda
        if len(nombresdataframes) > 10:
            #tan solo querremos mostrar los primeros 5 y los últimos 5
curvas en la leyenda para evitar que ocupe mucho
            lines_to_show = lines[:5] + lines[-5:]

            #ahora encontraremos las labels de las primeras 5 y las
últimas 5
            labels_to_show = [line.get_label() for line in
lines_to_show]


            if subplot == "grey":#si tenemos subplots querremos que el
tamaño no siga la misma regla
                ax.legend(lines_to_show,labels_to_show, loc = "upper
left", prop = { "size": (escala_leyenda*11)/( int( subplotmat[1] ) *
int(subplotmat[0] ) ) } )

            else:
                ax.legend(lines_to_show,labels_to_show, loc = "upper
left", prop = { "size": escala_leyenda*5 } )
        else:
            if subplot == "grey":#si tenemos subplots querremos que el
tamaño no siga la misma regla
```

```
                ax.legend(loc = "upper left", prop = { "size":
(escala_leyenda*11)/( int( subplotmat[1] ) *  int(subplotmat[0] ) ) }
)

            else:
                ax.legend(loc = "upper left", prop = { "size":
escala_leyenda*5 } )

        ax.grid(True)


        #introducimos el subplot en la lista
        self.ax.append( ax )
        self.line.append(lines)

        print("Nuevo subplot añadido a la lista de subplots, en la
posición " + str(self.ax.index(ax)) + " El tamaño total es de " +
str(len(self.ax)) + " ejes y " + str(len(self.line)) + " grupos de
lineas")

        self.fig.tight_layout()

        if not self.canvas:
            self.canvas = FigureCanvasTkAgg(self.fig , container)
            self.toolbar = NavigationToolbar2Tk(self.canvas,
container)

            #py.show()
            #toolbar.update()
            self.canvas.get_tk_widget().pack(side="bottom",
fill="both", expand=True)

        self.toolbar.update()
        self.canvas.draw_idle()


    def plotGener(self, dataframes_or,dataframes_selec, columnas,
container, estad, columnas_estad ,labels, mostrar_x
,subplot,subplotmat,subplotpos):

        #Comprobaremos que se ha seleccionado la opción de subplots
        if subplot == "grey":

            if self.canvas:#se borrara si había un canvas, si no lo
habia es que directamente no había plots
                self.clearSubplot(subplotpos[0], subplotpos[1])

            #ahora comprobaremos la coherencia de los datos, es decir,
que se hayan seleccionado tanto filas como
            #columnas y que la posición asociada el raw data está
dentro de los límites, si no, asignaremos una
            if 1 <= int(subplotpos[0]) <=   int(subplotmat[0]) and 1
<= int(subplotpos[1]) <=   int(subplotmat[1]):
                #El int(subplotmat[1])* es debido a que hemos puesto
que como máximo sean 3 columnas, si se cambiara, debería cambiarse
esto también
                ax = self.fig.add_subplot( int(subplotmat[0]),
int(subplotmat[1]), int(subplotmat[1])*( int(subplotpos[0]) - 1 ) +
int(subplotpos[1]) )
            else:
```

```
                messagebox.showerror("Error","No se ha seleccionado
una posición de subplot adecuada")
                return
        else:

            if self.canvas:#se borrara si había un canvas, si no lo
habia es que directamente no había plots
                self.clearSubplot(subplotpos[0], subplotpos[1])

            ax = self.fig.add_subplot( 1  , 1  ,1)



        ax.set_xlabel(labels[0])
        ax.set_ylabel(labels[1])
        ax.set_title(labels[2])

        if mostrar_x == "No":
            ax.set_xticklabels([])



        #creamos una lista donde guardaremos los objetos 'line' cada
vez que represenetemos una curva
        lines = [ ]
        ejey = None

        #Ahora, dependiendo de qué modelo de estadística que se haya
elegido, se procede d euna forma u otra
        if estad[0] == 'Análisis de sobrecarga':

            #Esta función requiere cambiar los nombres de las columnas
y adaptar los dataframes para adecuarlos a los nombres que
            #manejan las funciones de sborecarga

            #adaptamos los dataframes
            dataframes = copy.deepcopy(dataframes_or)
            grupodataframes = list( dataframes_selec.keys() )[0]
            nombresdataframes = dataframes_selec[grupodataframes]
            lista_dataframes = []

            #debemos adaptar al formato de este tipo de funciones, que
reciben una lista de dataframes
            for nombres_individuos in nombresdataframes:
                lista_dataframes.append(
dataframes[grupodataframes][nombres_individuos] )


            tipo_curva = None

            #si estamos ante cierto tipo de análisis deberemos saber
el tipo de curva, dentro de la lista estad
            if estad[2] in [ "Intervalo de limitación", "Energía",
"Corriente media limitación" , "Intervalo de transición"]:
                if estad[2] == "Intervalo de transición":
                    tipo_curva = columnas_estad[1][1]
                else:
                    tipo_curva = columnas_estad[2][1]

                columnas_estad = columnas_estad[ :-1 ]
```

```
            lista_de_nombres = columnas_estad
            # Iterar sobre cada DataFrame en la lista
            for df in lista_dataframes:
                # Crear diccionario de mapeo solo con las columnas
existentes en el DataFrame
                rename_dict = {nombre_existente: nuevo_nombre
                               for nuevo_nombre, nombre_existente in
lista_de_nombres
                               if nombre_existente in df.columns}
                # Renombrar las columnas aplicando el diccionario
                df.rename(columns=rename_dict, inplace=True)

            print("Tipo de curva:")
            print(tipo_curva)
            print("Lista de nombres de columna:")
            print(lista_de_nombres)
            #Una vez que se han cambiado los nombres de las columnas
seleccionadas, procedemos de diferente forma según el tipo de curva

            if estad[1] == 'Sobrecarga':


                if estad[2] == 'Valor máximo':

                    columna = columnas_estad[0][0]#la columna será la
escogida

                    self.valormax_SC(lista_dataframes, columna)

                    ejey = self.listagener

                elif estad[2] == 'Intervalo de limitación':

                    self.intervalo_SC(lista_dataframes, tipo_curva)

                    ejey = self.tiempoenergia

                elif estad[2] == 'Energía':

                    columnas = [ columnas_estad[0][0] ,
columnas_estad[1][0] ]

                    self.intervalo_SC(lista_dataframes, tipo_curva)
#debe ejecutarse antes el análisis del intervalo
                    self.energia_SC(lista_dataframes, columnas, 1)

                    ejey = self.listagener
                else:#si no, será corriente media

                    self.intervalo_SC(lista_dataframes, tipo_curva)
#debe ejecutarse antes el análisis del intervalo
                    self.corriente_media_SC(lista_dataframes)

                    ejey = self.listagener


            elif estad[1] == 'Carga capacitiva':

                if estad[2] == 'Valor máximo':
```

```
                    columna = columnas_estad[0][0]#la columna será la
escogida

                    self.valormax_inrush(lista_dataframes, columna)

                    ejey = self.listagener

                elif estad[2] == 'Intervalo de transición':

                    self.intervalo_inrush(lista_dataframes,
tipo_curva)

                    ejey = self.tiempoenergia

                else:#si no, será la energía
                    columnas = [ columnas_estad[0][0] ,
columnas_estad[1][0] ]

                    self.intervalo_inrush(lista_dataframes,
tipo_curva) #debe ejecutarse antes el análisis del intervalo
                    self.energia_inrush(lista_dataframes, columnas, 1)

                    ejey = self.listagener


            else:#será de cortocircuito
                if estad[2] == 'Valor máximo':

                    columna = columnas_estad[0][0]#la columna será la
escogida

                    self.valormax_PS(lista_dataframes, columna)

                    ejey = self.listagener

                elif estad[2] == 'Intervalo de limitación':

                    self.intervalo_PS(lista_dataframes, tipo_curva)

                    ejey = self.tiempoenergia

                elif estad[2] == 'Energía':

                    columnas = [ columnas_estad[0][0] ,
columnas_estad[1][0] ]

                    self.intervalo_PS(lista_dataframes, tipo_curva)
#debe ejecutarse antes el análisis del intervalo
                    self.energia_PS(lista_dataframes, columnas, 1)

                    ejey = self.listagener
                else:#si no, será corriente media

                    self.intervalo_PS(lista_dataframes, tipo_curva)
#debe ejecutarse antes el análisis del intervalo
                    self.corriente_media_PS(lista_dataframes)

                    ejey = self.listagener


            print(ejey)
```

```
            line, = ax.plot(nombresdataframes, ejey)
            # Aplicar rotación a las etiquetas del eje X
            # Establecer etiquetas rotadas
            for label in ax.get_xticklabels():
                label.set_rotation(45)  # o el ángulo que necesites
                label.set_fontsize(4)


            lines.append(line)



        else:# si no es de sobrecarga, es análisis temporales

            grupodataframes = list( dataframes_selec.keys() )[0]
            nombresdataframes = dataframes_selec[grupodataframes]

            if estad[1] == 'Derivada':

                columna_ejex = columnas_estad[0][1]
                columna_ejey = columnas_estad[1][1]
                window_length = columnas_estad[2][1]
                poly = columnas_estad[3][1]

                print(window_length)
                print(poly)

                for nombres_individuos in nombresdataframes:
                    ejex =
dataframes_or[grupodataframes][nombres_individuos][columna_ejex]
                    ejey =
dataframes_or[grupodataframes][nombres_individuos][columna_ejey]

                    self.derivada(ejex, ejey, int(window_length),
int(poly))
                    line, = ax.plot(ejex, self.listagener)

                    lines.append(line)







        ax.grid(True)
        #introducimos el subplot en la lista
        self.ax.append( ax )
        self.line.append(lines)

        print("Nuevo subplot añadido a la lista de subplots, en la
posición " + str(self.ax.index(ax)) + " El tamaño total es de " +
str(len(self.ax)) + " ejes y " + str(len(self.line)) + " grupos de
lineas")

        self.fig.tight_layout()

        if not self.canvas:
            self.canvas = FigureCanvasTkAgg(self.fig , container)
```

```
            self.toolbar = NavigationToolbar2Tk(self.canvas,
container)

            #py.show()
            #toolbar.update()
            self.canvas.get_tk_widget().pack(side="bottom",
fill="both", expand=True)

        self.toolbar.update()
        self.canvas.draw_idle()


    #esta función generará una lista de floats con la misma longitud
que el número de filas
    #debemos tener en cuenta que hay 5 escalas a la izquierda y 5 a la
derecha
    def generatiempo( self, escala,offset,numpuntos ):


        self.ejex_generado = np.arange( -5*escala + offset , 5*escala
+ offset, (10*escala)/numpuntos )

        print("el eje x tiene una cantidad de:")
        print(  str(len(self.ejex_generado)) + " puntos")
        print("Y va desde " + str(self.ejex_generado[0]) + " hasta " +
str(self.ejex_generado[-1]))

    #esta función generará una lista de floats con la misma longitud
que el número de filas
    #debemos tener en cuenta que hay 5 escalas a la izquierda y 5 a la
derecha
    def generaejex( self, escala,offset,dataframes, dataframes_selec
):

        if es_numero(escala) == False and es_numero(offset) == False:
            print("No se ha introducido correctamente la escala o el
offset")
            return

        grupodataframes = list( dataframes_selec.keys() )[0]
        nombresdataframes = dataframes_selec[grupodataframes]

        df = dataframes[grupodataframes][nombresdataframes[0]]

        numpuntos = df.shape[0]

        escala = float(escala)
        offset = float(offset)

        self.ejex_generado = np.arange( -5*escala + offset , 5*escala
+ offset, (10*escala)/numpuntos )

        print("el eje x tiene una cantidad de:")
        print(  str(len(self.ejex_generado)) + " puntos")
        print("Y va desde " + str(self.ejex_generado[0]) + " hasta " +
str(self.ejex_generado[-1]))

    #%%FUNCIONES PARA INTERPRETAR Y REALIZAR OPERACIONES MATH
    def evaluate_ast(self, node, df):
        """
```

```
        Evalúa recursivamente el AST validado de la expresión y
devuelve el resultado.

        Esta función asume que el nodo ya ha sido validado mediante
validar_expresion.
        Las operaciones se realizan element wise, aprovechando que los
valores en dummy_values
        son pandas Series, lo cual permite realizar operaciones
vectorizadas.

        Parámetros:
            node: Nodo AST a evaluar.

        Devuelve:
            El resultado de la evaluación (una pandas Series o un
escalar).

        Lanza:
            ValueError: Si se encuentra un operador o nodo no
soportado.
        """


        if isinstance(node, ast.Expression):
            return self.evaluate_ast(node.body, df)
        elif isinstance(node, ast.BinOp):
            # Evaluamos recursivamente el operando izquierdo y el
derecho.
            left = self.evaluate_ast(node.left,df)
            right = self.evaluate_ast(node.right,df)
            # Dependiendo del operador, se realiza la operación
element wise.
            if isinstance(node.op, ast.Add):
                return left + right
            elif isinstance(node.op, ast.Sub):
                return left - right
            elif isinstance(node.op, ast.Mult):
                return left * right
            else:
                raise ValueError(f"Operador no soportado:
{type(node.op).__name__}")
        elif isinstance(node, ast.UnaryOp):
            # Operadores unarios
            operand = self.evaluate_ast(node.operand,df)
            if isinstance(node.op, ast.UAdd):
                return +operand
            elif isinstance(node.op, ast.USub):
                return -operand
            else:
                raise ValueError("Operador unario no soportado")
        elif isinstance(node, ast.Constant):
            # Retornamos el valor numérico de la constante.
            return node.value
        elif isinstance(node, ast.Name):
            # Al evaluar un nodo de nombre, se sustituyen por los
valores dummy definidos.
            if node.id in df:
                return df[node.id]
            else:
                raise ValueError(f"Nombre no permitido en evaluación:
{node.id}")
```

```
        else:
            raise ValueError(f"Nodo no soportado en evaluación:
{type(node).__name__}")

    def realizar_operacion(self, node, df):
        """
        Función extra que:
          - Valida la expresión ingresada.
          - Evalúa la expresión usando los valores dummy (pandas
Series) de forma element wise.
          - Imprime el resultado en la línea de comandos.

        Parámetros:
            expr (str): La expresión a evaluar.
        """
        print("df")
        print(df)
        df = df.to_dict("series")
        print("dict")
        print(df)

        try:
            # Se evalúa recursivamente el AST.
            result = self.evaluate_ast(node, df)
            print("Resultado:")
            print(result)
            return result
        except Exception as e:
            print("Error al realizar la operación:", e)

    #%%FUNCIONES DE ANÁLISIS DE SOBRECARGA
    '''
        Función que calcula y almacena en una lista genérica los
valores máximos ajustados de una columna específica
        de una serie de dataframes. El cálculo del máximo se realiza
dentro de intervalos definidos, dependiendo del tipo de curva.

        Parámetros:

        dataframes: lista de objetos tipo DataFrame que contienen las
curvas a analizar.

        columna: cadena de caracteres que indica la columna específica
a procesar en cada DataFrame.
    '''
    def valormax_SC(self, dataframes, columna ):

        # Limpiamos la lista genérica que almacenará los valores
máximos procesados.
        self.listagener.clear()

        # Recorremos todos los DataFrames proporcionados.
        contador = 0
        for data in dataframes:

            # Extraemos los datos de la columna seleccionada, dentro
del rango [indicemin:indicemax],
            # y los convertimos a un array de NumPy para facilitar el
procesamiento numérico.
            intermediario =
data.loc[self.indicemin:self.indicemax][columna].to_numpy()
```

```
            npts = len(intermediario)  # Número de puntos en el array
            offset = 0  # Inicializamos el offset

            # El valor del offset se calcula promediando una pequeña
porción de la curva (según su tipo),
            # que se considera representativa de una región de
referencia o "baseline".
            # Luego se calcula el valor máximo dentro de un rango
específico y se ajusta restando el offset.

            if columna == "VdsJFET":  # Tensión del JFET
                offset = np.mean(intermediario[int(round(npts *
0.005)) - 1 : int(round(npts * 0.01)) - 1])

self.listagener.append(np.max(intermediario[int(round(npts * 0.1)) - 1
: int(round(npts * 0.3)) - 1]) - offset)

            elif columna == "VdsPMOS":  # Tensión del PMOS
                offset = np.mean(intermediario[int(round(npts *
0.005)) - 1 : int(round(npts * 0.01)) - 1])

self.listagener.append(np.max(intermediario[int(round(npts * 0.1)) - 1
: int(round(npts * 0.3)) - 1]) - offset)

            elif columna == "I":  # Corriente
                offset = np.mean(intermediario[int(round(npts * 0.96))
- 1 : int(round(npts * 0.965)) - 1])

self.listagener.append(np.max(intermediario[int(round(npts * 0.1)) - 1
: int(round(npts * 0.3)) - 1]) - offset)

            # Imprimimos información de progreso cada 100 dataframes
para evitar sobrecarga de salida.
            if contador % 100 == 0:
                print("Cargando dataframe " + str(contador))
                print("El offset es " + str(offset))

            contador += 1

        # Imprimimos la lista final con los valores máximos ajustados
y su longitud.
        print(self.listagener)
        print("El tamaño del vector de datos genéricos es " +
str(len(self.listagener)))


    '''
        Función que calcula y almacena en una lista genérica los
valores máximos ajustados de una columna específica
        de una serie de dataframes.

        Parámetros:

        dataframes: lista de objetos tipo DataFrame que contienen las
curvas a analizar.

        columna: cadena de caracteres que indica la columna específica
a procesar en cada DataFrame.
    '''
    def valormax_inrush(self, dataframes, columna ):
```

```
        self.listagener.clear()
        #Debemos crear un objeto tipo array de numpy para cada columna
con el .to_numpy()

        # Recorremos todos los DataFrames proporcionados.
        contador = 0
        for data in dataframes:

            # Extraemos los datos de la columna seleccionada, dentro
del rango [indicemin:indicemax],
            # y los convertimos a un array de NumPy para facilitar el
procesamiento numérico.
            intermediario =
data.loc[self.indicemin:self.indicemax][columna].to_numpy()

            npts = len(intermediario)  # Número de puntos en el array
            offset = 0  # Inicializamos el offset

            # El valor del offset se calcula promediando una pequeña
porción de la curva (según su tipo),
            # que se considera representativa de una región de
referencia o "baseline".
            # Luego se calcula el valor máximo dentro de un rango
específico y se ajusta restando el offset.

            if columna == "VdsJFET":  # Tensión del JFET
                offset = np.mean(intermediario[int(round(npts * 0.96))
- 1 : int(round(npts * 0.965)) - 1])

self.listagener.append(np.max(intermediario[int(round(npts * 0.1)) - 1
: int(round(npts * 0.4)) - 1]) - offset)
            elif columna == "VdsPMOS":  # Tensión del PMOS
                offset = np.mean(intermediario[int(round(npts * 0.96))
- 1 : int(round(npts * 0.965)) - 1])

self.listagener.append(np.max(intermediario[int(round(npts * 0.01)) -
1 : int(round(npts * 0.3)) - 1]) - offset)

            elif columna == "I":  # Corriente
                offset = np.mean(intermediario[int(round(npts *
0.005)) - 1 : int(round(npts * 0.01)) - 1])

self.listagener.append(np.max(intermediario[int(round(npts * 0.3)) - 1
: int(round(npts * 0.7)) - 1]) - offset)

            # Imprimimos información de progreso cada 100 dataframes
para evitar sobrecarga de salida.
            if contador % 100 == 0:
                print("Cargando dataframe " + str(contador))
                print("El offset es " + str(offset))

            contador += 1

        # Imprimimos la lista final con los valores máximos ajustados
y su longitud.
        print(self.listagener)
        print("El tamaño del vector de datos genéricos es " +
str(len(self.listagener)))
```

```
    '''
        Función que calcula y almacena en una lista genérica los
valores máximos ajustados de una columna específica
        de una serie de dataframes.

        Parámetros:

        dataframes: lista de objetos tipo DataFrame que contienen las
curvas a analizar.

        columna: cadena de caracteres que indica la columna específica
a procesar en cada DataFrame.
    '''
    def valormax_PS(self, dataframes, columna  ):


        # Limpiamos la lista genérica que almacenará los valores
máximos procesados.
        self.listagener.clear()

        # Recorremos todos los DataFrames proporcionados.
        contador = 0
        for data in dataframes:

            # Extraemos los datos de la columna seleccionada, dentro
del rango [indicemin:indicemax],
            # y los convertimos a un array de NumPy para facilitar el
procesamiento numérico.
            intermediario =
data.loc[self.indicemin:self.indicemax][columna].to_numpy()

            npts = len(intermediario)  # Número de puntos en el array
            offset = 0  # Inicializamos el offset

            # El valor del offset se calcula promediando una pequeña
porción de la curva (según su tipo),
            # que se considera representativa de una región de
referencia o "baseline".
            # Luego se calcula el valor máximo dentro de un rango
específico y se ajusta restando el offset.

            if columna == "VdsJFET":  # Tensión del JFET
                offset = np.mean(intermediario[int(round(npts * 0.4))
- 1 : int(round(npts * 0.405)) - 1])

self.listagener.append(np.max(intermediario[int(round(npts * 0.45)) -
1 : int(round(npts * 0.6)) - 1]) - offset)

            elif columna == "VdsPMOS":  # Tensión del PMOS
                offset = np.mean(intermediario[int(round(npts * 0.4))
- 1 : int(round(npts * 0.405)) - 1])
                self.listagener.append(np.max(intermediario) - offset)

            elif columna == "I":  # Corriente
                offset = np.mean(intermediario[int(round(npts *
0.005)) - 1 : int(round(npts * 0.01)) - 1])

self.listagener.append(np.max(intermediario[int(round(npts * 0.4)) - 1
: int(round(npts * 0.6)) - 1]) - offset)
```

```
            # Imprimimos información de progreso cada 100 dataframes
para evitar sobrecarga de salida.
            if contador % 100 == 0:
                print("Cargando dataframe " + str(contador))
                print("El offset es " + str(offset))

            contador += 1

        # Imprimimos la lista final con los valores máximos ajustados
y su longitud.
        print(self.listagener)
        print("El tamaño del vector de datos genéricos es " +
str(len(self.listagener)))



    '''
        Función que calculará el intervalo de tiempo en el que se
realizará el cálculo de la energía
        para las pruebas de SC.
        El cálculo se realiza buscando en la potencia en el JFET, como
producto de la corriente a través
        del JFET y del voltaje que cae en él, que es más fácil de
identificar.

        Parámetros:
        dataframes: lista de objetos tipo DataFrame que contienen las
curvas a analizar.
        tipo: cadena de caracteres que representa l tipo de curva (2A,
10A), que determinará por dónde empezar a buscar el inal del intervalo

    '''

    def intervalo_SC(self, dataframes, tipo):

        # Limpiamos las listas que almacenarán los tiempos de energía
y los intervalos de energía.
        self.tiempoenergia.clear()
        self.intervaloenergia.clear()

        # Recorremos todos los DataFrames proporcionados.
        contador = 0
        for data in dataframes:

            #========= CARGA DE DATOS Y OPERACIONES BÁSICAS =========
            if contador % 100 == 0:  # Mostramos el progreso cada 100
iteraciones para no saturar la salida.
                print("Cargando dataframe " + str(contador))

            # Obtenemos los datos de voltaje y corriente en el JFET
para el rango de índices entre indicemin y indicemax.
            intermediario1 =
data.loc[self.indicemin:self.indicemax]["VdsJFET"].to_numpy()  #
Voltaje en el JFET
            intermediario2 =
data.loc[self.indicemin:self.indicemax]["I"].to_numpy()  # Corriente

            npts = len(intermediario1)  # Número de puntos en los
datos
```

```
            # Calculamos los offsets de voltaje y corriente usando
promedios de segmentos pequeños
            offset_JFET = np.mean(intermediario1[int(round(npts *
0.005) - 1): int(round(npts * 0.01) - 1)])  # Offset de VdsJFET
            offset_I = np.mean(intermediario2[int(round(npts * 0.96) -
1): int(round(npts * 0.965) - 1)])  # Offset de I

            if contador % 100 == 0:
                print("El offset de V JFET es " + str(offset_JFET))
                print("El offset de I es " + str(offset_I))

            # Aplicamos un filtro de Savitzky-Golay para suavizar la
potencia calculada.
            # La potencia es el producto de (VdsJFET - offset_JFET) y
(I - offset_I)
            filtro = savgol_filter((intermediario1 - offset_JFET) *
(intermediario2 - offset_I), 99, 3)
            #=========================================================

            #=============== BÚSQUEDA DEL ÍNDICE DE COMIENZO
================
            # Buscamos el índice en el que la curva de potencia supera
un umbral.
            # Este índice marcará el inicio del intervalo de
limitación.
            indiceminimo = npts*0.2  # Inicializamos un valor grande
que posteriormente será reemplazado.
            umbral = 50  # Establecemos el umbral que debe superar la
media de los puntos de potencia para determinar el inicio.

            for j in range(int(npts * 0.2) - 1, int(npts * 0.202) - 1,
1):
                # Comprobamos si la media de un intervalo de puntos
supera el umbral
                if np.mean(filtro[j - 10: j]) >= umbral:
                    indiceminimo = j  # Al encontrar el punto,
guardamos el índice de inicio.

                    if contador % 100 == 0:
                        # Mostramos los valores para depuración si
estamos en una iteración múltiplo de 100
                        print(filtro[j])
                        print(indiceminimo)
                        print("El índice mínimo detectado se encuentra
en el tiempo " + str(self.ejex_generado[indiceminimo +
self.indicemin]))

                    break  # Salimos del bucle una vez que encontramos
el índice mínimo.

            #=============== BÚSQUEDA DEL ÍNDICE DE FIN
================
            # Ahora buscamos el índice en el que la potencia cae por
debajo de un umbral, indicando el final de la limitación.
            indicemaximo = self.indicemax  # Inicializamos el índice
máximo al valor predeterminado.

            #Según el tipo, empezaremos a buscar antes o después
            if tipo == "2A":
                inicio_busqueda = 0.4
            else:
```

```
                inicio_busqueda = 0.7

            # Buscamos entre el 40% y el final la curva.
            for j in range(int(npts * inicio_busqueda) - 1, npts - 1,
1):
                if np.mean(filtro[j - 10: j]) <= umbral:  # Si la
media de los puntos es menor que el umbral
                    indicemaximo = j  # Actualizamos el índice máximo.

                    if contador % 100 == 0:
                        # Mostramos los valores para depuración si
estamos en una iteración múltiplo de 100
                        print(filtro[j])
                        print(indicemaximo)
                        print("El índice máximo detectado se encuentra
en el tiempo " + str(self.ejex_generado[indicemaximo +
self.indicemin]))

                    break  # Salimos del bucle una vez que encontramos
el índice máximo.

            if contador % 100 == 0:
                # Mostramos el intervalo de tiempo entre el mínimo y
el máximo encontrado
                print("El tiempo entre el máximo y el mínimo es " +
str(self.ejex_generado[indicemaximo + self.indicemin] -
self.ejex_generado[indiceminimo + self.indicemin]))

            # Actualizamos las listas de tiempo y intervalos de
energía.
            self.tiempoenergia.append(self.ejex_generado[indicemaximo
+ self.indicemin] - self.ejex_generado[indiceminimo + self.indicemin])
            self.intervaloenergia.append([indiceminimo +
self.indicemin, indicemaximo + self.indicemin])
            contador += 1

        # Al finalizar el proceso, mostramos el tamaño del vector de
tiempos calculados.
        print("El tamaño del vector de tiempo es " +
str(len(self.tiempoenergia)))

    '''
    Crearemos la función de operación binaria para obtener la energía.
Esta función multiplicará dos columnas de
    cada dataframe y calculará la integral del resultado usando el
método trapezoidal.

    El comportamiento depende del parámetro `modo`:
    - Si `modo = 0`, se calcula la integral de todo el intervalo de
datos.
    - Si `modo = 1`, se calcula la integral en intervalos previamente
calculados.

    Parámetros:
    dataframes: Lista de objetos tipo DataFrame que contienen los
datos a analizar.
    columnas: Lista con las columnas específicas a procesar en cada
DataFrame.
    modo: Parámetro que indica el tipo de cálculo de la integral (0
para todo el intervalo, 1 para intervalos específicos).
    '''
```

```
    def energia_SC(self, dataframes, columnas, modo):

        # Limpiamos la lista que almacenará los resultados de la
energía calculada.
        self.listagener.clear()

        # Recorremos todos los DataFrames proporcionados.
        contador = 0
        for data in dataframes:

            # Mostramos el progreso cada 100 iteraciones para no
saturar la salida.
            if contador % 100 == 0:
                print("Cargando dataframe " + str(contador))

            # Obtenemos los datos de las dos columnas especificadas
para el rango de índices entre indicemin y indicemax.

            if columnas[0] == "VdsJFET" or columnas[0] == "VdsPMOS":
                columnaV = columnas[0]
                columnaI = columnas[1]
            else:
                columnaV = columnas[1]
                columnaI = columnas[0]


            intermediario1 =
data.loc[self.indicemin:self.indicemax][columnaV].to_numpy()  #
Columna 1 (Ej. voltaje)
            intermediario2 =
data.loc[self.indicemin:self.indicemax][columnaI].to_numpy()  #
Columna 2 (Ej. corriente)

            npts = len(intermediario1)  # Número de puntos en los
datos

            # Calculamos los offsets de cada columna usando promedios
de segmentos pequeños (al principio y al final).
            offset_1 = np.mean(intermediario1[int(round(npts * 0.005)
- 1): int(round(npts * 0.01) - 1)])  # Offset de la primera columna
            offset_2 = np.mean(intermediario2[int(round(npts * 0.96) -
1): int(round(npts * 0.965) - 1)])  # Offset de la segunda columna

            # Si el modo es 0, calculamos la integral de todo el
intervalo usando la regla trapezoidal
            if modo == 0:
                # La integral se calcula como la integral de la
multiplicación de las dos señales corregidas por los offsets
                self.listagener.append(np.trapz((intermediario1 -
offset_1) * (intermediario2 - offset_2), self.ejex_generado))
            else:
                # Si el modo es 1, cargamos los intervalos de energía
previamente calculados
                # Y calculamos la integral solo en ese intervalo
específico.
                intermediario1 =
data.loc[self.intervaloenergia[contador][0]:
self.intervaloenergia[contador][1]][columnas[0]].to_numpy()
```

```
                intermediario2 =
data.loc[self.intervaloenergia[contador][0]:
self.intervaloenergia[contador][1]][columnas[1]].to_numpy()

                # Calculamos la integral en el intervalo seleccionado
usando la regla trapezoidal.
                self.listagener.append(np.trapz((intermediario1 -
offset_1) * (intermediario2 - offset_2),

self.ejex_generado[self.intervaloenergia[contador][0]:
self.intervaloenergia[contador][1] + 1]))

            # Incrementamos el contador para procesar el siguiente
DataFrame.
            contador += 1

        # Al finalizar, mostramos el tamaño del vector de resultados
generados.
        print("El tamaño del vector de datos generados es " +
str(len(self.listagener)))


    '''
        Función que calculará el intervalo de tiempo en el que se
realizará el cálculo de la energía
        para las pruebas de Inrush.
        El cálculo se realiza utilizando la curva de corriente, debido
a que en ciertas situaciones existen fenómenos
        de oscilación que se ven muy pronunciados en la curva de
potencia y dificulta el análisis
        En las pruebas de Inrush solo nos fijamos en el transitorio de
corriente antes de la limitación
        por lo tanto el intervalo no es el de limitación sino el del
transitorio.
        Además, al cargarse una capacitancia, la subida es mucho más
suabe y difícil de detectar

        Parámetros:
        dataframes: lista de objetos tipo DataFrame que contienen las
curvas a analizar.

    '''

    def intervalo_inrush(self, dataframes, tipo):

        # Limpiamos las listas que almacenarán los tiempos de energía
y los intervalos de energía.
        self.tiempoenergia.clear()
        self.intervaloenergia.clear()

        # Recorremos todos los DataFrames proporcionados.
        contador = 0
        for data in dataframes:

            #========= CARGA DE DATOS Y OPERACIONES BÁSICAS =========
            if contador % 100 == 0:  # Mostramos el progreso cada 100
iteraciones para no saturar la salida.
                print("Cargando dataframe " + str(contador))

            # Obtenemos los datos de voltaje y corriente en el JFET
para el rango de índices entre indicemin y indicemax.
```

```
            intermediario1 =
data.loc[self.indicemin:self.indicemax]["I"].to_numpy()  # Voltaje en
el JFET

            npts = len(intermediario1)  # Número de puntos en los
datos

            #No necesitaremos corregir el offset debido a queno
trabajaremos comparando con un umbral absoluto, sino con factores de
un valor referencia

            # Aplicamos un filtro de Savitzky-Golay para suavizar la
potencia calculada.
            # La potencia es el producto de (VdsJFET - offset_JFET) y
(I - offset_I)
            filtro = savgol_filter((intermediario1 ), 99, 3)
            #============================================================

            #=============== BÚSQUEDA DEL ÍNDICE DE COMIENZO
================
            #Debido a que la subida es mucho más suabe debido a la
carca capacitiva, la búsqueda de la subida debe ser más exhaustiva
            #En primer lugar, tomaremos una media de referencia de una
zona característica antes de la subida,
            indiceminimo = npts*0.2  # Inicializamos un valor grande
que posteriormente será reemplazado.
            factor = 8  # Establecemos un factor que multiplicará a la
media de referencia, a partir del cual consideramos que se ha
producido la subida
            media_referencia = np.mean( filtro[ int(npts*0.25 - 1):
int(npts*0.26 - 1)] )

            if contador % 100 == 0:
                print("la media referencia del principio de la gráfica
es " + str(media_referencia))

            #comenzaremos a recorrer hacia atrás
            for j in range(int(npts * 0.35) - 1, int(npts * 0.25) - 1,
-1):
                # Comprobamos si la media de un intervalo de puntos
supera el umbral
                if filtro[j] < factor * abs(media_referencia):
                    indiceminimo = j  # Al encontrar el punto,
guardamos el índice de inicio.

                    if contador % 100 == 0:
                        # Mostramos los valores para depuración si
estamos en una iteración múltiplo de 100
                        print(filtro[j])
                        print(indiceminimo)
                        print("El índice mínimo detectado se encuentra
en el tiempo " + str(self.ejex_generado[indiceminimo +
self.indicemin]))

                    break  # Salimos del bucle una vez que encontramos
el índice mínimo.

            #=============== BÚSQUEDA DEL ÍNDICE DE FIN
================
            # Ahora buscamos el fin del transitorio de una manera
similar, solo que esta vez no emplearemos una media de referencia
```

```
            #lo haremos comparando tramos, iremos de derecha a
izquierda, comparando la media del punto y los 100 puntos más a la
derecha con la media del punto 5000 veces más a la izquierda y los 100
puntos más a la izquierda
            indicemaximo = self.indicemax  # Inicializamos el índice
máximo al valor predeterminado.

            #Según el tipo, empezaremos a buscar antes o después
            if tipo == "2A":
                inicio_busqueda = 0.8
            else:
                inicio_busqueda = 0.68


            for j in range(int(npts * inicio_busqueda) - 1, int(npts *
0.5) - 1, -1):
                if np.mean( filtro[  j - 5000 - 100 : j - 5000 ] ) >=
1.5*np.mean( filtro[ j : j + 100 ] ):
                    indicemaximo = j  # Actualizamos el índice máximo.

                    if contador % 100 == 0:
                        # Mostramos los valores para depuración si
estamos en una iteración múltiplo de 100
                        print(filtro[j])
                        print(indicemaximo)
                        print("El índice máximo detectado se encuentra
en el tiempo " + str(self.ejex_generado[indicemaximo +
self.indicemin]))

                    break  # Salimos del bucle una vez que encontramos
el índice máximo.

            if contador % 100 == 0:
                # Mostramos el intervalo de tiempo entre el mínimo y
el máximo encontrado
                print("El tiempo entre el máximo y el mínimo es " +
str(self.ejex_generado[indicemaximo + self.indicemin] -
self.ejex_generado[indiceminimo + self.indicemin]))

            # Actualizamos las listas de tiempo y intervalos de
energía.
            self.tiempoenergia.append(self.ejex_generado[indicemaximo
+ self.indicemin] - self.ejex_generado[indiceminimo + self.indicemin])
            self.intervaloenergia.append([indiceminimo +
self.indicemin, indicemaximo + self.indicemin])
            contador += 1
        # Al finalizar el proceso, mostramos el tamaño del vector de
tiempos calculados.
        print("El tamaño del vector de tiempo es " +
str(len(self.tiempoenergia)))


    '''
    Crearemos la función de operación binaria para obtener la energía.
Esta función multiplicará dos columnas de
    cada dataframe y calculará la integral del resultado usando el
método trapezoidal.

    El comportamiento depende del parámetro `modo`:
    - Si `modo = 0`, se calcula la integral de todo el intervalo de
datos.
```

```
    - Si `modo = 1`, se calcula la integral en intervalos previamente
calculados.

    Parámetros:
    dataframes: Lista de objetos tipo DataFrame que contienen los
datos a analizar.
    columnas: Lista con las columnas específicas a procesar en cada
DataFrame.
    modo: Parámetro que indica el tipo de cálculo de la integral (0
para todo el intervalo, 1 para intervalos específicos).
    '''

    def energia_inrush(self, dataframes, columnas, modo):

        # Limpiamos la lista que almacenará los resultados de la
energía calculada.
        self.listagener.clear()

        # Recorremos todos los DataFrames proporcionados.
        contador = 0
        for data in dataframes:

            # Mostramos el progreso cada 100 iteraciones para no
saturar la salida.
            if contador % 100 == 0:
                print("Cargando dataframe " + str(contador))

            # Obtenemos los datos de las dos columnas especificadas
para el rango de índices entre indicemin y indicemax.
            if columnas[0] == "VdsJFET" or columnas[0] == "VdsPMOS":
                columnaV = columnas[0]
                columnaI = columnas[1]
            else:
                columnaV = columnas[1]
                columnaI = columnas[0]


            intermediario1 =
data.loc[self.indicemin:self.indicemax][columnaV].to_numpy()  #
Columna 1 (Ej. voltaje)
            intermediario2 =
data.loc[self.indicemin:self.indicemax][columnaI].to_numpy()  #
Columna 2 (Ej. corriente)

            npts = len(intermediario1)  # Número de puntos en los
datos

            # Calculamos los offsets de cada columna usando promedios
de segmentos pequeños (al principio y al final).
            offset_1 = np.mean(intermediario1[int(round(npts * 0.96))
- 1 : int(round(npts * 0.965)) - 1])  # Offset de la primera columna
            offset_2 = np.mean(intermediario2[int(round(npts * 0.005))
- 1 : int(round(npts * 0.01)) - 1])  # Offset de la segunda columna

            # Si el modo es 0, calculamos la integral de todo el
intervalo usando la regla trapezoidal
            if modo == 0:
                # La integral se calcula como la integral de la
multiplicación de las dos señales corregidas por los offsets
                self.listagener.append(np.trapz((intermediario1 -
offset_1) * (intermediario2 - offset_2), self.ejex_generado))
```

```
            else:
                # Si el modo es 1, cargamos los intervalos de energía
previamente calculados
                # Y calculamos la integral solo en ese intervalo
específico.
                intermediario1 =
data.loc[self.intervaloenergia[contador][0]:
self.intervaloenergia[contador][1]][columnas[0]].to_numpy()
                intermediario2 =
data.loc[self.intervaloenergia[contador][0]:
self.intervaloenergia[contador][1]][columnas[1]].to_numpy()

                # Calculamos la integral en el intervalo seleccionado
usando la regla trapezoidal.
                self.listagener.append(np.trapz((intermediario1 -
offset_1) * (intermediario2 - offset_2),

self.ejex_generado[self.intervaloenergia[contador][0]:
self.intervaloenergia[contador][1] + 1]))

            # Incrementamos el contador para procesar el siguiente
DataFrame.
            contador += 1

        # Al finalizar, mostramos el tamaño del vector de resultados
generados.
        print("El tamaño del vector de datos generados es " +
str(len(self.listagener)))

    '''
        Función que calculará el intervalo de tiempo en el que se
realizará el cálculo de la energía
        para las pruebas de PS.
        El cálculo se realiza buscando en la potencia en el JFET, como
producto de la corriente a través
        del JFET y del voltaje que cae en él, que es más fácil de
identificar.

        Parámetros:
        dataframes: lista de objetos tipo DataFrame que contienen las
curvas a analizar.

    '''

    def intervalo_PS(self, dataframes, tipo):

        print("\n------------Analizando intervalo de limitación de
curvas PS------------\n")
        # Limpiamos las listas que almacenarán los tiempos de energía
y los intervalos de energía.
        self.tiempoenergia.clear()
        self.intervaloenergia.clear()

        # Recorremos todos los DataFrames proporcionados.
        contador = 0
        for data in dataframes:

            #========= CARGA DE DATOS Y OPERACIONES BÁSICAS =========
            if contador % 100 == 0:  # Mostramos el progreso cada 100
iteraciones para no saturar la salida.
                print("Cargando dataframe " + str(contador))
```

```
            # Obtenemos los datos de voltaje y corriente en el JFET
para el rango de índices entre indicemin y indicemax.
            intermediario1 =
data.loc[self.indicemin:self.indicemax]["VdsJFET"].to_numpy()  #
Voltaje en el JFET
            intermediario2 =
data.loc[self.indicemin:self.indicemax]["I"].to_numpy()  # Corriente

            npts = len(intermediario1)  # Número de puntos en los
datos

            # Calculamos los offsets de voltaje y corriente usando
promedios de segmentos pequeños
            offset_JFET = np.mean(intermediario1[int(round(npts *
0.4)) - 1 : int(round(npts * 0.405)) - 1])  # Offset de VdsJFET
            offset_I = np.mean(intermediario2[int(round(npts * 0.005))
- 1 : int(round(npts * 0.01)) - 1])  # Offset de I

            if contador % 100 == 0:
                print("El offset de V JFET es " + str(offset_JFET))
                print("El offset de I es " + str(offset_I))

            # Aplicamos un filtro de Savitzky-Golay para suavizar la
potencia calculada.
            # La potencia es el producto de (VdsJFET - offset_JFET) y
(I - offset_I)
            filtro = savgol_filter((intermediario1 - offset_JFET) *
(intermediario2 - offset_I), 99, 3)
            #=============================================================

            #=============== BÚSQUEDA DEL ÍNDICE DE COMIENZO
================
            # Buscamos el índice en el que la curva de potencia supera
un umbral.
            # Este índice marcará el inicio del intervalo de
limitación.
            indiceminimo = npts*0.5  # Inicializamos un valor grande
que posteriormente será reemplazado.
            umbral = 50  # Establecemos el umbral que debe superar la
media de los puntos de potencia para determinar el inicio.

            for j in range(int(npts * 0.5) - 1, int(npts * 0.505) - 1,
1):
                # Comprobamos si la media de un intervalo de puntos
supera el umbral
                if np.mean(filtro[j - 10: j]) >= umbral:
                    indiceminimo = j  # Al encontrar el punto,
guardamos el índice de inicio.

                    if contador % 100 == 0:
                        # Mostramos los valores para depuración si
estamos en una iteración múltiplo de 100
                        print(filtro[j])
                        print(indiceminimo)
                        print("El índice mínimo detectado se encuentra
en el tiempo " + str(self.ejex_generado[indiceminimo +
self.indicemin]))

                    break  # Salimos del bucle una vez que encontramos
el índice mínimo.
```

```
            #=============== BÚSQUEDA DEL ÍNDICE DE FIN
================
            # Ahora buscamos el índice en el que la potencia cae por
debajo de un umbral, indicando el final de la limitación.
            indicemaximo = self.indicemax  # Inicializamos el índice
máximo al valor predeterminado.

            #Según el tipo, empezaremos a buscar antes o después
            if tipo == "2A":
                inicio_busqueda = 0.6
            else:
                inicio_busqueda = 0.62

            # Buscamos entre el 40% y el final la curva.
            for j in range(int(npts * inicio_busqueda) - 1, npts - 1,
1):
                if np.mean(filtro[j - 10: j]) <= umbral:  # Si la
media de los puntos es menor que el umbral
                    indicemaximo = j  # Actualizamos el índice máximo.

                    if contador % 100 == 0:
                        # Mostramos los valores para depuración si
estamos en una iteración múltiplo de 100
                        print(filtro[j])
                        print(indicemaximo)
                        print("El índice máximo detectado se encuentra
en el tiempo " + str(self.ejex_generado[indicemaximo +
self.indicemin]))

                    break  # Salimos del bucle una vez que encontramos
el índice máximo.

            if contador % 100 == 0:
                # Mostramos el intervalo de tiempo entre el mínimo y
el máximo encontrado
                print("El tiempo entre el máximo y el mínimo es " +
str(self.ejex_generado[indicemaximo + self.indicemin] -
self.ejex_generado[indiceminimo + self.indicemin]))

            # Actualizamos las listas de tiempo y intervalos de
energía.
            self.tiempoenergia.append(self.ejex_generado[indicemaximo
+ self.indicemin] - self.ejex_generado[indiceminimo + self.indicemin])
            self.intervaloenergia.append([indiceminimo +
self.indicemin, indicemaximo + self.indicemin])
            contador += 1

        # Al finalizar el proceso, mostramos el tamaño del vector de
tiempos calculados.
        print("El tamaño del vector de tiempo es " +
str(len(self.tiempoenergia)))


    '''
    Crearemos la función de operación binaria para obtener la energía.
Esta función multiplicará dos columnas de
    cada dataframe y calculará la integral del resultado usando el
método trapezoidal.

    El comportamiento depende del parámetro `modo`:
```

```
    - Si `modo = 0`, se calcula la integral de todo el intervalo de
datos.
    - Si `modo = 1`, se calcula la integral en intervalos previamente
calculados.

    Parámetros:
    dataframes: Lista de objetos tipo DataFrame que contienen los
datos a analizar.
    columnas: Lista con las columnas específicas a procesar en cada
DataFrame.
    modo: Parámetro que indica el tipo de cálculo de la integral (0
para todo el intervalo, 1 para intervalos específicos).
    '''

    def energia_PS(self, dataframes, columnas, modo):

        print("\n------------Analizando energía entre " + columnas[0]
+ " y " + columnas[1] +  "de curvas PS------------\n")
        # Limpiamos la lista que almacenará los resultados de la
energía calculada.
        self.listagener.clear()

        # Recorremos todos los DataFrames proporcionados.
        contador = 0
        for data in dataframes:

            # Mostramos el progreso cada 100 iteraciones para no
saturar la salida.
            if contador % 100 == 0:
                print("Cargando dataframe " + str(contador))

            # Obtenemos los datos de las dos columnas especificadas
para el rango de índices entre indicemin y indicemax.
            if columnas[0] == "VdsJFET" or columnas[0] == "VdsPMOS":
                columnaV = columnas[0]
                columnaI = columnas[1]
            else:
                columnaV = columnas[1]
                columnaI = columnas[0]


            intermediario1 =
data.loc[self.indicemin:self.indicemax][columnaV].to_numpy()  #
Columna 1 (Ej. voltaje)
            intermediario2 =
data.loc[self.indicemin:self.indicemax][columnaI].to_numpy()  #
Columna 2 (Ej. corriente)

            npts = len(intermediario1)  # Número de puntos en los
datos

            # Calculamos los offsets de cada columna usando promedios
de segmentos pequeños (al principio y al final).
            offset_1 = np.mean(intermediario1[int(round(npts * 0.4)) -
1 : int(round(npts * 0.405)) - 1])  # Offset de VdsJFET
            offset_2 = np.mean(intermediario2[int(round(npts * 0.005))
- 1 : int(round(npts * 0.01)) - 1])  # Offset de I

            # Si el modo es 0, calculamos la integral de todo el
intervalo usando la regla trapezoidal
            if modo == 0:
```

```
                # La integral se calcula como la integral de la
multiplicación de las dos señales corregidas por los offsets
                self.listagener.append(np.trapz((intermediario1 -
offset_1) * (intermediario2 - offset_2), self.ejex_generado))
            else:
                # Si el modo es 1, cargamos los intervalos de energía
previamente calculados
                # Y calculamos la integral solo en ese intervalo
específico.
                intermediario1 =
data.loc[self.intervaloenergia[contador][0]:
self.intervaloenergia[contador][1]][columnas[0]].to_numpy()
                intermediario2 =
data.loc[self.intervaloenergia[contador][0]:
self.intervaloenergia[contador][1]][columnas[1]].to_numpy()

                # Calculamos la integral en el intervalo seleccionado
usando la regla trapezoidal.
                self.listagener.append(np.trapz((intermediario1 -
offset_1) * (intermediario2 - offset_2),

self.ejex_generado[self.intervaloenergia[contador][0]:
self.intervaloenergia[contador][1] + 1]))

            # Incrementamos el contador para procesar el siguiente
DataFrame.
            contador += 1

        # Al finalizar, mostramos el tamaño del vector de resultados
generados.
        print("El tamaño del vector de datos generados es " +
str(len(self.listagener)))


    '''
    Función que calculará la corriente media durante la etapa de
limitación
    para las pruebas de PS.
    El cálculo se realiza buscando en la curva de corriente, entre los
intervalos de limitación calculados previamente.

    Parámetros:
    dataframes: lista de objetos tipo DataFrame que contienen las
curvas a analizar.
    '''

    def corriente_media_PS(self, dataframes):

        # Limpiamos la lista donde almacenaremos los valores de
corriente media.
        self.listagener.clear()

        contador = 0
        for data in dataframes:

            # ========= CARGA DE DATOS Y OPERACIONES BÁSICAS =========
            if contador % 100 == 0:  # Mostramos mensaje cada 100
iteraciones para no saturar la consola
                print("Calculando media de corriente entre " +
str(self.ejex_generado[self.intervaloenergia[contador][0]]) +
```

```
                    " ( " + str(self.intervaloenergia[contador][0])
+ " ) " + " y " +

str(self.ejex_generado[self.intervaloenergia[contador][1]])  +
                    " ( " + str(self.intervaloenergia[contador][1])
+ " ) " )

            # Obtenemos los datos de corriente en el intervalo
completo definido
            intermediario1 =
data.loc[self.indicemin:self.indicemax]["I"].to_numpy()  # Corriente

            npts = len(intermediario1)  # Número total de puntos de
datos

            # ========= CÁLCULO DEL OFFSET =========
            # Calculamos el offset (nivel base) de la corriente a
partir de un pequeño segmento al inicio del intervalo
            offset_1 = np.mean(intermediario1[int(round(npts * 0.005))
- 1 : int(round(npts * 0.01)) - 1])

            if contador % 100 == 0:
                # Se imprime el rango que se usará para calcular la
media, ajustado al 6%-94% del intervalo original
                print("Desde el mínimo " +
str(round(self.intervaloenergia[contador][0]*1.06)) +
                    " hasta el máximo " +
str(round(self.intervaloenergia[contador][1]*0.94)))

            # ========= EXTRACCIÓN DEL INTERVALO AJUSTADO =========
            # Obtenemos los datos de corriente en un subintervalo más
estrecho dentro del intervalo de limitación
            intermediario1 =
data.loc[round(self.intervaloenergia[contador][0]*1.06) :

round(self.intervaloenergia[contador][1]*0.94)]["I"].to_numpy()

            if contador % 100 == 0:
                # Imprimimos los valores inicial y final del intervalo
de corriente corregidos por el offset
                print("Para la gráfica Desde el mínimo " +
str(intermediario1[0] - offset_1) +
                    " hasta el máximo " + str(intermediario1[-1] -
offset_1))

            # ========= CÁLCULO DE LA CORRIENTE MEDIA =========
            # Aplicamos el teorema del valor medio: integral del valor
corregido dividido por la duración del intervalo
            self.listagener.append(
                np.trapz(
                    (intermediario1 - offset_1),

self.ejex_generado[round(self.intervaloenergia[contador][0]*1.06) :

round(self.intervaloenergia[contador][1]*0.94) + 1]
                ) / (

self.ejex_generado[round(self.intervaloenergia[contador][1]*0.94)] -

self.ejex_generado[round(self.intervaloenergia[contador][0]*1.06)]
                )
```

```
            )

        # Mostramos el número de resultados obtenidos
        print("El tamaño del vector de datos genéricos es " +
str(len(self.listagener)))



    '''
    Función que calculará la corriente media durante la etapa de
limitación
    para las pruebas de PS.
    El cálculo se realiza buscando en la curva de corriente, entre los
intervalos de limitación calculados previamente.

    Parámetros:
    dataframes: lista de objetos tipo DataFrame que contienen las
curvas a analizar.
    '''

    def corriente_media_SC(self, dataframes):

        # Limpiamos la lista donde almacenaremos los valores de
corriente media.
        self.listagener.clear()

        contador = 0
        for data in dataframes:

            # ========= CARGA DE DATOS Y OPERACIONES BÁSICAS =========
            if contador % 100 == 0:  # Mostramos mensaje cada 100
iteraciones para no saturar la consola
                print("Calculando media de corriente entre " +
str(self.ejex_generado[self.intervaloenergia[contador][0]]) +
                      " ( " + str(self.intervaloenergia[contador][0])
+ " ) " + " y " +

str(self.ejex_generado[self.intervaloenergia[contador][1]])  +
                      " ( " + str(self.intervaloenergia[contador][1])
+ " ) " )

            # Obtenemos los datos de corriente en el intervalo
completo definido
            intermediario1 =
data.loc[self.indicemin:self.indicemax]["I"].to_numpy()  # Corriente

            npts = len(intermediario1)  # Número total de puntos de
datos

            # ========= CÁLCULO DEL OFFSET =========
            # Calculamos el offset (nivel base) de la corriente a
partir de un pequeño segmento al inicio del intervalo
            offset_1 = np.mean(intermediario1[int(round(npts * 0.005))
- 1 : int(round(npts * 0.01)) - 1])

            if contador % 100 == 0:
                # Se imprime el rango que se usará para calcular la
media, ajustado al 6%-94% del intervalo original
                print("Desde el mínimo " +
str(round(self.intervaloenergia[contador][0]*1.23)) +
```

```
                    " hasta el máximo " +
str(round(self.intervaloenergia[contador][1]*0.77)))

            # ========= EXTRACCIÓN DEL INTERVALO AJUSTADO =========
            # Obtenemos los datos de corriente en un subintervalo más
estrecho dentro del intervalo de limitación
            intermediario1 =
data.loc[round(self.intervaloenergia[contador][0]*1.23) :

round(self.intervaloenergia[contador][1]*0.77)]["I"].to_numpy()

            if contador % 100 == 0:
                # Imprimimos los valores inicial y final del intervalo
de corriente corregidos por el offset
                print("Para la gráfica Desde el mínimo " +
str(intermediario1[0] - offset_1) +
                    " hasta el máximo " + str(intermediario1[-1] -
offset_1))

            # ========= CÁLCULO DE LA CORRIENTE MEDIA =========
            # Aplicamos el teorema del valor medio: integral del valor
corregido dividido por la duración del intervalo
            self.listagener.append(
                np.trapz(
                    (intermediario1 - offset_1),

self.ejex_generado[round(self.intervaloenergia[contador][0]*1.23) :

round(self.intervaloenergia[contador][1]*0.77) + 1]
                ) / (

self.ejex_generado[round(self.intervaloenergia[contador][1]*0.77)] -

self.ejex_generado[round(self.intervaloenergia[contador][0]*1.23)]
                )
            )

        # Mostramos el número de resultados obtenidos
        print("El tamaño del vector de datos genéricos es " +
str(len(self.listagener)))


    #%%FUNCIONES DE ANÁLISIS TEMPORALES
    '''
    Función orientada a calcular la derivada de una función temporal
utilizando la aproximación de Savitz-Golay e incorpora el resultado a
la lista interna self.listagener

    Parámetros:

    ejex: array de numpy que representta el eje x de la función

    ejey: array de numpy que representta el eje y de la función

    window_lenght: parámetro de la función de Savitz-Golay del que
depende la claridad del resultado

    polyorder: parámetro de la función de Savitz-Golay del que depende
la claridad del resultado
```

```
    '''
    def derivada( self, ejex, ejey, window_length, polyorder ):
        # Limpiamos la lista genérica que almacenará los valores
máximos procesados.
        self.listagener.clear()

        # Calcular el espaciado promedio entre puntos en X (usado como
'delta' en el filtro)
        dx = np.mean(np.diff(ejex))

        # Calcular la derivada de Y usando el filtro Savitzky-Golay
        dy = savgol_filter(ejey, window_length, polyorder, deriv=1,
delta=dx)

        self.listagener=list(dy)

        # Imprimimos la lista final con los valores máximos ajustados
y su longitud.
        print(self.listagener)
        print("El tamaño del vector de datos genéricos es " +
str(len(self.listagener)))


    #%%FUNCIONES PARA RESALTAR LOS SUBPLOTS



    #%%FUNCIONES DE BORRADO DE PLOTS Y COMPARTIR EJES

    #Función para borrar el canvas entero, se usará con la opción
grande de borrar o incluso cuando apliquemos cambios
    #al cambair el tamaño de la matriz de subplots, si esta no
coincide con la que había antes, esto se hará si se le pasa un 1 en
    #la variable selec
    def clearPlotPage(self, selec , matrixnueva, matrixantigua):

        #Solo borraremos si ya hay un canvas
        if not self.canvas:
            pass
        else:
            if selec == 1:

                if matrixnueva != matrixantigua:
                    self.fig.clear() #clear your figure
                    self.canvas.draw_idle() #redraw your canvas so it
becomes empty
                    self.ax.clear( )#Borramos todas las instancias de
subplot que teníamos
                    self.line.clear()

                    print("Canvas borrado")
                    print("Ahora el tamaño de la lista de subplots es
de " + str(len(self.ax)))
            else:
                self.fig.clear() #clear your figure
                self.canvas.draw_idle() #redraw your canvas so it
becomes empty
                self.ax.clear( )
                self.line.clear()

                print("Canvas borrado")
```

```
                print("Ahora el tamaño de la lista de subplots es de "
+ str(len(self.ax)))

        #Para liberar la memoria
        gc.collect()




    #Creamos una fuunción especializada en borrar subplots
    def clearSubplot(self, posfil,poscol):

        #recorremos nuestra lista de subplots, si alguno coincide, lo
borramos
        for indice, i in enumerate(self.ax):
            if ( i.get_subplotspec().rowspan[0] == (int(posfil) - 1) )
and ( i.get_subplotspec().colspan[0] == (int(poscol) - 1) ):
                i.cla()
                i.axis("off")
                self.ax.remove(i)
                del self.line[indice]
                print("Se ha eliminado un subplot")
                print("Ahora el tamaño de la lista de subplots es de "
+ str(len(self.ax)))
                #Para liberar la memoria
                gc.collect()

        self.canvas.draw_idle()   #redraw your canvas so the subplot
leaves

    def compartir_ejes(self,subplot,subplotmat, opcion_coord ):
        print(opcion_coord)
        print(subplot)

        #Comprobaremos que se ha seleccionado la opción de subplots,
si no, no tiene sentido coordinar las gráficas

        if subplot == "grey" and len(self.ax) > 1: #no tiene sentido
ejecutar esto si tenemos menos de 2 subplots



            #ahora comprobamos en qué opción estamos

            if opcion_coord.lower() == "no coord.":#significa que no
queremos ningún tipo de coordinación y que queremos deshacer todas las
coordinaciones de ejes existentes

                #=======================================
                #Independientemente de si estaban coordinados o no,
borraremos los existentes y crearemos copias
                print("===============")
                print("Con una matriz de subplots de ")
                print(int(subplotmat[0]))
                print(int(subplotmat[1]))

                #primero, crearemos unas listas correspondientes a los
titulos, las labels, los
```

```
                #creamos unos 10 colores bien distinguibles, que serán
los más representativos para la leyenda
                colores =
['blue','orange','green','red','purple','brown','pink','gray','olive',
'cyan']

                #creamos una lista momentánea donde guardaremos los
nuevos objetos 'ax' que serán idénticos a los anteriores
                new_ax_list = []
                new_line_list = []
                #lo que haremos será crear nuevos subplots a partir de
los que habían pero sin el sharex y sharey
                for indice,sub in enumerate(self.ax):

                    #Variable que controla si el eje x de este plot
fue desactivado
                    ejex_desactivado = False

                    #debemos comprobar si el eje x son cadenas de
caracteres
                    caracteres = False
                    labels = [tick.get_text() for tick in
sub.get_xticklabels()]
                    for label in labels:
                        if not es_numero(label):
                            caracteres = True
                            break
                    #también comprobamos si tenía leyenda
                    legend = sub.get_legend()

                    print("Para el subplot número " + str(indice))
                    #Obtenemos los datos del objeto 'ax' que vamos a
eliminar
                    title = sub.get_title()
                    xlabel = sub.get_xlabel()
                    ylabel = sub.get_ylabel()
                    # Copiar la rotación y el tamaño de fuente de ax1
                    xticklabels1 = sub.get_xticklabels()

                    # Extraer las propiedades de rotación y tamaño de
las etiquetas de ax1
                    rotacion = xticklabels1[0].get_rotation()  #
Asumimos que todas tienen la misma rotación
                    size = xticklabels1[0].get_fontsize()  # Asumimos
que todas tienen el mismo tamaño



                    #lo eliminamos
                    sub.cla()
                    sub.axis("off")
                    gc.collect()

                    print("Con No fila y No columna: ")
                    print(sub.get_subplotspec().rowspan[0] + 1)
                    print(sub.get_subplotspec().colspan[0] + 1)

                    print("Creamos un nuevo subplot de " +
subplotmat[0] + "x" + subplotmat[1] + " en la posición " + str(
int(subplotmat[1])*( sub.get_subplotspec().rowspan[0]  ) +
sub.get_subplotspec().colspan[0] + 1 ))
```

```
                    #creamos una nueva figura en la misma posición que
estaba el anterior
                    ax_new = self.fig.add_subplot(int(subplotmat[0]),
int(subplotmat[1]) ,  int(subplotmat[1])*(
sub.get_subplotspec().rowspan[0]  ) + sub.get_subplotspec().colspan[0]
+ 1  )

                    new_line_sub = []
                    print("Este subplot tiene " +
str(len(self.line[indice])) + " lineas")
                    indice_colores = 0
                    for indice_line, line in
enumerate(self.line[indice]):


                        if caracteres:
                            if labels[0] == '': #Si eran espacios es
debido a que se desactivó ell ejex, por lo tanto no hay que asignar
caracteres de nuevo
                                ejex = line.get_xdata()
                                print("Estaba desactivado el ejex")
                                ejex_desactivado = True

                            else:
                                print("No estaba desactivado el ejex")
                                ejex = labels
                                print(ejex)
                        else:
                            ejex = line.get_xdata()
                        ejey = line.get_ydata()

                        label_sub = line.get_label()



                        if indice_line < 5 or indice_line >=
(len(self.line[indice]) - 5) :
                            line, = ax_new.plot(ejex , ejey, label =
label_sub, color = colores[indice_colores])
                            indice_colores += 1
                            new_line_sub.append(line)
                        else:
                            line, = ax_new.plot(ejex , ejey, label =
label_sub)
                            new_line_sub.append(line)


                        #representamos la gráfica
                        #line, = ax_new.plot(ejex, ejey, label=
labels)

                    if ejex_desactivado:
                        print("Se ha desactivado el eje x")

                        ax_new.tick_params(axis='x',
labelbottom=False)

                    if legend:
                        if len(self.line[indice]) > 10:
                            #tan solo querremos mostrar los primeros 5
y los últimos 5 curvas en la leyenda para evitar que ocupe mucho
```

```
                                lines_to_show = new_line_sub[:5] +
new_line_sub[-5:]

                                #ahora encontraremos las labels de las
primeras 5 y las últimas 5
                                labels_to_show = [line.get_label() for
line in lines_to_show]


                                if subplot == "grey":#si tenemos subplots
querremos que el tamaño no siga la misma regla

ax_new.legend(lines_to_show,labels_to_show, loc = "upper left", prop =
{ "size": 12/( int( subplotmat[1] ) *  int(subplotmat[0] ) ) } )

                                else:

ax_new.legend(lines_to_show,labels_to_show, loc = "upper left", prop =
{ "size": 5 } )
                            else:
                                if subplot == "grey":#si tenemos subplots
querremos que el tamaño no siga la misma regla
                                    ax_new.legend(loc = "upper left", prop
= { "size": 12/( int( subplotmat[1] ) *  int(subplotmat[0] ) ) } )

                                else:
                                    ax_new.legend(loc = "upper left", prop
= { "size": 5 } )

                        ax_new.grid(True)

                        #configuramos la inclinación y el tamaño de letra
                        for label in ax_new.get_xticklabels():
                            label.set_rotation(rotacion)
                            label.set_fontsize(size)

                        # Copiar otros elementos como título, etiquetas, y
límites
                        ax_new.set_title(title)
                        ax_new.set_xlabel(xlabel)
                        ax_new.set_ylabel(ylabel)


                        new_ax_list.append(ax_new)
                        new_line_list.append(new_line_sub)

                    self.ax = new_ax_list
                    self.line = new_line_list

                    #Para liberar la memoria
                    gc.collect()
                    self.toolbar.update()
                    self.canvas.draw_idle()    #redraw your canvas

                    print("====================")
                    print("Se han eliminado todas las posibles
dependencias de ejes")
                    #================================


                elif opcion_coord.lower() == "coord. col. x-y":
```

```
                print("se han hecho interdependientes los ejes de las
columnas")
                #En primer lugar debemos comprobar que la matriz de
subplots tiene más de 1 fila
                if int(subplotmat[0]) <= 1:
                    print("No hay más de una fila")
                    return
                else:

                    #=======================================
                    #Independientemente de si estaban coordinados o
no, borraremos los existentes y crearemos copias
                    print("===============")
                    print("Con una matriz de subplots de ")
                    print(int(subplotmat[0]))
                    print(int(subplotmat[1]))

                    #primero, crearemos unas listas correspondientes a
los titulos, las labels, los

                    #creamos unos 10 colores bien distinguibles, que
serán los más representativos para la leyenda
                    colores =
['blue','orange','green','red','purple','brown','pink','gray','olive',
'cyan']

                    #creamos una lista momentánea donde guardaremos
los nuevos objetos 'ax' que serán idénticos a los anteriores
                    new_ax_list = []
                    new_line_list = []
                    #lo que haremos será crear nuevos subplots a
partir de los que habían pero sin el sharex y sharey
                    for indice,sub in enumerate(self.ax):

                        #Variable que controla si el eje x de este
plot fue desactivado
                        ejex_desactivado = False

                        #debemos comprobar si el eje x son cadenas de
caracteres
                        caracteres = False
                        labels = [tick.get_text() for tick in
sub.get_xticklabels()]
                        for label in labels:
                            if not es_numero(label):
                                caracteres = True
                                break
                        #también comprobamos si tenía leyenda
                        legend = sub.get_legend()

                        print("Para el subplot número " + str(indice))
                        #Obtenemos los datos del objeto 'ax' que vamos
a eliminar
                        title = sub.get_title()
                        xlabel = sub.get_xlabel()
                        ylabel = sub.get_ylabel()
                        # Copiar la rotación y el tamaño de fuente de
ax1
                        xticklabels1 = sub.get_xticklabels()
```

```
                        # Extraer las propiedades de rotación y tamaño
de las etiquetas de ax1
                        rotacion = xticklabels1[0].get_rotation()  #
Asumimos que todas tienen la misma rotación
                        size = xticklabels1[0].get_fontsize()  #
Asumimos que todas tienen el mismo tamaño



                        #lo eliminamos
                        sub.cla()
                        sub.axis("off")
                        gc.collect()

                        print("Con No fila y No columna: ")
                        print(sub.get_subplotspec().rowspan[0] + 1)
                        print(sub.get_subplotspec().colspan[0] + 1)

                        print("Creamos un nuevo subplot de " +
subplotmat[0] + "x" + subplotmat[1] + " en la posición " + str(
int(subplotmat[1])*( sub.get_subplotspec().rowspan[0]  ) +
sub.get_subplotspec().colspan[0] + 1 ))
                        #creamos una nueva figura en la misma posición
que estaba el anterior
                        ax_new =
self.fig.add_subplot(int(subplotmat[0]), int(subplotmat[1]) ,
int(subplotmat[1])*( sub.get_subplotspec().rowspan[0]  ) +
sub.get_subplotspec().colspan[0] + 1  )

                        new_line_sub = []
                        print("Este subplot tiene " +
str(len(self.line[indice])) + " lineas")
                        indice_colores = 0
                        for indice_line, line in
enumerate(self.line[indice]):


                            if caracteres:
                                if labels[0] == '': #Si eran espacios
es debido a que se desactivó ell ejex, por lo tanto no hay que asignar
caracteres de nuevo
                                    ejex = line.get_xdata()
                                    print("Estaba desactivado el ejex")
                                    ejex_desactivado = True

                                else:
                                    print("No estaba desactivado el
ejex")
                                    ejex = labels
                                    print(ejex)
                            else:
                                ejex = line.get_xdata()
                            ejey = line.get_ydata()

                            label_sub = line.get_label()



                            if indice_line < 5 or indice_line >=
(len(self.line[indice]) - 5) :
```

```
                                line, = ax_new.plot(ejex , ejey, label
= label_sub, color = colores[indice_colores])
                                indice_colores += 1
                                new_line_sub.append(line)
                            else:
                                line, = ax_new.plot(ejex , ejey, label
= label_sub)
                                new_line_sub.append(line)


                            #representamos la gráfica
                            #line, = ax_new.plot(ejex, ejey, label=
labels)

                        if ejex_desactivado:
                            print("Se ha desactivado el eje x")

                            ax_new.tick_params(axis='x',
labelbottom=False)

                        if legend:
                            if len(self.line[indice]) > 10:
                                #tan solo querremos mostrar los
primeros 5 y los últimos 5 curvas en la leyenda para evitar que ocupe
mucho
                                lines_to_show = new_line_sub[:5] +
new_line_sub[-5:]

                                #ahora encontraremos las labels de las
primeras 5 y las últimas 5
                                labels_to_show = [line.get_label() for
line in lines_to_show]


                                if subplot == "grey":#si tenemos
subplots querremos que el tamaño no siga la misma regla

ax_new.legend(lines_to_show,labels_to_show, loc = "upper left", prop =
{ "size": 12/( int( subplotmat[1] ) *  int(subplotmat[0] ) ) } )

                                else:

ax_new.legend(lines_to_show,labels_to_show, loc = "upper left", prop =
{ "size": 5 } )
                            else:
                                if subplot == "grey":#si tenemos
subplots querremos que el tamaño no siga la misma regla
                                    ax_new.legend(loc = "upper left",
prop = { "size": 12/( int( subplotmat[1] ) *  int(subplotmat[0] ) ) }
)

                                else:
                                    ax_new.legend(loc = "upper left",
prop = { "size": 5 } )

                        ax_new.grid(True)

                        #configuramos la inclinación y el tamaño de
letra
                        for label in ax_new.get_xticklabels():
                            label.set_rotation(rotacion)
```

```
                            label.set_fontsize(size)

                        # Copiar otros elementos como título,
etiquetas, y límites
                        ax_new.set_title(title)
                        ax_new.set_xlabel(xlabel)
                        ax_new.set_ylabel(ylabel)


                        new_ax_list.append(ax_new)
                        new_line_list.append(new_line_sub)

                    self.ax = new_ax_list
                    self.line = new_line_list

                    #Para liberar la memoria
                    gc.collect()
                    self.toolbar.update()
                    self.canvas.draw_idle()   #redraw your canvas

                    print("====================")
                    print("Se han eliminado todas las posibles
dependencias de ejes")
                    #================================

                    #ahora que sabemos que hay más de una fila en la
matriz de subplots, recorremos
                    #los objetos 'ax' para comprobar si hay varios en
una misma columna

                    #crearemos una lista, , cada elemento
corresponderá
                    #con el índice del objeto 'ax' y con su columna,
de esta forma, podremos saber
                    #recursívamente si algunos comparten columna y
compartir los ejes

                    columnas = []


                    for indice, sub in enumerate(self.ax):

                        columna = sub.get_subplotspec().colspan[0]

                        saltar = False
                        #comprobamos si alguna gráfica contiene
palabras en lugar de números en su eje x, y si es así, no se comparten
                        labels = [tick.get_text() for tick in
sub.get_xticklabels()]
                        for label in labels:
                            if not es_numero(label):
                                messagebox.showerror("Error al
compartir ejes", "Uno de los gráficos no contiene números en el eje x,
su eje no se compartirá" )
                                saltar = True
                                break
                        if saltar:
                            continue

                        if columna in columnas: #si ya había otros
plots en esta columna
```

```
                            indice_anterior = columnas.index(columna)
#obtenemos el indice del primer elemento de esta columna
                            sub.sharex(self.ax[indice_anterior])
                            sub.sharey(self.ax[indice_anterior])

                        columnas.append(columna)

            elif opcion_coord.lower() == "coord. col. x":
                print("se han hecho interdependientes los ejes de las
columnas")
                #En primer lugar debemos comprobar que la matriz de
subplots tiene más de 1 fila
                if int(subplotmat[0]) <= 1:
                    print("No hay más de una fila")
                    return
                else:

                    #=======================================
                    #Independientemente de si estaban coordinados o
no, borraremos los existentes y crearemos copias
                    print("===============")
                    print("Con una matriz de subplots de ")
                    print(int(subplotmat[0]))
                    print(int(subplotmat[1]))

                    #primero, crearemos unas listas correspondientes a
los titulos, las labels, los

                    #creamos unos 10 colores bien distinguibles, que
serán los más representativos para la leyenda
                    colores =
['blue','orange','green','red','purple','brown','pink','gray','olive',
'cyan']

                    #creamos una lista momentánea donde guardaremos
los nuevos objetos 'ax' que serán idénticos a los anteriores
                    new_ax_list = []
                    new_line_list = []
                    #lo que haremos será crear nuevos subplots a
partir de los que habían pero sin el sharex y sharey
                    for indice,sub in enumerate(self.ax):

                        #Variable que controla si el eje x de este
plot fue desactivado
                        ejex_desactivado = False

                        #debemos comprobar si el eje x son cadenas de
caracteres
                        caracteres = False
                        labels = [tick.get_text() for tick in
sub.get_xticklabels()]
                        for label in labels:
                            if not es_numero(label):
                                caracteres = True
                                break
                        #también comprobamos si tenía leyenda
                        legend = sub.get_legend()

                        print("Para el subplot número " + str(indice))
                        #Obtenemos los datos del objeto 'ax' que vamos
a eliminar
```

```
                        title = sub.get_title()
                        xlabel = sub.get_xlabel()
                        ylabel = sub.get_ylabel()
                        # Copiar la rotación y el tamaño de fuente de
ax1
                        xticklabels1 = sub.get_xticklabels()

                        # Extraer las propiedades de rotación y tamaño
de las etiquetas de ax1
                        rotacion = xticklabels1[0].get_rotation()  #
Asumimos que todas tienen la misma rotación
                        size = xticklabels1[0].get_fontsize()  #
Asumimos que todas tienen el mismo tamaño



                        #lo eliminamos
                        sub.cla()
                        sub.axis("off")
                        gc.collect()

                        print("Con No fila y No columna: ")
                        print(sub.get_subplotspec().rowspan[0] + 1)
                        print(sub.get_subplotspec().colspan[0] + 1)

                        print("Creamos un nuevo subplot de " +
subplotmat[0] + "x" + subplotmat[1] + " en la posición " + str(
int(subplotmat[1])*( sub.get_subplotspec().rowspan[0]  ) +
sub.get_subplotspec().colspan[0] + 1 ))
                        #creamos una nueva figura en la misma posición
que estaba el anterior
                        ax_new =
self.fig.add_subplot(int(subplotmat[0]), int(subplotmat[1]) ,
int(subplotmat[1])*( sub.get_subplotspec().rowspan[0]  ) +
sub.get_subplotspec().colspan[0] + 1  )

                        new_line_sub = []
                        print("Este subplot tiene " +
str(len(self.line[indice])) + " lineas")
                        indice_colores = 0
                        for indice_line, line in
enumerate(self.line[indice]):


                            if caracteres:
                                if labels[0] == '': #Si eran espacios
es debido a que se desactivó ell ejex, por lo tanto no hay que asignar
caracteres de nuevo
                                    ejex = line.get_xdata()
                                    print("Estaba desactivado el ejex")
                                    ejex_desactivado = True

                                else:
                                    print("No estaba desactivado el
ejex")

                                    ejex = labels
                                    print(ejex)
                            else:
                                ejex = line.get_xdata()
                            ejey = line.get_ydata()
```

```
                            label_sub = line.get_label()


                            if indice_line < 5 or indice_line >=
(len(self.line[indice]) - 5) :
                                line, = ax_new.plot(ejex , ejey, label
= label_sub, color = colores[indice_colores])
                                indice_colores += 1
                                new_line_sub.append(line)
                            else:
                                line, = ax_new.plot(ejex , ejey, label
= label_sub)
                                new_line_sub.append(line)


                            #representamos la gráfica
                            #line, = ax_new.plot(ejex, ejey, label=
labels)

                        if ejex_desactivado:
                            print("Se ha desactivado el eje x")

                            ax_new.tick_params(axis='x',
labelbottom=False)

                        if legend:
                            if len(self.line[indice]) > 10:
                                #tan solo querremos mostrar los
primeros 5 y los últimos 5 curvas en la leyenda para evitar que ocupe
mucho
                                lines_to_show = new_line_sub[:5] +
new_line_sub[-5:]

                                #ahora encontraremos las labels de las
primeras 5 y las últimas 5
                                labels_to_show = [line.get_label() for
line in lines_to_show]


                                if subplot == "grey":#si tenemos
subplots querremos que el tamaño no siga la misma regla

ax_new.legend(lines_to_show,labels_to_show, loc = "upper left", prop =
{ "size": 12/( int( subplotmat[1] ) *  int(subplotmat[0] ) ) } )

                                else:

ax_new.legend(lines_to_show,labels_to_show, loc = "upper left", prop =
{ "size": 5 } )
                            else:
                                if subplot == "grey":#si tenemos
subplots querremos que el tamaño no siga la misma regla
                                    ax_new.legend(loc = "upper left",
prop = { "size": 12/( int( subplotmat[1] ) *  int(subplotmat[0] ) ) }
)

                                else:
                                    ax_new.legend(loc = "upper left",
prop = { "size": 5 } )
```

```
                        ax_new.grid(True)

                        #configuramos la inclinación y el tamaño de
letra
                        for label in ax_new.get_xticklabels():
                            label.set_rotation(rotacion)
                            label.set_fontsize(size)

                        # Copiar otros elementos como título,
etiquetas, y límites
                        ax_new.set_title(title)
                        ax_new.set_xlabel(xlabel)
                        ax_new.set_ylabel(ylabel)


                        new_ax_list.append(ax_new)
                        new_line_list.append(new_line_sub)

                    self.ax = new_ax_list
                    self.line = new_line_list

                    #Para liberar la memoria
                    gc.collect()
                    self.toolbar.update()
                    self.canvas.draw_idle()   #redraw your canvas

                    print("====================")
                    print("Se han eliminado todas las posibles
dependencias de ejes")
                    #================================

                    #ahora que sabemos que hay más de una fila en la
matriz de subplots, recorremos
                    #los objetos 'ax' para comprobar si hay varios en
una misma columna

                    #crearemos una lista, , cada elemento
corresponderá
                    #con el índice del objeto 'ax' y con su columna,
de esta forma, podremos saber
                    #recursívamente si algunos comparten columna y
compartir los ejes

                    columnas = []


                    for indice, sub in enumerate(self.ax):

                        columna = sub.get_subplotspec().colspan[0]

                        saltar = False
                        #comprobamos si alguna gráfica contiene
palabras en lugar de números en su eje x, y si es así, no se comparten
                        labels = [tick.get_text() for tick in
sub.get_xticklabels()]
                        for label in labels:
                            if not es_numero(label) and labels[0] !=
'':
                                messagebox.showerror("Error al
compartir ejes", "Uno de los gráficos no contiene números en el eje x,
su eje no se compartirá" )
```

```
                                saltar = True
                                break
                        if saltar:
                            continue

                        if columna in columnas: #si ya había otros
plots en esta columna
                            indice_anterior = columnas.index(columna)
#obtenemos el indice del primer elemento de esta columna
                            sub.sharex(self.ax[indice_anterior])


                        columnas.append(columna)


            elif opcion_coord.lower() == "coord. all x-y":

                #comprobamos si la primera gráfica contiene palabras
en lugar de números en su eje x, y si es así, no se comparten
                labels = [tick.get_text() for tick in
self.ax[0].get_xticklabels()]
                for label in labels:
                    if not es_numero(label):
                        messagebox.showerror("Error al compartir
ejes", "El primer gráfico (usado como referencia) no contiene números
en el eje x" )
                        return

                #Recorremos los objetos 'ax' guardados, y hacemos que
todos guarden referencia a los ejes del primero
                for sub in self.ax[1:]:

                    saltar = False
                    #comprobamos si alguna gráfica contiene palabras
en lugar de números en su eje x, y si es así, no se comparten
                    labels = [tick.get_text() for tick in
sub.get_xticklabels()]
                    for label in labels:
                        if not es_numero(label):
                            messagebox.showerror("Error al compartir
ejes", "Uno de los gráficos no contiene números en el eje x, su eje no
se compartirá" )
                            saltar = True
                            break

                    if saltar:
                        continue


                    sub.sharex(self.ax[0])
                    sub.sharey(self.ax[0])


                self.canvas.draw_idle()   #redraw your canvas
                print("se han hecho interdependientes los ejes de
todos los subplots")

            elif opcion_coord.lower() == "coord. all x":

                #comprobamos si la primera gráfica contiene palabras
en lugar de números en su eje x, y si es así, no se comparten
```

```
                labels = [tick.get_text() for tick in
self.ax[0].get_xticklabels()]
                for label in labels:
                    if not es_numero(label):
                        messagebox.showerror("Error al compartir
ejes", "El primer gráfico (usado como referencia) no contiene números
en el eje x" )
                        return

                #Recorremos los objetos 'ax' guardados, y hacemos que
todos guarden referencia a los ejes del primero
                for sub in self.ax[1:]:

                    saltar = False
                    #comprobamos si alguna gráfica contiene palabras
en lugar de números en su eje x, y si es así, no se comparten
                    labels = [tick.get_text() for tick in
sub.get_xticklabels()]
                    for label in labels:
                        if not es_numero(label):
                            messagebox.showerror("Error al compartir
ejes", "Uno de los gráficos no contiene números en el eje x, su eje no
se compartirá" )
                            saltar = True
                            break

                    if saltar:
                        continue


                    sub.sharex(self.ax[0])


                self.canvas.draw_idle()   #redraw your canvas
                print("se han hecho interdependientes los ejes de
todos los subplots")

        else:
            print("No se pueden compartir ejes aún ya que hay una
cantidad de " + str(len(self.ax)) + " subplots")
```

5. BIBLIOGRAFÍA

[1] "The European Green Deal," https://commission.europa.eu/strategy-and-policy/priorities-2019-2024/european-green-deal_en.

[2] "SUPPORTING THE GREEN TRANSITION," 2020, doi: 10.2775/98379.

[3] ESA, "ESA'S TECHNOLOGY STRATEGY," 2022.

[4] R. Rodrigues, Y. Du, A. Antoniazzi, and P. Cairoli, "A Review of Solid-State Circuit Breakers," *IEEE Trans Power Electron*, vol. 36, no. 1, pp. 364–377, Jan. 2021, doi: 10.1109/TPEL.2020.3003358.

[5] S. Beheshtaein, R. M. Cuzner, M. Forouzesh, M. Savaghebi, and J. M. Guerrero, "DC Microgrid Protection: A Comprehensive Review," *IEEE J Emerg Sel Top Power Electron*, 2024, doi: 10.1109/JESTPE.2019.2904588.

[6] D. Izquierdo, A. Barrado, C. Fernández, M. Sanz, and A. Lázaro, "SSPC active control strategy by optimal trajectory of the current for onboard system applications," *IEEE Transactions on Industrial Electronics*, vol. 60, no. 11, pp. 5195–5205, 2013, doi: 10.1109/TIE.2012.2219832.

[7] Z. Huang *et al.*, "Development of High-Current Solid-State Power Controllers for Aircraft High-Voltage DC Network Applications," *IEEE Access*, vol. 9, pp. 105048–105059, 2021, doi: 10.1109/ACCESS.2021.3099257.

[8] Z. Miao, G. Sabui, A. M. Roshandeh, and Z. J. Shen, "Design and Analysis of DC Solid-State Circuit Breakers Using SiC JFETs," *IEEE J Emerg Sel Top Power Electron*, vol. 4, no. 3, pp. 863–873, Sep. 2016, doi: 10.1109/JESTPE.2016.2558448.

[9] R. Kheirollahi, H. Zhang, S. Zhao, and F. Lu, "A DC Solid-State Circuit Breaker Based on Transient Current Commutation," *IEEE J Emerg Sel Top Power Electron*, vol. 10, no. 4, pp. 4614–4625, Aug. 2022, doi: 10.1109/JESTPE.2021.3116605.

[10] D. Marroquí, J. M. Blanes, A. Garrigós, and R. Gutiérrez, "Self-Powered 380 v DC SiC Solid-State Circuit Breaker and Fault Current Limiter," *IEEE Trans Power Electron*, vol. 34, no. 10, pp. 9600–9608, Oct. 2019, doi: 10.1109/TPEL.2019.2893104.

[11] "Space engineering Electrical design and interface requirements for power supply," 2016.

[12] J. M. Lauenstein, M. C. Casey, R. L. Ladbury, H. S. Kim, A. M. Phan, and A. D. Topper, "Space Radiation Effects on SiC Power Device Reliability," in *IEEE International Reliability Physics Symposium Proceedings*, Institute of Electrical and Electronics Engineers Inc., Mar. 2021. doi: 10.1109/IRPS46558.2021.9405180.

[13] Toshiba, *Reliability Handbook*. 2018.

[14] "Space product assurance Derating-EEE components ECSS Secretariat ESA-ESTEC Requirements & Standards Division Noordwijk, The Netherlands," 2011.

[15] JEDEC, *22A108G*. 2022.

[16] D. Marroquí, "Diseño e implementación de controladores de potencia de estado sólido SiC para aplicaciones DC," 2020.

[17] A. Ortiz-Conde, F. J. García-Sánchez, J. Muci, A. Terán Barrios, J. J. Liou, and C. S. Ho, "Revisiting MOSFET threshold voltage extraction methods," *Microelectronics Reliability*, vol. 53, no. 1, pp. 90–104, Jan. 2013, doi: 10.1016/j.microrel.2012.09.015.

[18] D. Harvey, “INTRUMENTAL ANALYSIS.” [Online]. Available: https://LibreTexts.org
[19] *IIPhDW : 2018 International Interdisciplinary PhD Workshop : 9-12 May 2018, Swinoujscie, Poland*. Institute of Electrical and Electronics Engineers, 2018.
[20] L. Cited ; Cohen, H. Bates, J. Hiittenrauch, and R. Pitter, “Smoothing and Differentiation of Data by Simplified Least Squares Pro,cedures,” 1951.
[21] I. Williams and J. Bridgmon, “Offset Error TI Precision Labs-Current Sense Amplifiers.”
[22] “MT-095 TUTORIAL EMI, RFI, and Shielding Concepts.”
[23] A. Kakaraparthy and J. M. Patel, “SpIitDF: Splitting Dataframes for Memory-Efficient Data Analysis,” in *Proceedings of the VLDB Endowment*, VLDB Endowment, 2024, pp. 2175–2184. doi: 10.14778/3665844.3665849.
[24] H. Yang and H. Inokawa, “A differential smoothing technique for the extraction of MOSFET threshold voltage using extrapolation in the linear region,” *Solid State Electron*, vol. 76, pp. 5–7, Oct. 2012, doi: 10.1016/j.sse.2012.05.065.
[25] I. Turner-Trauring, “Don’t bother trying to estimate Pandas memory usage,” https://pythonspeed.com/articles/estimating-pandas-memory/.