Sistema de Control de Oscilaciones

versión 1.0, 7 de febrero de 2026

Para uso con Python

Guía de usuario

PhD. Jimy Alexander Cortés Osorio

M. Sc Edward Andrés González Ríos

Lina Marcela Rodríguez Soto

María Camila Quintero Sánchez

Héctor Fabio Quesada Pascuas

Hugo Sebastián Gutiérrez Muñoz

Universidad Tecnológica de Pereira
2026

Índice

Índice de figuras

1. Introducción

El presente manual de usuario describe el uso y funcionamiento de una aplicación desarrollada en Python para el control, monitoreo y adquisición de datos de un prototipo experimental generador de oscilaciones mecánicas. La aplicación proporciona una interfaz gráfica que permite al usuario interactuar de manera directa con el sistema, facilitando la configuración de parámetros, la visualización de variables dinámicas y la gestión de datos experimentales.

La necesidad de esta aplicación surge del uso de un prototipo mecánico previamente desarrollado, basado en un mecanismo de yugo escocés accionado por un motor eléctrico, el cual requiere un control preciso de su velocidad de operación y una adquisición confiable de las variables asociadas al movimiento. En este contexto, la interfaz gráfica actúa como un medio unificado para establecer la comunicación con los dispositivos electrónicos del sistema y coordinar su operación.

El manual está orientado a usuarios con conocimientos básicos en programación, instrumentación y control, y tiene como objetivo servir como guía práctica para la correcta utilización de la aplicación, describiendo su estructura general, sus principales funcionalidades y el flujo de operación esperado durante la ejecución de experimentos.

1.1. Generalidades

La aplicación fue desarrollada en Python utilizando la biblioteca PyQt5, lo que permite su ejecución en distintos sistemas operativos y su integración con herramientas comunes de desarrollo y análisis. El software está diseñado como una solución centralizada que gestiona la comunicación serial con los dispositivos electrónicos del prototipo, separando de forma lógica las tareas de control y telemetría.

A través de la interfaz gráfica, el usuario puede establecer conexiones independientes con los módulos encargados del control del motor y de la medición de variables cinemáticas, configurar parámetros relevantes del sistema, y visualizar en tiempo real los datos adquiridos. Adicionalmente, la aplicación incorpora herramientas para el registro y exportación de datos, así como para la adquisición y organización de imágenes asociadas a las condiciones experimentales.

La estructura modular de la aplicación permite su uso tanto en tareas de ajuste y depuración como en la ejecución sistemática de experimentos, sin requerir modificaciones en el código fuente para la operación básica del sistema.

1.2. Ejecución de la aplicación

La aplicación OSCOS puede ejecutarse de dos formas distintas, dependiendo del tipo de uso previsto. Para usuarios finales, se recomienda el uso de los ejecutables pre-compilados, los cuales no requieren la instalación manual de dependencias. Para desarrolladores o usuarios avanzados que deseen modificar o depurar el código fuente, es posible ejecutar la aplicación directamente desde Python, lo cual requiere una instalación previa de las dependencias.

1.2.1. Opción 1: Ejecutables pre-compilados

La forma más simple de ejecutar la aplicación es mediante los ejecutables pre-compilados incluidos en el proyecto. Esta opción no requiere la instalación de Python ni de dependencias adicionales, ya que todos los componentes necesarios se encuentran dentro de los archivos de la aplicación.

En Linux:

Navegar a la carpeta `bin/linux/` y ejecutar:

```
./OSCOS
```

Alternativamente, ejecutar desde la raíz del proyecto:

```
./bin/linux/OSCOS
```

En Windows:

Navegar a la carpeta `bin\windows\` y ejecutar el archivo `OSCOS.exe`. Alternativamente, desde una terminal:

```
.\bin\windows\OSCOS.exe
```

1.2.2. Opción 2: Ejecución desde código fuente

La aplicación también puede ejecutarse directamente desde el código fuente en Python. Esta opción está orientada a desarrolladores que deseen modificar el software, realizar tareas de depuración o experimentar con nuevas funcionalidades.

Para esta modalidad es obligatorio realizar previamente la instalación de las dependencias del proyecto, como se describe en la siguiente sección.

Con el entorno virtual activado, ejecutar desde la carpeta raíz del proyecto:

```
python src/oscos/main.py
```

1.3. Instalación de dependencias

Para ejecutar la aplicación desde el código fuente, es necesario instalar las dependencias requeridas. Se recomienda encarecidamente utilizar un entorno virtual de Python para mantener aisladas las dependencias del proyecto.

Paso 1: Creación de un entorno virtual

En Linux/macOS:

```
python3 -m venv venv
source venv/bin/activate
```

En Windows:

```
python -m venv venv
venv\Scripts\activate
```

Paso 2: Instalación de dependencias

Con el entorno virtual activado, instalar las dependencias ejecutando:

```
pip install -r requirements.txt
```

El archivo `requirements.txt` contiene todas las dependencias externas necesarias para ejecutar o modificar la aplicación desde el código fuente:

```
numpy<2.3.0                          # Cálculos numéricos
opencv_python_headless==4.12.0.88    # Procesamiento de imágenes
Pillow==12.1.0                       # Manejo de imágenes
pypylon==4.2.0                       # SDK Basler para cámaras
PyQt5==5.15.11                       # Framework GUI
pyqt5_sip==12.17.2                   # Componente PyQt5
pyqtgraph==0.14.0                    # Gráficos interactivos
pyserial==3.5                        # Comunicación serial
```

Esta instalación descargará e instalará automáticamente todas las librerías necesarias en el entorno virtual.

1.4. Recomendaciones de uso

- Se recomienda utilizar los ejecutables pre-compilados para la operación normal del sistema, evitando configuraciones innecesarias.
- Para modificaciones del código fuente, utilizar siempre un entorno virtual de Python.
- Después de modificar el código fuente, reiniciar completamente la aplicación para asegurar que los cambios se apliquen correctamente.
- Al ejecutar desde código fuente, verificar que el entorno virtual se encuentre activado antes de iniciar la aplicación.
- Ante problemas con dependencias, se recomienda eliminar la carpeta `venv/` y repetir el proceso de instalación desde cero.

1.5. Compilación de ejecutables con PyInstaller

En caso de que en el futuro se deseen reconstruir los ejecutables tras aplicar cambios al código fuente, la construcción de los binarios se realiza utilizando **PyInstaller** y el archivo de especificación proporcionado `OSCOS.spec` ubicado en la raíz del proyecto.

Pasos recomendados:

1. Activar el entorno virtual (si corresponde) e instalar PyInstaller:

```
pip install pyinstaller
```

2. Ejecutar PyInstaller sobre el spec file desde la raíz del proyecto:

   ```
   pyinstaller OSCOS.spec
   ```

   Este comando generará las carpetas `build/` y `dist/`. Los ejecutables finales quedarán dentro de `dist/` (por ejemplo, `dist/OSCOS/` o `dist/OSCOS.exe` según la plataforma).

3. Probar el ejecutable resultante en la plataforma objetivo. PyInstaller no realiza compilación cruzada completa. Es necesario generar los binarios en cada sistema operativo destino (Linux en Linux, Windows en Windows).

2. Interfaz gráfica de usuario

2.1. Descripción general de la aplicación

La aplicación proporciona una interfaz gráfica desarrollada en Python que centraliza el control, la adquisición de datos y la gestión de imágenes asociadas a un sistema experimental de oscilaciones mecánicas. A través de esta interfaz, el usuario puede establecer la comunicación con los dispositivos electrónicos del sistema, configurar los parámetros de operación, visualizar variables dinámicas en tiempo real y coordinar la adquisición de imágenes junto con sus metadatos.

La interfaz está organizada en tres pestañas principales, cada una enfocada en una etapa específica del flujo experimental: la configuración de las comunicaciones, el control y monitoreo del sistema, y la adquisición de imágenes. Esta organización permite una operación estructurada y reduce la probabilidad de errores durante la ejecución de los experimentos.

El diseño de la aplicación prioriza la claridad en la disposición de los elementos interactivos, así como la separación funcional entre las tareas de control, telemetría y adquisición, facilitando tanto el uso operativo como las tareas de depuración y ajuste del sistema.

La Figura 1 muestra una vista general de la interfaz gráfica de la aplicación, resaltando los elementos de navegación principales.

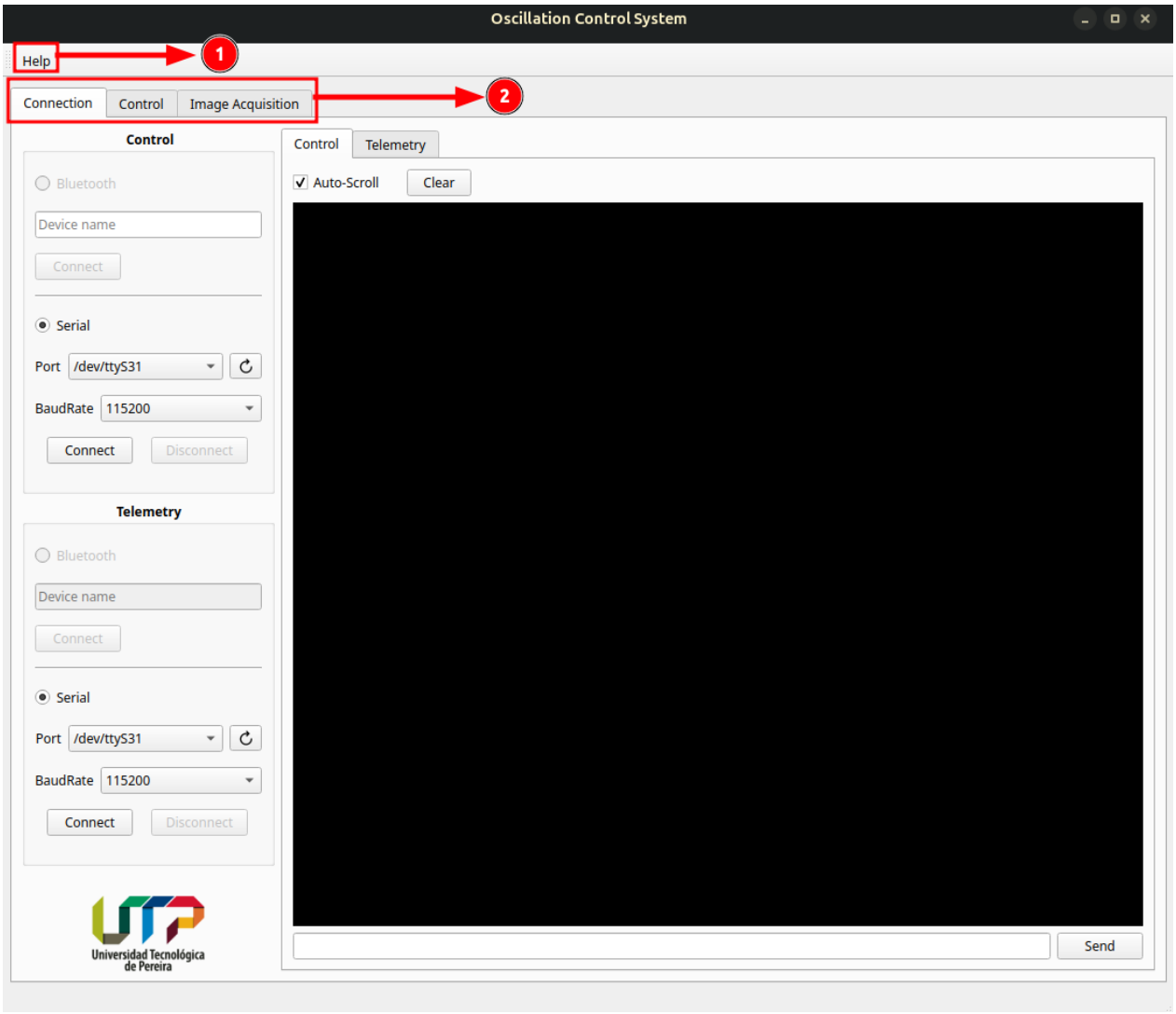

Figura 1: Interfaz general de la aplicación OSCOS.

En la Figura 1 se destacan los siguientes elementos:

1. **Botón de ayuda:** Permite acceder a un menú de ayuda integrado que describe las principales funcionalidades de la aplicación. Este menú se encuentra disponible tanto en idioma español como en inglés y está orientado a facilitar el uso general del sistema.

2. **Pestañas principales de la aplicación:** Corresponden a las tres secciones fundamentales del flujo de operación: *Connection*, *Control* e *Image Acquisition*, las cuales se describen en detalle en las secciones siguientes.

2.2. Flujo de uso previsto de la aplicación

La aplicación fue diseñada para ser utilizada siguiendo un orden de operación específico, el cual garantiza el correcto funcionamiento del sistema y la coherencia de los datos adquiridos.

En primer lugar, es necesario establecer las conexiones de comunicación con los dispositivos electrónicos del sistema mediante la pestaña *Connection*. Esta etapa es obligatoria antes de acceder a las funcionalidades de las demás pestañas, ya que tanto el control del sistema como la adquisición de datos dependen de una comunicación activa con los módulos de control y telemetría.

Una vez establecidas las conexiones, el usuario puede acceder a la pestaña *Control* para configurar los parámetros del sistema, iniciar o detener el movimiento y monitorear en tiempo real las variables dinámicas medidas. Posteriormente, y con el sistema operando bajo las condiciones deseadas, es posible utilizar la pestaña *Image Acquisition* para capturar imágenes sincronizadas con los parámetros de oscilación.

Finalmente, la aplicación permite exportar los datos adquiridos y cerrar de manera segura las conexiones antes de finalizar la ejecución del software. Este flujo de operación recomendado reduce errores de comunicación y asegura la correcta asociación entre los parámetros configurados, los datos adquiridos y las imágenes capturadas.

2.3. Pestaña Connection

La pestaña *Connection* es el punto de inicio de la aplicación y está destinada a la configuración y gestión de las comunicaciones entre el computador y los dispositivos electrónicos del sistema. Desde esta pestaña se establecen conexiones independientes para el módulo de control del motor y para el módulo de telemetría encargado de la adquisición de datos.

La Figura 2 muestra la interfaz correspondiente a esta pestaña, donde se han numerado los distintos elementos interactivos para facilitar su descripción.

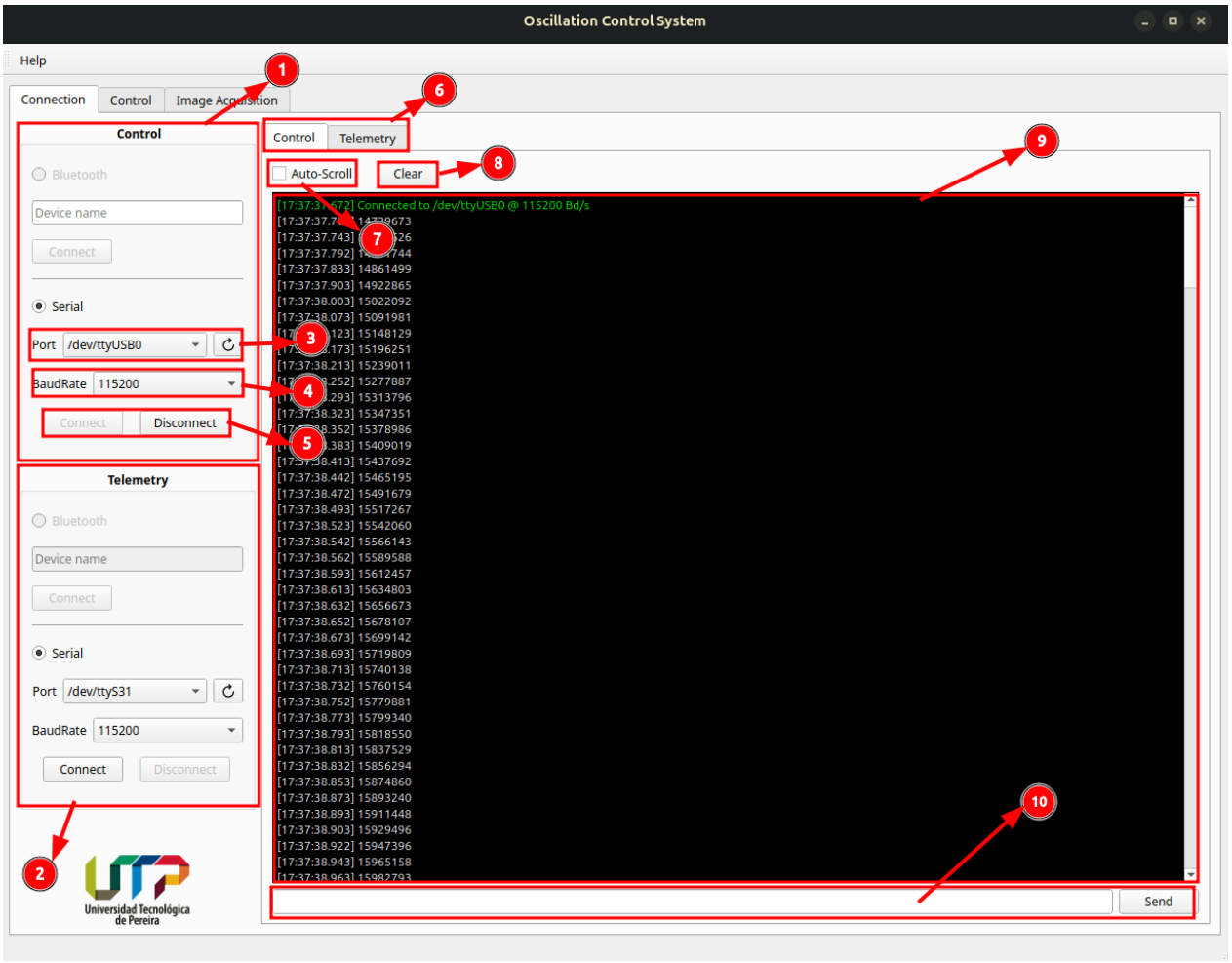

Figura 2: Interfaz de la pestaña *Connection*.

A continuación, se describen los elementos numerados en la Figura 2:

1. **GroupBox de Control:** Contiene los elementos necesarios para configurar la comunicación con el dispositivo encargado del control del motor. Incluye la selección del tipo de comunicación, el puerto serial, el baud rate y los botones de conexión. La opción de comunicación por Bluetooth se encuentra deshabilitada en la versión actual de la aplicación, ya que no ha sido implementada.

2. **GroupBox de Telemetría:** Presenta la misma estructura funcional que el GroupBox de control, pero está dedicado a la configuración de la comunicación con el dispositivo de telemetría que envía los datos de medición del sistema.

3. **Selección de puerto serial y botón de actualización:** Permite seleccionar el puerto COM al cual está conectado el dispositivo correspondiente. El botón de actualización refresca la

lista de puertos disponibles, lo cual resulta útil cuando un dispositivo es conectado después de haber iniciado la aplicación.

4. **Selección de baud rate:** Permite definir la velocidad de comunicación serial. El menú desplegable incluye valores típicos desde 300 hasta 230400 baudios. Por defecto, se selecciona el valor de 115200 baudios, que corresponde a la configuración estándar de los microcontroladores del sistema.

5. **Botones Connect y Disconnect:** Permiten iniciar y finalizar la comunicación serial utilizando los parámetros seleccionados. El botón de conexión solo está disponible cuando se ha seleccionado un puerto válido, y el botón de desconexión se habilita únicamente cuando la conexión está activa.

6. **Pestañas de consola Control y Telemetry:** Permiten alternar entre las consolas de comunicación correspondientes a cada dispositivo. Ambas consolas presentan la misma estructura y funcionalidad.

7. **Opción de auto-desplazamiento (Auto-scroll):** Al estar activada, la consola se desplaza automáticamente para mostrar los mensajes más recientes.

8. **Botón Clear:** Limpia el contenido de la consola, eliminando todos los mensajes mostrados hasta el momento.

9. **Consola de comunicación:** Muestra en tiempo real los mensajes enviados y recibidos a través de la comunicación serial, así como los mensajes de estado y error, cada uno comenzando con su respectivo timestamp. El texto está codificado por colores para facilitar su identificación: los errores se muestran en rojo, las conexiones exitosas en verde, los mensajes entrantes en blanco y los mensajes salientes en amarillo.

10. **Campo de envío manual y botón Send:** Permite al usuario enviar comandos manuales a través de la comunicación serial. Esta funcionalidad está orientada principalmente a tareas de depuración y pruebas del sistema.

2.4. Pestaña Control

La pestaña *Control* permite configurar los parámetros de operación del sistema, monitorear en tiempo real las variables dinámicas medidas y visualizar gráficas asociadas al comportamiento del movimiento oscilatorio. Esta pestaña constituye el núcleo operativo de la aplicación una vez que las conexiones han sido establecidas correctamente.

La interfaz presenta un panel lateral con desplazamiento vertical, el cual contiene las distintas secciones de configuración y visualización. Debido a la extensión de este panel, su descripción se divide en dos partes. La Figura 3 muestra los elementos visibles en la vista inicial de la pestaña, mientras que la sección inferior correspondiente a la exportación de datos se describe posteriormente.

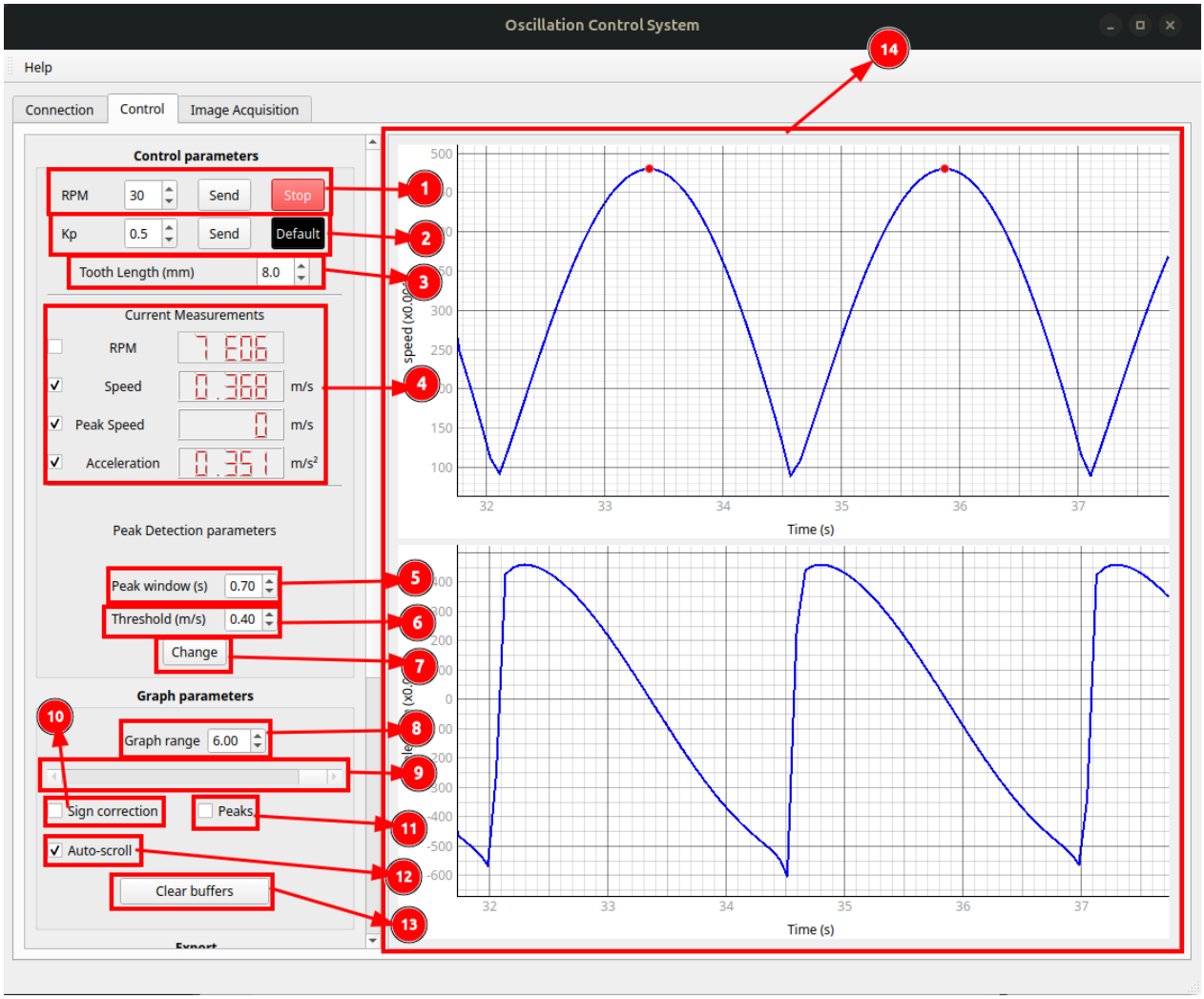

Figura 3: Interfaz de la pestaña *Control*: parámetros de operación y visualización.

A continuación, se describen los elementos numerados en la Figura 3:

1. **Control de velocidad (RPM):** Permite definir la velocidad de rotación deseada del sistema mediante un campo numérico. El botón *Send* envía el valor configurado al microcontrolador de control, mientras que el botón *Stop* envía un comando de parada inmediata del sistema.

2. **Parámetro de control proporcional (Kp):** Permite configurar la ganancia proporcional utilizada en el control del motor. El botón *Send* aplica el nuevo valor al sistema, y el botón *Default* restaura el valor predeterminado del parámetro.

3. **Tooth Length:** Define la longitud de los dientes de la peineta metálica en milímetros. Este parámetro es utilizado en el cálculo de la velocidad, la aceleración y otras variables derivadas a partir de las mediciones temporales del sensor.

4. **Selección de variables a visualizar:** Conjunto de casillas de verificación que permite seleccionar qué variables serán mostradas en las gráficas. Las opciones incluyen RPM, velocidad lineal, velocidad pico y aceleración. Cada variable seleccionada se acompaña de un indicador numérico que muestra su valor instantáneo.

5. **Peak window:** Configura el intervalo mínimo de tiempo, en segundos, que debe existir entre picos consecutivos para que estos sean considerados válidos durante el proceso de detección de picos en la señal de velocidad.

6. **Threshold:** Define el valor mínimo de velocidad, en metros por segundo, que una señal debe superar para ser considerada como un pico válido.

7. **Botón Change:** Aplica los cambios realizados en los parámetros de detección de picos definidos en los campos anteriores.

8. **Graph range:** Configura el rango temporal, en segundos, que se mostrará en las gráficas de velocidad y aceleración.

9. **Barra de desplazamiento horizontal:** Permite seleccionar manualmente la porción de los datos que se desea visualizar en las gráficas. Esta opción se deshabilita automáticamente cuando la función de auto-desplazamiento está activa.

10. **Sign correction:** Habilita o deshabilita el algoritmo de corrección de signo aplicado a la señal de velocidad. Este algoritmo se utiliza para reconstruir el signo de la velocidad a partir de mediciones que, por la naturaleza del sensor, se obtienen en valor absoluto.

11. **Opción Peaks:** Permite habilitar la visualización de los picos detectados en la señal de velocidad dentro de la gráfica correspondiente.

12. **Auto-scroll:** Al estar activada, la gráfica se desplaza automáticamente para mostrar la porción más reciente de los datos. Esta opción aplica únicamente a las gráficas de velocidad y aceleración.

13. **Clear buffers:** Elimina todos los datos actualmente almacenados en los buffers internos de la aplicación, limpiando las gráficas y permitiendo iniciar una nueva sesión de adquisición.

14. **Área de visualización de gráficas:** Muestra las gráficas correspondientes a las variables seleccionadas. Cuando la visualización de picos está habilitada, estos se representan como puntos sobre la gráfica de velocidad.

La parte inferior del panel de la pestaña *Control* está dedicada a la exportación de los datos adquiridos durante la ejecución del sistema. Esta sección permite seleccionar qué variables serán almacenadas y definir las opciones de guardado de los archivos generados.

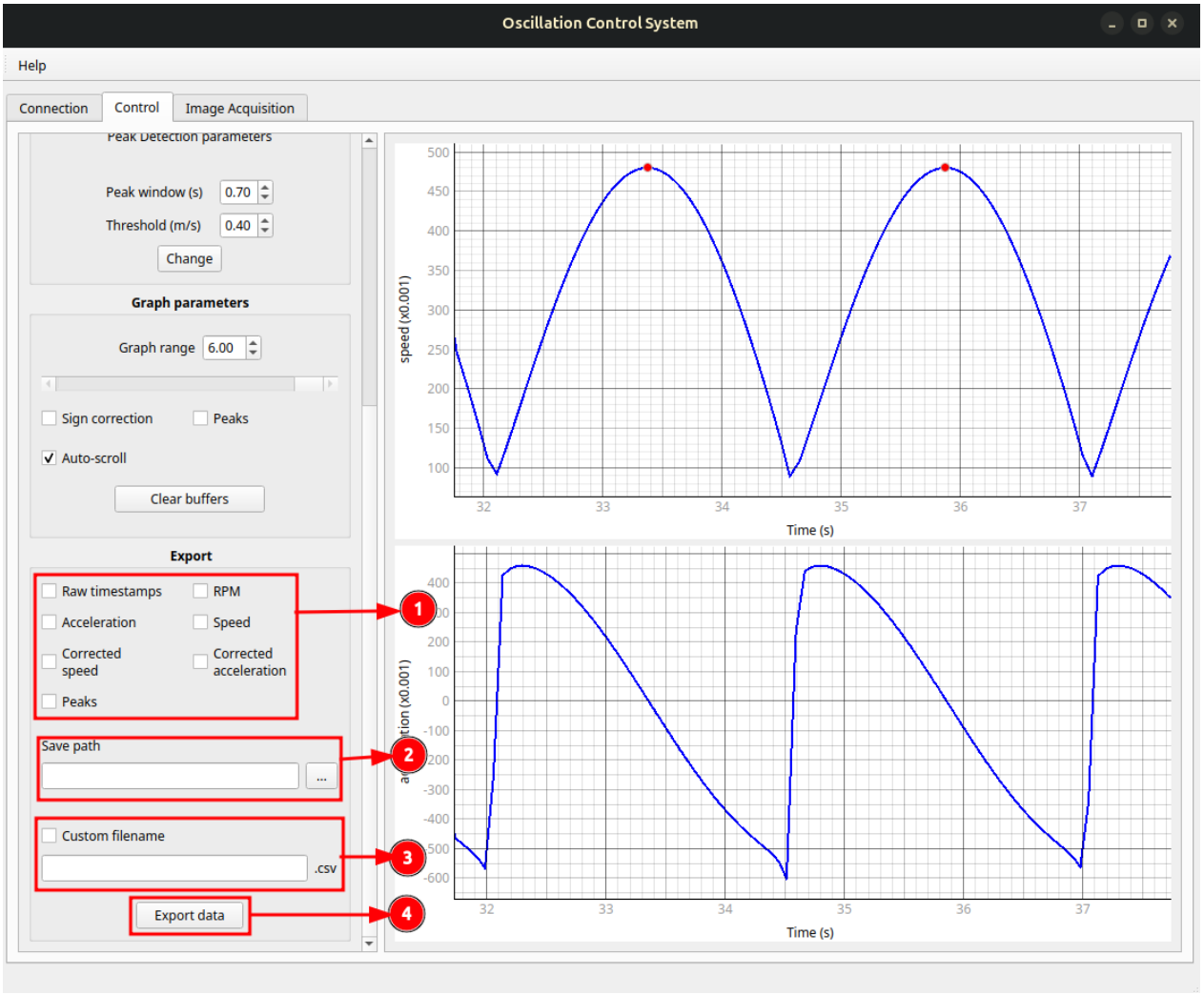

Figura 4: Interfaz de la pestaña *Control*: sección de exportación de datos.

La Figura 4 muestra la sección correspondiente a la exportación de datos, la cual no es visible en la vista inicial debido al desplazamiento vertical del panel.

A continuación, se describen los elementos numerados en la Figura 4:

1. **Selección de datos a exportar:** Conjunto de casillas de verificación que permite definir qué variables serán incluidas en el archivo de exportación. Las opciones disponibles son:
   - *Raw timestamps:* Marcas de tiempo originales recibidas desde el sistema de telemetría, sin ningún tipo de procesamiento.
   - *RPM:* Velocidad angular del sistema.
   - *Speed:* Velocidad lineal calculada a partir de las mediciones del sensor.
   - *Acceleration:* Aceleración lineal obtenida a partir de la derivación temporal de la velocidad.
   - *Corrected Speed:* Velocidad lineal con el algoritmo de corrección de signo aplicado.
   - *Corrected Acceleration:* Aceleración calculada a partir de la velocidad corregida.
   - *Peaks:* Valores correspondientes a los picos de velocidad detectados durante la adquisición.
2. **Save path:** Permite seleccionar el directorio donde se almacenará el archivo de datos exportado. El botón asociado abre un cuadro de diálogo para la selección de carpetas en el sistema operativo.
3. **Custom filename:** Al activar esta opción, el usuario puede especificar manualmente el nombre del archivo de exportación mediante un campo de texto. Si esta opción no está habilitada, la aplicación asigna automáticamente un nombre genérico basado en la fecha y hora de generación del archivo.
4. **Export data:** Genera el archivo de datos en formato CSV utilizando las opciones de selección y guardado configuradas previamente.

2.5. Pestaña Image Acquisition

La pestaña *Image Acquisition* permite la captura, organización y visualización de imágenes asociadas a un experimento, así como el almacenamiento de metadatos relacionados con las condiciones de operación del sistema en el momento de la adquisición. Esta pestaña está diseñada para facilitar la gestión de conjuntos de imágenes (*sets*) y su sincronización con los parámetros de oscilación configurados previamente.

La Figura 5 muestra la interfaz correspondiente a esta pestaña, donde se han numerado los distintos elementos interactivos para su descripción.

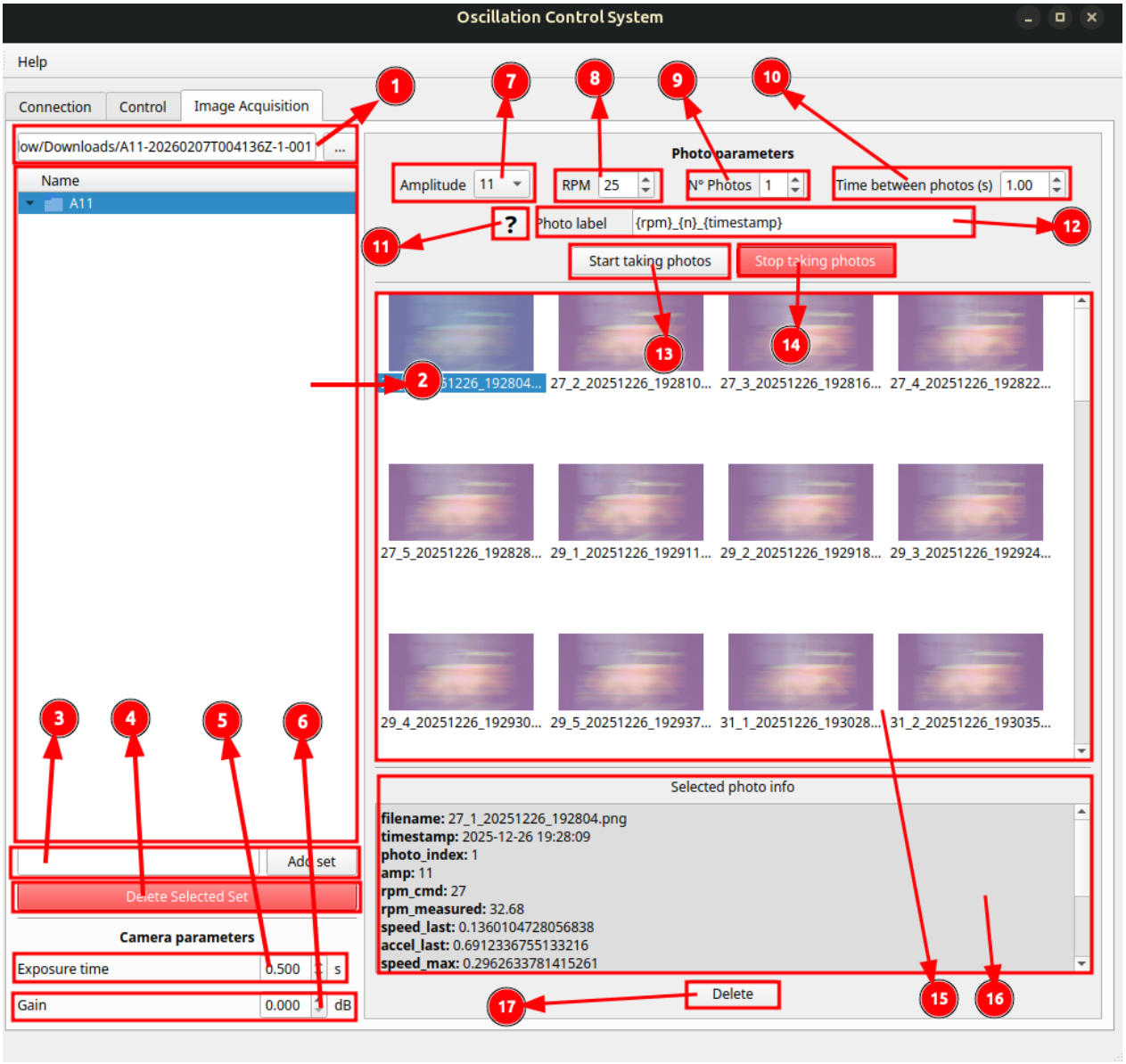

Figura 5: Interfaz de la pestaña *Image Acquisition*.

A continuación, se describen los elementos numerados en la Figura 5:

1. **Selección de ruta de trabajo:** Permite seleccionar la carpeta principal donde se visualizarán y almacenarán los conjuntos de imágenes. Esta ruta define el espacio de trabajo de la pestaña.

2. **Vista de carpetas:** Muestra la estructura de carpetas y subcarpetas contenidas dentro de la ruta seleccionada. Cada carpeta puede representar un conjunto de imágenes correspondiente a un experimento.

3. **Creación de sets de imágenes:** Campo de texto y botón *Add Set* que permiten crear una nueva carpeta dentro de la carpeta seleccionada, destinada a almacenar un nuevo conjunto de imágenes.

4. **Delete Selected Set:** Elimina el conjunto de imágenes seleccionado actualmente. Esta acción elimina la carpeta correspondiente y su contenido.

5. **Tiempo de exposición:** Permite definir el tiempo de exposición de la cámara para la captura de imágenes.

6. **Ganancia:** Configura la ganancia del sensor de la cámara durante la adquisición.

7. **Amplitud:** Permite seleccionar la amplitud del movimiento oscilatorio asociada al conjunto de imágenes. Las opciones disponibles corresponden a las amplitudes físicas del dispositivo mecánico.

8. **RPM del set:** Define la velocidad de rotación que será almacenada en los metadatos del set de imágenes a tomar. Es importante notar que la información aquí configurada NO cambiará las RPM deseadas de la pestaña de control.

9. **Número de fotografías:** Establece la cantidad de imágenes consecutivas que se capturarán durante la adquisición.

10. **Tiempo entre fotografías:** Define el intervalo temporal, en segundos, entre capturas consecutivas.

11. **Ayuda del sistema de etiquetado:** Abre un menú informativo que describe el sistema de etiquetado dinámico utilizado para generar los nombres de los archivos de imagen, así como las etiquetas disponibles para su construcción.

12. **Campo de etiqueta de fotografías:** Permite definir el nombre de los archivos de imagen mediante el uso de etiquetas dinámicas. Este nombre se aplica a todas las imágenes del conjunto, garantizando una nomenclatura consistente.

13. **Start Taking Photos:** Inicia el proceso de adquisición de imágenes utilizando los parámetros configurados.

14. **Stop Taking Photos:** Detiene inmediatamente el proceso de captura de imágenes.

15. **Vista de imágenes del set:** Muestra la lista de imágenes contenidas en el conjunto seleccionado, incluyendo su nombre y una miniatura, de forma similar a un explorador de archivos.
16. **Información de la imagen seleccionada:** Muestra los metadatos asociados a la imagen seleccionada, tales como parámetros de adquisición, valores dinámicos del sistema y configuraciones de control vigentes durante la captura.
17. **Delete Photo:** Elimina la imagen seleccionada actualmente del conjunto.

3. Arquitectura del Software

Esta sección está dirigida a desarrolladores o investigadores que deseen modificar, extender o auditar el funcionamiento interno de la aplicación.

3.1. Descripción general y principios de diseño

La aplicación OSCOS está desarrollada en Python utilizando el *framework* **PyQt5** para la construcción de la interfaz gráfica. El diseño sigue una **arquitectura de capas bien definidas**, separando claramente la interfaz de usuario, la lógica de control, el procesamiento de datos y la comunicación con hardware.

Los principios fundamentales que rigen el diseño son:

1. **Separación de responsabilidades:** Cada módulo tiene una responsabilidad única y bien definida.
2. **Desacoplamiento mediante señales:** Los componentes se comunican a través de señales de Qt, evitando dependencias directas.
3. **Thread-safety:** Las operaciones de entrada-salida ocurren en hilos separados, sin bloquear la interfaz gráfica.
4. **Extensibilidad:** El diseño modelado facilita la adición de nuevos procesadores, controladores o funcionalidades sin modificaciones invasivas.
5. **Patrón de flujo de datos:** Los datos fluyen desde sensores hacia buffers, procesamientos, visualización y exportación.

3.2. Estructura general del proyecto

El proyecto se organiza en una estructura jerárquica de directorios, omitiendo directorios generados automáticamente como `__pycache__`:

```
oscos/
├── main.py                    # Punto de entrada principal
├── requirements.txt           # Dependencias de Python
├── resources.qrc              # Definición de recursos Qt
├── resources_rc.py            # Recursos compilados
│
├── ui/                        # Capa de presentación
│   ├── __init__.py
│   ├── gui.py                 # UI generada (1176 líneas)
│   └── gui.ui                 # Definición de interfaz Qt
│
├── controllers/               # Controladores (lógica de UI)
│   ├── __init__.py
│   ├── connection.py          # Control de conexiones (190 líneas)
│   ├── control.py             # Control motor y gráficos (482 líneas)
│   ├── image.py               # Captura de fotos (730 líneas)
│   └── help_dialog.py         # Diálogo de ayuda
│
├── core/                      # Lógica central
│   ├── __init__.py
│   ├── serial_manager.py      # Gestor de comunicación (130 líneas)
│   ├── data_buffer.py         # Buffers circulares (50+ líneas)
│   ├── processors.py          # Procesadores de señal (215 líneas)
│   └── take_photo.py          # Captura Basler (114 líneas)
│
└── workers/                   # Threads no-bloqueantes
    ├── __init__.py
    └── serial_worker.py       # Worker serial (59 líneas)
```

Esta estructura refleja una arquitectura típica de aplicaciones Qt basada en el patrón MVC (Model-View-Controller) adaptado a las necesidades específicas del sistema.

3.3. Archivo principal: main.py

El archivo `main.py` constituye el punto de entrada único de la aplicación. Su rol es mínimo pero crítico: orquestar la creación de la ventana principal e instanciar los controladores.

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.ui = Ui_MainWindow()
        self.ui.setupUi(self)

        # Instanciar controladores
        self.connection_controller = ConnectionController(self.ui)
        self.control_controller = ControlController(self.ui)
        self.image_controller = ImageController(self.ui)

        # Conectar Help
        self.ui.actionHelp.triggered.connect(self.show_help)
```

Aspecto crítico del diseño: `MainWindow` no contiene lógica de negocio. Actúa como un orquestador que delega toda la funcionalidad a los controladores específicos. Esto permite modificar la lógica sin tocar la estructura principal de la aplicación.

3.4. Capa de controladores

El directorio `controllers/` contiene los módulos que implementan la lógica de negocio asociada a cada elemento de la interfaz gráfica. Los controladores actúan como intermediarios entre la UI y el núcleo lógico.

3.4.1. ConnectionController (connection.py)

Responsabilidad: Gestionar la configuración y estado de dos canales seriales independientes: control y telemetría.

Atributos principales:

- `self.ui`: Referencia al objeto `Ui_MainWindow`.
- `self.control_thread`, `self.telemetry_thread`: Threads de los workers seriales.
- `self.control_worker`, `self.telemetry_worker`: Instancias de `SerialWorker`.

Métodos principales:

Método	Descripción
`populate_ports()`	Enumera puertos COM disponibles en el sistema
`populate_baudrates()`	Carga velocidades estándar (300 - 230400 baud) en combo-boxes
`control_COM_connect()`	Abre conexión de control con parámetros seleccionados
`control_COM_disconnect()`	Cierra conexión de control limpiamente
`control_connected(port, baud)`	Callback: actualiza UI cuando conexión exitosa
`update_control_console(text)`	Actualiza consola con datos recibidos
`send_control_console()`	Envía comando manual por puerto de control

Patrón de arquitectura: Este controlador utiliza exclusivamente señales para comunicación. No realiza I/O serial directamente; en su lugar, emite señales que son procesadas por el `SerialManager`.

Formato de consola: Cada mensaje incluye timestamp con formato `[HH:MM:SS.fff]`:

```
[12:34:56.123] Conectado a COM3 @ 115200 Bd/s        [verde]
[12:34:57.456] velocidad = 45.2 m/s                  [blanco]
[12:34:58.789][SENT] r;30                            [amarillo]
[12:34:59.012][ERROR] Puerto no disponible           [rojo]
```

3.4.2. ControlController (control.py)

Responsabilidad: Gestionar control del motor, visualización de gráficas, procesamiento de telemetría y exportación de datos.

Este es el componente más complejo de la aplicación. Sus responsabilidades incluyen:

- Recibir datos telemétricos desde `SerialManager`.
- Alimentar procesadores de señal para transformar datos crudos.

- Mantener buffers históricos de datos procesados.
- Actualizar gráficas a 60 FPS mediante un temporizador interno.
- Permitir envío de comandos de control (RPM, Kp).
- Exportar buffers a archivos CSV.

Procesadores de señal instanciados:

```
self.speed_processor = SpeedProcessor(
    tooth_length_mm, buffer.speed)
self.accel_processor = AccelerationProcessor(buffer.acceleration)
self.speed_corrected_processor = SpeedCorrectedProcessor(
    buffer.speed_corrected)
self.speed_peak_processor = SpeedPeakDetection(buffer.speed_peaks)
```

Flujo de datos telemétricos:

```
SerialManager.telemetry_rx(string)
    ↓ [parsea formato]
ControlController.refresh_t_buffer(text)
    ↓ [alimenta procesadores]
SpeedProcessor → buffer.speed
AccelerationProcessor → buffer.acceleration
SpeedCorrectedProcessor → buffer.speed_corrected
SpeedPeakDetection → buffer.speed_peaks
    ↓ [cada 16.67ms - 60 FPS]
ControlController.refresh_graph()
    ↓ [actualiza]
PlotWidget + LCD displays
```

Métodos de comandos de control:

Método	Comando serial	Efecto
`change_rpm()`	`r;30`	Cambia RPM a valor del SpinBox
`stop_rpm()`	`r;0`	Detiene motor inmediatamente
`send_kp()`	`k;0.8`	Envía parámetro Kp
`set_kp_default()`	`k;0.5`	Reset Kp a valor por defecto

Atributo importante - signal_registry:

Este diccionario mapea nombres de señales a sus widgets asociados:

```
self.signal_registry = {
    "speed": {"lcd": self.ui.lcdSpeed, "scrollable": True},
    "acceleration": {"lcd": self.ui.lcdAcceleration, "scrollable": True},
    "rpm": {"lcd": self.ui.lcdRPM, "scrollable": False},
    "peak": {"lcd": self.ui.lcdPeak, "scrollable": None},
}
```

Permite agregar nuevas señales sin modificaciones en múltiples lugares.

3.4.3. ImageController (image.py)

Responsabilidad: Captura automática de imágenes, organización en sets y generación de metadatos.

Características principales:

- **Captura automática:** Timer interno permite capturar N fotos con intervalo configurable.
- **Plantillas de nombres:** Sistema de macros para nombrar archivos dinámicamente.
- **Persistencia:** Usa `QSettings` para recordar rutas y configuración entre ejecuciones.
- **Metadatos:** Guarda exposure, gain, RPM y timestamp con cada imagen.

Macros soportadas en plantillas:

```
Adquisición: {amp}, {rpm}, {exposure}, {exposure_s}, {gain}
Indexación: {n}, {n0}
Tiempo: {year}, {month}, {day}, {hour}, {minute}, {second}, {timestamp}
Contexto: {set}
```

Ejemplo: La plantilla `foto_{set}_{rpm}rpm_{n0}.jpg` genera nombres como:

```
foto_acceleration_test_3000rpm_001.jpg
foto_acceleration_test_3000rpm_002.jpg
...
```

3.5. Capa de interfaz gráfica (ui/)

El directorio `ui/` contiene exclusivamente archivos relacionados con la presentación visual:

- `gui.ui`: Archivo XML generado por Qt Designer que describe la estructura visual.
- `gui.py`: Código Python autogenerado. **No debe editarse manualmente.**

Flujo de trabajo para modificaciones visuales:

1. Abrir `gui.ui` con Qt Designer.
2. Realizar cambios visuales en el designer.
3. Guardar el archivo `.ui`.
4. Ejecutar: `pyuic5 gui.ui >gui.py`
5. Verificar que los cambios se propaguen correctamente.

Nota importante: Nunca editar `gui.py` directamente, ya que cualquier regeneración desde Qt Designer sobrescribirá los cambios.

3.6. Núcleo de la aplicación (core/)

El directorio `core/` contiene la lógica fundamental que no depende de la interfaz gráfica.

3.6.1. SerialManager (serial_manager.py)

Patrón: Pub-Sub (Publicador-Suscriptor) con señales de Qt.

Rol: Hub centralizado de comunicación serial que desacopla completamente la UI de los threads de comunicación.

Estructura de dos canales independientes:

```
SerialManager
├── control (puerto serial de control)
│   ├── control_worker (SerialWorker en thread)
│   ├── control_thread (QThread)
│   └── Signals: control_rx, control_sent, control_connected
│
└── telemetry (puerto serial de telemetría)
    ├── telemetry_worker (SerialWorker en thread)
    ├── telemetry_thread (QThread)
    └── Signals: telemetry_rx, telemetry_sent, telemetry_connected
```

Signals emitidos (interfaz pública):

Signal	Parámetros y significado
`control_rx`	`(str)` - Dato recibido por puerto de control
`control_sent`	`(str)` - Confirmación de envío
`control_connected`	`(str, int)` - (puerto, baudrate) - Conexión exitosa
`control_disconnected`	`()` - Cierre de conexión
`control_error`	`(str)` - Mensaje de error

Métodos críticos:

```
def connect_control(port, baud):
    # Crea worker y thread, conecta signals
    # El worker inicia la lectura serial en el nuevo thread

def disconnect_control():
    # Emite shutdown_control signal → worker.stop()
    # Aguarda a que el thread termine (thread.wait())

def send_control(text):
    # Emite send_control_signal → worker.send_text()
```

Implementación de singleton: Al final del archivo se instancia globalmente:

```
serial_mgr = SerialManager()
```

Esto permite a cualquier módulo acceder como: `from core import serial_mgr`.

3.6.2. TelemetryBuffer y BufferRegistry (data_buffer.py)

Propósito: Almacenamiento thread-safe de datos históricos con patrón observer integrado.

Clase TelemetryBuffer:

Implementa un buffer circular FIFO con capacidad máxima configurable (100000 elementos por defecto).

```
class TelemetryBuffer:
    def __init__(self, maxlen=100000):
        self.timestamps = deque(maxlen=maxlen)
        self.values = deque(maxlen=maxlen)
        self.lock = Lock()  # Thread-safety
        self._subscribers = []
        self._t0 = None  # Timestamp base

    def add(self, value, timestamp=None):
        with self.lock:
            # Cálculo de timestamp relativo
            if self._t0 is None:
                self._t0 = timestamp
            self.timestamps.append(timestamp - self._t0)
            self.values.append(value)

        # Notificar subscribers (sincronamente)
        for cb in self._subscribers:
            cb(timestamp, value)
```

Métodos principales:

- `add(value, timestamp=None)`: Agrega dato y notifica subscribers sincronamente.
- `subscribe(callback)`: Registra callback que se ejecuta al agregar datos.
- `get_all()`: Retorna copia thread-safe de todos los datos.
- `get_latest(n)`: Retorna últimos N elementos.
- `clear()`: Limpia buffer y resetea tiempo base.

Clase BufferRegistry (singleton global):

```
class BufferRegistry:
    def __init__(self):
        self.raw_timestamps = TelemetryBuffer()
        self.speed = TelemetryBuffer()
        self.acceleration = TelemetryBuffer()
        self.rpm = TelemetryBuffer()
        self.speed_corrected = TelemetryBuffer()
        self.acceleration_corrected = TelemetryBuffer()
        self.speed_peaks = TelemetryBuffer()

buffer = BufferRegistry()  # Instancia global
```

Arquitectura de capas de datos:

```
raw_timestamps (datos crudos del sensor)
    ↓
speed (velocidad = tooth_length / dt)
    ├─ rama 1 ─→ acceleration (aceleración = dv/dt)
    │
    └─ rama 2 ─→ speed_corrected (con corrección de signo)
                    ↓
                acceleration_corrected
                    ↓
                speed_peaks (picos detectados)
```

3.6.3. Procesadores de señal (processors.py)

Patrón de arquitectura: Procesadores encadenables que implementan operaciones de DSP (Digital Signal Processing).

Arquitectura base:

```
StreamProcessor (clase base abstracta)
    ├── SpeedProcessor
    ├── AccelerationProcessor
    ├── SpeedCorrectedProcessor
    └── SpeedPeakDetection
```

Clase SpeedProcessor:

Convierte timestamps de eventos (impulsos de sensor) en velocidad lineal.

- **Entrada:** Timestamps de eventos de encoder.
- **Parámetro:** Longitud de diente L (milímetros).
- **Cálculo:**
$$v_n = \frac{L}{t_n - t_{n-1}}$$
- **Salida:** Velocidad en m/s a tiempo $t_{\text{mid}} = \frac{t_n + t_{n-1}}{2}$.

Clase AccelerationProcessor:

Diferencia numérica de velocidad respecto al tiempo.

- **Entrada:** Stream de velocidad (valores síncronos).
- **Cálculo:**
$$a_n = \frac{v_n - v_{n-1}}{t_n - t_{n-1}}$$
- **Salida:** Aceleración en m/s^2.

Clase SpeedCorrectedProcessor - Algoritmo de corrección de signo:

Problema: El sensor solo entrega valores absolutos. Al cambiar de dirección, hay oscilaciones ruidosas alrededor de cero.

Solución: Máquina de estados con histéresis y suavizado temporal que reconstruye el signo.

Algoritmo en cuatro etapas:

1. **Suavizado exponencial (EMA):**

$$v_{\text{smooth},n} = \alpha \cdot v_n + (1 - \alpha) \cdot v_{\text{smooth},n-1}$$

   Donde $\alpha = 0{,}2$ por defecto (ajustable).

2. **Detección de pendiente:**

$$\text{slope}_n = \frac{v_{\text{smooth},n} - v_{\text{smooth},n-1}}{\Delta t}$$

3. **Máquina de estados:**
   - Estado "ascending": slope $> \epsilon$ (0.02 por defecto).
   - Estado "descending": slope $< -\epsilon$.
   - Al detectar transición, requiere N muestras consecutivas (3 por defecto) para confirmar.

4. **Gating temporal:** No permite flips más frecuentes que `MIN_FLIP_DT` $= 0.2$ segundos.

Parámetros ajustables:

Parámetro	Defecto	Significado
`smoothing_alpha`	0.2	Coeficiente EMA (0.0-1.0)
`SLOPE_EPS`	0.02	Deadband de pendiente (m/s^2)
`CONFIRM_SAMPLES`	3	Muestras para confirmar cambio
`MIN_FLIP_DT`	0.2	Segundos mínimos entre flips

Clase SpeedPeakDetection - Detección de máximos con anti-aliasing:

Problema: Detectar picos reales evitando picos espurios causados por ruido de alta frecuencia.

Algoritmo:

1. Mantiene un **candidato a máximo** (valor y timestamp).
2. Si llega valor mayor: **actualiza candidato**.
3. Si pasan `window_seconds` sin valor mayor: **emite pico como evento** al buffer.
4. Después de emitir: **bloquea nuevos picos** por `window_seconds`.
5. Solo emite si `valor >threshold`.

Parámetros:

Parámetro	Significado
`window_seconds`	0.5 s - Ventana temporal mínima entre picos
`threshold`	0.4 m/s - Valor mínimo para validar pico

Encadenamiento de procesadores (subscriber pattern):

```
buffer.speed.subscribe(accel_processor)
# → Cada vez que se agrega a buffer.speed,
#   accel_processor.__call__(t,v) es invocado
#   → accel_processor emite a buffer.acceleration

buffer.speed.subscribe(speed_corrected_processor)
# → Paralelamente, speed_corrected_processor
#   procesa los datos de buffer.speed

buffer.speed_corrected.subscribe(accel_corrected_processor)
# → La aceleración corregida se obtiene de
#   la velocidad corregida

buffer.speed.subscribe(speed_peak_processor)
# → Detección de picos en paralelo
```

3.6.4. Captura de imágenes (take_photo.py)

Función: `take_photo(exposure_us=2000000, gain_db=0.0)`

Captura imagen con cámara Basler usando el SDK `pypylon`.

Pasos internos:

1. Obtiene `TlFactory` (Basler Transport Layer).
2. Enumera dispositivos conectados.
3. Instancia primera cámara disponible.
4. **Itera formatos de pixel:** Intenta RGB8, BGR8, Bayer... hasta encontrar compatible.
5. Desactiva auto-ganancia y auto-exposición.
6. Configura parámetros: `Gain`, `ExposureTime`.
7. Configura resolución máxima: 2040 $\times$ 1086 píxeles.
8. Desactiva trigger (captura inmediata bajo demanda).
9. Captura exactamente 1 frame.
10. Retorna imagen en formato color (BGR) y escala de grises.

Excepciones:

- `RuntimeError`: Si no hay cámara Basler conectada.

Integración con threads: Esta función es bloqueante (puede tardarse varios segundos). Por eso es llamada desde `ImageController` mediante timers para no bloquear la UI.

3.7. Workers y ejecución en hilos (workers/)

3.7.1. SerialWorker (serial_worker.py)

Patrón: Worker thread con comunicación mediante signals de Qt.

Ciclo de vida completo:

```
Creación (main thread):
worker = SerialWorker(port, baudrate)
thread = QThread()
worker.moveToThread(thread)

Inicio:
thread.started.connect(worker.start)
thread.start()

Operación (worker thread, cada 10ms):
read_serial():
    if self.ser.in_waiting > 0:
        data = self.ser.read(self.ser.in_waiting)
        rx_buffer.extend(data)

        while b"\n" in rx_buffer:
            line, _, rx_buffer = rx_buffer.partition(b"\n")
            text = line.decode(errors="ignore").strip()
            data_received.emit(text)  # Signal → main thread

Parada (main thread):
shutdown_control.emit()  → slot stop()
worker.stop()            → para timer, cierra puerto
finished.emit()          → thread.quit()
thread.wait()            → espera limpieza
```

Métodos slots principales:

Slot	Descripción
`@pyqtSlot() start()`	Abre puerto serial, inicia timer de 10ms
`@pyqtSlot() stop()`	Para timer, cierra puerto, emite finished
`@pyqtSlot() read_serial()`	Llamado cada 10ms por timer
`@pyqtSlot(str) send_text(text)`	Escribe texto codificado en puerto

Buffer RX circulante:

El worker mantiene un `rx_buffer` (bytearray) que acumula bytes recibidos hasta encontrar '\n'. Esto permite parsear líneas completas incluso si el sistema operativo entrega datos en fragmentos arbitrarios.

Signals emitidos:

- `connected(str port, int baud)`: Conexión serial exitosa.
- `data_received(str)`: Línea completa recibida y decodificada.
- `sent_data(str)`: Confirmación de datos enviados.
- `error(str)`: Mensaje de error en I/O.
- `finished()`: Hilo preparándose para terminar.

3.8. Recursos gráficos

El archivo `resources.qrc` define iconos e imágenes utilizados por la interfaz gráfica. Se compila a `resources_rc.py` mediante:

```
pyrcc5 resources.qrc -o resources_rc.py
```

Este archivo compilado es importado automáticamente por `gui.py`.