



UNIVERSIDAD DE GRANADA

TRABAJO FIN DE GRADO
INGENIERÍA INFORMÁTICA

Generador Automático de Programas de Simulación Continua

Autor

José Pineda Serrano

Director

Luis Miguel de Campos Ibáñez



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE TELECOMUNICACIÓN

Granada, Junio de 2025

Generador Automático de Programas de Simulación Continua

José Pineda Serrano

Palabras clave: Simulación continua, generación de código, métodos numéricos, ecuaciones diferenciales, interfaz gráfica, Python.

Resumen

El presente Trabajo de Fin de Grado tiene como objetivo principal el desarrollo de una herramienta software que permita generar automáticamente programas de simulación continua a partir de modelos definidos por el usuario. Esta herramienta facilita la definición estructurada de ecuaciones diferenciales, condiciones iniciales y parámetros del sistema, a través de una interfaz gráfica desarrollada en *CustomTkinter*.

El sistema implementado permite traducir estos modelos a código fuente en tres lenguajes de programación: C++, Python y Java, integrando métodos numéricos como Euler y Runge-Kutta para la resolución de las ecuaciones. Asimismo, se ha garantizado que el código generado sea autónomo, legible y portable, sin necesidad de librerías externas complejas.

Además, la aplicación incluye un módulo de simulación que permite ejecutar el código generado y visualizar los resultados directamente desde la interfaz. Para validar la funcionalidad de la herramienta, se han realizado pruebas con distintos modelos y se han analizado casos no esperados, comprobando la estabilidad, precisión y coherencia de los resultados obtenidos.

Automatic Generator of Continuous Simulation Programs

José Pineda Serrano

Keywords: Continuous simulation, code generation, numerical methods, differential equations, graphical user interface, Python.

Abstract

This Final Degree Project aims to develop a software tool that automatically generates continuous simulation programs from user-defined models. The tool facilitates the structured definition of differential equations, initial conditions, and system parameters through a graphical user interface developed with *CustomTkinter*.

The implemented system translates these models into source code in three programming languages: C++, Python, and Java, integrating numerical methods such as Euler and Runge-Kutta to solve the equations. The generated code is designed to be self-contained, readable, and portable, without requiring complex external libraries.

In addition, the application includes a simulation module that enables users to execute the generated code and visualize the results directly through the interface. To validate the functionality of the tool, tests have been conducted with various models, including edge cases, confirming the stability, accuracy, and consistency of the results.

This work demonstrates the feasibility of automating the generation of numerical simulations and highlights the value of integrating symbolic analysis techniques and numerical methods into an environment accessible to users with varying levels of technical expertise.

Yo, **José Pineda Serrano**, alumno de la titulación Graduado en Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 50640090R, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: José Pineda Serrano

Granada a 01 de Junio de 2025 .

D. **Luis Miguel de Campos Ibáñez**, Catedrático del Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Generador Automático de Programas de Simulación Continua***, ha sido realizado bajo su supervisión por **Luis Miguel de Campos Ibáñez**, y autoriza la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a X de Junio de 2025 .

Los directores:

Luis Miguel de Campos Ibáñez

Agradecimientos

“Solos podemos hacer poco; juntos podemos hacer mucho.”

— *Helen Keller*

Agradecer a la Universidad de Granada y la Escuela Técnica de Ingenierías Informática y Telecomunicaciones, la oportunidad de formarme con sus profesionales y su comunidad.

A mi tutor en este proyecto y docente ejemplar, Luis Miguel de Campos Ibáñez, quien ha sabido guiarme y aconsejarme con dedicación a lo largo de todo este proceso.

A todas aquellas personas que me han acompañado durante estos maravillosos años:

A mis compañeros de piso, Asier Lizama Muñoz, Carlos Martín Gallardo y Alejandro Rueda Conejero, sin ellos no hubiera tenido el apoyo que tanto necesitaba al llegar a casa después de un duro día de trabajo.

A mis compañeros de clase y amigos, Álvaro Ruiz López, Carlos Muñoz Barco y Daniel Lozano Moya, con quienes he compartido los mejores momentos durante mi estancia en Granada.

A mis amigos de mi ciudad natal, Víctor Caballero Rodríguez y Javier Muñoz García, quienes han estado ahí siempre que lo he necesitado.

A mi mejor amiga, María José Rueda Montes, quien me convenció para empezar a estudiar Ingeniería Informática y me enseñó mis primeros pasos en programación.

A mi cuñada, Lucía Caballero Granados, quien ha compartido tantos momentos conmigo, regalándome su compañía, su alegría y su cariño en cada etapa de este camino.

A mi pareja, Elena Caballero Granados, por estar a mi lado en todo momento, apoyándome y acompañándome en mis mejores y peores momentos.

Y, por supuesto, a mi familia, por su amor, comprensión y constante apoyo a lo largo de este camino académico y personal. Sin ellos nada de esto habría sido posible.

Índice general

1. Introducción	1
1.1. Contexto	2
1.2. Motivación	3
1.3. Objetivos	3
1.4. Estructura de la memoria	4
2. Conceptos Básicos de Simulación	5
2.1. Conceptos teóricos sobre la simulación.	5
2.2. Clasificación de modelos.	6
2.3. Conveniencia de la simulación	7
2.4. Modelos de Simulación Continuos	7
2.5. Métodos numéricos.	9
2.5.1. El método de Euler.	9
2.5.2. Heun.	11
2.5.3. Runge-Kutta de Cuarto Orden (RK-4).	12
2.5.4. Limitaciones de los Métodos de Paso Fijo.	12
2.5.5. Runge-Kutta-Fehlberg (RKF-45).	13
3. Estado del Arte	15
3.1. Descripción del dominio del problema	15
3.2. Metodologías potenciales	16
3.3. Tecnologías potenciales	17
3.4. Trabajos Relacionados	19
4. Análisis	21
4.1. Requisitos funcionales	21
4.2. Requisitos no funcionales	22
4.3. Requisitos de información	22
4.4. Casos de Uso	23
4.5. Planificación	30
4.6. Presupuesto	31
5. Diseño	33
5.1. Diagrama de clases	33
5.2. Diagramas de secuencia	39
5.3. Diagrama de arquitectura	46
5.4. Diseño de la interfaz	47
6. Implementación	51
6.1. Tecnologías elegidas	51
6.2. Desarrollo de componentes	53
6.2.1. Clase <code>GUI_CTK</code>	53
6.2.2. Clase <code>GUI_Equation</code>	55
6.2.3. Clase <code>GUI_Condition</code>	57

6.2.4. Clase <code>GUI_Simulation</code>	59
6.2.5. Clase <code>GeneratorController</code>	62
6.2.6. Clase <code>LogHandler</code>	63
6.2.7. Clase <code>ContinuousModelGenerator</code>	64
6.2.8. Clase <code>SimulationModelGenerator</code>	68
6.2.9. Clase <code>CppSimulationGenerator</code>	70
6.2.10. Clase <code>JavaSimulationGenerator</code>	71
6.2.11. Clase <code>PythonSimulationGenerator</code>	72
6.2.12. Clase <code>Equation</code>	73
6.2.13. Clase <code>Condition</code>	74
6.3. Directorio de implementación	76
6.4. Creación de código fuente	76
6.5. Despliegue de aplicación	77
6.5.1. Instalación mediante <i>pip</i>	77
6.5.2. Generación del ejecutable mediante <i>PyInstaller</i>	78
6.5.3. Limitación en entornos virtualizados sin acceso a <i>pip</i>	78
7. Pruebas	79
7.1. Preparación del entorno de pruebas	79
7.2. Pruebas por funcionalidad	79
7.2.1. Prueba del módulo principal	80
7.2.2. Prueba del módulo de ecuaciones	80
7.2.3. Prueba del módulo de condiciones	81
7.2.4. Prueba del sistema de ejecución de simulaciones	82
7.2.5. Validación de requisitos funcionales	83
8. Conclusiones	85
9. Trabajos Futuros	87
Bibliografía	90
Anexos	96

Índice de figuras

2.1. Método de Euler. Error Local	10
2.2. Método de Heun vs Euler. Error Local	11
4.1. Diagrama de Casos de Uso del Sistema.	29
4.2. Diagrama de Gantt del proyecto - Cronograma Febrero-Junio 2025.	30
5.1. Diagrama de clases del sistema. Parte 1/3	36
5.2. Diagrama de clases del sistema. Parte 2/3	37
5.3. Diagrama de clases del sistema. Parte 3/3	38
5.4. Diagrama Secuencia CU-001. Añadir ecuación introducida por el usuario al sistema.	39
5.5. Diagrama Secuencia CU-002. Editar la ecuación con índice <i>idx</i> almacenada en el sistema.	40
5.6. Diagrama Secuencia CU-003. Eliminar la ecuación con índice <i>idx</i> almacenada en el sistema.	41
5.7. Diagrama Secuencia CU-004. Añadir condición introducida por el usuario al sistema.	42
5.8. Diagrama Secuencia CU-005. Editar la condición con índice <i>idx</i> almacenada en el sistema.	43
5.9. Diagrama Secuencia CU-006. Eliminar la condición con índice <i>idx</i> almacenada en el sistema.	44
5.10. Diagrama Secuencia CU-007. Generación de código fuente.	45
5.11. Diagrama de arquitectura del patrón MVC	46
5.12. Prototipo interfaz. Pantalla inicial	47
5.13. Prototipo interfaz. Pantalla de Añadir/Editar Ecuación	48
5.14. Prototipo interfaz. Pantalla de Añadir/Editar Condición	49
5.15. Prototipo interfaz. Pantalla de Simulación	50
1. Pantalla principal de la aplicación	91
2. Pantalla de añadir/editar ecuación.	92
3. Pantalla de añadir/editar condición.	93
4. Pantalla de simulación de modelos.	94
5. Ejemplo de fichero de exportación resultados.	95

Índice de cuadros

2.1. Tabla de Butcher del método Runge-Kutta-Fehlberg 4(5)	13
4.9. Estimación de costes de personal para el desarrollo del proyecto	31
4.10. Gastos de ejecución	31
6.1. Atributos de la clase GUI_CTK	53
6.2. Métodos de la clase GUI_CTK	54
6.3. Atributos de la clase GUI_Equation	55
6.4. Métodos de la clase GUI_Equation	56
6.5. Atributos de la clase GUI_Condition	57
6.6. Métodos de la clase GUI_Condition	58
6.7. Atributos de la clase GUI_Simulation	59
6.8. Métodos de la clase GUI_Simulation	61
6.9. Atributos de la clase GeneratorController	62
6.10. Métodos de la clase GeneratorController	62
6.11. Atributos de la clase LogHandler	64
6.12. Métodos de la clase LogHandler	64
6.13. Atributos de la clase ContinuousModelGenerator	65
6.14. Métodos de la clase ContinuousModelGenerator	65
6.15. Atributos de la clase SimulationModelGenerator	68
6.16. Métodos de la clase SimulationModelGenerator	68
6.17. Atributos de la clase CppSimulationGenerator	70
6.18. Métodos de la clase CppSimulationGenerator	70
6.19. Atributos de la clase JavaSimulationGenerator	71
6.20. Métodos de la clase JavaSimulationGenerator	71
6.21. Atributos de la clase PythonSimulationGenerator	72
6.22. Métodos de la clase PythonSimulationGenerator	72
6.23. Atributos de la clase Equation	73
6.24. Métodos de la clase Equation	73
6.25. Atributos de la clase Condition	74
6.26. Métodos de la clase Condition	75
7.1. Verificación del cumplimiento de los requisitos funcionales	83

Capítulo 1

Introducción

Actualmente, nos encontramos con multitud de sistemas alrededor, donde para controlarlos se deben tomar continuamente decisiones sobre su comportamiento. Estas decisiones deben desencadenar que el sistema actúe acorde a los objetivos que se han marcado.

Evaluar el comportamiento del sistema frente a las acciones que se definan podría parecer un paso trivial, pero nada más lejos de la realidad. Existen sistemas sencillos en los que el coste del proceso de *experimentación* es totalmente asumible y otros en los que, ya sea por factores de seguridad, disponibilidad u otros, es inviable [21].

Ante este tipo de limitaciones, la **simulación** surge como una alternativa eficaz, ya que permite recrear escenarios reales de forma controlada, sin comprometer recursos físicos ni la integridad del propio sistema [21, 3].

Gracias a la simulación, es posible anticipar comportamiento, analizar situaciones hipotéticas y tomar decisiones fundamentadas antes de actuar, si fuese posible, en el sistema real [21].

Sin embargo, utilizar la simulación no está exento de numerosos **desafíos**. Es necesario comprender que los resultados dependen en gran medida de las condiciones iniciales definidas y las suposiciones que se consideren durante el diseño del propio entorno simulado. Además de esto, interpretar correctamente los resultados obtenidos requiere **unos conocimientos avanzados** del sistema original, así como una actitud crítica ante los posibles sesgos o limitaciones del proceso de simulación [3].

Teniendo esto en cuenta, la simulación no debe entenderse como una simple herramienta automatizada, sino que se trata de un proceso riguroso de análisis que complementa y, en muchos casos, sustituye a la experimentación directa en el mundo real.

1.1. Contexto

Este proyecto se centra en el desarrollo de una herramienta enmarcada dentro del ámbito de la **simulación**, por lo que resulta esencial comenzar contextualizando su significado y las implicaciones que ha tenido el uso de esta herramienta durante la historia.

Uno de los pioneros de la simulación, *Tomas H. Naylor*, la definió así: “La simulación es una técnica numérica para conducir experimentos en una computadora digital. Estos experimentos comprenden ciertos tipos de relaciones matemáticas y lógicas, las cuales son necesarias para describir el comportamiento y la estructura de sistemas complejos del mundo real a través de largos periodos de tiempo” [2].

H. Maisel y *G. Gnugnoli* definen la simulación como: “una técnica numérica para realizar experimentos en una computadora digital. Estos experimentos involucran ciertos tipos de modelos matemáticos y lógicos que describen el comportamiento de sistemas de negocios, económicos, sociales, biológicos, físicos o químicos a través de largos períodos de tiempo” [14].

Según *Shannon*, “la simulación es el proceso de diseñar un modelo de un sistema real y llevar a cabo experiencias con él, con la finalidad de aprender el comportamiento del sistema o de evaluar diversas estrategias para el funcionamiento del sistema” [21].

De entre las distintas definiciones presentadas, la definición de *Shannon* se adecúa de mejor forma al ámbito en el que nos centraremos en este proyecto, la experimentación con modelos que representen sistemas reales con el fin de comprender o evaluar estrategias de operación.

El uso de la simulación se remonta a los años veinte del siglo pasado, siendo impulsada por la aeronáutica, la investigación militar y, posteriormente, por la informática:

- En el año 1929 *Edwin A. Link* comercializó el primer simulador de vuelo electromecánico plenamente funcional que transformó la instrucción de pilotos y estableció la simulación como la nueva herramienta de entrenamiento aeronáutico.
- En la Segunda Guerra Mundial la simulación tomó un papel fundamental en el Proyecto Manhattan con el desarrollo de las armas nucleares. El desarrollo exigió la modelización de procesos de difusión de neutrones y termodinámica extrema. Estos esfuerzos dieron lugar a unos de los métodos más significativos dentro de la simulación, el método de Monte Carlo, ideado por *Stanislaw Ulam* y *John Von Neumann* en Los Álamos.
- En la década de 1960 IBM lanzó GPSS (*General Purpose Simulation System*), uno de los primeros lenguajes para la simulación de eventos discretos orientados a colas y procesos de fabricación. Además de esto RAND Corporation, *H. Markowitz* y *B. Hausner* presentaron SIMSCRIPT(1962), otro lenguaje para la simulación de eventos discretos.
- En las décadas siguientes, las técnicas de simulación se amplían con variantes de Monte Carlo, simulación continua y discreta-híbrida, apareciendo entornos integrados como (SIMULA-67, MATLAB/Simulink, STELLA), aplicados en ámbitos, no solo de la defensa o aviación, sino también a logística, salud, finanzas, etc.

Dentro del amplio campo de la simulación, este proyecto se centra específicamente en el área de la **simulación continua**, una técnica que permite modelar y analizar sistemas cuyo comportamiento varía de forma continua a lo largo del tiempo, generalmente mediante ecuaciones diferenciales ordinarias.

1.2. Motivación

Una vez contextualizado el papel de la simulación y delimitado el ámbito específico de este proyecto, es crucial exponer las razones que motivan su desarrollo.

Pese a los avances en el campo de la simulación continua, aún persisten barreras que dificultan su implementación por parte de estudiantes, investigadores y profesionales sin formación específica en programación o uso de entornos complejos.

Algunas de las herramientas que se encuentran en la actualidad, (MATLAB/Simulink, Modelica, Scilab/Xcos), entre otras, ofrecen un abanico de funcionalidades muy potentes, pero tienen una serie de limitaciones:

- **Licencias de pago**, especialmente en el caso de MATLAB/Simulink.
- **Curva de aprendizaje elevada**. Algunas de estas herramientas requieren de uso de lenguajes o interfaces poco intuitivas para usuarios sin experiencia previa.
- **Dificultades para exportar código comprensible y editable**. OpenModelica permite la exportación a otros lenguajes como C o Python, pero su generación no siempre es directa o incluso el código puede llegar a ser poco comprensible.
- **Complejidad estructural**. La mayoría de estas herramientas están orientadas a la modelización de sistemas físicos complejos y puede resultar difícil para representar modelos puramente matemáticos.

Estas limitaciones han motivado la necesidad de una herramienta que automatice el proceso de generación de programas de simulación continua, sin depender de entornos privativos, requiriendo de pocos conocimientos en cuestiones de programación y evitando interfaces gráficas complejas.

1.3. Objetivos

Habiendo expuesto las limitaciones detectadas en herramientas existentes para la simulación, este proyecto plantea una solución fundamentada en el desarrollo de una herramienta software especializada.

El **objetivo principal** de este Trabajo de Fin de Grado es el diseño y desarrollo de una aplicación que permita la generación automática de programas de simulación continua. Este objetivo se desglosa en los siguientes objetivos específicos:

- **OE1:** Diseñar una interfaz que permita al usuario definir modelos continuos de simulación de forma estructurada y accesible.
- **OE2:** Implementar un sistema de análisis y procesamiento que permita vincular variables, condiciones iniciales y parámetros del sistema.
- **OE3:** Integrar métodos numéricos para la resolución de ecuaciones diferenciales que garanticen la correcta simulación de los sistemas continuos.
- **OE4:** Garantizar que el código fuente generado sea autónomo, legible y portable sin dependencia de plataformas específicas y/o librerías externas complejas.
- **OE5:** Integrar el proceso de simulación dentro del software para facilitar el uso del código fuente generado por parte del usuario.
- **OE6:** Validar la funcionalidad de la herramienta mediante casos de prueba, evaluando las respuestas del software ante casos no esperados, comprobar la calidad del código generado y la precisión de los resultados obtenidos.

1.4. Estructura de la memoria

Tras definir el contexto, justificar la motivación y establecer los objetivos fundamentales del proyecto, el resto de esta memoria se estructura como se detalla a continuación:

1. En el capítulo 2 se definen las principales tecnologías, métodos, técnicas y conocimientos teóricos que constituyen la base para el desarrollo de la herramienta.
2. En el capítulo 3 se presenta el estado del arte, donde se analizan los estudios y herramientas existentes relacionados con el ámbito de este proyecto, generadores automáticos de programas de simulación continua, permitiendo establecer una base teórica y contextual para el desarrollo del proyecto.
3. En el capítulo 4 se presenta el análisis de la propuesta, desglosando los requisitos y funcionalidades esperadas desde la perspectiva de la ingeniería del software.
4. El capítulo 5 contiene la especificación y arquitectura de los principales módulos e interfaces que hemos desarrollado.
5. El capítulo 6 aborda la implementación de la herramienta desarrollada, incluyendo la justificación de las tecnologías seleccionadas y los aspectos técnicos más relevantes de cada uno de sus componentes.
6. En el capítulo 7 se valida el funcionamiento de la herramienta mediante la realización de pruebas, analizando distintos casos de uso y verificando que los resultados obtenidos son acordes con los requisitos establecidos.
7. En el capítulo 8 se abordan las conclusiones finales sobre el proyecto.
8. En el capítulo 9 se detallan los trabajos futuros que podrían aplicarse a la herramienta para mejorar su funcionalidad.

Capítulo 2

Conceptos Básicos de Simulación

Retomando la definición de *Shannon*: “la simulación es el proceso de diseñar un modelo de un sistema real y llevar a cabo experiencias con él, con la finalidad de aprender el comportamiento del sistema o de evaluar diversas estrategias para el funcionamiento del sistema” [21].

A partir de esta definición, surgen dos conceptos esenciales: **sistema** y **modelo**.

Shannon [21] define **sistema** como “un conjunto de objetos o ideas que están interrelacionados entre sí como una unidad para la consecución de un fin”.

Para *Minsky* [12], “un objeto X es un **modelo** del objeto Y para el observador Z, si Z puede emplear X para responder cuestiones que le interesan acerca de Y”.

Estas definiciones engloban las bases para adentrarnos en el marco teórico que sustenta el uso de modelos en simulación. Comprender cómo se construyen, interpretan y utilizan estos modelos es imprescindible para abordar el problema planteado.

2.1. Conceptos teóricos sobre la simulación.

Un **modelo**, en el contexto de la simulación, se compone por un conjunto de supuestos, a menudo en forma de relaciones matemáticas o lógicas que nos permiten comprender cómo se comporta un sistema [21].

El diseño de un modelo a partir de un sistema real no es una tarea trivial, e incluso podría considerarse un arte. La **modelización** de un sistema implica analizar el problema, identificar las características principales, seleccionar y modificar los supuestos básicos, y enriquecer el modelo hasta obtener una aproximación válida hacia nuestros objetivos [2].

Un **modelo** debe cumplir con las siguientes características:

- Ser comprensible para el usuario.
- Debe estar orientado hacia metas u objetivos específicos.
- Producir resultados coherentes y evitar respuestas ilógicas o sin sentido.
- Permitir una interacción sencilla con el usuario, facilitando su manejo y control.
- Incluir todos los aspectos relevantes del sistema, es decir, ser completo.
- Permitir modificaciones o actualizaciones de forma sencilla.
- Comenzar con una estructura simple y permitir su progresiva complejidad con el tiempo.

Estas características fundamentales nos permiten construir modelos eficaces y útiles para el análisis y la toma de decisiones. Sin embargo, no existe un único tipo de modelo capaz de

adaptarse a todas las situaciones. Dependiendo del propósito del estudio, del nivel de detalle requerido y de la naturaleza del sistema real, se pueden emplear distintos tipos de modelos [14].

2.2. Clasificación de modelos.

Los **modelos de simulación** se pueden clasificar dependiendo de diferentes ámbitos:

1. Según su Representación:

- **Modelos Físicos:** Son representaciones tangibles del sistema real.
- **Modelos Simbólicos:** Son conjuntos de supuestos, en forma de relaciones matemáticas o lógicas. Son el tipo principal de modelo en la simulación por ordenador, en el cual el modelo se encuentra implementado en un lenguaje de programación.

2. Según la Evolución del Tiempo:

- **Modelos Estáticos:** Representan un sistema donde el tiempo no juega un papel importante.
- **Modelos Dinámicos:** Representan cómo evoluciona un sistema en el tiempo.

3. Según la Inclusión de Aleatoriedad:

- **Modelos Determinísticos:** No contienen componentes probabilísticos o aleatorios. La salida se encuentra completamente determinada por las entradas.
- **Modelos Estocásticos:** Incluyen al menos algún componente de entrada aleatorio. La salida es aleatoria y se considera una estimación.

4. Según el Nivel de Conocimiento Interno Modelado:

- **Modelos Teóricos:** Intentan reproducir las relaciones funcionales internas del sistema real. Requieren menos datos experimentales para su validación y su rango de validez se basa en la teoría.
- **Modelos Experimentales:** También denominados modelos de caja negra, estos se enfocan en reproducir solo las salidas del sistema real sin intentar modelar su comportamiento interno. Dichos modelos requieren una gran cantidad de datos para su calibración y el rango de validez se limita al conjunto de datos utilizado.

5. Según cómo cambian las Variables de Estado:

- **Modelos de Eventos Discretos:** Las variables de estado cambian instantáneamente en un número de puntos separados en el tiempo, llamados **eventos**.
- **Modelos de Continuos:** Las variables de estado cambian continuamente en el tiempo.

En este proyecto abordaremos este último tipo de modelos, los **Modelos de Simulación Continuos**.

2.3. Conveniencia de la simulación

A continuación mostramos qué tipos de sistemas son aptos para ser simulados:

- No existe una formulación matemática analíticamente resoluble. Existen sistemas reales que no pueden llegar a ser modelados matemáticamente con las herramientas de las que disponemos actualmente, por ejemplo el sistema de control de temperatura de un horno industrial.
- Existe una formulación matemática, pero es complicado obtener una solución analítica. Un ejemplo de esto es el estudio de un péndulo doble que queda definido por ecuaciones diferenciales no lineales acopladas.
- No existe el sistema. En la generación de prototipos o creación de una idea, si se cuenta con un modelo para realizar la experimentación mejorará notablemente el diseño del sistema real.
- Imposibilidad en la realización de experimentos debido a factores económicos, de calidad, de seguridad o incluso éticos.
- El sistema evoluciona en el tiempo muy rápidamente o lentamente. Un ejemplo sería el estudio del proceso de desertificación de un área previamente fértil, un proceso realmente lento ya que implica la degradación progresiva del suelo a la multitud de factores en las que el estudio en el sistema real puede dar lugar a décadas o incluso siglos.

También ha de tenerse en cuenta que la elección de simulación como herramienta de estudio de un sistema conlleva una serie de desventajas tanto dependientes del sistema como de la herramienta:

- **Errores dependientes del modelo.** Debemos tener en cuenta que el proceso de experimentación se realiza con el modelo y no con el sistema real, por lo que una mala modelización del sistema o un uso incorrecto del modelo darán resultados incorrectos.
- **Errores de precisión.** Un ordenador solo puede retener un número fijo de cifras significativas durante los cálculos.
- **Coste asociado a la creación del modelo.** La modelización de un sistema puede llegar a ser un proceso complejo y costoso.
- **Incertidumbre en los resultados.** Al simular los modelos normalmente se plantean situaciones que nunca han sido probadas en el sistema real, esto supone no tener información para estimar la correspondencia entre los datos obtenidos por el modelo y el sistema real.

2.4. Modelos de Simulación Continuos

Los **modelos de simulación continua** son un tipo de modelos de **simulación dinámica**, en el que representan un sistema a medida que evoluciona con el tiempo. Tal y como introducimos previamente, las variables de estado en este tipo de modelos cambian de forma continua con el tiempo. Matemáticamente, el estado cambia en una cantidad infinita de puntos en el tiempo.

En estos modelos, el comportamiento del sistema se describe por la evolución de las variables de estado a lo largo del tiempo, a partir de un conjunto de condiciones iniciales y, en muchos casos, condiciones de frontera. La evolución de este tipo de sistemas se encuentra gobernada por leyes físicas, químicas, biológicas o de otro tipo, expresadas matemáticamente.

A diferencia de los **modelos de simulación discreta**, donde el cambio de estado ocurre en momentos puntuales y específicos, los modelos continuos consideran un cambio fluido y permanente, lo cual permite captar con mayor precisión fenómenos naturales como, por ejemplo:

- El crecimiento poblacional.
- La propagación del calor o sustancias químicas.
- La dinámica de fluidos.
- Sistemas eléctricos y mecánicos.

Para la representación matemática de estos fenómenos, se emplean **ecuaciones diferenciales ordinarias** o, en casos más complejos, **sistemas de ecuaciones diferenciales**, que permiten modelar como varían las magnitudes involucradas a lo largo del tiempo o del espacio. Estas ecuaciones describen relaciones entre una o varias funciones desconocidas y sus derivadas, reflejando el comportamiento dinámico del sistema [3].

EN el análisis de este tipo de modelos, se recurre a la resolución mediante técnicas **analíticas** o **numéricas**, dependiendo de la naturaleza del sistema y de los objetivos del estudio. Cuando sea posible, se antepone la resolución analítica frente a la resolución numérica.

Una resolución analítica proporciona una expresión explícita del comportamiento del sistema en función del tiempo y los parámetros del modelo. Esta solución simbólica permite realizar un análisis detallado del sistema, identificar puntos de equilibrio, inestabilidades, comportamientos periódicos o tendencias asintóticas, facilitando la comparación entre diferentes escenarios mediante la manipulación algebraica de la solución general. Sin embargo, solo es posible utilizar este método para **modelos simples** o con **condiciones ideales** [3].

Tal y como se expone en *Chapra y Canale* [3], la resolución analítica se vuelve inviable o incluso imposible cuando el modelo contiene:

- **Geometrías irregulares.** Las soluciones analíticas de ecuaciones diferenciales suelen obtenerse bajo la suposición de que el dominio espacial del problema presenta una forma geométrica regular. En estos casos, se pueden aplicar técnicas como la **separación de variables** o el **uso de funciones ortogonales específicas**. Sin embargo cuando el sistema tiene una geometría compleja, dichas técnicas dejan de ser aplicables. La falta de simetría y la imposibilidad de encontrar una base funcional adecuada para el dominio impiden el desarrollo de una solución en forma cerrada.
- **Condiciones de frontera complejas.** Las condiciones en los bordes del dominio son fundamentales para formular correctamente un modelo. Si estas condiciones cambian con el tiempo, varían especialmente de forma no uniforme, o dependen de la propia solución del problema (no lineales), no pueden abordarse mediante técnicas analíticas convencionales.
- **No linealidad.** Estas ecuaciones no permiten aplicar principios clave en muchas de técnicas analíticas (superposición, ...). Aunque si bien es cierto que existen soluciones aproximadas o métodos específicos para ciertos casos particulares, en la mayoría de los problemas reales con no linealidades se requiere otro tipo de enfoque.
- **Variables acopladas.** En la mayoría de los sistemas complejos, las variables interactúan entre sí y evolucionan simultáneamente. Esto genera sistemas de ecuaciones diferenciales acopladas, donde cada ecuación depende de varias variables al mismo tiempo. Resolver este tipo de sistemas de forma analítica se vuelve extremadamente complejo, o puede llegar a ser imposible, especialmente cuando también hay no linealidades o condiciones de frontera complejas.

En estos casos, la resolución mediante métodos numéricos resulta ser la opción ideal.

2.5. Métodos numéricos.

Los métodos numéricos constituyen un conjunto de herramientas y algoritmos destinados a realizar cálculos complejos cuando las soluciones analíticas son inviables de obtener.

Su funcionamiento se basa principalmente en cálculos iterativos que generan aproximaciones sucesivas a la solución, empleando representaciones aproximadas de operaciones [3]. Estos métodos simplifican matemáticas avanzadas a operaciones aritméticas básicas implementadas normalmente mediante ordenadores.

Como ya se ha mencionado anteriormente, las **ecuaciones diferenciales ordinarias** (EDO), son herramientas fundamentales para modelar el comportamiento dinámico de sistemas continuos. De forma general, una EDO expresa cómo varía una variable dependiente en función de una sola variable independiente, normalmente el tiempo. Su forma más común es:

$$\frac{dy}{dt} = f(t, y) \quad (2.1)$$

donde $y(t)$ representa una magnitud de interés (como la temperatura o velocidad) y $f(t, y)$ describe la ley que rige su evolución.

En el contexto de la simulación continua, donde se modelan sistemas dinámicos cuyas variables de estado cambian de forma continua en el tiempo, los métodos numéricos son fundamentales [3].

A continuación, se describen los métodos numéricos utilizados en este trabajo, detallando su funcionamiento y los errores asociados a cada uno.

2.5.1. El método de Euler.

El **método de Euler** es un enfoque numérico simple para resolver ecuaciones diferenciales ordinarias (EDOs).

El principio fundamental de este método es utilizar la pendiente de la función en un punto dado para proyectar linealmente y estimar el valor en el siguiente punto. Matemáticamente se puede representar como:

$$y_{i+1} = y_i + h * f(x_i, y_i) \quad (2.2)$$

donde:

- y_i es el valor actual de la variable interdependiente en x_i .
- y_{i+1} es el valor estimado en x_{i+1} .
- x_i es el valor actual de la variable independiente.
- h es el tamaño del paso o distancia entre dos valores consecutivos de x : $h = x_{i+1} - x_i$.
- $f(x_i, y_i)$ es una estimación de la pendiente promedio sobre el intervalo, calculada a partir de la EDO original.

Esta expresión se puede aplicar paso a paso para estimar valores futuros y trazar la trayectoria de la solución de la EDO, es decir, a partir de (x_0, y_0) se calcula y_1 , y así sucesivamente, construyendo una aproximación escalonada [3].

Errores asociados

Un factor importante a tener en cuenta en el uso de métodos de integración numérica es el **error de truncamiento**. Este error se origina debido a la naturaleza de la aproximación empleada. En particular, el método solo utiliza un número finito de términos de la expansión de la serie de Taylor. Concretamente, el método de Euler únicamente utiliza el primer término no constante de la expansión de la serie de Taylor, los términos de orden superior que no se incluyen constituyen el error de truncamiento.

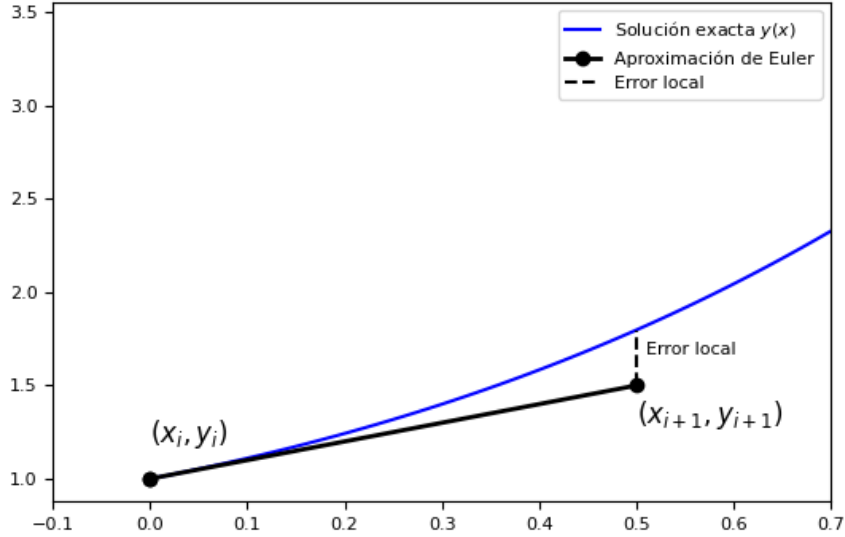


Figura 2.1: Método de Euler. Error Local

El **error de truncatura** se compone de dos partes:

- **Local.** Es el error cometido en cada paso individual.
- **Propagado.** Resultante de las aproximaciones hechas en pasos anteriores.

La suma de ambos es el **error de truncatura**. Para pasos lo suficientemente pequeños h , el error de truncamiento local aproximado de Euler es $O(h^2)$, es decir, que el error local es proporcional al cuadrado del tamaño del paso.

Esto se puede explicar partiendo de la solución de $y(x)$ en la expansión de la serie de Taylor de la alrededor del punto x_i :

$$y(x_{i+1}) = y(x_i) + hy'(x_i) + \frac{h^2}{2}y''(\xi), \quad \text{para algún } \xi \in (x_i, x_{i+1}). \quad (2.3)$$

Teniendo en cuenta que la ecuación diferencial es $y' = f(x, y)$ se tiene:

$$y(x_{i+1}) = y(x_i) + hf(x_i, y_i) + \frac{h^2}{2}y''(\xi). \quad (2.4)$$

Sin embargo, el método de Euler proporciona la siguiente aproximación, utilizando únicamente el primer término no constante:

$$y_{i+1} = y_i + hf(x_i, y_i). \quad (2.5)$$

Siendo el error local cometido en un único paso:

$$\text{Error local} = y(x_{i+1}) - y_{i+1} = \frac{h^2}{2}y''(\xi). \quad (2.6)$$

lo que demuestra que el error local de truncamiento del método de Euler es de orden $O(h^2)$. Sin embargo el error global es de $O(h)$.

Por tanto, para obtener un buen nivel de precisión, es necesario fijar pasos muy pequeños lo que requiere un gran esfuerzo computacional.

2.5.2. Heun.

El método de Heun es una de las versiones mejoradas del método de Euler. Este método es considerado como una técnica Runge-Kutta de segundo orden.

La idea central del método es utilizar el promedio de las dos estimaciones de la pendiente: una en el comienzo del intervalo y otra al final del intervalo.

1. **Paso Predictor:** Primero, se utiliza la expresión de Euler estándar para hacer una predicción inicial del valor en el punto futuro x_{i+1} .

$$y_{i+1}^{\text{pred}} = y_i + f(x_i, y_i) * h \quad (2.7)$$

2. **Paso Corrector:** Luego, se calcula la pendiente en el punto futuro x_{i+1} utilizando el valor predicho y_{i+1} . Esta pendiente es $f(x_{i+1}, y_{i+1})$. A continuación se promedian ambas pendientes y se usa para calcular el valor corregido y final en x_{i+1} .

$$y_{i+1} = y_i + \frac{h}{2} \left[f(x_i, y_i) + f(x_{i+1}, y_{i+1}^{\text{pred}}) \right] \quad (2.8)$$

Este paso corrector puede aplicarse de forma iterativa, usando el valor corregido actual para calcular una nueva pendiente en x_{i+1} y recalculando una estimación aún más definida para y_{i+1} . Sin embargo, incluso sin iterar el corrector (conocido como “*Simple Heun*”), el método proporciona una mejora significativa frente a Euler.

Errores Asociados

A diferencia de Euler, el método de Heun (con o sin iteración del corrector) su error de truncamiento local es $O(h^3)$, y su error de truncamiento global es $O(h^2)$. Esto significa que al reducir el tamaño del paso h , el error global disminuye a una tasa cuadrática, una mejora considerable frente a la tasa lineal que caracterizaba el método de Euler.

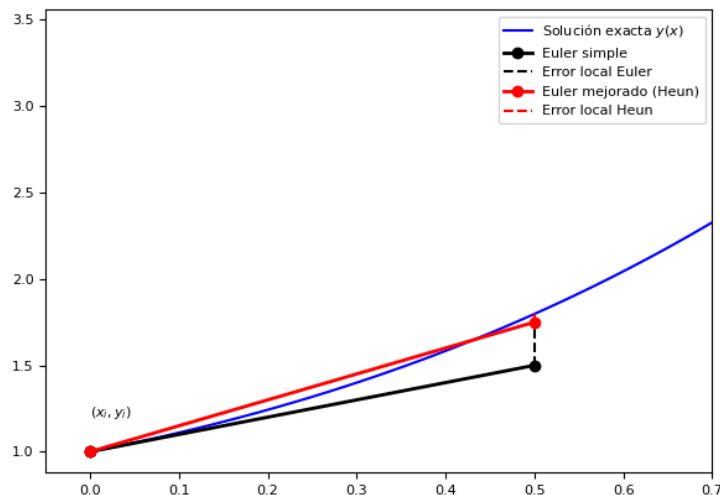


Figura 2.2: Método de Heun vs Euler. Error Local

2.5.3. Runge-Kutta de Cuarto Orden (RK-4).

El método de Runge-Kutta de cuarto orden busca obtener una mayor precisión que los métodos de órdenes inferiores como Euler o Heun.

A diferencia de los métodos anteriores, este utiliza una combinación ponderada de varias estimaciones de la pendiente dentro del intervalo de integración. Estas estimaciones se calculan en distintos puntos y luego son combinadas para una mejor aproximación de la solución.

La formulación matemática del método de Runge-Kutta de cuarto orden se expresa:

$$y_{i+1} = y_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \quad (2.9)$$

siendo procedimiento de cálculo de las estimaciones el siguiente:

$$\begin{aligned} k_1 &= f(x_i, y_i), \\ k_2 &= f\left(x_i + \frac{h}{2}, y_i + \frac{h}{2}k_1\right), \\ k_3 &= f\left(x_i + \frac{h}{2}, y_i + \frac{h}{2}k_2\right), \\ k_4 &= f(x_i + h, y_i + hk_3) \end{aligned} \quad (2.10)$$

Los pesos asignados a cada una de las estimaciones intermedias en el método de Runge-Kutta de cuarto orden se eligen de forma que la combinación lineal de estas pendientes reproduzca la expansión en serie de Taylor de la solución exacta de la ecuación diferencial hasta términos de cuarto orden. De esta manera, se garantiza una mayor precisión sin necesidad de calcular derivadas de orden superior de $f(x, y)$.

La solución exacta de una ecuación diferencial ordinaria puede aproximarse mediante su desarrollo en serie de Taylor alrededor del punto x :

$$y(x+h) = y(x) + hy'(x) + \frac{h^2}{2!}y''(x) + \frac{h^3}{3!}y^{(3)}(x) + \frac{h^4}{4!}y^{(4)}(x) + \dots \quad (2.11)$$

El método de Runge-Kutta de cuarto orden busca aproximar esta expresión utilizando únicamente evaluaciones de la función $f(x, y)$, sin requerir el cálculo explícito de derivadas de orden superior.

Errores Asociados

El método de Runge-Kutta de cuarto orden **error de truncatura local** de orden $O(h^5)$ y un **error de truncatura global** de orden $O(h^4)$. Esto lo hace significativamente más preciso que los métodos de orden inferior, como el método de Euler o el método de Heun.

2.5.4. Limitaciones de los Métodos de Paso Fijo.

Los métodos de Runge-Kutta de paso fijo utilizan un tamaño de paso constante a lo largo de todo el intervalo de integración. Por tanto, si la solución de la EDO presenta regiones donde cambia rápidamente y otras donde es más suave, el uso un tamaño de paso fijo presenta los siguientes problemas:

- Un paso de tamaño grande no capturaría adecuadamente los cambios bruscos en la solución y acumular un error de truncamiento local considerable en esas regiones.

- Un paso de tamaño pequeño para garantizar la precisión en las regiones donde los cambios son considerables conlleva a desperdiciar una gran cantidad de recursos en las regiones donde la solución varía lentamente.

Estos inconvenientes motivan el uso de métodos adaptativos, como el método de Runge-Kutta-Fehlberg (RKF-45), que ajustan automáticamente el tamaño del paso en función del comportamiento local de la solución.

2.5.5. Runge-Kutta-Fehlberg (RKF-45).

El método Runge-Kutta-Fehlberg es una extensión adaptativa del método de Runge-Kutta clásico, que introduce una característica crucial: **la adaptabilidad del tamaño del paso**.

La adaptabilidad del paso busca solventar el error de truncamiento durante la computación. Este método hace uso de un método Runge-Kutta de quinto orden que emplea las evaluaciones de la función de un método Runge-Kutta de cuarto orden, pudiendo estimar el error basándose en seis evaluaciones de la función.

Las evaluaciones serán calculadas a partir de las constantes que se indican en la Tabla de Butcher del propio método:

0						
$\frac{1}{4}$	$\frac{1}{4}$					
$\frac{3}{8}$	$\frac{3}{32}$	$\frac{9}{32}$				
$\frac{12}{13}$	$\frac{1932}{2197}$	$-\frac{7200}{2197}$	$\frac{7296}{2197}$			
1	$\frac{439}{216}$	-8	$\frac{3680}{513}$	$-\frac{845}{4104}$		
$\frac{1}{2}$	$-\frac{8}{27}$	2	$-\frac{3544}{2565}$	$\frac{1859}{4104}$	$-\frac{11}{40}$	
	$\frac{16}{135}$	0	$\frac{6656}{12825}$	$\frac{28561}{56430}$	$-\frac{9}{50}$	$\frac{2}{55}$
	$\frac{25}{216}$	0	$\frac{1408}{2565}$	$\frac{2197}{4104}$	$-\frac{1}{5}$	0

Cuadro 2.1: Tabla de Butcher del método Runge-Kutta-Fehlberg 4(5)

Estas seis evaluaciones son denotadas como k_1 a k_6 , se calculan utilizando la función f que define el sistema diferencial, según las siguientes expresiones:

$$k_1 = f(x_i, y_i) \quad (2.12)$$

$$k_2 = f(x_i + a_2h, y_i + b_{21}k_1h) \quad (2.13)$$

$$k_3 = f(x_i + a_3h, y_i + b_{31}k_1h + b_{32}k_2h) \quad (2.14)$$

$$k_4 = f(x_i + a_4h, y_i + b_{41}k_1h + b_{42}k_2h + b_{43}k_3h) \quad (2.15)$$

$$k_5 = f(x_i + a_5h, y_i + b_{51}k_1h + b_{52}k_2h + b_{53}k_3h + b_{54}k_4h) \quad (2.16)$$

$$k_6 = f(x_i + a_6h, y_i + b_{61}k_1h + b_{62}k_2h + b_{63}k_3h + b_{64}k_4h + b_{65}k_5h) \quad (2.17)$$

Una vez obtenidos los resultados intermedios podemos obtener las dos soluciones:

$$y^{(5)} = y_i + h \sum_{i=1}^6 c_{5i} k_i \quad (2.18)$$

$$y^{(4)} = y_i + h \sum_{i=1}^6 c_{4i} k_i \quad (2.19)$$

$$\varepsilon = |y^{(5)} - y^{(4)}| \quad (2.20)$$

El error estimado en cada paso, determina si la integración en ese paso es válida o no. Si el error es menor o igual a la tolerancia especificada, el paso se acepta y se avanza, aumentando el tamaño del paso. En caso contrario, si el error supera la tolerancia, el paso se rechaza, se reduce el tamaño del paso y se repite la integración en el mismo punto, garantizando así la precisión requerida.

El nuevo paso es calculado según el error estimado obtenido:

$$h_{\text{nuevo}} = S \cdot h \cdot \left(\frac{\text{TOL}}{|\varepsilon|} \right)^{\frac{1}{p+1}} \quad (2.21)$$

donde:

- S es un factor de seguridad, típicamente $0 < S < 1$,
- p es el orden del método de menor precisión (para RKF45, $p = 4$),
- $\varepsilon = |y^{(5)} - y^{(4)}|$ es la estimación del error local,
- TOL es la tolerancia de error establecida por el usuario.

Capítulo 3

Estado del Arte

En este capítulo establecemos las bases conceptuales y tecnológicas utilizadas en el desarrollo del proyecto. Para ello, se analiza el estado actual del conocimiento en torno al dominio del problema, las metodologías existentes aplicables, las tecnologías disponibles para la implementación y los trabajos previos relacionados con la temática abordada.

En primer lugar, se describe el dominio del problema, indicando los desafíos y la necesidad de automatizar la generación de programas de simulación.

Posteriormente, se exponen las principales metodologías empleadas en el desarrollo de herramientas de software, así como las tecnologías más relevantes para su implementación.

Finalmente, se revisan trabajos relacionados que abordan problemáticas similares con el fin de identificar enfoques existentes.

3.1. Descripción del dominio del problema

La generación de programas orientados a la simulación continua aborda la intersección entre modelado matemático, automatización del desarrollo software, simulación numérica y análisis de resultados. Este dominio tiene como objetivo facilitar la construcción de programas que contengan modelos capaces de representar el comportamiento dinámico de sistemas reales, su posterior implementación automática en lenguajes de programación y el análisis de los resultados obtenidos de las distintas simulaciones.

En multitud de áreas del conocimiento, como la ingeniería, la física, la biología y la economía, los sistemas que evolucionan de forma continua se modelan mediante ecuaciones diferenciales ordinarias. La resolución numérica de estos modelos es fundamental para su análisis y simulación, lo que implica el diseño manual del código, la elección e incorporación del método integración numérica adecuado y la gestión de estructuras de datos que representen el modelo.

Sin embargo, este proceso resulta complejo y propenso a errores, especialmente en los siguientes casos:

- **Falta de experiencia en programación.** Muchos usuarios que formulan este tipo de modelos no poseen conocimientos avanzados en lenguajes de programación ni en estructuras de software necesarias para implementar simuladores.
- **Errores humanos en la codificación manual.** La implementación directa de modelos matemáticos en código fuente conlleva un alto riesgo de errores, como una transcripción inadecuada de las ecuaciones, uso incorrecto de variables, definición inadecuada de condiciones iniciales o fallos en la configuración del paso de integración, comprometiendo así la validez de los resultados de la simulación.

En el contexto actual, caracterizado por la necesidad de soluciones rápidas, reutilizables y estandarizadas, resulta fundamental que instituciones, investigadores y profesionales cuenten con herramientas que optimicen el proceso de construcción de modelos computacionales. Estas soluciones mejoran la productividad, garantizan la correcta unión entre el modelo teórico y su implementación, y promueven la estandarización del desarrollo de simulaciones, contribuyendo al avance científico y tecnológico en diversos campos.

3.2. Metodologías potenciales

El desarrollo de un software implica definir qué metodologías existen para poder elegir la más adecuada según las necesidades del proyecto.

Según *Roger S. Pressman* [18], las metodologías de desarrollo que encontramos son:

- **Modelo en Cascada.** Se trata de una metodología de desarrollo secuencial en la que el proyecto avanza de forma lineal a través de diferentes fases: requisitos, diseño, implementación, verificación y mantenimiento. Cada etapa debe completarse antes de pasar a la siguiente, lo que por un lado facilita la planificación pero limita la flexibilidad ante cambios posteriores.
- **Modelo Incremental.** El desarrollo del software se divide en pequeñas entregas funcionales denominadas “incrementos”. Estos incrementos añaden funcionalidades al sistema de forma gradual, comenzando con una versión básica y completando el software en sucesivas iteraciones. Esta metodología permite obtener versiones utilizables de forma temprana, facilitando la retroalimentación continua y mejora la gestión del riesgo en comparación a los métodos secuenciales.
- **Modelo en Espiral.** Esta metodología se centra en la gestión del riesgo. La construcción del software se realiza en diferentes y sucesivos ciclos. En cada uno de estos ciclos el sistema es contruido y refinado mediante las fases de planificación, análisis de riesgos, diseño, implementación y evaluación. En cada ciclo se identifican los objetivos, se analizan los posibles riesgos, se desarrollan prototipos o versiones parciales del sistema, y se planifica la siguiente fase. Esta estructura permite adaptar el desarrollo a medida que se obtienen nuevos requisitos o se identifican problemas, siendo muy útil en proyectos grandes, complejos o con mucha incertidumbre.
- **Metodologías Ágiles.** Se tratan de enfoques de desarrollo software centrados en la entrega continua de valor, la colaboración constante con el cliente y la adaptación al cambio. A diferencia de los métodos tradicionales, se promueve ciclos cortos de desarrollo e iterativos llamados *sprints*, en los que se entrega el software funcional de forma incremental. En estas metodologías se prioriza la comunicación directa, la retroalimentación continua y la simplicidad en los procesos. Existen diferentes tipos tales como *Scrum*, *Kanban*, *XP* (Extreme Programming) y *Lean Software Development*. Estas metodologías son útiles y efectivas en entornos cambiantes o en aquellos en los que los requisitos evolucionan con el tiempo.

Atendiendo a las características del proyecto, el desarrollo del software mediante el **modelo en cascada** es el más conveniente debido a los siguientes motivos:

- **Requisitos bien definidos.** El proyecto parte de objetivos claros y estables desde el principio de su desarrollo.
- **Estructura clara del desarrollo.** Esta metodología permite avanzar paso a paso a través de la diferentes fases, facilitando el seguimiento del progreso.

- **Ausencia de cliente para la retroalimentación.** Dado que el proyecto no cuenta con un cliente del que recibir una retroalimentación constante, no se requiere una adaptación a las nuevas demandas, lo que refuerza el uso de un modelo lineal.

3.3. Tecnologías potenciales

El estudio de las tecnologías disponibles para el desarrollo de la herramienta es un paso fundamental, ya que definirá las posibilidades y limitaciones en cuanto a rendimiento, escalabilidad, mantenimiento y compatibilidad multiplataforma. Para ello, es necesario analizar distintos lenguajes de programación, frameworks para interfaces gráficas, el uso o no de bibliotecas de integración numérica y las herramientas de empaquetado disponibles.

La primera cuestión a abordar es la elección de la plataforma en la que implementar la herramienta, es decir, si se va a tratar de una herramienta web o una aplicación de escritorio. Para ello se muestran los siguientes criterios por los que se va a evaluar la conveniencia de cada opción:

- **Accesibilidad.**
 - **Aplicación Web.** Tiene la ventaja de ser accesible desde cualquier dispositivo con un navegador web y conexión a internet, sin necesidad de instalación local.
 - **Aplicación de escritorio.** El software únicamente se encontrará disponible en aquellos dispositivos que contengan la aplicación desplegada.
- **Interacción con el sistema operativo.**
 - **Aplicación Web.** Opera bajo un entorno restringido denominado “*sandbox*” que limita el acceso a funciones del sistema operativo.
 - **Aplicación escritorio.** Tiene acceso completo al sistema operativo.
- **Rendimiento.**
 - **Aplicación Web.** Se encuentra limitada por las capacidades del navegador. Las tareas computacionalmente intensivas, pueden verse afectadas negativamente.
 - **Aplicación de escritorio.** Permite un uso más eficiente de los recursos del sistema, con soporte completo para bibliotecas de alto rendimiento y posibilidad de optimización a nivel de sistema operativo.
- **Distribución y actualización.**
 - **Aplicación Web.** No necesita instalación y se actualiza automáticamente desde el servidor.
 - **Aplicación de escritorio.** Requiere una distribución manual y es necesario establecer un mecanismo de actualización si se desea distribuir nuevas versiones.
- **Experiencia de usuario.**
 - **Aplicación web.** Permiten interfaces flexibles y adaptables.
 - **Aplicación de Escritorio.** Permite un control más directo de la interfaz junto con la integración con componentes del sistema.
- **Trabajo offline**
 - **Aplicación Web.** En la mayoría de casos requieren una conexión permanente a internet. Existen soluciones PWA (Progressive Web Apps), aunque su implementación es más compleja y limitada.

- **Aplicación escritorio.** Funciona de manera completamente autónoma una vez instalada.

En el caso de este proyecto, se ha realizado una **evaluación comparativa** entre una posible implementación como aplicación web o de escritorio, considerando los criterios previamente definidos. Si bien las aplicaciones web permiten el acceso multiplataforma desde cualquier navegador sin necesidad de instalación, en este proyecto no se prioriza la disponibilidad remota, sino que se valora positivamente un entorno controlado y funcional localmente, por lo que esta ventaja resulta secundaria en términos de **accesibilidad**. En cuanto a la **interacción con el sistema operativo**, una aplicación de escritorio nos permite ejecutar scripts, compilar código en distintos lenguajes y acceder a archivos locales, funciones que están restringidas en un entorno web por políticas de seguridad del navegador. Abordando el **rendimiento**, las simulaciones numéricas y los métodos de integración requieren una cantidad de cómputo considerable y bibliotecas específicas que no pueden aprovecharse completamente en un entorno web, mientras que en una aplicación de escritorio no tenemos esos inconvenientes y además permite optimizar el uso de recursos y ejecutar procesos de forma directa. En cuanto a la **experiencia de usuario**, si bien las interfaces web ofrecen mayor flexibilidad, una aplicación de escritorio garantiza una mayor integración con el sistema permitiendo la gestión de archivos, control de errores y exportación/visualización de resultados de una forma más directa. Por último, en relación con la **desconexión**, la capacidad de ejecutar la herramienta sin necesidad de conexión a internet es esencial para garantizar su uso en laboratorios, aulas o entornos con conectividad limitada, lo que refuerza de forma más acentuada la elección del entorno de escritorio como el más adecuado para este proyecto.

Otro aspecto a evaluar es el **lenguaje de programación** en el que se va a desarrollar el proyecto. Como opciones se consideran Python, Java y C++, por ser lenguajes ampliamente conocidos y dominados en el entorno del autor, lo que permite un mayor control durante las fases de desarrollo.

Python destaca por su sintaxis clara y concisa, que favorece una rápida implementación y facilita el mantenimiento del código. Además, ofrece un ecosistema muy completo de bibliotecas, tanto para la construcción de interfaces gráficas (como *Tkinter* o *CustomTkinter*), manipulación de archivos, automatización de procesos (*subprocess*) y tratamiento de expresiones matemáticas simbólicas (*SymPy*). A esto se le incluye la gran comunidad que está continuamente ampliando las funcionalidades ofrecidas por el propio lenguaje.

En comparación, **C++** tiene una mayor complejidad sintáctica que puede influir en el tiempo de desarrollo. Un aspecto importante podría ser la gestión manual de la memoria, necesaria para proyectos donde la eficiencia es uno de los principales requisitos. Sin embargo, este no es el caso. La simulación se realizará mediante la ejecución de los programas generados por la herramienta y no por el propio programa per se.

Java presenta un equilibrio entre ambos enfoques. Su portabilidad gracias a la JVM y el soporte para interfaces gráficas lo hacen adecuado para aplicaciones de escritorio. Sin embargo, la ausencia de una biblioteca tan completa y potente como *SymPy* hace que este lenguaje no sea tan apto para el desarrollo de este proyecto.

Por estos motivos, se selecciona **Python** como lenguaje de programación principal para el desarrollo de la herramienta.

Otro factor importante a contemplar es el uso, o no, de **bibliotecas de integración numéricas** en los programas generados por la herramienta. Aunque su uso no depende directamente de la herramienta a desarrollar en este proyecto, sí influye en la legibilidad y comprensión del código fuente generado. Por ello se evita el uso de dichas bibliotecas, realizando una implementación basada en la literatura, propuesta por *J.M.A. Dandby* [4].

Por último, se evalúan las herramientas de empaquetado y distribución del software disponibles en el lenguaje de programación seleccionado. En el caso de Python, existen soluciones

que permiten distribuir tanto bibliotecas como aplicaciones de usuario final. Dado que en este proyecto se persigue la generación de un ejecutable autónomo, que no requiera de instalación previa del intérprete de Python ni de dependencias externas, es fundamental seleccionar una herramienta que facilite la generación de binarios autoejecutables multiplataforma.

Entre las alternativas más destacadas se encuentran *PyInstaller* y *cxFreeze*, ambas ampliamente utilizadas en el ecosistema de Python. *Pyinstaller* destaca por su simplicidad de uso, su alto grado de compatibilidad con diferentes sistemas operativos (Windows, Linux, macOS) y su capacidad para detectar e incluir automáticamente las dependencias necesarias del proyecto. Además, permite incluir recursos como iconos, archivos de datos y bibliotecas externas. Por otro lado, *cxFreeze* ofrece un mayor control sobre el proceso de empaquetado, aunque presenta menor soporte en sistemas Unix en comparación con *PyInstaller*. Por estas razones, se elige *PyInstaller* como herramienta principal de empaquetado.

3.4. Trabajos Relacionados

Con el fin de contextualizar el presente proyecto dentro del estado actual del conocimiento, en este apartado se analizan diversos trabajos y herramientas que abordan problemas similares o complementarios al que aquí se plantea.

- **Modelica / OpenModelica.** Modelica es un lenguaje de modelado orientado a objetos, diseñado específicamente para describir sistemas físicos complejos mediante ecuaciones diferenciales algebraicas (EDA). A diferencia de los lenguajes imperativos tradicionales, Modelica permite especificar el comportamiento del sistema de forma declarativa, es decir, sin necesidad de establecer de forma explícita el orden de ejecución. Entre sus implementaciones destaca *OpenModelica*, una plataforma de código abierto que proporciona un entorno completo para el desarrollo, simulación y análisis de modelos contruidos en este lenguaje. Aunque es muy potente para el modelado físico multidominio (eléctrico, mecánico, etc.), la necesidad de aprender una sintaxis puede ser una barrera para usuarios no especializados. Además, la exportación directa de código en otros lenguajes como *Python* o *C++* no está totalmente automatizada, lo que limita su utilidad como generador de programas.
- **Simulink.** *Simulink* es una extensión de MATLAB que permite la simulación de sistemas dinámicos mediante diagramas de bloques. Es muy utilizada en entornos industriales y académicos, especialmente en ingeniería de control, aeroespacial, electrónica y automatización. Su interfaz gráfica facilita el diseño de modelos sin necesidad de una codificación directa, lo que facilita su uso. Si bien permite la generación automática de código *C* y *C++* mediante herramientas como *Simulink Coder*, se trata de un entorno propietario en el que se requiere licencias comerciales costosas [11].
- **Scilab/Xcos.** Entorno de programación científica de código abierto similar a MATLAB, que incluye Xcos, una herramienta de simulación gráfica comparable a Simulink. Xcos permite modelar y simular sistemas dinámicos utilizando bloques y conexiones. Se trata de una alternativa a Simulink, útil en contextos educativos. Sin embargo, su ecosistema es más limitado y la personalización o generación automática de código fuente a partir de modelos no se caracteriza por ser la más potente. Esto lo aleja del enfoque de herramientas que transforman modelos matemáticos en simuladores ejecutables en múltiples lenguajes.
- **ACSL** (*Advanced Continuous Simulation Language*). Lenguaje especializado para simulación de sistemas dinámicos continuos definidos mediante ecuaciones diferenciales [13]. Su sintaxis está orientada a la representación explícita de modelos matemáticos, lo que lo convierte en una herramienta potente para usuarios con formación técnica. Sin embargo, se trata de una solución propietaria y actualmente desactualizada, sin soporte activo ni

adaptación a entornos modernos de desarrollo. A esto se le añade que su capacidad para generar programas autónomos en múltiples lenguajes es limitada, lo que lo aleja del enfoque de automatización y portabilidad que persigue el presente proyecto.

- **MIST** (*MicroSimulation Tool*). Es una herramienta de simulación basada en técnicas de Monte Carlo, orientada al análisis de sistemas estocásticos a nivel individual, como los modelos de comportamiento poblacional en sanidad o economía. Permite definir reglas y parámetros para representar dinámicas complejas que evolucionan a lo largo del tiempo mediante simulaciones individualizadas. Aunque resulta útil en estudios donde la aleatoriedad y la variabilidad individual son relevantes, su naturaleza discreta y su enfoque en la simulación probabilística la distancian del objetivo de este proyecto.

Como se ha expuesto, existen una gran cantidad de herramientas y entornos de simulación que permiten abordar problemas de simulación continua desde diferentes enfoques. No obstante, muchas de estas soluciones están orientadas a contextos más amplios y/o especializados, como el modelado físico multidominio, la simulación de sistemas embebidos o el análisis mediante métodos estocásticos. Además, muchas de ellas presentan limitaciones en cuanto a accesibilidad, dependencia de entornos propietarios, complejidad de uso o falta de automatización completa en la generación de código en múltiples lenguajes. En consecuencia, resulta complicado encontrar una herramienta que se enfoque específicamente en la automatización del flujo completo, desde la introducción simbólica de ecuaciones diferenciales hasta la generación directa de programas ejecutables. Por ello, este proyecto propone una solución más acotada y especializada, que facilite el desarrollo de simulaciones continuas.

Capítulo 4

Análisis

Una vez desarrolladas las bases teóricas del proyecto y expuesto el objetivo que se persigue con su desarrollo, se procede a analizar los requisitos que han sido necesarios desde la perspectiva de la ingeniería del software.

En primer lugar, se exponen los **requisitos funcionales y no funcionales**, definiendo así las acciones que el sistema debe ser capaz de realizar, los criterios de calidad, rendimiento y otros atributos importantes para su correcto funcionamiento.

A continuación, se abordan los **requisitos de datos e información**, en los que se especifica el tipo, la estructura y el tratamiento de datos necesarios para que el sistema funcione acorde a sus objetivos.

Posteriormente, se desarrollan los **casos de uso** que detallan los diferentes escenarios de interacción entre el usuario y el sistema, especificando las funcionalidades y el flujo de acciones asociado a cada uno. Estos casos de uso se acompañarán de sus respectivos diagramas de secuencia permitiendo así una mejor comprensión del funcionamiento .

Por último lugar, se presentan los aspectos relativos a la **planificación** y el **presupuesto** del proyecto, en el que se incluirá los recursos empleados, estimación de costes, cronograma de desarrollo y las fases que han estructurado los procesos siguiendo la metodología seleccionada.

4.1. Requisitos funcionales

Los **requisitos funcionales** que definen el sistema son:

RF.1 La **generación de programas de simulación continua** a partir de ecuaciones diferenciales y condiciones.

RF.1.1 Generación de código fuente en C++, Python o Java según la configuración seleccionada.

RF.2 La **introducción, edición y eliminación** de ecuaciones diferenciales.

RF.3 La **introducción, edición y eliminación** de las condiciones que afecten a la variables del modelo.

RF.4 Seleccionar el **lenguaje de programación** de salida del código fuente.

RF.5 Seleccionar el **método de integración numérica**.

RF.6 **Simular el modelo**, mediante la ejecución del código fuente generado.

RF.7 Introducción de las características de la simulación personalizada al modelo.

RF.7.1 Definición de las condiciones iniciales de las variables.

RF.7.2 Definición del valor de las constantes.

RF.7.3 Definición del tamaño del paso o tolerancia del método numérico usado.

RF.8 **Compilación** del código fuente generado por el sistema.

RF.9 Guardado en un archivo de configuración de las características del modelo.

RF.9.1 Almacenamiento de las ecuaciones junto con las variables y constantes que las definen.

RF.9.2 Almacenamiento de las condiciones junto con la expresión lógica, acciones, variables y constantes que las forman.

RF.9.3 Almacenamiento del lenguaje y método numérico seleccionado.

RF.10 Cargado del archivo de configuración.

RF.11 Eliminar los datos actuales para iniciar una nueva definición de un modelo.

4.2. Requisitos no funcionales

Los **requisitos no funcionales** del sistema son:

RNF.1 Debe tener una interfaz gráfica intuitiva y amigable.

RNF.2 Facilidad a la hora de la implementación de nuevos métodos numéricos.

RNF.3 El código debe estar documentado y organizado de forma modular, facilitando así la mantenibilidad.

RNF.4 La generación y compilación del código deben realizarse en tiempos razonables.

RNF.5 Validación de las entradas del usuario, previniendo los errores en la generación y ejecución del código fuente generado.

RNF.6 Los programas generados deben poder ejecutarse en compiladores comunes (C++, Java) e interpretes (Python).

RNF.7 Mensajes de confirmación y/o error de acciones en el uso de la aplicación.

4.3. Requisitos de información

El sistema debe tratar con los siguientes datos en su funcionamiento:

RI.1 Ecuaciones diferenciales ordinarias (EDO) que define el usuario.

RI.2 Variables de las ecuaciones y condiciones.

RI.3 Constantes de las ecuaciones y condiciones.

RI.4 Condiciones asociadas a algunas de las variables del sistema.

RI.5 Valores iniciales de las variables.

RI.6 Valores de las constantes.

RI.7 Lenguaje de programación seleccionado.

RI.8 Método de integración seleccionado.

RI.9 Archivos de configuración generados.

RI.10 Tiempo total de la simulación.

RI.11 Resultados de las ejecuciones de los programas generados.

RI.12 Directorio donde se almacena el código fuente generado.

RI.13 Mensajes de error/éxito adaptados a cada situación.

4.4. Casos de Uso

Los casos de uso obtenidos a partir de los requisitos que previamente se han definido son los siguientes:

ID	CU-001
Título	Añadir ecuación
Referencia a RF	RF-2
Actor	Usuario
Precondiciones	No existen precondiciones.
Postcondiciones	La ecuación queda añadida al sistema.
Flujo Principal	<ol style="list-style-type: none"> 1. El usuario selecciona “Añadir ecuación”. 2. El sistema actualiza la ventana y muestra los siguientes campos de entrada: <ul style="list-style-type: none"> • Campo expresión. • Campo Variables. • Campo Constantes. 3. El usuario introduce la expresión de la ecuación. <ol style="list-style-type: none"> 3.1. El sistema actualiza la vista previa de la ecuación por cada pulsación del usuario. 4. El usuario introduce las variables y constantes que componen la ecuación. 5. El sistema verifica las entradas del usuario. <ol style="list-style-type: none"> 5.1. Si los datos son correctos: <ul style="list-style-type: none"> • El sistema añade la ecuación al modelo mostrando un mensaje de éxito. • El sistema actualiza la ventana al menú principal. 5.2. Si los datos no son correctos: <ul style="list-style-type: none"> • El sistema muestra un mensaje con el error indicado.
Flujos Alternativos	<p>A1 : Si el usuario pulsa el botón de “Añadir” sin introducir la expresión:</p> <ul style="list-style-type: none"> • El sistema muestra un mensaje de error “Introduzca una expresión válida.”. <p>A2 : Si el usuario pulsa el botón de “Cancelar” en la vista de ecuación:</p> <ul style="list-style-type: none"> • El sistema actualiza la vista al menú principal.

ID	CU-002
Título	Editar ecuación
Referencia a RF	RF-2
Actor	Usuario
Precondiciones	Debe haber al menos una ecuación añadida al sistema.
Postcondiciones	La ecuación seleccionada es modificada en el sistema.
Flujo Principal	<ol style="list-style-type: none"> 1. El usuario selecciona la ecuación de la lista de ecuaciones añadidas al sistema y selecciona “Editar ecuación” . 2. El sistema actualiza la vista y carga en los campos de entrada los datos de la ecuación seleccionada. 3. El usuario edita los campos necesarios y selecciona “Editar” 4. El sistema verifica las entradas del usuario: <ol style="list-style-type: none"> 4.1. Si los datos son correctos: <ul style="list-style-type: none"> • El sistema edita la ecuación seleccionada. • El sistema actualiza la vista al menú principal. • El sistema muestra un mensaje de éxito al editar la ecuación. 4.2. Si los datos son incorrectos: <ul style="list-style-type: none"> • El sistema muestra un mensaje de error.
Flujos Alternativos	A1. Si el usuario pulsa el botón de “Cancelar” en la vista de ecuación: <ul style="list-style-type: none"> • El sistema actualiza la vista al menú principal.
Excepciones	E1. Si el usuario pulsa el botón “Editar” y no se encuentra ninguna ecuación seleccionada: <ul style="list-style-type: none"> • El sistema muestra un mensaje de error y permanece en el menú principal.

ID	CU-003
Título	Eliminar ecuación
Referencia a RF	RF-7
Actor	Usuario
Precondiciones	Debe haber al menos una ecuación añadida al sistema.
Postcondiciones	La ecuación seleccionada ya no forma parte del sistema.
Flujo Principal	<ol style="list-style-type: none"> 1. El usuario selecciona la ecuación de la lista de ecuaciones añadidas al sistema y selecciona “Eliminar ecuación” . 2. El sistema actualiza la vista. 3. El sistema muestra un mensaje de éxito al eliminar la ecuación.
Flujos Alternativos	

Excepciones	<p>E1. Si el usuario pulsa el botón “Eliminar” y no se encuentra ninguna ecuación seleccionada:</p> <ul style="list-style-type: none"> • El sistema muestra un mensaje de error y permanece en el menú principal.
--------------------	--

ID	CU-004
Título	Añadir condición
Referencia a RF	RF-3
Actor	Usuario
Precondiciones	No existen precondiciones.
Postcondiciones	La condición queda añadida en el sistema.
Flujo Principal	<ol style="list-style-type: none"> 1. El usuario selecciona “Añadir condición”. 2. El sistema actualiza la ventana y muestra los siguientes campos de entrada: <ul style="list-style-type: none"> • Campo Expresión lógica. • Campo Acción • Campo Variables. • Campo Constantes. 3. El usuario introduce la expresión lógica . <ol style="list-style-type: none"> 3.1. El sistema actualiza la vista previa de la condición por cada pulsación del usuario. 4. El usuario introduce las acciones de la condición. <ol style="list-style-type: none"> 4.1. El sistema actualiza la vista previa de la condición por cada pulsación del usuario. 5. El usuario introduce las variables y constantes que componen la ecuación. 6. El usuario pulsa el botón “Añadir” 7. El sistema verifica las entradas del usuario. <ol style="list-style-type: none"> 7.1. Si los datos son correctos: <ul style="list-style-type: none"> • El sistema añade la condición al modelo mostrando un mensaje de éxito. • El sistema actualiza la ventana al menú principal. 7.2. Si los datos no son correctos: <ul style="list-style-type: none"> • El sistema muestra un mensaje con el error indicado.
Flujos Alternativos	<p>A1 : Si el usuario pulsa el botón de “Añadir” sin introducir la expresión:</p> <ul style="list-style-type: none"> • El sistema muestra un mensaje de error “Introduzca una expresión válida.”. <p>A2 : Si el usuario pulsa el botón de “Cancelar” en la vista de condición:</p> <ul style="list-style-type: none"> • El sistema actualiza la vista al menú principal.

ID	CU-005
Título	Editar condición
Referencia a RF	RF-3
Actor	Usuario
Precondiciones	Debe haber al menos una condición añadida al sistema.
Postcondiciones	La condición se modifica en el sistema.
Flujo Principal	<ol style="list-style-type: none"> 1. El usuario selecciona la condición de la lista de condiciones añadidas al sistema y selecciona “Editar condición” . 2. El sistema actualiza la vista y carga en los campos de entrada los datos de la condición seleccionada. 3. El usuario edita los campos necesarios y selecciona “Editar” 4. El sistema verifica las entradas del usuario: <ol style="list-style-type: none"> 4.1. Si los datos son correctos: <ul style="list-style-type: none"> • El sistema edita la condición seleccionada. • El sistema actualiza la vista al menú principal. • El sistema muestra un mensaje de éxito al editar la condición. 4.2. Si los datos son incorrectos: <ul style="list-style-type: none"> • El sistema muestra un mensaje de error.
Flujos Alternativos	A1. Si el usuario pulsa el botón de “Cancelar” en la vista de condición: <ul style="list-style-type: none"> • El sistema actualiza la vista al menú principal.
Excepciones	E1. Si el usuario pulsa el botón “Editar” y no se encuentra ninguna condición seleccionada: <ul style="list-style-type: none"> • El sistema muestra un mensaje de error y permanece en el menú principal.

ID	CU-006
Título	Eliminar condición
Referencia a RF	RF-3
Actor	Usuario
Precondiciones	Debe haber al menos una condición añadida al sistema.
Postcondiciones	La condición ya no forma parte del sistema
Flujo Principal	<ol style="list-style-type: none"> 1. El usuario selecciona la condición de la lista de condiciones añadidas al sistema y selecciona “Eliminar” . 2. El sistema actualiza la vista. 3. El sistema muestra un mensaje de éxito al eliminar la condición.
Flujos Alternativos	
Excepciones	<p>E1. Si el usuario pulsa el botón “Eliminar” y no se encuentra ninguna condición seleccionada:</p> <ul style="list-style-type: none"> • El sistema muestra un mensaje de error y permanece en el menú principal.

ID	CU-007
Título	Generación de código fuente
Referencia a RF	RF-1, RF-4, RF-5
Actor	Usuario
Precondiciones	<ul style="list-style-type: none"> • El lenguaje del código fuente debe estar seleccionado. • El método de integración debe estar seleccionado. • Debe existir al menos una ecuación añadida al sistema. • Las variables que identifican al modelo deben estar presentes únicamente como resultado de una ecuación.
Postcondiciones	El sistema genera el código fuente en el directorio indicado.
Flujo Principal	<ol style="list-style-type: none"> 1. El usuario pulsa el boton “Generar” en la vista principal. 2. El sistema muestra una pantalla emergente para seleccionar el destino del codigo fuente generado. 3. El usuario selecciona el directorio destino e indica el nombre
Flujos Alternativos	

ID	CU-008
Título	Simulación de modelo
Referencia a RF	RF-6, RF-7, RF-8
Actor	Usuario
Precondiciones	El código fuente del modelo debe haberse generado correctamente.
Postcondiciones	Se genera un archivo de salida resultado de la ejecución del código fuente.
Flujo Principal	<ol style="list-style-type: none"> 1. El usuario introduce el tiempo, el tamaño del paso / tolerancia, los valores iniciales de las variables y los valores de las constantes de la simulación. 2. El usuario pulsa el botón “Simular”. 3. El sistema ejecuta el programa con los argumentos. 4. El sistema carga el archivo de salida del programa y muestra los datos de forma gráfica y de forma tabular.
Flujos Alternativos	A1. Si el usuario pulsa el botón “Salir”, la ventana correspondiente a la simulación se cerrará.
Excepciones	E1. Si el usuario pulsa el botón “Simular”, y se encuentra alguno de los campos sin valor, el sistema mostrará un error indicando que existen campos sin valor.

Finalmente, se incluye el diagrama de casos de uso que modela las principales interacciones entre el usuario y el sistema.

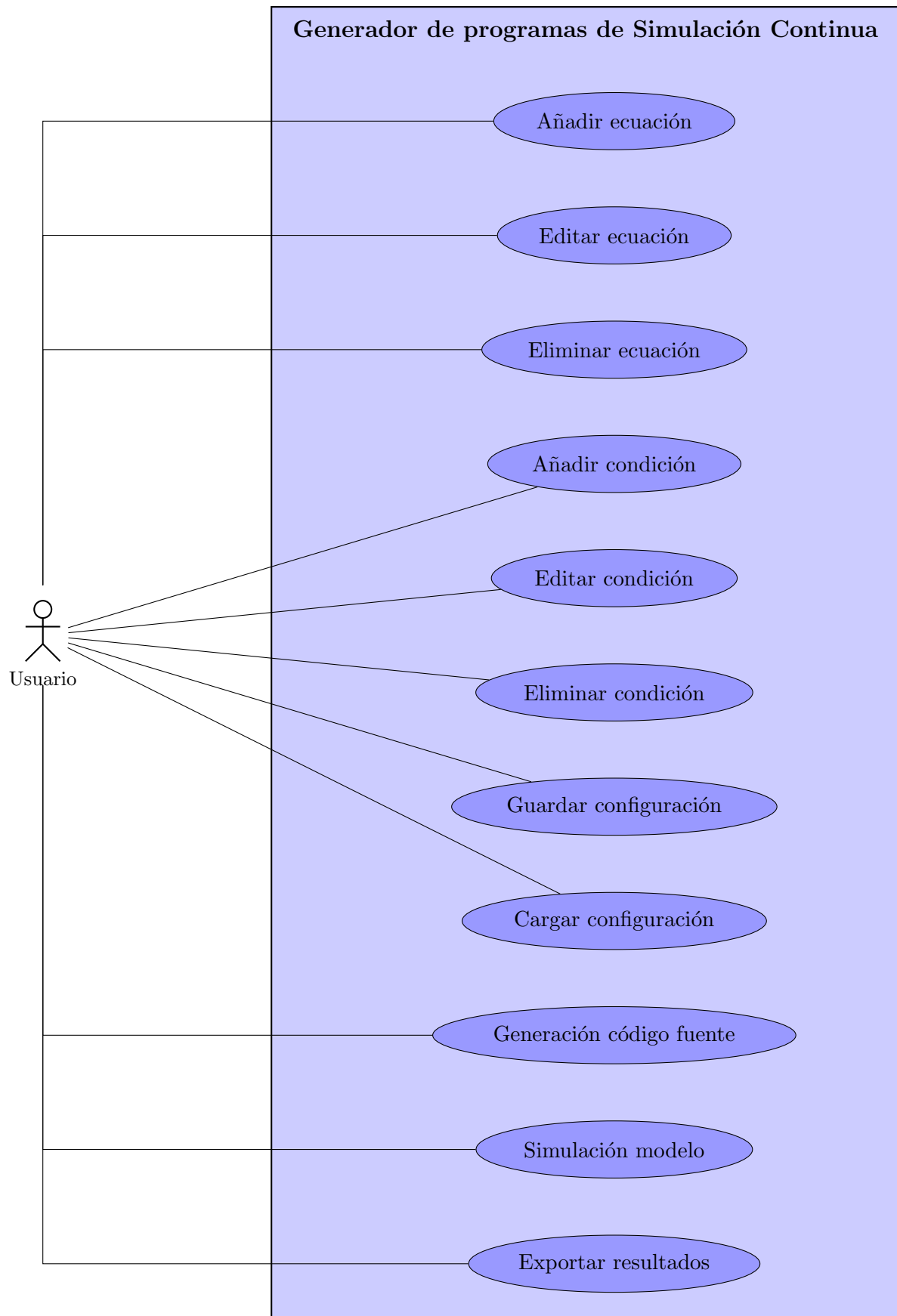


Figura 4.1: Diagrama de Casos de Uso del Sistema.

4.5. Planificación

La planificación de este proyecto se ha estructurado de forma afín a la metodología de desarrollo seleccionada, modelo en cascada. Esta metodología, como introducimos previamente, se caracteriza por una secuencia lineal de etapas, en las que cada fase depende de la finalización de la anterior. Sin embargo, las etapas de implementación, pruebas y validación, se ha optado por superponerlas parcialmente con el objetivo de optimizar el tiempo de desarrollo y facilitar la detección temprana de errores.

A continuación, se describen las tareas específicas asociadas a cada una de las fases que componen el modelo en cascada:

- **Fase de Análisis de Requisitos.** Durante esta etapa se recopilamos, analizamos y definieron los requisitos funcionales y no funcionales del sistema. Esta tarea incluyó la elaboración de los casos de uso, la definición de los actores y la identificación de las funcionalidades principales que debe ofrecer la aplicación.
- **Fase de Diseño del Sistema.** En esta fase se realiza el diseño de la arquitectura del sistema siguiendo el patrón Modelo-Vista-Controlador (MVC), así como la elaboración de diagramas de clases, diagramas de secuencia, esquemas de interfaces gráficas de usuario y el diseño de la estructura del generador de código fuente.
- **Fase de Implementación.** Comprende el desarrollo del sistema conforme al diseño previamente establecido, implementando los módulos que generan el código fuente en los distintos lenguajes objetivos (Python, C++ y Java), la interfaz gráfica y la lógica de integración de los componentes. Esta etapa es abordada de forma incremental, permitiendo pruebas intermedias.
- **Fase de Pruebas y Validación.** En esta fase se ejecutaron pruebas funcionales, de integración y validación, garantizando que el sistema cumple con los requisitos especificados. Además se verificó la generación y ejecución de los programas de simulación continua, la visualización de resultados, pudiendo asegurar la fiabilidad del sistema desarrollado.

La temporización de cada una de las fase queda definida mediante el siguiente Diagrama de Gantt:

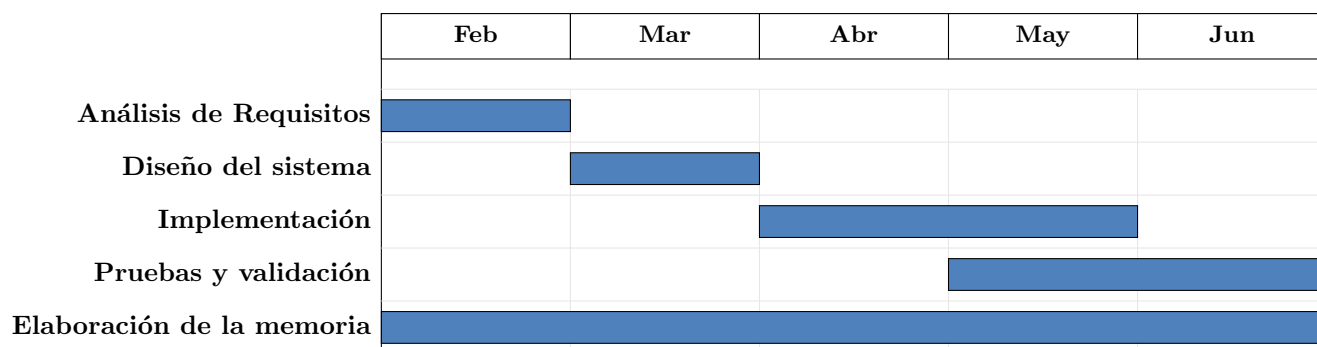


Figura 4.2: Diagrama de Gantt del proyecto - Cronograma Febrero-Junio 2025.

Este diagrama representa una estimación inicial de la temporización del desarrollo del proyecto. Posteriormente se comprobará si se ha cumplido con la planificación, y en su caso, se detallarán los motivos por los cuales no ha sido posible el cumplimiento de estos plazos conforme se ha indicado.

4.6. Presupuesto

Todo proyecto conlleva, de forma implícita, una serie de costes asociados que deben ser considerados. En este caso, aunque el presente proyecto ha sido desarrollado en un entorno académico y sin coste económico real, se presenta a continuación una estimación aproximada de los gastos de personal y ejecución, bajo el supuesto de que el proyecto hubiese sido encargado a una persona con el perfil técnico adecuado.

Tomando como referencia ofertas publicadas en portales especializados como Infojobs, LinkedIn e Indeed, los salarios brutos anuales de programadores junior con experiencia en desarrollo de aplicaciones de escritorio y conocimientos en lenguajes como Python, Java y C++ oscilan entre los 18.000 € y los 24.000 €. Para esta estimación, se considera un salario medio de 21.600 € brutos anuales, lo que equivale a 1.800 € brutos mensuales.

Este importe bruto no refleja el coste total del personal a la empresa que se interese en el desarrollo de la herramienta. Por ello, se realiza un desglose aproximado de los costes reales mensuales:

Concepto	Importe mensual (€)
Salario bruto del desarrollador	1.800
Cotizaciones a la Seguridad Social (empresa, aprox. 30 %)	540
Coste total mensual	2.340
Duración estimada del proyecto	5 meses
Coste total estimado	11.700

Cuadro 4.9: Estimación de costes de personal para el desarrollo del proyecto

A esto se debe añadir los costes indirectos, que no son repercutibles al desarrollo, pero son los necesarios en el mantenimiento de la actividad por parte de la empresa.

Descripción	Uds.	Precio (€)	IVA (21 %)	(€)
Material inventariable				
Ordenador personal (uso desarrollo)	1	978,11	183,40	1.161,51
Total material inventariable				1.161,51
Contratos y alquileres				
Alquiler del local	5	350,00	—	1750,00
Factura de luz	5	54,36	—	271,80
Factura de agua	5	25,12	—	125,60
Internet y línea	5	49,99	—	249,95
Total contratos y alquileres				2.397,35
Total gastos				3.558,86

Cuadro 4.10: Gastos de ejecución

Teniendo en cuenta todos los gastos, tanto de personal, como de mantenimiento de la empresa durante el desarrollo hace un total de 15.258,86 €

Capítulo 5

Diseño

Este capítulo presenta el diseño detallado del sistema desarrollado, con el objetivo de proporcionar una visión clara y estructurada de su arquitectura interna, el comportamiento de sus componentes y la interacción con el usuario, sirviendo como enlace entre el análisis de requisitos y la implementación final.

Para ello se abordan los distintos tipos de diagramas que permiten representar el sistema desde varias perspectivas complementarias:

- El **diagrama de clases** describe la estructura estática del sistema, identificando las clases principales, sus atributos, métodos y relaciones.
- Los **diagramas de secuencia** muestran la comunicación entre objetos durante la ejecución de los principales casos de uso, permitiendo comprender su comportamiento dinámico.
- El **diagrama de arquitectura** proporciona una visión global de la organización del sistema.
- Los **diagramas de interfaces de usuario** permiten visualizar el diseño de la interfaz gráfica, definiendo la disposición de los elementos interactivos y la correspondencia existente con las funcionalidades del sistema.

5.1. Diagrama de clases

Antes de presentar el diagrama de clases propuesto para el diseño del sistema es esencial justificar la elección y organización de las distintas clases que lo componen. La estructura del modelo de clases persigue los principios de diseño orientado a objetos, priorizando la modularidad, la reutilización de código y la separación de responsabilidades.

Las clases que componen el sistema son:

- **GUI.CTK.** Define la interfaz gráfica principal de la aplicación principal, que permite al usuario seleccionar el lenguaje de programación, el tipo de salida, el método de integración numérica, la gestión de ecuaciones diferenciales y las condiciones que afectan al modelo. Aparte de esto, contiene elementos para iniciar la generación y definición del directorio objetivo del código fuente, comienzo de la ejecución del mismo, carga de archivos de configuración de un modelo previamente creado y guardado de un modelo generado. Esta clase actúa como una parte de la vista dentro del patrón Modelo-Vista-Controlador, comunicándose con el controlador que gestiona la lógica subyacente.
- **GUI.Simulation.** Define la interfaz gráfica secundaria de la aplicación, encargada de gestionar la ejecución y visualización de las simulaciones. Esta clase permite al usuario

establecer los parámetros de la simulación (tiempo inicial, tiempo final, paso de integración o tolerancia del método numérico), introducir los valores iniciales de las variables y parámetros del modelo. La interfaz contiene un sistema de pestañas dinámico, lo que permite al usuario realizar múltiples simulaciones, cada una con su propia área de resultados y representación gráfica. Incluye una opción para exportar los resultados a un archivo PDF para su posterior observación, si así lo quisiese el usuario. Esta clase forma parte de las múltiples clases que componen la vista dentro del patron MVC.

- **GUI_Equation.** Forma parte también de la interfaz gráfica y tiene como objetivo proporcionar el entorno para la creación y edición de ecuaciones diferenciales. Permite al usuario introducir una ecuación simbólica junto con su variables y constantes asociadas, mostrando en tiempo real su representación en formato \LaTeX mediante un canvas generado. Esta clase funciona tanto en modo de adición como de edición, cargando y actualizando los datos según corresponda. Su diseño incorpora *placeholders*, textos predefinidos de ejemplo, que guían al usuario en la introducción ecuaciones diferenciales de forma adecuada para el sistema.
- **GUI_Condition.** Define la interfaz gráfica que proporciona el entorno de creación y edición de condiciones lógicas dentro del sistema. Esta clase permite al usuario introducir expresiones condicionales, acciones asociadas, variables y constantes, todo ello acompañado de una vista previa en lenguaje natural para facilitar la comprensión de la lógica definida. Además, al igual que la clase *GUI_Equation*, está adaptada a los dos modos de funcionamiento, adición y edición, incluyendo *placeholder* para guiar al usuario en la definición de condiciones.
- **GeneratorController.** Actúa como intermediario entre la vista y el modelo dentro de la arquitectura MVC, gestionando las entradas del usuario, validando las ecuaciones diferenciales, condiciones, variables y parámetros, y coordinando su almacenamiento y procesamiento en el modelo. Además se encarga de notificar errores o confirmaciones al usuario en la vista mediante un sistema de *logs*.
- **LogHandler.** Clase encargada de la gestión de errores y eventos al usuario junto con la clase *GeneratorController*, mediante el uso de códigos de error y de eventos adaptados para cada una de las situaciones que puedan darse en el uso del programa.
- **ContinuousModelGenerator.** Es el núcleo de la herramienta que permite generar y simular los modelos de simulación continua en los distintos lenguajes de programación. Sus responsabilidades abarcan el administrar las ecuaciones, condiciones iniciales y restricciones, traducir los modelos al lenguaje seleccionado utilizando los diferentes métodos numéricos implementados, gestiona la generación del código fuente, su compilación (si fuera necesario), la ejecución de la simulación, obtención de los resultados de la ejecución, exportar e importar archivos de configuración de un modelo creado por el usuario y la exportación de resultados en formato PDF. La clase también contiene elementos de validación de consistencia del modelo, la disponibilidad de los paquetes necesarios en el sistema operativo para la compilación del código fuente y ejecución de los programas.
- **SimulationModelGenerator.** Clase abstracta que sirve como base para la generación del código fuente en los distintos lenguajes. Su objetivo principal es organizar y preparar la información necesaria para traducir un modelo definido por ecuaciones diferenciales, condiciones y parámetros de simulación. Gestiona los atributos necesarios para ello, ecuaciones, condiciones, constantes, condiciones iniciales, tiempo de simulación, método numérico, y la ruta y nombre del código fuente de salida. Otro factor importante de esta clase es la asignación de identificadores únicos a las variables, y la verificación de que las condiciones sean coherentes con las ecuaciones.
- **PythonSimulationGenerator.** Clase que hereda de *SimulationModelGenerator*, se encarga de generar el código fuente en el lenguaje *Python* y su ejecución.

- **CppSimulationGenerator.** Clase que hereda de *SimulationModelGenerator*, se encarga de generar el código fuente en el lenguaje *C++*, su compilación y ejecución.
- **JavaSimulationGenerator.** Clase que hereda de *SimulationModelGenerator*, se encarga de generar el código fuente en el lenguaje *Java*, su compilación y ejecución.
- **Equation.** Clase encargada de representar las ecuaciones diferenciales que forman parte del modelo de simulación, su validación sintáctica y almacenarlas en una forma simbólica utilizando *sympy*, permitiendo así su posterior análisis y traducción a código fuente.
- **Condition.** Clase encargada de representar las condiciones que afectan al modelo durante su simulación, almacenando las expresiones lógicas, acciones, variables y constantes que la componen. También se encarga de realizar la validez de los elementos que la forman.

A continuación, se muestra el diagrama de clases del sistema dividido en tres partes para facilitar su visualización:

- En la Figura 5.1 se muestran las clases correspondientes al *Controlador* (*GeneratorController* y *LogHandler*) y parte de las clases del *Modelo* (*ContinuousModelGenerator*) de la arquitectura *MVC*.
- En la Figura 5.2 se muestran las clases correspondientes únicamente a la *Vista*.
- En la Figura 5.3 se muestran el resto de clases pertenecientes al *Modelo*.

En todas las figuras se indican clases que se encuentran “vacías” con el objetivo de una correcta vinculación entre cada una de ellas y facilitar al lector la interpretación del diagrama.

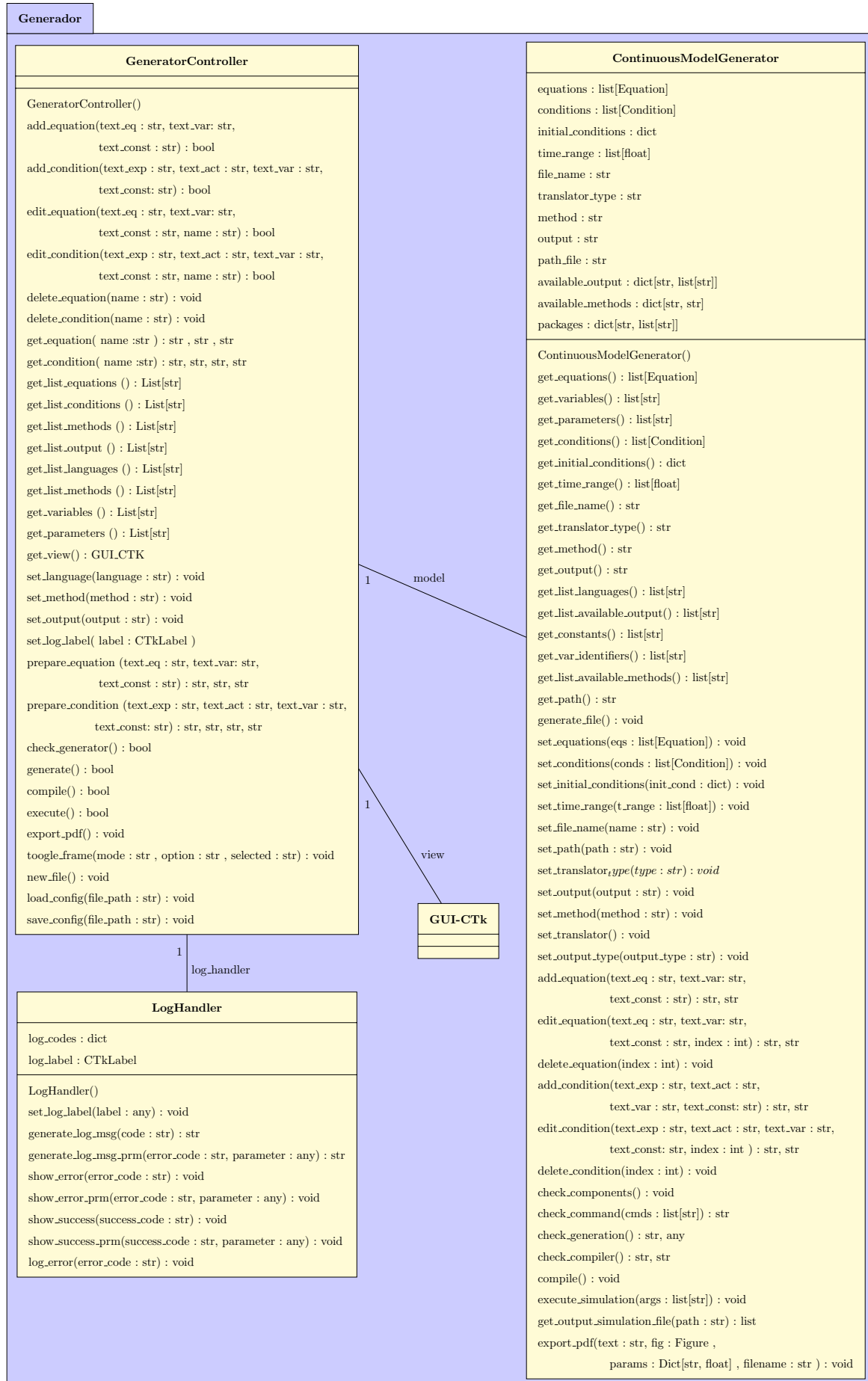


Figura 5.1: Diagrama de clases del sistema. Parte 1/3

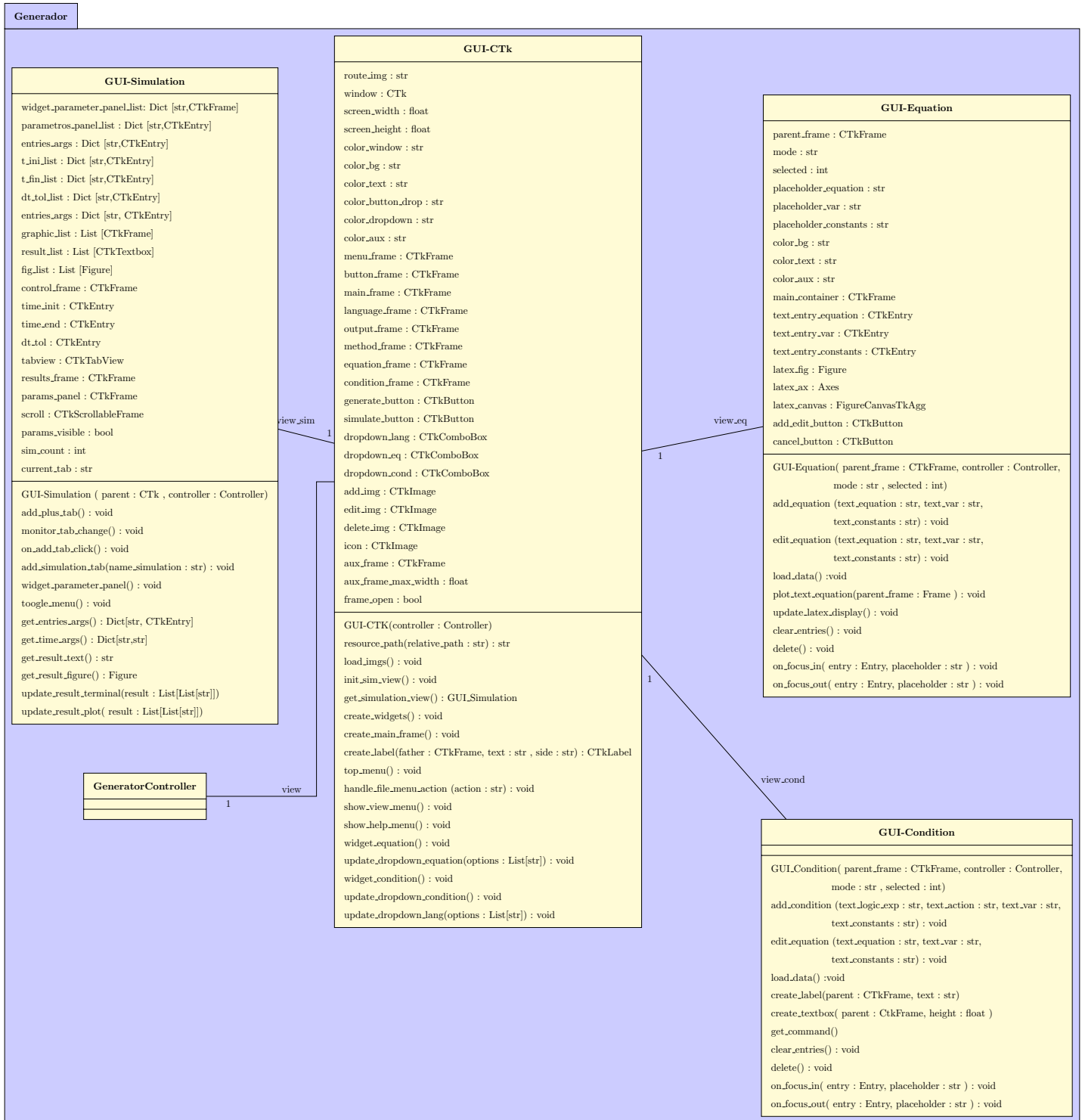


Figura 5.2: Diagrama de clases del sistema. Parte 2/3

5.2. Diagramas de secuencia

Los **diagramas de secuencia** son una herramienta clave para representar gráficamente el comportamiento dinámico del sistema [18].

Incluir estos diagramas en la documentación del proyecto resulta esencial, ya que facilita la comprensión de los flujos de ejecución, la coordinación entre clases, y la lógica que existe detrás de funcionalidades concretas.

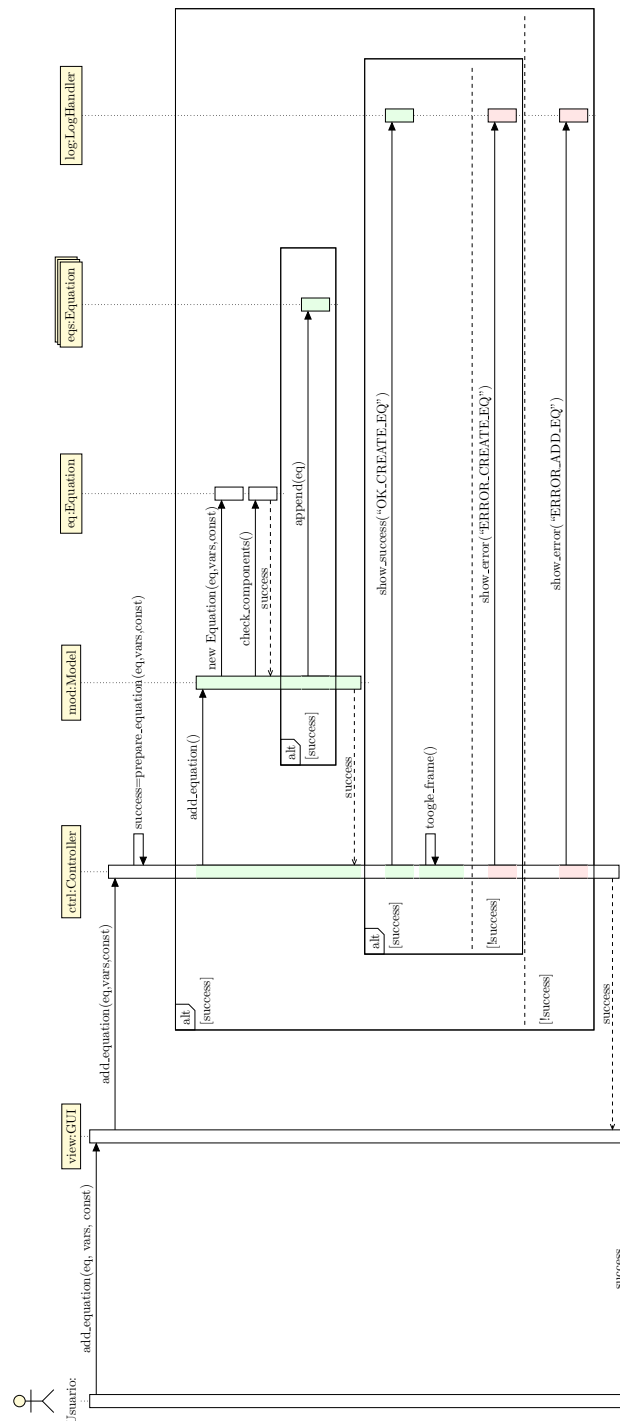


Figura 5.4: Diagrama Secuencia CU-001. Añadir ecuación introducida por el usuario al sistema.

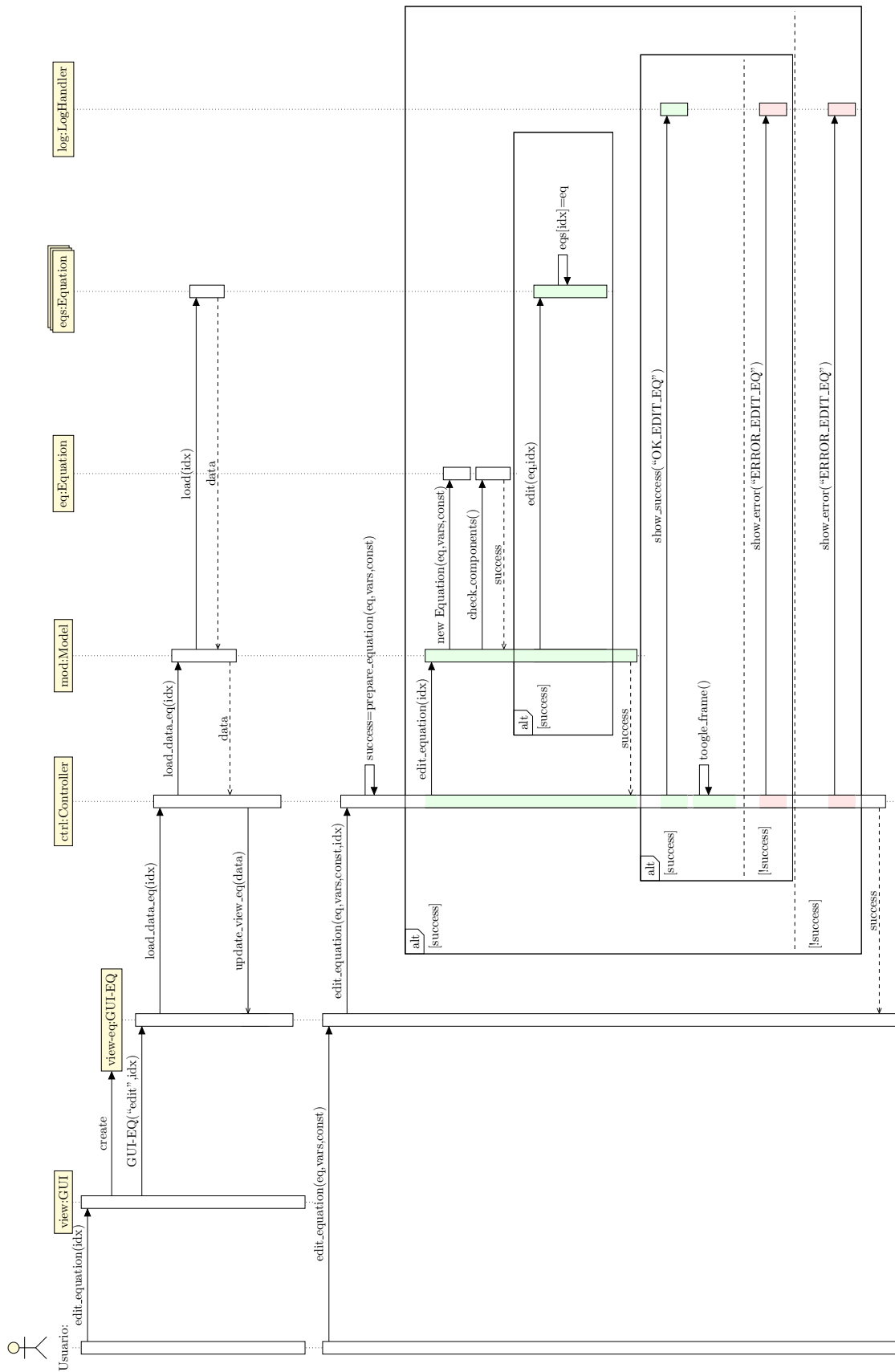


Figura 5.5: Diagrama Secuencia CU-002. Editar la ecuación con índice *idx* almacenada en el sistema.

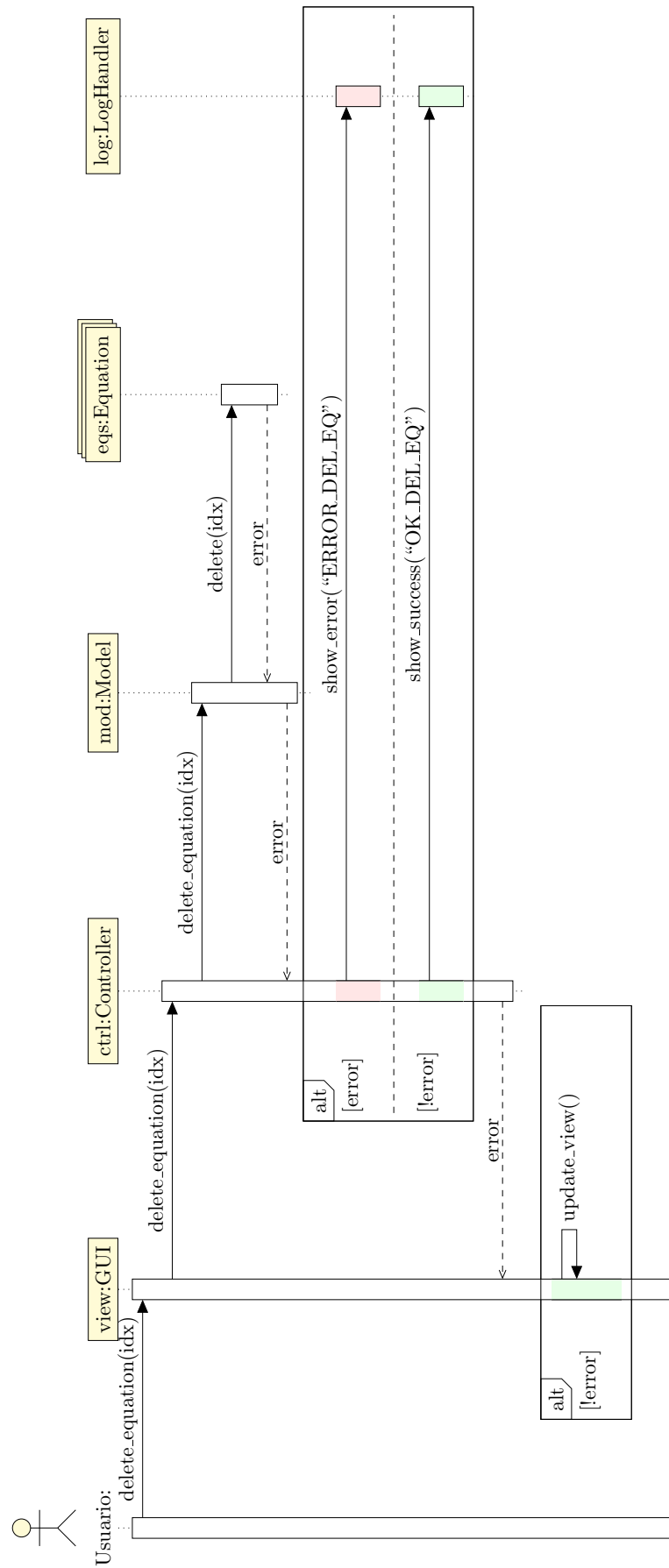


Figura 5.6: Diagrama Secuencia CU-003. Eliminar la ecuación con índice *idx* almacenada en el sistema.

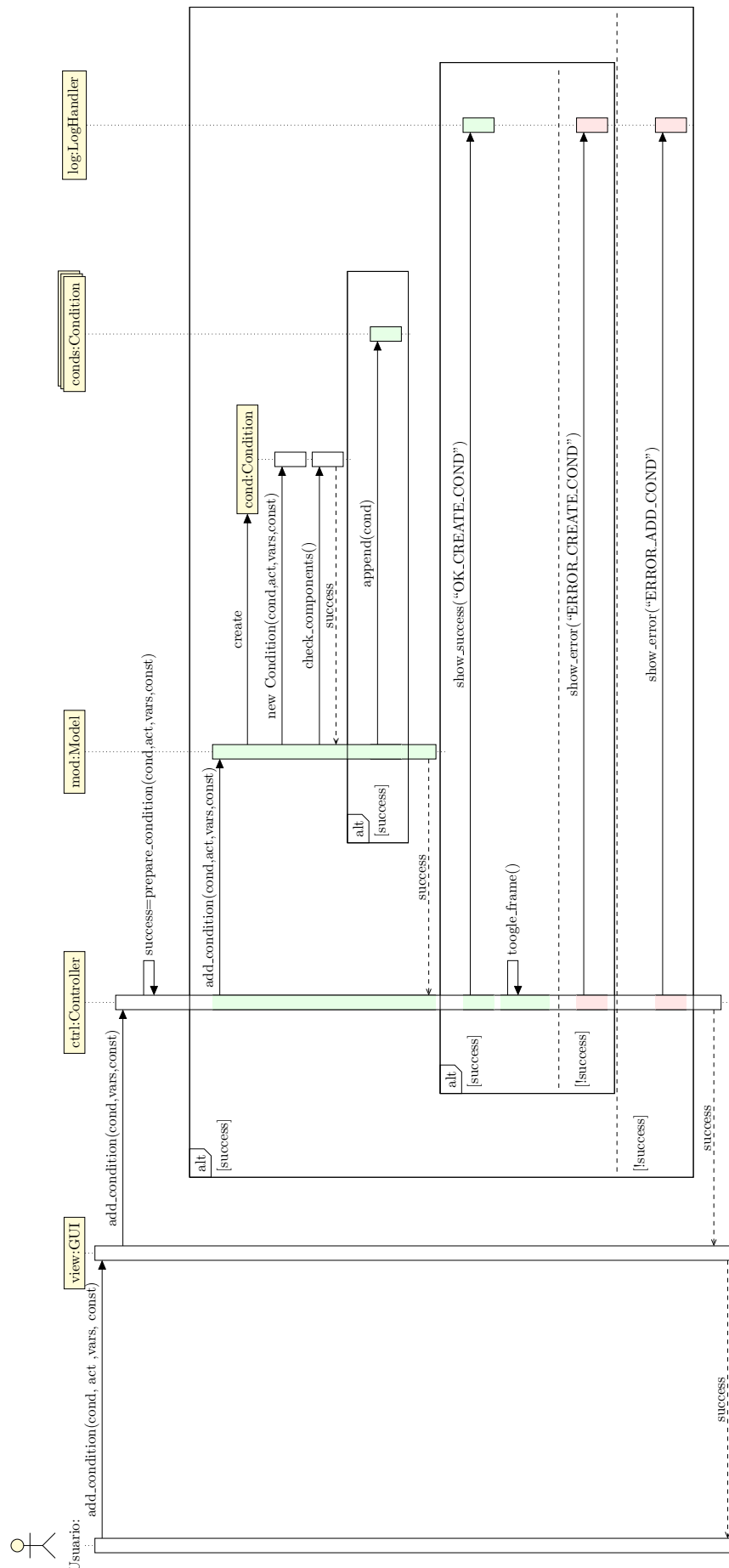


Figura 5.7: Diagrama Secuencia CU-004. Añadir condición introducida por el usuario al sistema.

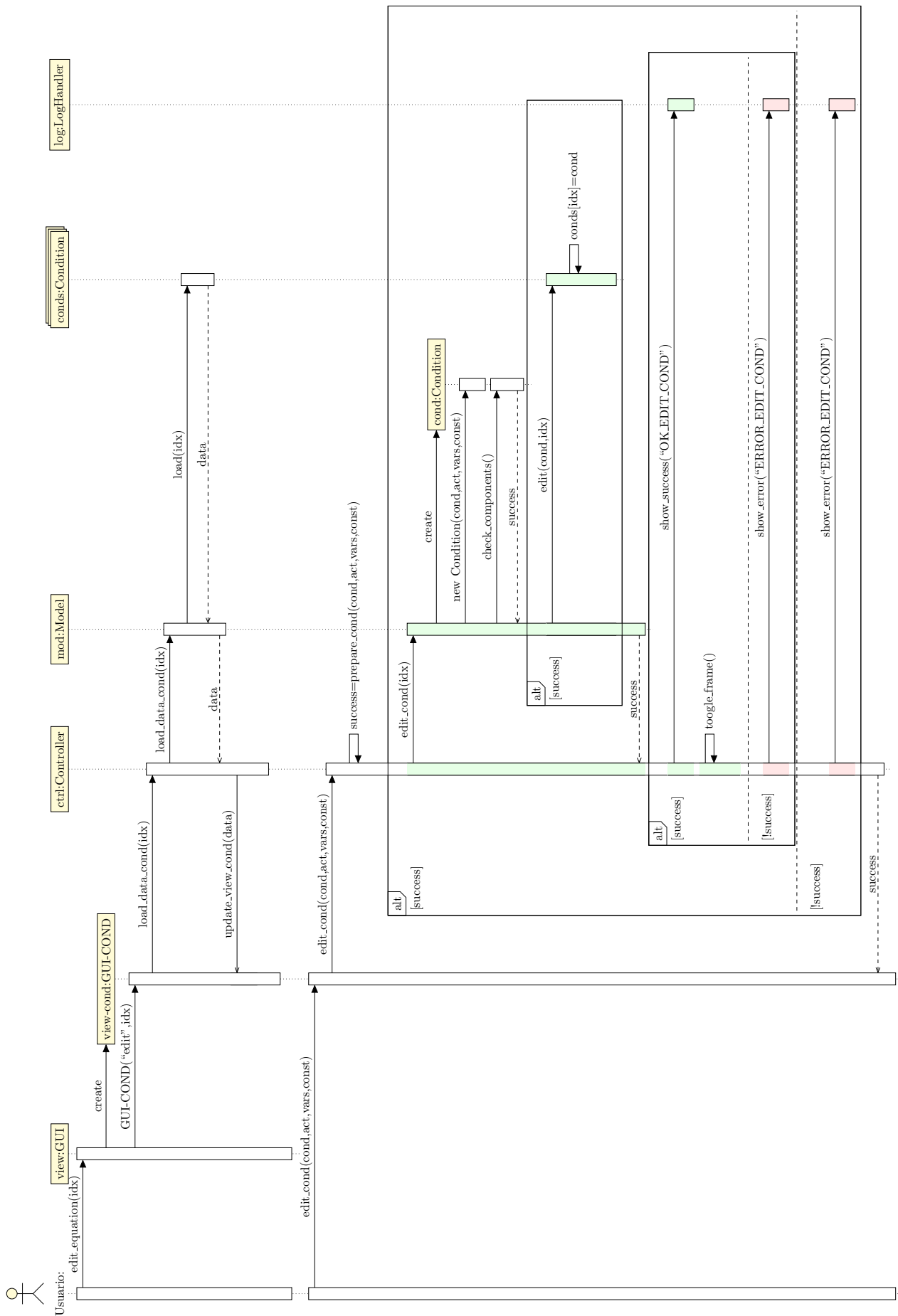


Figura 5.8: Diagrama Secuencia CU-005. Editar la condición con índice *idx* almacenada en el sistema.

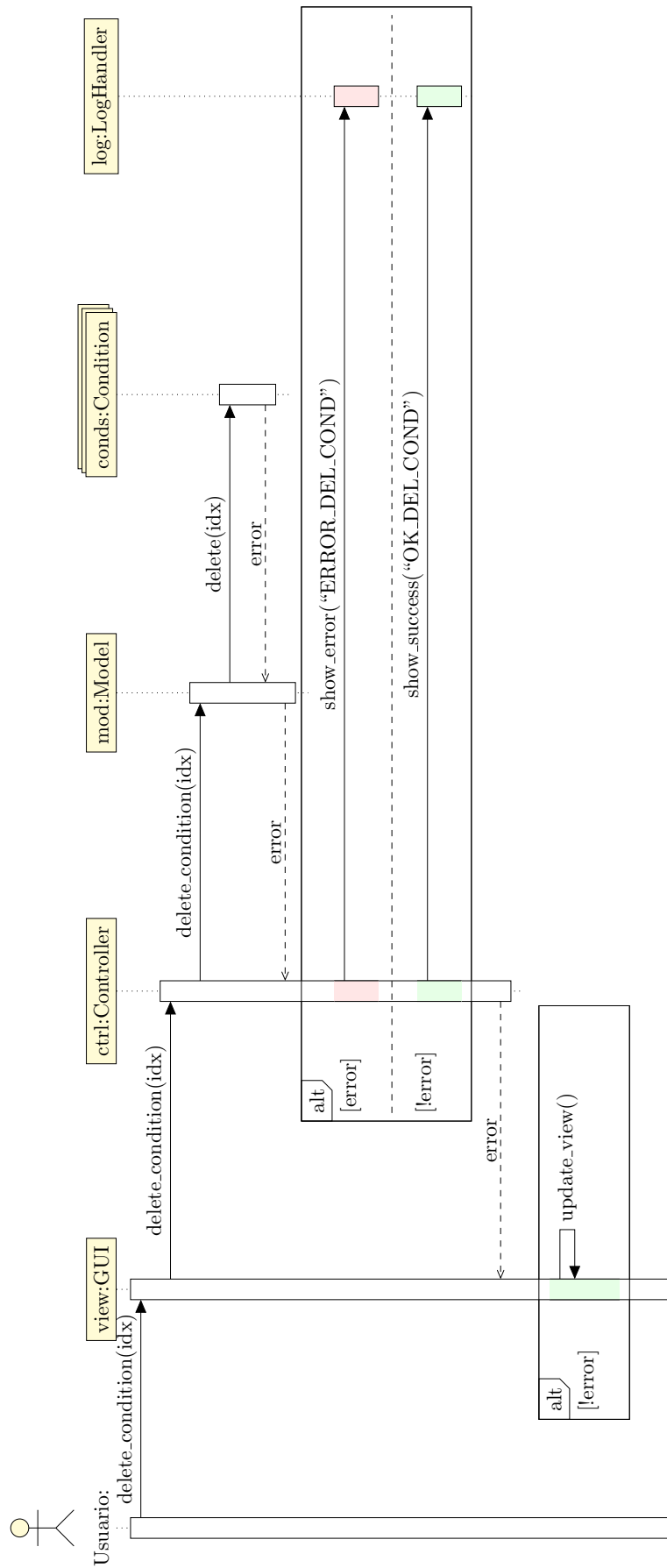


Figura 5.9: Diagrama Secuencia CU-006. Eliminar la condición con índice *idx* almacenada en el sistema.

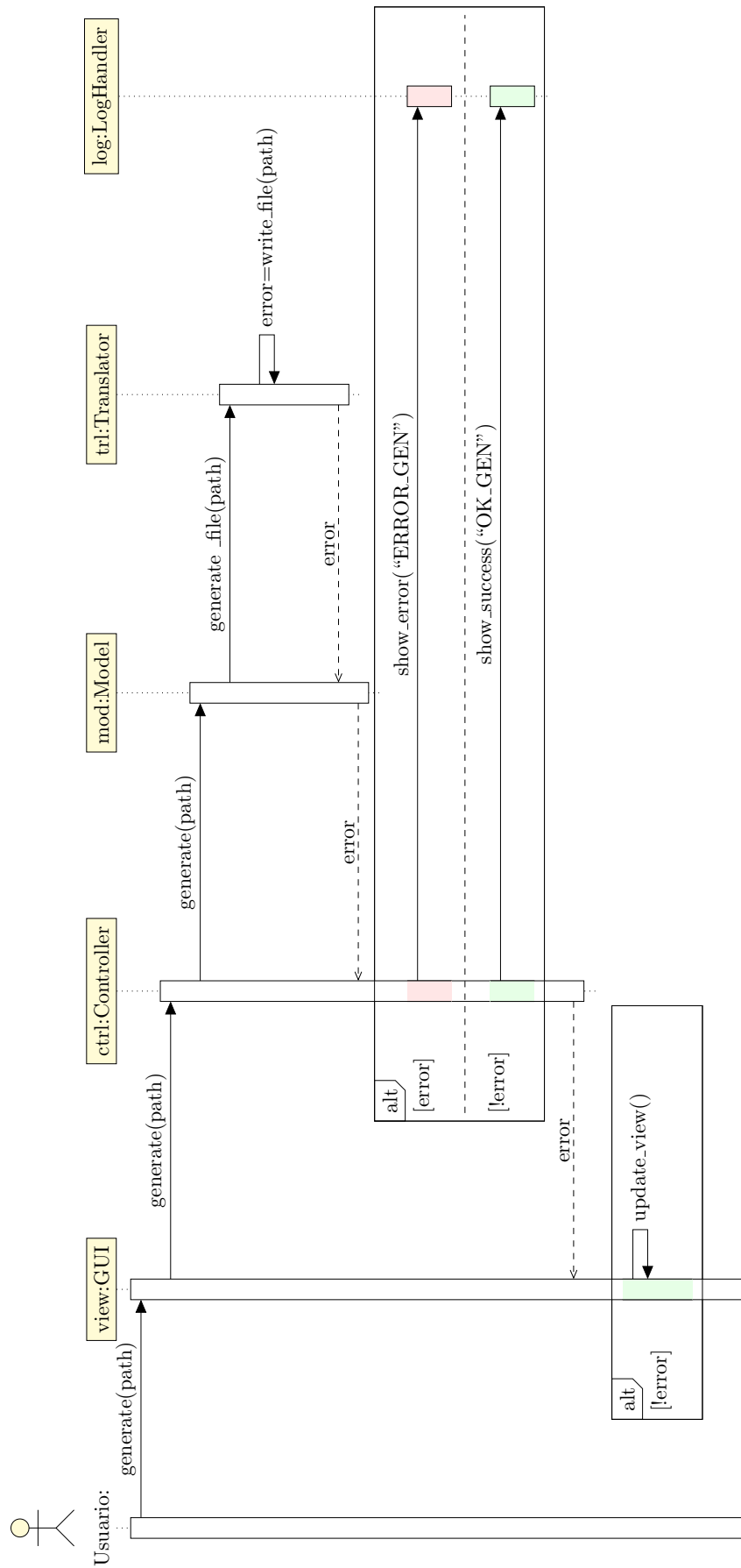


Figura 5.10: Diagrama Secuencia CU-007. Generación de código fuente.

5.3. Diagrama de arquitectura

El desarrollo del presente proyecto se ha realizado acorde al patrón de diseño **Modelo-Vista-Controlador** (MVC), una arquitectura que permite una separación clara entre la lógica del sistema, la presentación y el control de los eventos del usuario [20].

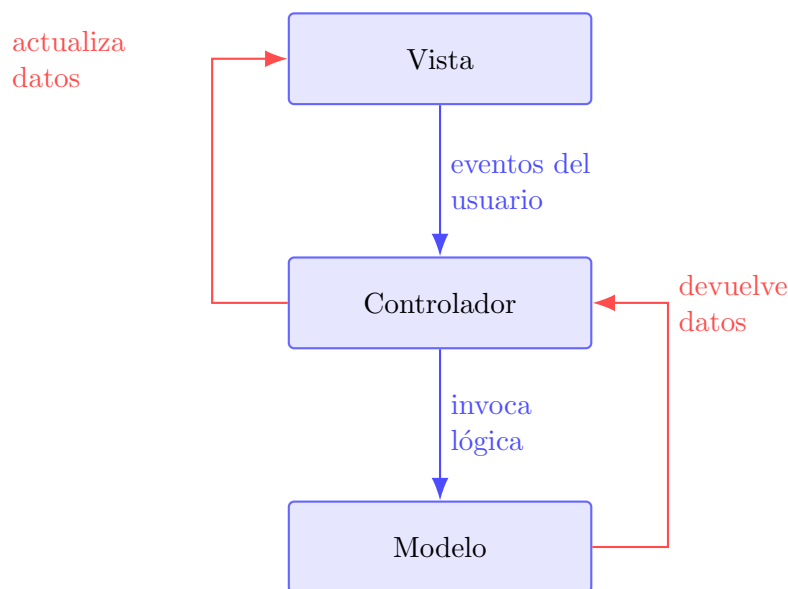


Figura 5.11: Diagrama de arquitectura del patrón MVC

Como se muestra en la Figura 5.11, el sistema se estructura siguiendo dicho patrón, donde:

- **Modelo.** Esta capa es responsable de, la gestión de las ecuaciones, condiciones, variables y constantes junto con sus respectivos valores, exportación de resultados, y guardado y carga de los archivos de configuración. Esta capa se encuentra compuesta por las clases: `Condition`, `Equation`, `SimulationModelGenerator`, `JavaSimulationGenerator`, `CppSimulationGenerator`, `PythonSimulationGenerator` y `ContinuousModelGenerator`.
- **Vista.** Capa responsable de la interacción del sistema con el usuario. Está compuesta por todas las clases que contienen elementos de interfaz gráfica: `GUI_CTK`, `GUI_Condition`, `GUI_Equation` y `GUI_Simulation`.
- **Controlador.** Esta última capa realiza la conexión entre la vista y la lógica del sistema. Se encarga de adaptar las entradas, detectar acciones del usuario y la gestión/-notificación de los errores si fuera necesario. Esta capa está representada por las clases `GeneratorController` y `LogHandler`.

En este tipo de sistemas, en los que la gran cantidad de componentes interconectados dificulta la mantenibilidad, escalabilidad y su validación, el patrón MVC aporta las siguientes ventajas [20]:

- **Aislar responsabilidades.** Cada módulo tiene una función claramente definida, facilitando la depuración y corrección de los errores.
- **Facilitar el mantenimiento y pruebas.** Al separar la lógica del sistema de la interfaz, el código resulta más explicable y modular para aquellos desarrolladores que no hayan participado en el proceso de desarrollo, además que facilita la realización de pruebas unitarias de manera más eficiente.

- **Permitir futuras extensiones.** En el caso de incorporar nuevos lenguajes de exportación no es necesario modificar la lógica de la interfaz.
- **Favorecer la reutilización del código.** Algunas de las clases pertenecientes a la capa de modelo son totalmente autónomas, por lo que podría reutilizarse en otros proyectos en los que sus necesidades sean similares.

En resumen, el uso de este patrón resulta fundamental en el desarrollo del proyecto, ya que permite organizar de manera clara y eficiente los requisitos que debe cumplir el sistema, gestionar la planificación temporal de cada una de sus partes y facilitar la adaptación a futuros requerimientos.

5.4. Diseño de la interfaz

La **interfaz de usuario**, *UI*, constituye uno de los elementos más relevantes del sistema. Tal y como señala *Roger S. Pressman* [18], una interfaz deficiente puede limitar severamente la capacidad del usuario para aprovechar el contenido informacional y el potencial de una aplicación, incluso si esta ha sido correctamente diseñada a nivel funcional y técnico. En este sentido, la percepción del usuario respecto al sistema está fuertemente condicionada por la calidad de su interfaz, convirtiéndose así en un factor determinante para la aceptación y éxito del *software*. Por ello, durante el desarrollo del proyecto se ha puesto especial énfasis en diseñar una interfaz intuitiva, coherente y funcional, capaz de ofrecer una experiencia de usuario satisfactoria.

Con el objetivo de guiar el diseño de la interfaz gráfica, se elaboró una serie de bocetos iniciales que sirvieran como referencia visual durante las primeras etapas del desarrollo. Estos bocetos, creados a partir de la herramienta *Figma* [6], permiten representar de forma esquemática la distribución de los elementos de la interfaz.

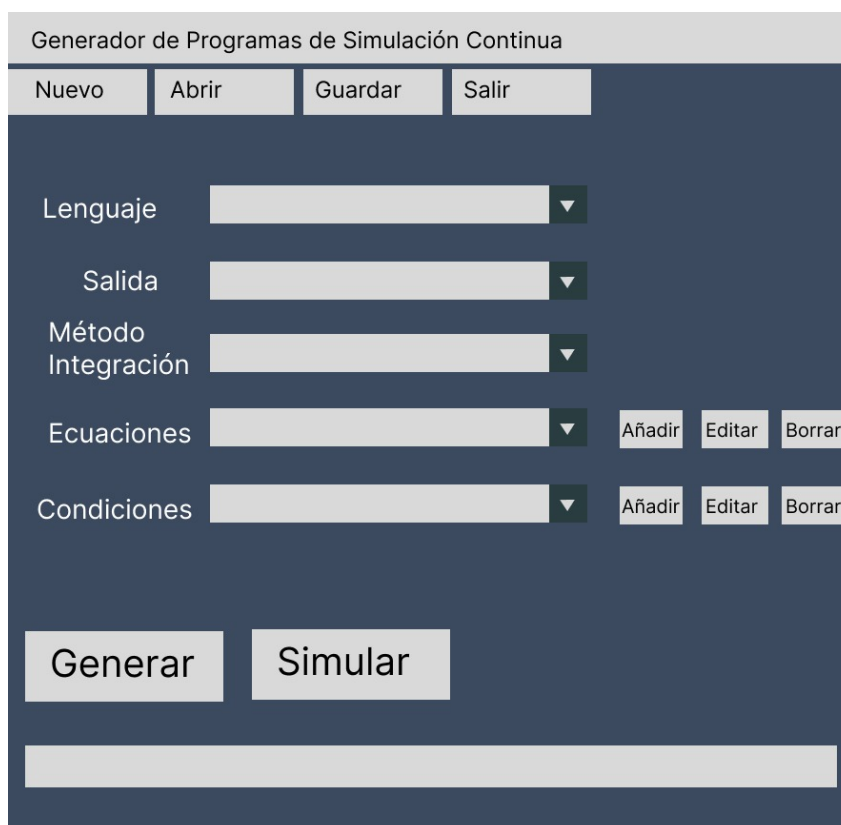


Figura 5.12: Prototipo interfaz. Pantalla inicial

En la Figura 5.12 se muestra el boceto inicial de la interfaz gráfica principal, en el que se incorporan los elementos esenciales para facilitar tanto la configuración del código fuente generado como su ejecución.

En la parte superior de la ventana, se dispone de una barra de menú con las opciones *Nuevo*, *Abrir*, *Guardar* y *Salir*, que permiten gestionar los archivos de configuración del proyecto de forma sencilla, así como el reinicio y cierre de la aplicación. A continuación, en el panel principal, se observa la presencia de varias listas desplegables. Algunas de ellas, como *Lenguaje*, *Salida* y *Método de Integración*, estarán disponibles desde el inicio, permitiendo al usuario seleccionar las características del código fuente que se desea generar. Por otro lado, las listas desplegables correspondientes a *Ecuaciones* y *Condiciones* estarán inicialmente vacías y a medida que el usuario introduzca información en el sistema irán apareciendo las opciones. Estas listas cuentan además con botones adyacentes que permiten editar o eliminar los elementos seleccionados.

En la parte inferior de la interfaz se encuentran dos botones principales: *Generar* y *Simular*, que ejecutan la generación del código fuente y la simulación del modelo, respectivamente. Finalmente, se incluye una barra de notificaciones del sistema, en el que se mostraran mensajes relevantes sobre los cambios que se van realizando sobre el sistema, tales como errores y mensajes de éxito, facilitando así el seguimiento durante el uso de la aplicación.

Figura 5.13: Prototipo interfaz. Pantalla de Añadir/Editar Ecuación

En la Figura 5.13 se muestra el boceto de la interfaz de gestión de las EDO. Esta pantalla está compuesta por tres campos de entrada de texto, donde el usuario puede introducir, respectivamente, la ecuación diferencial, el conjunto de variables involucradas y las constantes empleadas. Esta separación favorece la organización de los elementos del modelo y permite una validación más precisa de la entrada del usuario.

Uno de los elementos más relevantes de esta interfaz es el área de previsualización, donde se renderiza la ecuación introducida en formato $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$. Este componente resulta fundamental, ya que permite al usuario verificar en tiempo real la representación simbólica de la expresión, asegurando que la sintaxis y la estructura de la ecuación se ajustan a lo deseado para proceder a su almacenamiento.

En la parte inferior de la ventana se encuentran los botones de acción. El botón *Añadir/Editar* permite guardar o actualizar la información introducida en los campos, mientras que el botón

Cancelar permite cerrar la vista sin aplicar cambios.

El prototipo de interfaz de usuario para la pantalla de Añadir/Editar Condición se estructura de la siguiente manera:

- Condición**: Encabezado principal de la sección de configuración.
- Expresiones lógicas**: Campo de entrada para definir la condición lógica.
- Acción**: Campo de entrada para definir la acción a ejecutar.
- Variables**: Campo de entrada para definir las variables involucradas.
- Constantes**: Campo de entrada para definir las constantes involucradas.
- Vista Previa**: Área para visualizar la condición y acción definidas.
- Botones**: Botones **Añadir/Editar** y **Cancelar** para guardar o descartar los cambios.

Figura 5.14: Prototipo interfaz. Pantalla de Añadir/Editar Condición

En la Figura 5.14 se muestra el boceto correspondiente a la interfaz para la gestión de condiciones. Esta pantalla sigue una estructura similar a la utilizada en la sección de ecuaciones, incluyendo los campos de entrada para variables y constantes, así como una vista previa, pero en este caso el formato será más asociado a “lenguaje natural” con una sentencia “*Si <expresiones lógicas> entonces: <acciones>*” que permite al usuario verificar visualmente la expresión lógica introducida antes de su incorporación al sistema.

En lugar del campo para ecuaciones diferenciales, esta interfaz dispone de dos secciones específicas: un campo de expresión lógica, donde el usuario define la condición que debe cumplirse durante la simulación, y un campo de acciones, en el que se especifican las instrucciones que deben ejecutarse cuando la condición se cumpla.

En la parte inferior de la pantalla se encuentran los botones *Añadir/Editar* que permite guardar o actualizar la información introducida en los campos, mientras que con el botón de *Cancelar* descarta los cambios realizados.

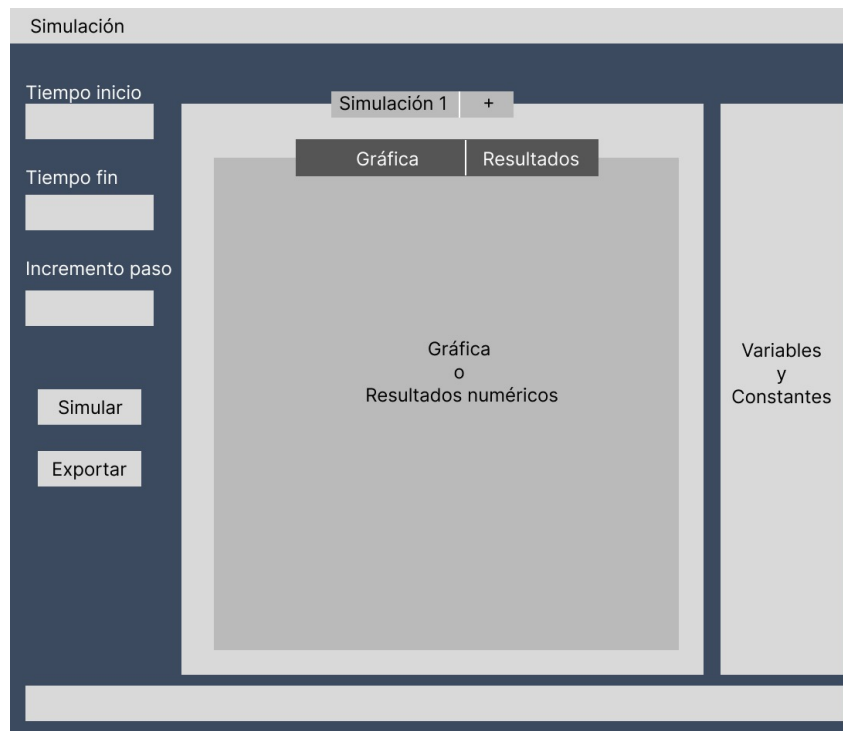


Figura 5.15: Prototipo interfaz. Pantalla de Simulación

Como último boceto, en la Figura 5.15 se muestra la interfaz principal de la simulación. Esta interfaz se divide en tres secciones diferenciadas:

- En la **parte izquierda** se encuentran los campos de configuración general de la simulación, incluyendo el tiempo de inicio y fin de la simulación, el incremento del paso (o la tolerancia, en función del método numérico seleccionado), así como los botones para inicializar la simulación y exportar resultados.
- En la **zona central** está dedicada a la visualización de los resultados. Se organiza mediante un sistema de pestañas, donde cada una representa una simulación distinta. Estas pestañas incluyen un gráfico de evolución temporal de las variables relevantes y una tabla numérica de resultados que facilita su análisis detallado. Además se proporciona un botón para añadir nuevas simulaciones, lo que permite al usuario gestionar múltiples ejecuciones de forma simultánea.
- Por último, en la **parte derecha** se encuentra el panel destinado a la definición de los valores de las variables y constantes del modelo. En este apartado el usuario puede consultar y modificar los valores asociados a cada parámetro del sistema antes de la ejecución de la simulación.

Con el objetivo de mejorar la experiencia de uso y minimizar la curva de aprendizaje, tanto en la interfaz de ecuaciones como en la de condiciones, los campos destinados a la introducción de texto incorporan *placeholders*¹. Estos elementos actúan como ejemplos representativos que orientan al usuario sobre la sintaxis esperada por el sistema. Gracias a esta funcionalidad, se facilita una comprensión rápida del formato requerido para introducir expresiones válidas, lo que contribuye a reducir errores y mejorar la usabilidad de la aplicación.

¹Texto breve que aparece de forma atenuada dentro de un campo de entrada y que sirve como guía para indicar el formato o contenido esperado.

Capítulo 6

Implementación

En este capítulo se describe en detalle el proceso de implementación de la aplicación desarrollada, abordando tanto los aspectos técnicos como las decisiones adoptadas durante el desarrollo del sistema.

En primer lugar, se presentan las **tecnologías empleadas**, detallando las librerías, dependencias y herramientas específicas que han sido utilizadas, ampliando la información ya introducida en los capítulos iniciales del proyecto.

En segundo lugar, se expone el **desarrollo de los componentes del sistema**, ofreciendo un desglose estructurado de las clases que conforman la aplicación. Para cada clase se describen sus atributos y métodos principales, así como su funcionalidad dentro del sistema.

En tercer lugar, se define la lógica seguida para la creación del código fuente en los lenguajes disponibles.

En cuarto lugar, se muestra el árbol de directorios y archivos generados durante el proceso de implementación del software.

Finalmente, se explica el proceso de **despliegue de la aplicación**, tanto en formato de paquete instalable mediante *pip*, como en forma de ejecutables para diferentes sistemas operativos generados con *PyInstaller*, incluyendo los pasos seguidos para obtener versiones compatibles con sistemas *Windows* y *Linux*.

6.1. Tecnologías elegidas

En el estado del arte de este proyecto se mencionaron de manera preliminar algunas de las tecnologías que se contemplaban para el desarrollo de la aplicación, incluyendo decisiones fundamentales como la elección del lenguaje de programación y el tipo de entorno sobre el que se trabaja. En este apartado, se amplía y completa esa información inicial, detallando con mayor precisión las bibliotecas, dependencias y herramientas específicas empleadas a lo largo del proceso de implementación. Además, se ofrece una justificación técnica para la selección de cada una de estas tecnologías, en función de los requisitos y objetivos ya planteados.

Bibliotecas:

- *CustomTkinter* [5]. Extensión de la biblioteca incluida en *Python* llamada *Tkinter*. Proporciona componentes visuales con un diseño actualizado y mayor capacidad de personalización, usada para la implementación de la interfaz gráfica de usuario.
- *datetime* [19]. Incluida en *Python*, permite la gestión de fechas y horas de forma precisa y eficiente. Usada en el proyecto para complementar las notificaciones de errores y éxito, facilitando al usuario la temporización de las notificaciones.

- *ReportLab* [9]. Biblioteca orientada a la generación dinámica de documentos en formato *PDF*, con soporte para estilos, tablas, imágenes y gráficos. Forma parte del núcleo de generación de los informes de cada simulación.
- *tempfile* [19]. Facilita la creación y gestión segura de archivos temporales. Se emplea para el almacenamiento temporal de gráficos generados durante la exportación de resultados, sin requerir una gestión manual de su eliminación.
- *re* [19]. Proporciona herramientas para manipulación avanzada de cadenas de texto mediante expresiones regulares definidas. En el proyecto es usada en la generación del código fuente en cada lenguaje, principalmente en la gestión de los operadores específicos.
- *functools.partial* [19]. Permite la creación de funciones con argumentos predefinidos. Usada para asociar acciones específicas a eventos dentro de la UI, mejorando la modularidad y legibilidad del código.
- *Matplotlib* [22]. Biblioteca especializada en la generación de gráficos estáticos e interactivos en *Python*. Se integra en la interfaz para visualización gráfica de los resultados de las simulaciones, proporcionando una representación visual clara.
- *Sympy* [24]. Biblioteca enfocada en el cálculo simbólico, que posibilita la manipulación de expresiones matemáticas y la resolución simbólica de ecuaciones. En el proyecto es usada para procesar, validar y manipular las expresiones matemáticas y condiciones definidas por el usuario, así como para la generación de código simbólico.
- *os* [19]. Proporciona funcionalidad para la interacción con el sistema operativo, incluyendo la gestión de archivos y rutas. Usada para las operaciones con la administración de archivos, tales como el almacenamiento de resultados y manejo de rutas de acceso.
- *subprocess* [19]. Permite la ejecución de procesos externos desde el programa principal. En el proyecto, se emplea para ejecutar el código fuente generado como simulación independiente, pasando los argumentos necesarios desde la interfaz y permitiendo controlar la visibilidad de la consola en distintos sistemas operativos.
- *abc* [19]. Permite definir clases y métodos abstractos, estableciendo interfaces formales que deben ser implementadas por las subclasses. Usada para la estructuración del código mediante la definición de clases base que garantizan la coherencia y uniformidad en la implementación de sus derivados.

Herramientas y dependencias. Es importante tener en cuenta la existencia de una comunicación con el sistema operativo anfitrión de la aplicación en su uso, ya que parte de la funcionalidad de la aplicación radica en el uso de herramientas externas al propio programa. En particular, la ejecución de los modelos generados en distintos lenguajes requiere que el entorno cuente con los compiladores e intérpretes correspondientes previamente instalados. A continuación, se detallan los paquetes y herramientas necesarias para el correcto funcionamiento de la aplicación:

- *C++*. Es imprescindible disponer del compilador *g++*. En sistemas *Linux*, suele formar parte del paquete *build-essential*, mientras que en entornos *Windows* puede instalarse mediante *MinGW* [25].
- *Java*. Para el caso de los programas generados en este lenguaje se requiere el compilador *javac*, que forma parte del JDK (*Java Development Kit*) y permite compilar los archivos con extensión “*.java*”. Asimismo, es necesario contar con el ejecutador *java*, encargado de interpretar y ejecutar dicho *bytecode* en la máquina virtual de *Java* (JVM).

- *Python*. Dado que la propia aplicación está desarrollada en este lenguaje, es necesario tener instalado el intérprete en el sistema. Si bien se dispone de una aplicación autónoma, con las dependencias incluidas dentro del ejecutable evitando la necesidad del intérprete, sigue siendo imprescindible para la ejecución de los programas de simulación generados en este lenguaje.
- *pip* [1]. Facilita la instalación de las bibliotecas necesarias para la ejecución de la aplicación desde el propio intérprete de *Python*. En el caso de que se use la aplicación desde el ejecutable proporcionado con el despliegue no será necesario este requisito.

Con el objetivo de garantizar la correcta ejecución de los programas de simulación generados, la aplicación incorpora mecanismos de notificación de errores que permiten alertar al usuario en caso de que no se encuentren instalados ciertos componentes clave en el entorno. En particular, se realiza una verificación de la disponibilidad de los compiladores, ejecutores e intérpretes.

6.2. Desarrollo de componentes

6.2.1. Clase GUI_CTK

Descripción: Clase que representa la ventana principal de la interfaz gráfica del generador de programas de simulación continua. Permite al usuario seleccionar el lenguaje, introducir ecuaciones y condiciones, elegir el método de integración, generar el código correspondiente y lanzar una simulación visual.

Cuadro 6.1: Atributos de la clase GUI_CTK

Nombre	Tipo	Descripción
controller	Controller	Instancia del controlador principal que gestiona la lógica de la aplicación, coordina la interacción entre la interfaz gráfica y el motor de simulación, y procesa las acciones del usuario.
route_img	str	Ruta base donde se encuentran las imágenes de recursos.
window	CTk	Ventana principal de la interfaz.
screen_width	float	Ancho de la pantalla.
screen_height	float	Alto de la pantalla.
color_window	str	Color de fondo de la ventana principal.
color_bg	str	Color de fondo general para widgets.
color_text	str	Color del texto en la interfaz.
color_button_drop	str	Color de fondo para botones desplegables.
color_dropdown	str	Color de fondo para menús desplegables.
color_aux	str	Color auxiliar para detalles de diseño.
menu_frame	CTkFrame	Marco contenedor para el menú principal.
button_frame	CTkFrame	Marco para contener los botones principales.

Continúa en la siguiente página

Cuadro 6.1 – continuación

Nombre	Tipo	Descripción
<code>main_frame</code>	<code>CTkFrame</code>	Marco principal de la interfaz.
<code>language_frame</code>	<code>CTkFrame</code>	Marco específico para selección de idioma.
<code>output_frame</code>	<code>CTkFrame</code>	Marco para opciones de salida y visualización.
<code>method_frame</code>	<code>CTkFrame</code>	Marco para selección de método de integración.
<code>equation_frame</code>	<code>CTkFrame</code>	Marco para introducir ecuaciones.
<code>condition_frame</code>	<code>CTkFrame</code>	Marco para introducir condiciones iniciales.
<code>generate_button</code>	<code>CTkButton</code>	Botón para generar código fuente.
<code>simulate_button</code>	<code>CTkButton</code>	Botón para ejecutar la simulación.
<code>dropdown_lang</code>	<code>CTkComboBox</code>	Menú desplegable para seleccionar idioma.
<code>dropdown_eq</code>	<code>CTkComboBox</code>	Menú desplegable para seleccionar tipo de ecuación.
<code>dropdown_cond</code>	<code>CTkComboBox</code>	Menú desplegable para seleccionar condición inicial.
<code>add_img</code>	<code>CTkImage</code>	Imagen para icono de añadir elemento.
<code>edit_img</code>	<code>CTkImage</code>	Imagen para icono de editar elemento.
<code>delete_img</code>	<code>CTkImage</code>	Imagen para icono de eliminar elemento.
<code>icon</code>	<code>CTkImage</code>	Icono general de la aplicación.
<code>aux_frame</code>	<code>CTkFrame</code>	Marco auxiliar para organización de widgets.
<code>aux_frame_max_width</code>	<code>float</code>	Ancho máximo permitido para el marco auxiliar.
<code>frame_open</code>	<code>bool</code>	Indica si un marco auxiliar está abierto.

Cuadro 6.2: Métodos de la clase `GUI_CTK`

Nombre	Parámetros	Descripción
<code>GUI_CTK</code>	<code>controller :</code> <code>Controller</code>	Constructor que inicializa la interfaz y el controlador asociado.
<code>resource_path</code>	<code>relative_path : str</code>	Devuelve la ruta absoluta a un recurso, útil para empaquetar con <code>PyInstaller</code> .
<code>load_imgs</code>	ninguno	Carga las imágenes utilizadas en la interfaz gráfica.
<code>init_sim_view</code>	ninguno	Inicializa la vista de simulación asociada.
<code>get_simulation_view</code>	ninguno	Devuelve el objeto de la vista de simulación.

Continúa en la siguiente página

Cuadro 6.2 – continuación

Nombre	Parámetros	Descripción
<code>create_widgets</code>	ninguno	Crea todos los widgets gráficos de la ventana principal.
<code>create_main_frame</code>	ninguno	Construye el marco principal y organiza su contenido.
<code>create_label</code>	<code>father : CTkFrame,</code> <code>text : str, side :</code> <code>str</code>	Crea y retorna una etiqueta (label) con texto dentro de un marco padre.
<code>top_menu</code>	ninguno	Configura el menú superior de la ventana.
<code>handle_file_menu_action</code>	<code>action : str</code>	Gestiona las acciones del menú archivo según la opción seleccionada.
<code>show_view_menu</code>	ninguno	Muestra el menú de vista.
<code>show_help_menu</code>	ninguno	Muestra el menú de ayuda.
<code>widget_equation</code>	ninguno	Crea y maneja los widgets relacionados con las ecuaciones.
<code>update_dropdown_equation</code>	<code>options : List[str]</code>	Actualiza las opciones del menú desplegable de ecuaciones.
<code>widget_condition</code>	ninguno	Crea y maneja los widgets para las condiciones iniciales.
<code>update_dropdown_condition</code>	ninguno	Actualiza las opciones del menú desplegable de condiciones iniciales.
<code>update_dropdown_lang</code>	<code>options : List[str]</code>	Actualiza las opciones del menú desplegable de idiomas.

6.2.2. Clase `GUI_Equation`

Descripción: Clase que gestiona la interfaz gráfica para añadir o editar ecuaciones diferenciales, variables y constantes, mostrando además una representación \LaTeX de la ecuación.

Cuadro 6.3: Atributos de la clase `GUI_Equation`

Nombre	Tipo	Descripción
<code>parent_frame</code>	<code>CTkFrame</code>	Marco contenedor donde se inserta el componente gráfico de la ecuación.
<code>mode</code>	<code>str</code>	Modo actual, puede ser para añadir o editar una ecuación.
<code>selected</code>	<code>int</code>	Índice o identificador de la ecuación seleccionada para editar.

Continúa en la siguiente página

Cuadro 6.3 – continuación

Nombre	Tipo	Descripción
placeholder_equation	str	Texto guía para el campo de entrada de la ecuación.
placeholder_var	str	Texto guía para el campo de entrada de la variable.
placeholder_constants	str	Texto guía para el campo de entrada de las constantes.
color_bg	str	Color de fondo utilizado en la interfaz.
color_text	str	Color del texto mostrado en los campos.
color_aux	str	Color auxiliar para elementos secundarios.
main_container	CTkFrame	Marco principal que agrupa todos los widgets del formulario.
text_entry_equation	CTkEntry	Campo de texto para introducir la ecuación.
text_entry_var	CTkEntry	Campo de texto para introducir la variable dependiente.
text_entry_constants	CTkEntry	Campo de texto para introducir las constantes del modelo.
latex_fig	Figure	Objeto figura para mostrar la representación LaTeX.
latex_ax	Axes	Ejes asociados a la figura LaTeX.
latex_canvas	FigureCanvasTkAgg	Canvas Tkinter para renderizar la figura LaTeX.
add_edit_button	CTkButton	Botón para añadir o editar la ecuación según el modo.
cancel_button	CTkButton	Botón para cancelar la operación y limpiar entradas.

Cuadro 6.4: Métodos de la clase `GUI_Equation`

Nombre	Parámetros	Descripción
<code>GUI_Equation()</code>	<code>parent_frame</code> : <code>CTkFrame</code> , <code>controller</code> : <code>Controller</code> , <code>mode</code> : <code>str</code> , <code>selected</code> : <code>int</code>	Constructor que inicializa la interfaz de la ecuación con los parámetros indicados.

Continúa en la siguiente página

Cuadro 6.4 – continuación

Nombre	Parámetros	Descripción
<code>add_equation()</code>	<code>text_equation :</code> <code>str</code> , <code>text_var :</code> <code>str</code> , <code>text_constants :</code> <code>str</code>	Añade una nueva ecuación con sus variables y constantes al sistema.
<code>edit_equation()</code>	<code>text_equation :</code> <code>str</code> , <code>text_var :</code> <code>str</code> , <code>text_constants :</code> <code>str</code>	Modifica la ecuación seleccionada con los nuevos valores proporcionados.
<code>load_data()</code>	ninguno	Carga los datos actuales para mostrar en la interfaz, útil en modo edición.
<code>plot_text_equation()</code>	<code>parent_frame :</code> <code>Frame</code>	Genera y muestra la representación visual \LaTeX de la ecuación en el marco indicado.
<code>update_latex_display()</code>	ninguno	Actualiza la visualización \LaTeX cuando cambian los datos.
<code>clear_entries()</code>	ninguno	Limpia los campos de entrada para ecuación, variable y constantes.
<code>delete()</code>	ninguno	Elimina el widget de la interfaz y libera recursos.
<code>on_focus_in()</code>	<code>entry :</code> <code>Entry</code> , <code>placeholder :</code> <code>str</code>	Maneja el evento de entrada de foco, limpiando el texto placeholder si está activo.
<code>on_focus_out()</code>	<code>entry :</code> <code>Entry</code> , <code>placeholder :</code> <code>str</code>	Maneja el evento de salida de foco, restaurando el placeholder si el campo está vacío.

6.2.3. Clase `GUI_Condition`

Descripción: Clase que gestiona la interfaz para introducir y modificar las condiciones del modelo de simulación continua. Permite al usuario añadir y editar condiciones desde la ventana principal del generador.

Cuadro 6.5: Atributos de la clase `GUI_Condition`

Nombre	Tipo	Descripción
<code>parent_frame</code>	<code>CTkFrame</code>	Marco padre donde se insertan los componentes de la interfaz gráfica para condiciones.
<code>controller</code>	<code>Controller</code>	Controlador principal que maneja la lógica asociada a las condiciones (añadir, editar, obtener).
<code>mode</code>	<code>str</code>	Modo de operación, puede ser ‘‘add’’ para añadir una nueva condición o ‘‘edit’’ para modificar una existente.

Continúa en la siguiente página

Cuadro 6.5 – continuación

Nombre	Tipo	Descripción
<code>selected</code>	<code>int</code>	Identificador de la condición seleccionada cuando se está en modo edición.
<code>placeholder_condition</code>	<code>str</code>	Texto de ejemplo para el campo de expresiones lógicas.
<code>placeholder_action</code>	<code>str</code>	Texto de ejemplo para el campo de acciones.
<code>placeholder_var</code>	<code>str</code>	Texto de ejemplo para el campo de variables.
<code>placeholder_constants</code>	<code>str</code>	Texto de ejemplo para el campo de constantes.
<code>color_window</code>	<code>str</code>	Color principal de fondo para la ventana de condiciones.
<code>color_bg</code>	<code>str</code>	Color de fondo secundario para marcos y bordes.
<code>color_text</code>	<code>str</code>	Color del texto de las etiquetas y entradas.
<code>color_aux</code>	<code>str</code>	Color auxiliar utilizado en botones y bordes.
<code>main_container</code>	<code>CTkFrame</code>	Contenedor principal que aloja todos los componentes de la clase.
<code>text_entry_condition</code>	<code>CTkTextbox</code>	Cuadro de texto donde se ingresan las expresiones lógicas de la condición.
<code>text_entry_action</code>	<code>CTkTextbox</code>	Cuadro de texto donde se ingresan las acciones que se ejecutan si la condición se cumple.
<code>text_entry_var</code>	<code>CTkTextbox</code>	Cuadro de texto para ingresar las variables usadas en la condición.
<code>text_entry_constants</code>	<code>CTkTextbox</code>	Cuadro de texto para ingresar las constantes utilizadas.
<code>text_preview</code>	<code>CTkTextbox</code>	Vista previa que muestra cómo se interpreta la condición escrita por el usuario.
<code>add_edit_button</code>	<code>CTkButton</code>	Botón que activa la acción de añadir o editar una condición según el modo actual.
<code>cancel_button</code>	<code>CTkButton</code>	Botón para cancelar la operación y volver al menú anterior.

Cuadro 6.6: Métodos de la clase `GUI.Condition`

Nombre	Parámetros	Descripción
<code>GUI.Condition</code>	<code>parent_frame : CTkFrame,</code> <code>controller : Controller,</code> <code>mode : str, selected :</code> <code>Optional[str] = None</code>	Constructor que inicializa la interfaz para crear o editar una condición.
<code>create_label</code>	<code>parent : CTkFrame, text :</code> <code>str</code>	Crea y añade una etiqueta de texto al contenedor especificado.

Continúa en la siguiente página

Cuadro 6.6 – continuación

Nombre	Parámetros	Descripción
<code>create_textbox</code>	<code>parent : CtkFrame, height : int = 10</code>	Crea y retorna un campo de texto ('CtkTextbox') con el alto indicado.
<code>get_command</code>	ninguno	Devuelve la función correspondiente (añadir o editar) según el modo actual.
<code>add_condition</code>	<code>text_logic_exp : str, text_action : str, text_var : str, text_constants : str</code>	Añade una nueva condición a través del controlador.
<code>edit_condition</code>	<code>text_logic_exp : str, text_action : str, text_var : str, text_constants : str</code>	Edita la condición seleccionada usando el controlador.
<code>load_data</code>	ninguno	Carga los datos de una condición existente en los campos correspondientes (modo edición).
<code>preview_condition</code>	ninguno	Genera y muestra una vista previa de la condición introducida.
<code>delete</code>	ninguno	Oculto el contenedor principal de la interfaz gráfica.
<code>on_focus_in</code>	<code>textbox : CtkTextbox, placeholder : str, event</code>	Limpia el contenido del campo si contiene el texto por defecto al obtener foco.
<code>on_focus_out</code>	<code>textbox : CtkTextbox, placeholder : str, event</code>	Restaura el texto por defecto si el campo está vacío al perder foco.

6.2.4. Clase GUI.Simulation

Descripción: Clase que gestiona la ventana de simulación, incluyendo la interfaz con pestañas para mostrar resultados gráficos y textuales, panel de parámetros, control de simulaciones múltiples y exportación de resultados a formato *PDF*.

Cuadro 6.7: Atributos de la clase GUI.Simulation

Nombre	Tipo	Descripción
<code>widget_parameter_panel_list</code>	<code>Dict[str, CtkFrame]</code>	Diccionario que contiene los paneles de parámetros de cada simulación.
<code>parametros_panel_list</code>	<code>Dict[str, CtkEntry]</code>	Diccionario que almacena las entradas de parámetros personalizados.

Continúa en la siguiente página

Cuadro 6.7 – continuación

Nombre	Tipo	Descripción
entries_args	Dict[str, CtkEntry]	Diccionario que almacena los campos de entrada para parámetros de simulación, indexados por nombre.
t_ini_list	Dict[str, CtkEntry]	Diccionario que contiene los campos de entrada para el tiempo inicial de cada simulación.
t_fin_list	Dict[str, CtkEntry]	Diccionario que contiene los campos de entrada para el tiempo final de cada simulación.
dt_tol_list	Dict[str, CtkEntry]	Diccionario con los campos de entrada del paso de integración y tolerancia para cada simulación.
graphic_list	List[CtkFrame]	Lista de marcos gráficos que contienen los gráficos de las simulaciones.
entries_args	Dict[str, CtkEntry]	Diccionario que almacena los campos de entrada para parámetros de simulación, indexados por nombre.
graphic_list	List[CtkFrame]	Lista de marcos gráficos que contienen los gráficos de las simulaciones.
result_list	List[CtkTextbox]	Lista de cajas de texto que muestran los resultados numéricos de las simulaciones.
fig_list	List[Figure]	Lista de objetos figura de Matplotlib usados para representar gráficas.
control_frame	CtkFrame	Marco que contiene los controles generales de la simulación (botones, entradas, etc.).
time_init	CtkEntry	Campo de entrada para el tiempo inicial de la simulación.
time_end	CtkEntry	Campo de entrada para el tiempo final de la simulación.
dt_tol	CtkEntry	Campo de entrada para el paso de tiempo o tolerancia de integración.
tabview	CtkTabView	Widget que gestiona las pestañas para múltiples simulaciones y resultados.
results_frame	CtkFrame	Marco contenedor de la sección de resultados.
params_panel	CtkFrame	Panel donde se muestran los parámetros editables de la simulación.

Continúa en la siguiente página

Cuadro 6.7 – continuación

Nombre	Tipo	Descripción
<code>scroll</code>	<code>CTkScrollableFrame</code>	Marco con barra de desplazamiento para contener elementos extensos.
<code>params_visible</code>	<code>bool</code>	Indica si el panel de parámetros está visible o colapsado.
<code>sim_count</code>	<code>int</code>	Contador para la cantidad de simulaciones abiertas.
<code>current_tab</code>	<code>str</code>	Nombre de la pestaña activa actualmente.

Cuadro 6.8: Métodos de la clase `GUI_Simulation`

Nombre	Parámetros	Descripción
<code>GUI_Simulation()</code>	<code>parent : CTk,</code> <code>controller : Controller</code>	Constructor que inicializa la ventana de simulación con sus componentes y parámetros.
<code>add_plus_tab()</code>	ninguno	Añade una pestaña especial para crear nuevas simulaciones.
<code>monitor_tab_change()</code>	ninguno	Detecta y gestiona el cambio entre pestañas de simulaciones.
<code>on_add_tab_click()</code>	ninguno	Acción al pulsar para agregar una nueva pestaña de simulación.
<code>add_simulation_tab()</code>	<code>name_simulation : str</code>	Crea y añade una nueva pestaña con el nombre dado para una simulación.
<code>widget_parameter_panel()</code>	ninguno	Construye el panel de parámetros editables para la simulación actual.
<code>toggle_menu()</code>	ninguno	Alterna la visibilidad del panel de parámetros.
<code>get_entries_args()</code>	ninguno	Retorna el diccionario con los campos de entrada y sus valores.
<code>get_time_args()</code>	ninguno	Obtiene los valores de tiempo de inicio, fin y paso para la simulación.
<code>get_result_text()</code>	ninguno	Obtiene el texto con los resultados numéricos de la simulación.
<code>get_result_figure()</code>	ninguno	Obtiene la figura gráfica de resultados generada.
<code>update_result_terminal()</code>	<code>result : List[List[str]]</code>	Actualiza la caja de texto con resultados numéricos.
<code>update_result_plot()</code>	<code>result : List[List[str]]</code>	Actualiza la gráfica con los nuevos datos de simulación.

6.2.5. Clase `GeneratorController`

Descripción: Clase que actúa como controlador principal del generador de programas de simulación continua. Coordina la lógica entre la interfaz gráfica y las operaciones del modelo, gestionando ecuaciones, condiciones, configuración del entorno, generación, compilación, ejecución y exportación del código fuente simulado.

Cuadro 6.9: Atributos de la clase `GeneratorController`

Nombre	Tipo	Descripción
<code>view</code>	<code>GUI_CTK</code>	Vista principal de la interfaz gráfica.
<code>log_label</code>	<code>CTkLabel</code>	Etiqueta usada para mostrar mensajes en la interfaz.

Cuadro 6.10: Métodos de la clase `GeneratorController`

Nombre	Parámetros	Descripción
<code>GeneratorController</code>	ninguno	Constructor que inicializa el controlador y la vista principal.
<code>add_equation</code>	<code>text_eq</code> , <code>text_var</code> , <code>text_const</code>	Añade una ecuación al sistema.
<code>add_condition</code>	<code>text_exp</code> , <code>text_act</code> , <code>text_var</code> , <code>text_const</code>	Añade una condición inicial.
<code>edit_equation</code>	<code>text_eq</code> , <code>text_var</code> , <code>text_const</code> , <code>name</code>	Edita una ecuación existente.
<code>edit_condition</code>	<code>text_exp</code> , <code>text_act</code> , <code>text_var</code> , <code>text_const</code> , <code>name</code>	Edita una condición inicial existente.
<code>delete_equation</code>	<code>name</code>	Elimina una ecuación por su nombre.
<code>delete_condition</code>	<code>name</code>	Elimina una condición por su nombre.
<code>get_equation</code>	<code>name</code>	Devuelve los componentes de una ecuación por nombre.
<code>get_condition</code>	<code>name</code>	Devuelve los componentes de una condición por nombre.
<code>get_list_equations</code>	ninguno	Devuelve la lista de ecuaciones registradas.
<code>get_list_conditions</code>	ninguno	Devuelve la lista de condiciones registradas.
<code>get_list_methods</code>	ninguno	Devuelve la lista de métodos de integración disponibles.
<code>get_list_output</code>	ninguno	Devuelve la lista de tipos de salida disponibles.

Continúa en la siguiente página

Cuadro 6.10 – continuación

Nombre	Parámetros	Descripción
<code>get_list_languages</code>	ninguno	Devuelve la lista de lenguajes de programación disponibles.
<code>get_variables</code>	ninguno	Devuelve la lista de variables usadas.
<code>get_parameters</code>	ninguno	Devuelve la lista de parámetros usados.
<code>get_view</code>	ninguno	Devuelve la vista gráfica principal asociada.
<code>set_language</code>	<code>language</code>	Establece el lenguaje de programación.
<code>set_method</code>	<code>method</code>	Establece el método de integración.
<code>set_output</code>	<code>output</code>	Establece el tipo de salida.
<code>set_log_label</code>	<code>label</code>	Asocia una etiqueta para mostrar mensajes.
<code>prepare_equation</code>	<code>text_eq</code> , <code>text_var</code> , <code>text_const</code>	Prepara y normaliza una ecuación antes de añadirla.
<code>prepare_condition</code>	<code>text_exp</code> , <code>text_act</code> , <code>text_var</code> , <code>text_const</code>	Prepara y normaliza una condición antes de añadirla.
<code>check_generator</code>	ninguno	Verifica que la configuración del generador sea válida.
<code>generate</code>	ninguno	Genera el código fuente según la configuración actual.
<code>compile</code>	ninguno	Compila el código generado si es necesario.
<code>execute</code>	ninguno	Ejecuta el programa generado.
<code>export_pdf</code>	ninguno	Exporta los datos o configuraciones a un archivo PDF.
<code>toggle_frame</code>	<code>mode</code> , <code>option</code> , <code>selected</code>	Cambia entre marcos en la interfaz gráfica.
<code>new_file</code>	ninguno	Limpia y reinicia la configuración actual.
<code>load_config</code>	<code>file_path</code>	Carga una configuración previamente guardada.
<code>save_config</code>	<code>file_path</code>	Guarda la configuración actual en un archivo.

6.2.6. Clase LogHandler

Descripción: Clase encargada de gestionar los mensajes de log para la interfaz gráfica. Asocia códigos de error o éxito con mensajes predefinidos y muestra dichos mensajes en una etiqueta

de la interfaz. Facilita la depuración y retroalimentación al usuario.

Cuadro 6.11: Atributos de la clase `LogHandler`

Nombre	Tipo	Descripción
<code>log_codes</code>	<code>dict</code>	Diccionario que contiene códigos de mensajes y sus textos asociados.
<code>log_label</code>	<code>CTkLabel</code>	Etiqueta de la interfaz donde se mostrarán los mensajes.

Cuadro 6.12: Métodos de la clase `LogHandler`

Nombre	Parámetros	Descripción
<code>LogHandler</code>	ninguno	Constructor que inicializa el diccionario de códigos y la etiqueta vacía.
<code>set_log_label</code>	<code>label :</code> <code>CTkLabel</code>	Asocia una etiqueta donde se mostrarán los mensajes.
<code>generate_log_msg</code>	<code>code : str</code>	Genera el mensaje correspondiente a un código dado.
<code>generate_log_msg_prm</code>	<code>error_code :</code> <code>str, parameter</code> <code>: str</code>	Genera un mensaje usando un código y un parámetro adicional.
<code>show_error</code>	<code>error_code :</code> <code>str</code>	Muestra un mensaje de error en la etiqueta.
<code>show_error_prm</code>	<code>error_code :</code> <code>str, parameter</code> <code>: str</code>	Muestra un mensaje de error con parámetro.
<code>show_success</code>	<code>success_code :</code> <code>str</code>	Muestra un mensaje de éxito en la etiqueta.
<code>show_success_prm</code>	<code>success_code :</code> <code>str, parameter</code> <code>: str</code>	Muestra un mensaje de éxito con parámetro.
<code>log_error</code>	<code>error_code :</code> <code>str</code>	Registra un mensaje de error en la etiqueta sin mostrarlo como pop-up.

6.2.7. Clase `ContinuousModelGenerator`

Descripción: Clase principal encargada de gestionar la información del modelo de simulación continua. Almacena ecuaciones, condiciones, parámetros de integración y configuración general. Se encarga de preparar los datos para el traductor seleccionado y de generar, compilar, ejecutar y exportar los resultados de la simulación.

Cuadro 6.13: Atributos de la clase ContinuousModelGenerator

Nombre	Tipo	Descripción
equations	list[Equation]	Lista de ecuaciones diferenciales del modelo.
conditions	list[Condition]	Lista de condiciones para la simulación.
initial_conditions	dict	Diccionario con las condiciones iniciales de las variables.
time_range	list[float]	Intervalo de tiempo de la simulación.
file_name	str	Nombre del archivo generado.
translator_type	str	Lenguaje de programación seleccionado (Python, C++, Java).
method	str	Método de integración numérica seleccionado.
output	str	Formato de salida de la simulación (gráfico, tabla, etc.).
path_file	str	Ruta del archivo fuente generado.
available_output	dict[str, list[str]]	Salidas disponibles por lenguaje.
available_methods	dict[str, str]	Métodos disponibles por lenguaje.
packages	dict[str, list[str]]	Paquetes necesarios por lenguaje.

Cuadro 6.14: Métodos de la clase ContinuousModelGenerator

Nombre	Parámetros	Descripción
ContinuousModelGenerator	ninguno	Constructor que inicializa todos los atributos del modelo.
get_equations	ninguno	Retorna la lista de ecuaciones.
get_variables	ninguno	Devuelve las variables utilizadas en el modelo.
get_parameters	ninguno	Devuelve las constantes utilizadas en las ecuaciones.
get_conditions	ninguno	Retorna la lista de condiciones.
get_initial_conditions	ninguno	Retorna el diccionario de condiciones iniciales.

Continúa en la siguiente página

Cuadro 6.14 – continuación

Nombre	Parámetros	Descripción
<code>get_time_range</code>	ninguno	Devuelve el intervalo de tiempo de la simulación.
<code>get_file_name</code>	ninguno	Retorna el nombre del archivo fuente.
<code>get_translator_type</code>	ninguno	Devuelve el lenguaje de programación seleccionado.
<code>get_method</code>	ninguno	Retorna el método de integración seleccionado.
<code>get_output</code>	ninguno	Devuelve el tipo de salida seleccionada.
<code>get_list_languages</code>	ninguno	Devuelve la lista de lenguajes disponibles.
<code>get_list_available_output</code>	ninguno	Devuelve las salidas disponibles para el lenguaje.
<code>get_constants</code>	ninguno	Retorna la lista de constantes usadas.
<code>get_var_identifiers</code>	ninguno	Retorna los identificadores de las variables.
<code>get_list_available_methods</code>	ninguno	Retorna los métodos disponibles.
<code>get_path</code>	ninguno	Devuelve la ruta del archivo generado.
<code>generate_file</code>	ninguno	Genera el archivo fuente del modelo.
<code>set_equations</code>	<code>eqs :</code> <code>list[Equation]</code>	Asigna una nueva lista de ecuaciones.
<code>set_conditions</code>	<code>conds :</code> <code>list[Condition]</code>	Asigna una nueva lista de condiciones.
<code>set_initial_conditions</code>	<code>init_cond :</code> <code>dict</code>	Establece las condiciones iniciales.
<code>set_time_range</code>	<code>t_range :</code> <code>list[float]</code>	Establece el intervalo de tiempo.
<code>set_file_name</code>	<code>name : str</code>	Asigna el nombre al archivo fuente.
<code>set_path</code>	<code>path : str</code>	Establece la ruta del archivo generado.
<code>set_translator_type</code>	<code>type : str</code>	Define el lenguaje de programación.
<code>set_output</code>	<code>output : str</code>	Define el tipo de salida de la simulación.
<code>set_method</code>	<code>method : str</code>	Establece el método numérico.
<code>set_translator</code>	ninguno	Configura el traductor correspondiente.

Continúa en la siguiente página

Cuadro 6.14 – continuación

Nombre	Parámetros	Descripción
set_output_type	output_type : str	Asigna un nuevo tipo de salida.
add_equation	text_eq : str, text_var : str, text_const : str	Añade una ecuación al modelo.
edit_equation	text_eq : str, text_var : str, text_const : str, index : int	Edita una ecuación existente.
delete_equation	index : int	Elimina una ecuación del modelo.
add_condition	text_exp, text_act, text_var, text_const	Añade una condición.
edit_condition	text_exp, text_act, text_var, text_const, index	Edita una condición existente.
delete_condition	index : int	Elimina una condición del modelo.
check_components	ninguno	Verifica si todos los elementos del modelo están definidos.
check_command	cmds : list[str]	Verifica que los comandos estén correctamente definidos.
check_generation	ninguno	Comprueba si el modelo está listo para generar el archivo.
check_compiler	ninguno	Comprueba si el compilador está disponible.
compile	ninguno	Compila el archivo generado si es necesario.
execute_simulation	args : list[str]	Ejecuta el programa de simulación.
get_output_simulation_file	path : str	Lee y devuelve los datos de salida de la simulación.
export_pdf	text, fig, params, filename	Exporta los resultados a un archivo PDF.

6.2.8. Clase SimulationModelGenerator

Descripción: Clase abstracta que define la estructura base para los generadores de código fuente de modelos de simulación continua. Gestiona el estado inicial, el tiempo de simulación, la ruta y nombre del archivo, el método numérico, y las funciones necesarias para escribir, compilar y ejecutar un archivo fuente.

Cuadro 6.15: Atributos de la clase SimulationModelGenerator

Nombre	Tipo	Descripción
<code>initial_state</code>	<code>Dict[str, float]</code>	Diccionario que contiene las variables de estado con sus valores iniciales.
<code>simulation_time</code>	<code>Tuple[float, float, float]</code>	Tupla con tiempo inicial, tiempo final y paso/tolerancia de integración.
<code>path_file</code>	<code>str</code>	Ruta del directorio donde se guardará el archivo fuente generado.
<code>name_file</code>	<code>str</code>	Nombre del archivo fuente que se generará.
<code>file</code>	<code>TextIOWrapper</code>	Objeto de archivo que representa el archivo fuente abierto para escritura.
<code>numerical_method</code>	<code>str</code>	Nombre del método numérico a utilizar (por ejemplo, 'Euler', 'RK4', etc.).
<code>equations</code>	<code>List[Equation]</code>	Lista que contiene las ecuaciones que representan el modelo de simulación.
<code>conditions</code>	<code>List[Condition]</code>	Lista que contiene las condiciones que afectan al modelo de simulación.

Cuadro 6.16: Métodos de la clase SimulationModelGenerator

Nombre	Parámetros	Descripción
<code>generate_file()</code>	ninguno	Método principal que coordina la generación completa del archivo fuente.
<code>set_constants()</code>	ninguno	Establece las constantes necesarias para el modelo y las escribe en el archivo.
<code>write_head_file()</code>	ninguno	Escribe la cabecera del archivo (includes, imports, etc.) según el lenguaje.

Continúa en la siguiente página

Cuadro 6.16 – continuación

Nombre	Parámetros	Descripción
<code>write_model_parameters()</code>	ninguno	Escribe los parámetros del modelo como variables globales o locales.
<code>write_equations()</code>	ninguno	Escribe las ecuaciones diferenciales del modelo.
<code>write_conditionals()</code>	ninguno	Escribe las condiciones adicionales que afectan la simulación (ej. ifs, eventos).
<code>write_euler_method()</code>	ninguno	Escribe la implementación del método de Euler explícito.
<code>write_euler_improved_method()</code>	ninguno	Escribe la implementación del método de Euler mejorado (Heun).
<code>write_runge_kutta_4_method()</code>	ninguno	Escribe la implementación del método de Runge-Kutta de orden 4.
<code>write_runge_kutta_fehlberg_method()</code>	ninguno	Escribe la implementación del método de Runge-Kutta-Fehlberg adaptativo.
<code>write_main()</code>	ninguno	Escribe la función principal (main) del archivo, que ejecuta la simulación.
<code>compile()</code>	ninguno	Compila el archivo fuente si es necesario (para C++ o Java).
<code>run()</code>	<code>args : str</code>	Ejecuta el archivo generado, pasando argumentos adicionales si es necesario.

6.2.9. Clase CppSimulationGenerator

Descripción: Clase derivada de `SimulationModelGenerator` que genera código fuente en lenguaje *C++* a partir de un modelo de simulación continua. Se encarga de adaptar las ecuaciones y condiciones del modelo al formato y sintaxis del lenguaje *C++*, incluyendo el manejo de operadores y expresiones como potencias.

Cuadro 6.17: Atributos de la clase `CppSimulationGenerator`

Nombre	Tipo	Descripción
<code>operators</code>	<code>Dict[str, str]</code>	Diccionario que almacena las traducciones de operadores de Python/sympy al formato de C++.
<code>pattern_pow</code>	<code>str</code>	Expresión regular que permite identificar y procesar potencias para convertirlas a la función <code>pow(base, exponente)</code> en C++.

Cuadro 6.18: Métodos de la clase `CppSimulationGenerator`

Nombre	Parámetros	Descripción
<code>CppSimulationGenerator()</code>	<code>eq :</code> <code>List[Equation],</code> <code>cond :</code> <code>List[Condition],</code> <code>ini_state :</code> <code>Dict[str, float],</code> <code>sim_time :</code> <code>Tuple[float, float,</code> <code>float],</code> <code>path_file : str,</code> <code>name_file : str,</code> <code>num_method : str</code>	Constructor que inicializa el generador con las ecuaciones, condiciones, estado inicial, parámetros de simulación, ruta y método numérico para generar código en C++.
<code>prepare_equations()</code>	<code>equation :</code> <code>Equation, subs_dict</code> <code>: Dict[str, str]</code>	Recibe una ecuación simbólica y un diccionario de sustituciones, y devuelve una cadena con la ecuación adaptada al lenguaje C++ (con uso de operadores adecuados y funciones como <code>pow</code>).

6.2.10. Clase JavaSimulationGenerator

Descripción: Clase derivada de `SimulationModelGenerator` encargada de generar el código fuente de un programa de simulación continua en el lenguaje `Java`. Adapta la estructura del modelo, ecuaciones y condiciones a la sintaxis de `Java`, considerando sus operadores y estructuras específicas. Además, incluye la generación de clases auxiliares como `Pair`.

Cuadro 6.19: Atributos de la clase `JavaSimulationGenerator`

Nombre	Tipo	Descripción
<code>operators</code>	<code>Dict[str, str]</code>	Diccionario que almacena los operadores necesarios para traducir expresiones simbólicas al formato de <code>Java</code> . Por ejemplo, transforma exponentes o funciones especiales al formato adecuado.
<code>pattern_pow</code>	<code>str</code>	Expresión regular usada para detectar y convertir potencias en expresiones tipo <code>Math.pow(base, exponente)</code> en <code>Java</code> .

Cuadro 6.20: Métodos de la clase `JavaSimulationGenerator`

Nombre	Parámetros	Descripción
<code>JavaSimulationGenerator()</code>	<code>eq :</code> <code>List[Equation],</code> <code>cond :</code> <code>List[Condition],</code> <code>ini_state :</code> <code>Dict[str,</code> <code>float],</code> <code>sim_time :</code> <code>Tuple[float,</code> <code>float, float],</code> <code>path_file :</code> <code>str, name_file</code> <code>: str,</code> <code>num_method :</code> <code>str</code>	Constructor que inicializa el generador con ecuaciones, condiciones, estado inicial, parámetros de simulación, ruta de archivo y método numérico. Establece también las reglas de conversión de operadores a <code>Java</code> .
<code>prepare_equations()</code>	<code>equation :</code> <code>Equation,</code> <code>subs_dict :</code> <code>Dict[str, str]</code>	Convierte una ecuación simbólica en una cadena compatible con <code>Java</code> , aplicando operadores y funciones como <code>Math.pow()</code> .
<code>write_pair_class()</code>	ninguno	Genera e inserta en el archivo fuente una clase auxiliar <code>Pair<A,B></code> para representar pares ordenados, necesaria para el manejo de datos en <code>Java</code> .

6.2.11. Clase PythonSimulationGenerator

Descripción: Clase derivada de `SimulationModelGenerator` encargada de generar código fuente para modelos de simulación continua en lenguaje *Python*. Se encarga de traducir ecuaciones simbólicas, escribir el archivo de simulación en Python y generar tanto la salida de datos como su representación gráfica.

Cuadro 6.21: Atributos de la clase `PythonSimulationGenerator`

Nombre	Tipo	Descripción
<code>operators</code>	<code>Dict[str, str]</code>	Diccionario con los operadores y funciones simbólicas necesarias para la traducción al formato Python. Por ejemplo, exponentes o funciones matemáticas.
<code>pattern_pow</code>	<code>str</code>	Expresión regular usada para detectar potencias y transformarlas a la función <code>**</code> de Python.

Cuadro 6.22: Métodos de la clase `PythonSimulationGenerator`

Nombre	Parámetros	Descripción
<code>PythonSimulationGenerator()</code>	<code>eq :</code> <code>List[Equation],</code> <code>cond :</code> <code>List[Condition],</code> <code>ini_state :</code> <code>Dict[str,</code> <code>float],</code> <code>sim_time :</code> <code>Tuple[float,</code> <code>float, float],</code> <code>path_file :</code> <code>str, name_file</code> <code>: str,</code> <code>num_method :</code> <code>str</code>	Constructor que inicializa el generador con los datos del modelo. Configura también los operadores simbólicos que serán utilizados para la conversión a Python.
<code>prepare_equations()</code>	<code>equation :</code> <code>Equation,</code> <code>subs_dict :</code> <code>Dict[str, str]</code>	Convierte una ecuación simbólica a su representación en código Python, aplicando los operadores definidos.
<code>write_results()</code>	ninguno	Escribe las instrucciones en el archivo Python para imprimir y guardar los resultados de la simulación en un archivo de texto o CSV.

Continúa en la siguiente página

Cuadro 6.22 – continuación

Nombre	Parámetros	Descripción
<code>plot_results()</code>	ninguno	Añade al archivo de salida el código necesario para generar gráficos de los resultados utilizando bibliotecas como <code>matplotlib</code> .

6.2.12. Clase Equation

Descripción: Representa una ecuación diferencial ingresada como texto, que se transforma en una expresión simbólica utilizando *sympy*. Esta clase identifica los símbolos y constantes presentes, almacena los valores de las constantes para su uso en el proceso de generación del código fuente.

Cuadro 6.23: Atributos de la clase Equation

Nombre	Tipo	Descripción
<code>text_equation</code>	<code>list[str]</code>	Representación textual de la ecuación.
<code>equation</code>	<code>sympy.Expr</code>	Expresión simbólica generada a partir del texto.
<code>symbols</code>	<code>list[sympy.Symbol]</code>	Lista de variables simbólicas involucradas.
<code>constants</code>	<code>list[sympy.Symbol]</code>	Lista de constantes simbólicas.
<code>constant_values</code>	<code>dict[str, float]</code>	Diccionario con los valores numéricos de las constantes.

Cuadro 6.24: Métodos de la clase Equation

Nombre	Parámetros	Descripción
<code>Equation</code>	<code>name : str,</code> <code>equation : str,</code> <code>symbols : str,</code> <code>constant_values : dict[str, float]</code>	Constructor de la clase. Procesa los componentes textuales y genera la expresión simbólica.
<code>get_name</code>	ninguno	Retorna el nombre de la ecuación.
<code>get_equation</code>	ninguno	Retorna la expresión simbólica de la ecuación.
<code>get_constants</code>	ninguno	Devuelve las constantes simbólicas utilizadas.
<code>get_constants_values</code>	ninguno	Retorna el diccionario de valores numéricos de las constantes.

Continúa en la siguiente página

Cuadro 6.24 – continuación

Nombre	Parámetros	Descripción
<code>get_symbol</code>	ninguno	Devuelve la lista de símbolos de variables.
<code>get_text_equation</code>	ninguno	Devuelve el texto original de la ecuación.
<code>get_text_symbols</code>	ninguno	Devuelve la representación textual de los símbolos.
<code>process_equations</code>	ninguno	Procesa y valida la ecuación para asegurar su estructura simbólica.
<code>check_components</code>	ninguno	Verifica que los componentes (símbolos y constantes) estén definidos correctamente.

6.2.13. Clase Condition

Descripción: representa una condición de frontera al modelo de simulación continua. Convierte condiciones ingresadas como texto en expresiones simbólicas utilizando *sympy*, identifica sus componentes (símbolos, operadores y constantes), y las procesa para su validación y uso en la generación del código fuente.

Cuadro 6.25: Atributos de la clase Condition

Nombre	Tipo	Descripción
<code>text_condition</code>	<code>list[str]</code>	Lista de condiciones en formato textual.
<code>available_operators</code>	<code>list[str]</code>	Operadores permitidos para las condiciones (por ejemplo, '=', '!=', etc.).
<code>constants</code>	<code>dict[str, float]</code>	Diccionario de constantes y sus valores numéricos.
<code>vars</code>	<code>list[sympy.Symbol]</code>	Variables involucradas en las condiciones.
<code>result</code>	<code>list[sympy.Equality]</code>	Resultados simbólicos obtenidos tras procesar las condiciones.
<code>conditions</code>	<code>list[sympy.Expr]</code>	Expresiones simbólicas de las condiciones procesadas.

Cuadro 6.26: Métodos de la clase `Condition`

Nombre	Parámetros	Descripción
<code>Condition</code>	<code>text_condition</code> : <code>list[str]</code> , <code>text_result</code> : <code>list[str]</code> , <code>text_vars</code> : <code>str</code> , <code>constants</code> : <code>dict[str, float]</code>	Constructor de la clase. Inicializa las condiciones y sus componentes simbólicos.
<code>process_condition</code>	ninguno	Procesa las condiciones textuales y genera sus formas simbólicas.
<code>get_available_operators</code>	ninguno	Devuelve la lista de operadores permitidos para condiciones.
<code>get_symbols</code>	ninguno	Devuelve la lista de símbolos utilizados en las condiciones.
<code>get_text_symbols</code>	ninguno	Devuelve la representación textual de los símbolos usados.
<code>get_conditions</code>	ninguno	Devuelve las condiciones como expresiones simbólicas.
<code>get_text_condition</code>	ninguno	Devuelve el texto original de las condiciones.
<code>get_result</code>	ninguno	Devuelve las igualdades simbólicas que representan los resultados.
<code>get_text_result</code>	ninguno	Devuelve la representación textual de los resultados.
<code>get_results_var</code>	ninguno	Devuelve la lista de variables resultado de las condiciones.
<code>get_constants</code>	ninguno	Devuelve la lista de constantes simbólicas utilizadas.
<code>get_text_constants</code>	ninguno	Devuelve la representación textual de las constantes.
<code>get_constants_values</code>	ninguno	Devuelve el diccionario de valores numéricos de las constantes.
<code>show_condition</code>	ninguno	Devuelve una lista con la representación textual completa de las condiciones procesadas.
<code>check_components</code>	ninguno	Verifica que símbolos y constantes estén definidos correctamente; devuelve una tupla indicando su validez.

6.3. Directorio de implementación

A continuación se presenta la estructura de directorios del proyecto. Esta organización ha sido diseñada siguiendo principios de modularidad y separación de responsabilidades:

```
TFG-MODELOS-CONTINUOS/
|
+-- ContinuousModelGenerator/      # Módulo principal de la aplicación
|   |
|   +-- resources/img/             # Imágenes para botones e interfaz
|       +-- add.png
|       +-- delete.png
|       +-- edit.png
|   |
|   +-- translators/               # Contiene los traductores
|       +-- cpp_translator.py      # Traduce a código C++
|       +-- java_translator.py     # Traduce a código Java
|       +-- python_translator.py   # Traduce a código Python
|       +-- translator.py          # Clase base para todos los traductores
|   |
|   +-- view/                     # Vistas de la interfaz gráfica (CustomTkinter)
|       +-- condition_view_ckt.py  # Vista para condiciones iniciales
|       +-- equation_view_ckt.py   # Vista para introducir ecuaciones
|       +-- main_view_ckt.py       # Vista principal de la aplicación
|       +-- simulation_view.py     # Vista para mostrar los resultados de simulación
|   |
|   +-- controller.py              # Controlador general de la aplicación (patrón MVC)
|   +-- conditions.py              # Lógica asociada a condiciones iniciales
|   +-- equation.py                # Lógica para manipular ecuaciones simbólicas
|   +-- generator.py               # Punto de entrada del software
|   +-- log_codes.py               # Códigos de log definidos para control interno
|   +-- log_handler.py             # Manejador de logs del sistema
|   +-- model.py                   # Modelo de datos principal
|
+-- dist/                          # Ejecutables generados por PyInstaller
|   +-- GeneratorSimulator          # Ejecutable para entornos Linux
|   +-- GeneratorSimulator.exe      # Ejecutable principal para Windows
|
+-- models/                        # (Opcional) Carpeta para almacenar los códigos fuente del usuario
|
+-- test/                          #Ejemplos de archivos de configuración
|
+-- generator.spec                  # Archivo de configuración para PyInstaller
+-- requirements.txt                # Dependencias necesarias para ejecutar el proyecto
+-- setup.py                       # Script de instalación del paquete (pip install)
+-- TFG-Modelos-Continuos.rar       # Archivo comprimido del proyecto completo
```

6.4. Creación de código fuente

La creación del código fuente por parte de la aplicación se basa en la modularidad que contienen este tipo de programas que representan sistemas de simulación continuos.

Es por ello, que se ha definido una estructura común independiente del lenguaje para los programas, compuesta por los siguientes bloques:

1. **Importación de bibliotecas necesarias:** se incluyen las librerías requeridas para cada lenguaje, como `math` en *C++*, `numpy` en *Python* o `java.util.*` en *Java*. Estas bibliote-

cas proporcionan soporte para operaciones matemáticas, estructuras de datos y funciones auxiliares.

2. **Definición de constantes y parámetros de simulación:** en este bloque se declaran las constantes del modelo, el tiempo de simulación, el incremento de tiempo (o la tolerancia en método adaptativos), y las estructuras de datos donde se almacenarán los resultados de las variables simuladas.
3. **Función que contiene las ecuaciones diferenciales y condiciones:** se genera una función que encapsula el sistema de ecuaciones diferenciales ordinarias junto con las condiciones de frontera impuestas. Esta función toma como argumentos los valores actuales de las variables y el tiempo, y devuelve las derivadas respectivas.
4. **Función del método de integración:** se incluye el método numérico de integración seleccionado por el usuario. Esta función realiza los cálculos iterativos para avanzar en el tiempo y evalúa las ecuaciones diferenciales en cada paso.
5. **Bloque principal:** representa el punto de entrada del programa. Aquí se inicializan las variables, se invocan las funciones de simulación y se gestiona la salida de los resultados. Además, este bloque incluye funciones de comprobación de argumentos de entrada adaptados al modelo del que se genera el código fuente.

Los códigos fuentes generados están preparados para ser ejecutados sin necesidad de usar el software, lo que garantiza su portabilidad y reutilización.

En el caso específico de *Java*, es necesario incluir la estructura de clase para ser reconocido por el compilador, además de una clase adicional `Pair` necesaria para algunos de los métodos de integración implementados.

6.5. Despliegue de aplicación

El proceso de despliegue de la aplicación se ha abordado desde dos enfoques complementarios: por un lado, mediante la instalación del paquete a través de *pip*, y por otro, mediante la generación de un ejecutable para sistemas operativos *Windows* y *Linux* utilizando *PyInstaller*.

6.5.1. Instalación mediante *pip*

Para permitir la instalación del generador como un paquete *Python* estándar, se ha estructurado el proyecto conforme a las buenas prácticas del ecosistema de paquetes del lenguaje. En concreto, se ha creado un archivo `requirements.txt` en el que se indican todas las dependencias necesarias para el correcto funcionamiento de la aplicación, necesario para el instalador `setup.py`. Ambos archivos permiten instalar todas las bibliotecas requeridas y la adición del paquete de manera automática mediante la ejecución del siguiente comando en el directorio raíz del programa:

```
pip install .
```

Una vez instalado el paquete, podrá iniciarse mediante el siguiente comando :

```
cmg
```

Este enfoque permite que el generador sea fácilmente integrable en cualquier entorno *Python*, además de facilitar su distribución a través de repositorios como *PyPi* si en el futuro así se decidiera.

6.5.2. Generación del ejecutable mediante *PyInstaller*

Esta opción está destinada para aquellos usuarios que prefieran utilizar la aplicación como un ejecutable autónomo. *PyInstaller* permite empaquetar la aplicación junto con todas sus dependencias en un único archivo ejecutable.

La generación del ejecutable se realiza a partir de un archivo de especificaciones (*generation.spec*), que define los parámetros de compilación, las rutas a los archivos necesarios, los recursos adicionales, y el punto de entrada de la aplicación.

A partir de este archivo `.spec`, el ejecutable se genera con:

```
pyinstaller generator.spec
```

Es importante añadir que *PyInstaller* crea ejecutables específicos para el sistema operativo desde el que se realiza la compilación. Es decir, para generar un ejecutable para *Windows*, es necesario realizar la compilación desde una máquina con dicho sistema operativo. Del mismo modo, para generar el ejecutable compatible con distribuciones *Linux*, se ha utilizado el sub-sistema de *Windows* para *Linux* (WSL), que proporciona un entorno nativo de *Linux* desde el propio *Windows*, permitiendo así realizar la compilación adecuada sin necesidad de una máquina virtual o equipo independiente.

Este enfoque garantiza que los usuarios finales de ambos sistemas operativos puedan ejecutar la aplicación sin necesidad de realizar configuraciones adicionales, facilitando su uso y distribución.

Sin embargo, esta opción no soluciona los problemas de dependencias necesarias para la simulación de los códigos fuente generados, entre ellas se encuentran:

- Los compiladores de *C++*, *Java* e intérprete de *Python*.
- Para *Python* las bibliotecas *numpy* y *matplotlib*.

Estos deberán ser instalados manualmente por el usuario para poder ejecutar y/o simular los códigos fuente generados, ya sea de forma autónoma o a través de la herramienta.

6.5.3. Limitación en entornos virtualizados sin acceso a pip

Durante el desarrollo y pruebas de la herramienta, se ha identificado una limitación relevante al ejecutar el generador de modelos para simular código fuente en *Python* desde una máquina virtual *Linux* sin acceso al gestor de paquetes `pip`. En estos entornos, suele recomendarse el uso de `pipx` como alternativa segura para instalar y aislar aplicaciones *Python*. Sin embargo, no se ha utilizado `pipx` porque su implementación requeriría introducir características adicionales en la ejecución del código que interferirían con la detección automática de entornos que dispongan, o no, de `pip`.

Esta limitación no existe en los casos de la ejecución de los códigos fuentes tanto en *Java* como *C++*, siendo totalmente operativos en entornos virtualizados.

Capítulo 7

Pruebas

La fase de pruebas constituye un elemento fundamental en el desarrollo de cualquier sistema de software, especialmente en aplicaciones científicas y de simulación donde la precisión y confiabilidad de los resultados son críticas. Las pruebas no solo permiten verificar que el sistema cumple con los requisitos funcionales establecidos, sino que también garantizan la robustez, estabilidad y usabilidad del software.

En el contexto de nuestro sistema, las pruebas adquieren una importancia particular debido a la naturaleza compleja de los cálculos matemáticos involucrados y la necesidad de generar código ejecutable confiable. Un error en la implementación podría propagarse a través de todo el proceso de simulación, comprometiendo la validez de los resultados obtenidos.

7.1. Preparación del entorno de pruebas

El entorno de pruebas que se va a utilizar consta con las siguientes especificaciones:

- **Procesador.** 12th Gen Intel(R) Core(TM) i7-1260P 2.10 GHz
- **RAM.** 32,0 GB.
- **Sistema Operativo.** Windows 11 Home Versión: 24H2.
- **Almacenamiento.** 1TB.

La aplicación será puesta a prueba a través de la instalación comentada en el capítulo de *Implementación/Despliegue de la aplicación /Instalación mediante pip*.

7.2. Pruebas por funcionalidad

Las pruebas se dividen entre módulos para facilitar la comprobación de todos los componentes. Los módulos que formarán este apartado son:

- **Módulo principal.** Se valorará las respuestas de la ventana principal ante entradas no esperadas y la generación del código fuente.
- **Módulo de ecuaciones.** Se valorará la adaptabilidad ante las entradas del usuario, mostrándose los errores o mensajes de éxito pertinentes.
- **Módulo de condiciones.** Se valorará los mismos objetivos que en el módulo de ecuaciones.
- **Módulo de simulación.** Se valorará la comprobación de entradas, la salida correcta de los datos y una exportación de resultados en formato *PDF* de forma correcta.

7.2.1. Prueba del módulo principal

En este apartado se realiza las siguientes pruebas a la ventana principal de la herramienta:

- Generar código fuente sin definir ninguna configuración e ir progresivamente añadiendo la configuración necesaria según los errores que vaya mostrando la herramienta:
 1. Generamos sin definir ninguna configuración. La herramienta muestra el error *“Error: No se ha definido lenguaje de salida”*.
 2. Seleccionamos el lenguaje de salida y generamos. Muestra el error *“Error: No se ha definido la salida”*.
 3. Seleccionamos la salida del código fuente y generamos. Muestra el error *“Error: No se ha definido ninguna ecuación”*.
 4. Introducimos una ecuación en la herramienta y generamos. Muestra el error *“Error: No se ha elegido un método de numérico de integración”*.
 5. Seleccionamos el método numérico y generamos. Aparece una ventana para seleccionar el directorio y nombre del código fuente generado.
 6. Si se ha generado correctamente se muestra un mensaje al usuario *“Éxito: Generación de código directorio/nombre_archivo exitosa.”*, en caso contrario se muestra un mensaje de error *“Error: La generación de código ha fallado.”*.
- Editar ecuación sin seleccionar ninguna. El sistema muestra el error *“Error: Seleccione una ecuación para editarla”*.
- Eliminar ecuación sin seleccionar ninguna. El sistema muestra el error *“Error: Seleccione una ecuación para editarla”*.
- Editar condición sin seleccionar ninguna. El sistema muestra el error *“Error: Seleccione una condición para editarla”*.
- Eliminar condición sin seleccionar ninguna. El sistema muestra el error *“Error: Seleccione una condición para editarla”*.

Los errores provenientes de la generación del código normalmente se deben a falta de permisos de escritura en el directorio destino o sobrescribir algún archivo que se encuentre en uso.

Por otro lado, se evita la comprobación de existencia de condiciones ya que no todos los modelos de simulación continua las contienen.

En resumen, el resultado se adecúa perfectamente a lo esperado de este módulo. La salida de errores guía de forma correcta al usuario en su uso.

7.2.2. Prueba del módulo de ecuaciones

En este apartado se realizan las siguientes prueba a la ventana de añadir/editar ecuaciones.

- Introducir una ecuación sin ningún campo definido. Se muestra el error : *“Error: La expresión de la ecuación no está bien definida”*.
- Introducir una ecuación sin especificar en su parte derecha $d(variable)/dt$. Se muestra el error : *“Error: El nombre de la ecuación no es válido. Debe ser de tipo $d(var)/dt$ ”*.
- Introducir una ecuación con una expresión inválida. Se muestra el error : *“Error: La expresión de la ecuación no está bien definida”*.

- Introducir una ecuación con variables o constantes que sean símbolos reservados de la biblioteca `sympy`. Se muestra el error : *“Error: El nombre del “símbolo” no es válido. (Nombre reservado)”*.
- Introducir una ecuación con símbolos que se encuentren definidos como variables y como constantes simultáneamente. Se muestra el error : *“Error: El símbolo “símbolo” se encuentra definido como variable y constante”*.
- Introducir una ecuación con símbolos no declarados como variable ni como constante. Se muestra el error : *“Error: El símbolo “símbolo” no se encuentra definido como variable o constante”*.
- Introducir una ecuación correctamente definida. Se muestra un mensaje de éxito: *“Éxito:Ecuación_“índice” se ha añadido correctamente.”*.

Con este tipo de errores se intenta evitar cualquier entrada inválida que provoque la generación de código fuente inválido. Además, se comprueba que el programa se comporta según lo esperado.

7.2.3. Prueba del módulo de condiciones

En este apartado se realizan las siguientes pruebas a la ventana de añadir/editar condiciones.

- Introducir una condición con una expresión vacía o mal formada. Se muestra el error: *“Error: La expresión de la condición no está bien definida”*.
- Introducir una condición con más de un operador lógico o de comparación. Se muestra el error: *“Error: La expresión de la condición contiene más de un operador: “operador””*.
- Introducir una condición sin definir ninguna variable involucrada. Se muestra el error: *“Error: No se encuentran definidas variables”*.
- Introducir una condición sin definir ninguna acción a ejecutar. Se muestra el error: *“Error: No se encuentran definidas acciones”*.
- Introducir una condición que contenga símbolos no declarados como variables ni como constantes. Se muestra el error: *“Error: El símbolo “símbolo” no se encuentra definido como variable o constante”*.
- Introducir una condición que contenga símbolos definidos simultáneamente como variables y constantes. Se muestra el error: *“Error: El símbolo “símbolo” se encuentra definido como variable y constante”*.
- Introducir una condición con un símbolo inválido (por ejemplo, reservado o con caracteres no permitidos). Se muestra el error: *“Error: El símbolo “símbolo” no es válido en la expresión de la condición”*.
- Introducir una condición correctamente definida con su expresión, variables y acciones válidas. Se muestra un mensaje de éxito: *“Éxito: Condición_“índice” se ha añadido correctamente.”*.

Con este conjunto de validaciones se busca prevenir cualquier entrada incorrecta que pudiera derivar en la generación de código fuente inválido o erróneo. Además, estos controles permiten verificar que el comportamiento del programa ante entradas mal formadas es coherente con lo esperado.

7.2.4. Prueba del sistema de ejecución de simulaciones

Dado que la herramienta incorpora validaciones previas estrictas sobre ecuaciones y condiciones, la mayoría de los errores se detectan antes de la generación y ejecución del código. Por ello, las pruebas en esta etapa son pocas y se centran en verificar aspectos básicos como la disponibilidad de compiladores, la validez de las entradas y la correcta finalización del proceso. Las pruebas realizadas son las siguientes:

- Iniciar la generación de código sin tener instalados los compiladores o paquetes necesarios. Se muestra el error: *“Error: No se encuentran los paquetes necesarios “nombre_paquete”*”.
- Ingresar un valor inválido en uno de los campos de entrada para la simulación. Se muestra el error: *“Error: El valor de “campo” no es válido para la entrada de la simulación”*.
- Exportar correctamente los resultados de una simulación a un archivo *PDF*. Se muestra un mensaje de éxito: *“Éxito: Exportación de la simulación a “nombre_archivo.pdf.exitosa”*”.
- Completar correctamente el proceso de generación de código. Se muestra un mensaje de éxito: *“Éxito: Generación de código “nombre_archivo” exitosa”*.
- Ejecutar correctamente una simulación. Se muestra un mensaje de éxito: *“Éxito: Ejecución de la simulación exitosa”*.

Se abarcan todas las situaciones de error posibles para finalizar con un resultado acorde a los objetivos.

7.2.5. Validación de requisitos funcionales

A continuación, se presenta la Tabla 7.1 que resume el cumplimiento de los requisitos funcionales definidos para la herramienta desarrollada. Cada funcionalidad ha sido verificada y se confirma que el sistema responde correctamente a las especificaciones establecidas.

Cuadro 7.1: Verificación del cumplimiento de los requisitos funcionales

ID	Requisito Funcional	Cumplido
RF.1.1	Generación de código fuente en C++, Python o Java según la configuración seleccionada.	Sí
RF.2	Introducción, edición y eliminación de ecuaciones diferenciales.	Sí
RF.3	Introducción, edición y eliminación de las condiciones que afectan a las variables del modelo.	Sí
RF.4	Selección del lenguaje de programación de salida del código fuente.	Sí
RF.5	Selección del método de integración numérica.	Sí
RF.6	Simulación del modelo mediante la ejecución del código fuente generado.	Sí
RF.7.1	Definición de las condiciones iniciales de las variables.	Sí
RF.7.2	Definición del valor de las constantes.	Sí
RF.7.3	Definición del tamaño del paso o tolerancia del método numérico usado.	Sí
RF.8	Compilación del código fuente generado por el sistema.	Sí
RF.9.1	Almacenamiento de las ecuaciones junto con las variables y constantes que las definen.	Sí
RF.9.2	Almacenamiento de las condiciones junto con su expresión lógica, acciones, variables y constantes.	Sí
RF.9.3	Almacenamiento del lenguaje y método numérico seleccionado.	Sí
RF.10	Carga del archivo de configuración.	Sí
RF.11	Eliminación de los datos actuales para iniciar una nueva definición del modelo.	Sí

Capítulo 8

Conclusiones

El objetivo general de este Trabajo de Fin de Grado ha sido el desarrollo de una herramienta capaz de generar automáticamente programas de simulación continua en distintos lenguajes de programación, permitiendo su ejecución y visualización de resultados. A lo largo del desarrollo se han superado los principales retos técnicos, alcanzando resultados funcionales y cumpliendo con el objetivo general planteado.

A continuación, se realiza un repaso detallado de los objetivos específicos establecidos al inicio del proyecto:

- **OE1:** Diseñar una interfaz que permita al usuario definir modelos continuos de simulación de forma estructurada y accesible.

Porcentaje de realización: 87 %

Se ha desarrollado una interfaz en CustomTkinter que permite introducir ecuaciones, constantes, condiciones, simulación y visión de forma clara y guiada. Sin embargo debido a limitaciones en el desarrollo de la aplicación en entornos *Windows* y *Linux* no se ha conseguido todo lo que se buscaba.

Evidencia: Capítulo 5 y Sección 5.4.

- **OE2:** Implementar un sistema de análisis y procesamiento que permita vincular variables, condiciones iniciales y parámetros del sistema.

Porcentaje de realización: 100 %

Se ha diseñado un sistema que usa *sympy* para interpretar expresiones simbólicas y validar entradas, enlazando automáticamente variables y parámetros entre ecuaciones y condiciones.

Evidencia: Capítulo 6, Sección 6.1.

- **OE3:** Integrar métodos numéricos para la resolución de ecuaciones diferenciales que garanticen la correcta simulación de los sistemas continuos.

Porcentaje de realización: 100 %

Se han integrado métodos como Euler y Runge-Kutta (de distintos órdenes) para permitir al usuario elegir el más adecuado según su modelo.

Evidencia: Capítulo 6, Sección 6.2.

- **OE4:** Garantizar que el código fuente generado sea autónomo, legible y portable sin dependencia de plataformas específicas y/o librerías externas complejas.

Porcentaje de realización: 100 %

El código generado en C++, Python o Java es independiente y puede ejecutarse sin requerimientos externos, facilitando su distribución y reutilización.

Evidencia: Capítulo 6, Sección 6.4.

- **OE5:** Integrar el proceso de simulación dentro del software para facilitar el uso del código fuente generado por parte del usuario.

Porcentaje de realización: 100 %

Se ha implementado un sistema de ejecución interna del código generado, permitiendo visualizar directamente los resultados desde la propia interfaz.

Evidencia: Capítulo 6 y Capítulo 5 , Sección 5.4.

- **OE6:** Validar la funcionalidad de la herramienta mediante casos de prueba, evaluando las respuestas del software ante casos no esperados.

Porcentaje de realización: 100 %

Se han llevado a cabo pruebas de validación del código generado, así como pruebas ante entradas erróneas, con resultados coherentes y sin fallos críticos.

Evidencia: Capítulo 7 y Tabla 7.1.

Durante la realización de este proyecto, mi formación previa en asignaturas como Programación Orientada a Objetos, Simulación de Sistemas y Estructuras de Datos ha sido esencial para abordar el proyecto. Sin embargo, también he tenido que enfrentarme a nuevos retos, como el empaquetado multiplataforma, el control de bibliotecas nunca utilizadas como *sympy* y *CustomTkinter*, o el diseño de interfaces modernas, los cuales he resuelto mediante la autoformación, lectura de documentación oficial y el apoyo en herramientas de inteligencia artificial.

Desde una perspectiva ética, el proyecto busca facilitar el acceso al modelado matemático a estudiantes o profesionales sin conocimientos técnicos avanzados, alineándose con los principios de accesibilidad y democratización del conocimiento. Si bien no se ha abordado un ODS de forma directa, sí se contribuye de forma indirecta a la mejora de la educación (ODS 4).

Finalmente, a nivel personal, me siento muy satisfecho con el trabajo realizado. Este proyecto me ha permitido aplicar gran parte de los conocimientos adquiridos durante el grado y, al mismo tiempo, mejorar mis habilidades en planificación, pensamiento crítico y resolución de problemas. Aunque ha habido momentos de dificultad, considero que el resultado final refleja un trabajo riguroso, útil y técnicamente sólido.

Capítulo 9

Trabajos Futuros

Aunque la herramienta desarrollada cumple con todos los objetivos propuestos, existen diversas líneas de mejora y ampliación que podrían abordarse en futuros desarrollos. A continuación, se presentan algunas de las más destacables:

- **Importación de ecuaciones y condiciones desde formatos visuales y usables:** Una futura mejora interesante sería permitir la introducción de ecuaciones y condiciones mediante métodos alternativos a la entrada manual de texto, como la importación desde imágenes (reconocimiento OCR de expresiones matemáticas), documentos de texto estructurados (por ejemplo, \LaTeX o Markdown), o incluso mediante entrada manuscrita digitalizada. Esto facilitaría el uso de la herramienta por parte de usuarios con distintos niveles de familiaridad con el lenguaje matemático simbólico y reduciría errores de entrada.
- **Ampliación del número de lenguajes de programación soportados:** Actualmente, el generador es capaz de producir código fuente en C++, Python y Java. Sin embargo, sería interesante extender el soporte a otros lenguajes populares en ingeniería y simulación, como MATLAB, Julia, Fortran o incluso lenguajes orientados a web como JavaScript/WebAssembly, lo que permitiría ejecutar simulaciones directamente en navegadores web.
- **Exportación de resultados a múltiples formatos:** En versiones futuras, se podría añadir la opción de exportar los resultados de las simulaciones a formatos estándar como CSV, Excel, JSON o incluso gráficos en PDF/PNG. Esto facilitaría la integración de los resultados en informes, publicaciones o análisis posteriores con otras herramientas externas.
- **Implementación de modelos híbridos o sistemas discretos:** Aunque actualmente la herramienta está orientada a la simulación continua, se podría considerar la inclusión de soporte para modelos híbridos (combinación de eventos discretos con dinámica continua) o modelos puramente discretos, ampliando así su campo de aplicación.
- **Simulaciones en tiempo real y control de parámetros dinámicos:** Una mejora valiosa sería permitir la simulación interactiva en tiempo real, con la posibilidad de modificar parámetros del modelo mientras se ejecuta la simulación y observar el impacto de forma instantánea.
- **Historial de versiones y almacenamiento en la nube:** Integrar un sistema de gestión de versiones de modelos y sincronización con servicios en la nube (como Google Drive o GitHub) permitiría a los usuarios guardar su progreso, colaborar y compartir modelos de manera más eficiente.
- **Mejoras en accesibilidad y experiencia de usuario (UX):** Futuras versiones podrían incluir mejoras en la interfaz gráfica para facilitar la navegación, así como opciones de accesibilidad como lectura de pantalla, modos de alto contraste, o integración con asistentes de voz.

Estas líneas de desarrollo suponen un camino prometedor para seguir evolucionando la herramienta, ampliar su utilidad en distintos contextos educativos y profesionales, y acercarla a un público más amplio.

Bibliografía

- [1] Python Packaging Authority. pip: The python package installer. <https://pip.pypa.io/>, 2025. Accedido el 25 de febrero de 2025.
- [2] R. C. Bú. *Simulación: un enfoque práctico*. Editorial Limusa, México, 1994.
- [3] Steven C. Chapra and Raymond P. Canale. *Numerical Methods for Engineers*. McGraw-Hill Higher Education, Boston, 6th edition, 2010.
- [4] J. M. A. Danby. *Computer modeling: From sports to spaceflight – from order to Chaos*. Willmann-Bell, Richmond, Va, 1997.
- [5] CustomTkinter Developers. Customtkinter: Modern ui library for python tkinter. <https://customtkinter.tomschimansky.com/>, 2023. Accedido el 13 de febrero de 2025.
- [6] Figma Inc. Figma — collaborative interface design tool. <https://www.figma.com/>, 2024. Último acceso: 7 de marzo de 2025.
- [7] Free Software Foundation. Gnu compiler collection (gcc) - c++ compiler (g++). <https://gcc.gnu.org/>, 2025. Versión utilizada en Linux. Último acceso: 2025.
- [8] GitHub. GitHub Copilot [asistente de programación basado en ia]. <https://github.com/features/copilot>, 2023. Accedido el 10 de marzo de 2025.
- [9] ReportLab Inc. Reportlab open source pdf library. <https://www.reportlab.com/opensource/>, 2025. Accedido el 12 de abril de 2025.
- [10] International Organization for Standardization. Iso/iec 14882:2020: Programming languages — c++. <https://isocpp.org/std/the-standard>, 2020. Accedido el 21 de febrero de 2025.
- [11] MathWorks. Simulink - simulation and model-based design. <https://www.mathworks.com/products/simulink.html>, 2025. Consultado el 10 de marzo de 2025.
- [12] Marvin L. Minsky. *Semantic Information Processing*. MIT Press, Cambridge, 1965.
- [13] Edward E. L. Mitchell and Joseph S. Gauthier. Advanced continuous simulation language (acsl). *SIMULATION*, 26(3):72–78, 1976.
- [14] T. H. Naylor, Manuel Sunderland, Lian Karp, and Rodolfo Luthe García. *Técnicas de simulación en computadoras*. Limusa, México, 1982. Versión española de Manuel Sunderland y Lian Karp; revisión de Rodolfo Luthe García.
- [15] Thomas H. Naylor, J. M. Finger, James L. McKenney, William E. Schrank, and Charles C. Holt. Verification of computer simulation models. *Management Science*, 14(2):B92–B106, 1967.
- [16] OpenAI. ChatGPT (versión gpt-4) [modelo de lenguaje grande]. <https://chat.openai.com>, 2023. Accedido el 10 de marzo de 2025.

- [17] Oracle Corporation. Java se 21 documentation. <https://docs.oracle.com/en/java/javase/21/>, 2023. Accedido el 21 de febrero de 2025.
- [18] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. Palgrave Macmillan, New York, 6th edition, 2005.
- [19] Python Software Foundation. Python 3.13.14 documentation. <https://docs.python.org/3/>, 2023. Accedido el 21 de febrero de 2025.
- [20] Y Fernández Romero and Y Díaz González. Patrón modelo-vista-controlador. *Revista Telemática*, 11(1):47–57, 2012.
- [21] R. E. Shannon. *Simulación de Sistemas. Diseño, desarrollo e implementación*. Trillas, México, 1988.
- [22] Matplotlib Development Team. Matplotlib: Visualization with python. <https://matplotlib.org/>, 2025. Accedido el 25 de febrero de 2025.
- [23] PyInstaller Development Team. Pyinstaller: Freeze python applications into stand-alone executables. <https://pyinstaller.org/>, 2025. Último acceso: 2025.
- [24] SymPy Development Team. Sympy: Python library for symbolic mathematics. <https://www.sympy.org/>, 2025. Accedido el 25 de febrero de 2025.
- [25] MinGW w64 Project. Mingw-w64 - gcc for windows 64 & 32 bits. <https://www.mingw-w64.org/>, 2025. Compilador g++ para Windows. Último acceso: 2025.

Anexos

A. Manual de usuario

Este manual explica los componentes de la aplicación y su uso a nivel de usuario.

Pantalla principal

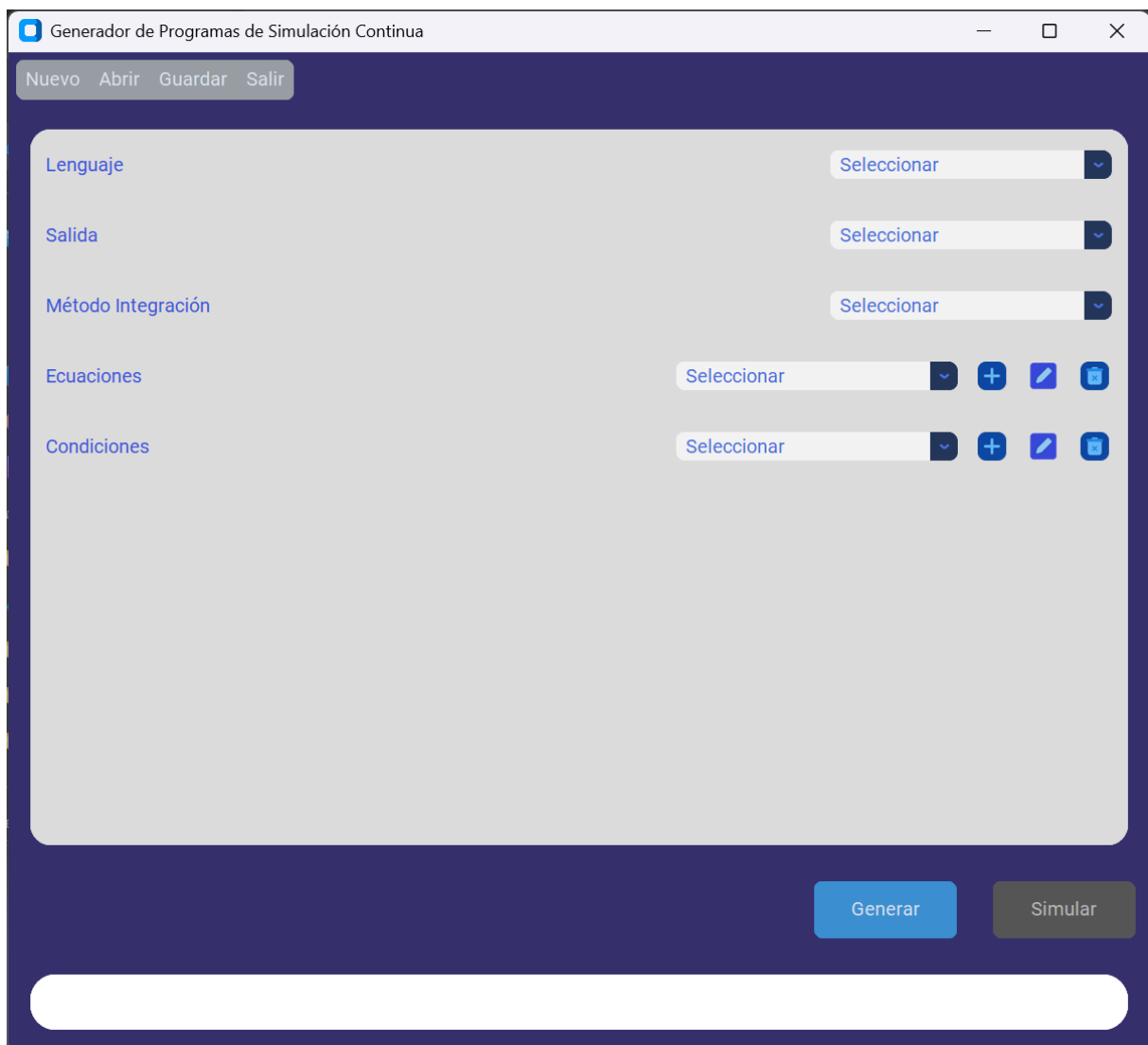


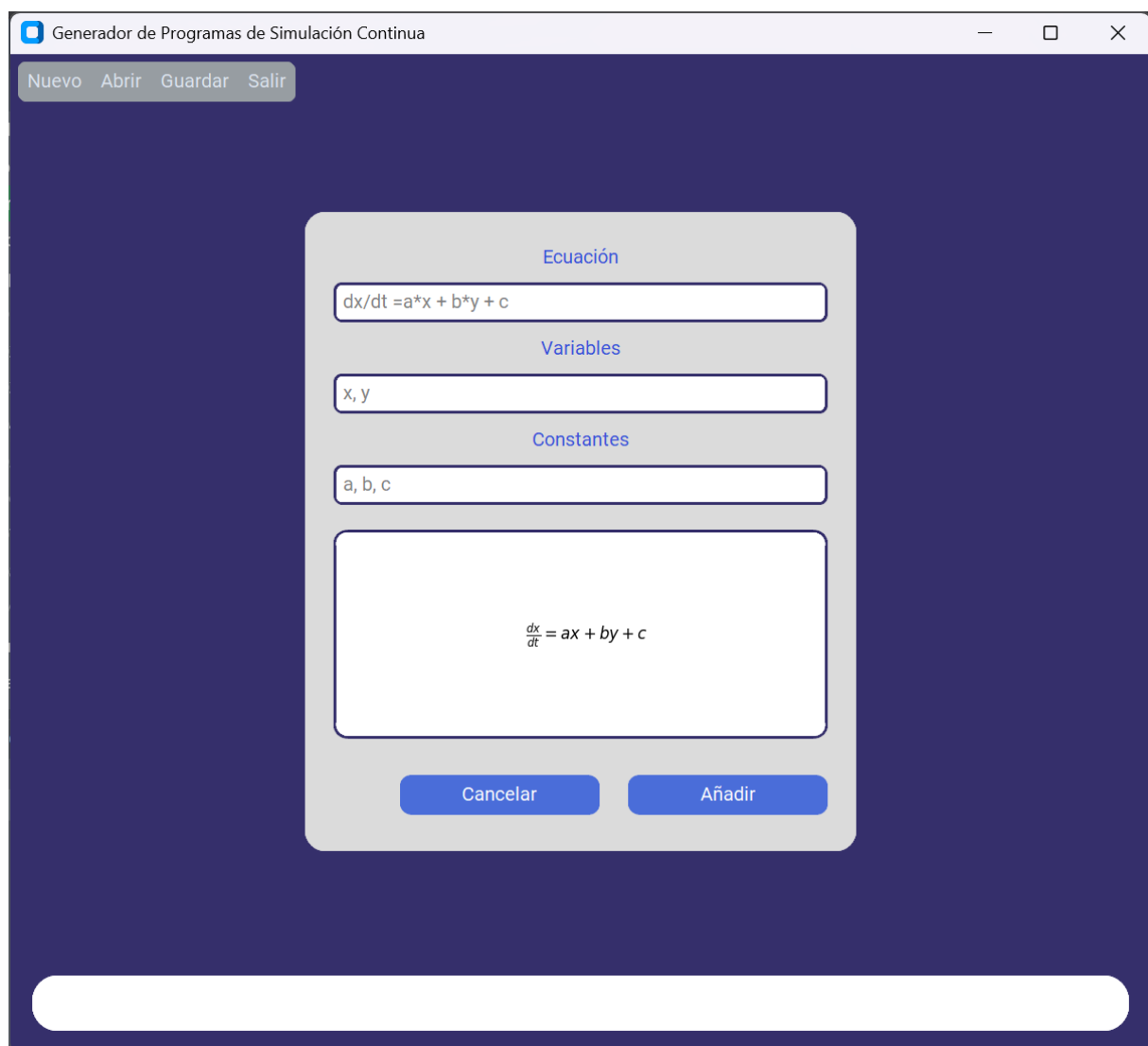
Figura 1: Pantalla principal de la aplicación

Desde esta pantalla el usuario puede:

- Utilizar los botones de la barra de tareas para crear un nuevo proyecto, abrir uno existente, guardar o salir.
- Seleccionar el **lenguaje** en el que se generará el código fuente (*Python*, *C++* o *Java*).
- Elegir el tipo de **salida**. Este campo dependerá del lenguaje seleccionado. Actualmente se encuentran disponibles *csv* y *plot* (Solo en *Python*).
- Escoger el **método de integración**.
- Añadir, editar o eliminar **ecuaciones** y **condiciones** mediante los botones correspondientes.
- Generar o simular el modelo. La opción de simular sólo está disponible si el usuario selecciona *csv* como método de salida.

En el caso de que se obtenga un error en la generación del código fuente deberá comprobar si tiene permisos de escritura en el directorio.

Pantalla ecuación



The screenshot shows a web application window titled "Generador de Programas de Simulación Continua". At the top, there is a navigation bar with buttons: "Nuevo", "Abrir", "Guardar", and "Salir". The main content area is dark blue. In the center, a light gray dialog box titled "Ecuación" is open. It contains three input fields: "Ecuación" with the text "dx/dt = a*x + b*y + c", "Variables" with the text "x, y", and "Constantes" with the text "a, b, c". Below these fields is a large white box displaying the mathematical equation $\frac{dx}{dt} = ax + by + c$. At the bottom of the dialog box are two buttons: "Cancelar" and "Añadir".

Figura 2: Pantalla de añadir/editar ecuación.

En esta pantalla se lleva a cabo el proceso de añadir/editar ecuaciones al sistema. Existen diferentes campos nombrados con *placeholders* para facilitar la entrada correcta de los campos.

- **Ecuación.** En ella se detalla la expresión de la ecuación diferencial. La entrada esperada es del tipo $d(var)/dt = exp$, donde *var* representa el nombre de la variable que se obtiene en el proceso de derivación y *exp*, la expresión que representa el cálculo de las variables. En el campo expresión se pueden incluir funciones matemáticas como *sin*, *cos*, *tan*, *sqrt*, *e* y π .
- **Variables.** En ella se deben definir las variables, separadas por comas, que forman parte de la derecha de la expresión (a partir del igual).
- **Constantes.** En ella se deben definir las constantes con el mismo formato que las variables y con las mismas restricciones.

Esta pantalla contiene mecanismos de seguridad en el caso de que no se cumpla con el formato esperado en el sistema, mostrando un mensaje de error en el caso de que la entrada fuera incorrecta.

Pantalla Condición

Figura 3: Pantalla de añadir/editar condición.

Para añadir/editar una condición se deben introducir los siguientes campos:

- **Expresiones lógicas.** Estarán definidas las expresiones que representan la condición, separadas por comas. Los operadores disponibles son: $=$, $<=$, $>=$, $>$, $!$, $=$.
- **Acción.** Son los cambios en las que se realizarán durante la simulación. Deben estar separados por comas y con el formato *iden = acción*.
- **Variables.** En ella se deben definir las variables, separadas por comas, que forman parte de la derecha de la expresión (a partir del igual).
- **Constantes.** En ella se deben definir las constantes con el mismo formato que las variables y con las mismas restricciones.

Pantalla Simulación

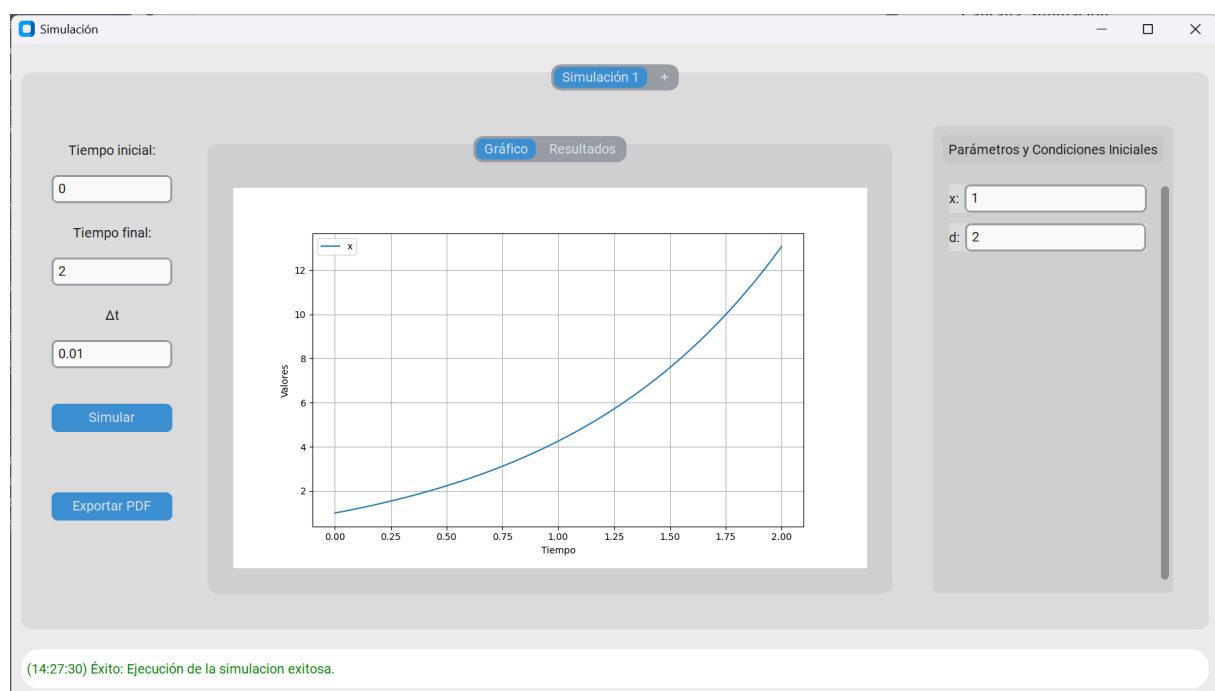


Figura 4: Pantalla de simulación de modelos.

En esta pantalla se podrán realizar múltiples simulaciones y visualizar los resultados obtenidos ya sea de forma gráfica o tabular.

La interfaz permite definir distintos parámetros:

- Tiempo inicial, tiempo final y paso de integración (o tolerancia).
- En el panel lateral derecho se muestran las variables y constantes para definir los valores de ellas en la simulación.

Una vez definidos los valores de cada uno de los elementos puede procederse a simular y, a visualizar los resultados cuando termine la ejecución. Además se incluye la exportación a PDF por si el usuario lo necesita (Ejemplo Figura 5) , que se almacenará en el mismo directorio que el ejecutable.

Resultados de la Simulación

Tipo de Traductor: Python

Método Numérico: Euler

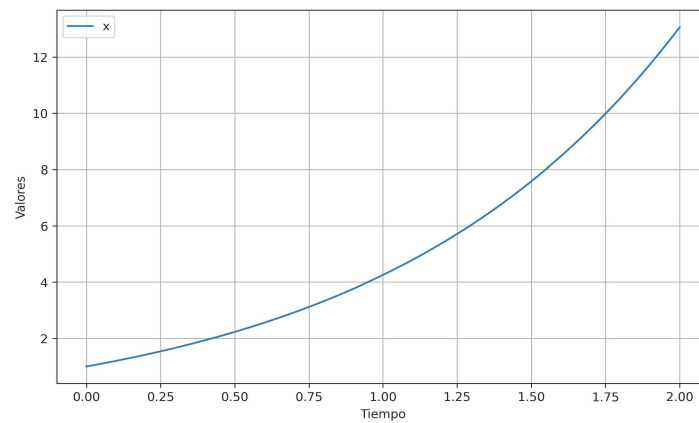
Rango de Tiempo: 0 a 2 con paso de 0.01

Parámetros y Variables:

Nombre	Valor
d	2
x	1

Ecuaciones:

Ecuación
$\frac{dx}{dt} = x + \sin(d)$



t	x
0.0000	1.0000
0.0100	1.0191
0.0200	1.0384
0.0300	1.0579
0.0400	1.0775

Figura 5: Ejemplo de fichero de exportación resultados.

B. Código fuente

Dado que el desarrollo de la herramienta se ha realizado mediante el uso de *GitHub*, se indica a continuación el enlace al repositorio donde podrá comprobarse el historial de *commits* realizados y descargar el código fuente.

<https://github.com/josepise/TFG-Modelos-Continuos>

