

ARQUITECTURA ASEQUIBLE PARA MONITORIZAR Y CONTROLAR TURBINAS OFFSHORE



Autores

Belén Sánchez Centeno
Martín Fernández de Diego

Tutores

Matilde Santos Peñas
Segundo Esteban San Román

Facultad de Informática
Universidad Complutense de Madrid

2021-2022

FINAL DEGREE PROJECT PRESENTED FOR THE DEGREE IN COMPUTER SCIENCE

AFFORDABLE ARCHITECTURE TO MONITOR AND CONTROL OFFSHORE TURBINES



Authors

Belén Sánchez Centeno
Martín Fernández de Diego

Tutors

Matilde Santos Peñas
Segundo Esteban San Román

Faculty of Computer Science
Complutense University of Madrid

2021-2022

A nuestros güelos

Resumen

La energía eólica marina juega un papel clave en la transición ecológica. Las turbinas eólicas flotantes requieren de nuevos y más complejos algoritmos de control y métodos de comunicación. Este proyecto propone un sistema asequible de monitorización y control de diferentes prototipos de turbina, reales o simulados.

En primer lugar, se analizan el modelado y las técnicas de control de las turbinas eólicas, así como la problemática de su implantación en alta mar. También se introduce el concepto de Gemelo Digital junto con sus utilidades en la materia.

La aplicación de un lazo de control básico y la monitorización del estado del sistema en tiempo real sobre un prototipo de baja fidelidad evidencia la necesidad de más de un hilo de ejecución. Se solventa con el uso de un microcontrolador de varios núcleos, colas de comunicación entre hilos y del formato JSON para la encapsulación y subida de datos al servidor de ThingSpeak.

La interfaz de control del sistema se materializa en un Gemelo Digital. Su desarrollo se lleva a cabo en el entorno MATLAB con el patrón de diseño Modelo-Vista-Controlador, lo que facilita su uso para ingenieros de otras ramas y favorece su escalabilidad. El Gemelo Digital permite monitorizar en tiempo real las turbinas de una granja eólica, formada particularmente por un prototipo físico y por una simulación, así como enviarle comandos, ajustando de forma remota el ángulo de las palas, la carga eléctrica o ejecutando paradas de emergencia.

El software integrado con ejecución multihilo se encarga del control de los componentes del prototipo de turbina escalado y es capaz de comunicarse bidireccionalmente con el Gemelo Digital. Resulta de una fusión con algoritmos de control PID modelados por miembros de otras disciplinas del grupo de trabajo.

Por último, se muestran diferentes casos de uso del funcionamiento del prototipo de turbina eólica y del simulador a través de la interfaz del Gemelo Digital.

Palabras clave: *Control, Monitorización, Gemelo Digital, Energía eólica marina, Turbina eólica flotante.*

Abstract

Offshore wind energy plays a key role in the ecological transition. Floating wind turbines require new and more complex control algorithms and communication methods. This project proposes an affordable monitoring and control system for different turbine prototypes, real or simulated.

Firstly, the modelling and control techniques of wind turbines are analysed, as well as the problems of offshore deployment. The concept of Digital Twin is also introduced along with its utilities in the field.

The implementation of a basic control loop and real-time system status monitoring on a low-fidelity prototype shows the need for more than one thread of execution. This is solved with the use of a multi-core microcontroller, inter-thread communication queues and the JSON format for encapsulation and uploading data to ThingSpeak server.

The control interface of the system is materialised in a Digital Twin. Its development is carried out in the MATLAB environment with the Model-View-Controller design pattern, which facilitates its use for engineers from other branches and favours its scalability. The Digital Twin makes possible to monitor the turbines of a wind farm in real time, consisting of a physical prototype and a simulation, and to send commands to it, remotely adjusting the angle of the blades, the electrical load or executing emergency stops.

The embedded software with multi-threaded execution takes care of the control of the scaled turbine prototype components and is able to communicate bidirectionally with the Digital Twin. It results from a fusion with PID control algorithms modelled by members of other disciplines in the working group.

Finally, different use cases of the operation of the wind turbine prototype and the simulator are shown through the Digital Twin interface.

Keywords: *Control, Monitoring, Digital Twin, Offshore wind energy, Floating wind turbine.*

Agradecimientos

Queremos agradecer el trabajo, el apoyo y el entusiasmo de los tutores en este proyecto. Gracias a ellos hemos descubierto el campo técnico de la energía eólica. A Matilde Santos Peñas por todas las oportunidades de formación que nos ha ofrecido. Entre ellas, la publicación de nuestro primer artículo. También por su ayuda en la parte teórica y formal del trabajo.

A Segundo Esteban San Román por toda la ayuda que nos ha brindado durante el proceso de implementación en el prototipo. También por enseñarnos a utilizar nuevos entornos software y desarrollar el simulador funcional.

Además, queremos agradecer la ayuda de Giordy Alexander Andrade Aimara en el proceso de fusión de sus algoritmos de control con la lógica de nuestro software.

Índice general

Resumen	VII
Abstract	IX
Agradecimientos	XI
I Introducción	1
1. Preliminares	3
1.1. Motivación	3
1.2. Objetivos	3
1.3. Asignaturas relacionadas	4
1.4. Plan de trabajo	4
1.4.1. Metodología	4
1.4.2. Contribuciones de los autores	6
1.4.3. Diagrama de Gantt	10
1.5. Estructura de la memoria	11
1. Preliminaries	13
1.1. Motivation	13
1.2. Objectives	13
1.3. Related subjects	14
1.4. Work plan	14
1.4.1. Methodology	14
1.4.2. Author's contributions	16
1.4.3. Gantt diagram	20
1.5. Report structure	21
2. Turbinas eólicas	23
2.1. Modelo	25
2.2. Esquema de control	28

2.2.1. Objetivos de control	29
2.2.2. Controlador PID	30
2.3. Problemática	31
3. Gemelo Digital	33
3.1. Definición	33
3.2. Origen	34
3.3. Tipos	34
3.4. Aplicación y beneficios	35
II Desarrollo del sistema	37
4. Aprendizaje y dominio de las tecnologías	39
4.1. Desarrollo de un lazo de control básico	39
4.1.1. Arduino IDE	40
4.2. Monitorización y almacenamiento en un servidor	42
4.2.1. ThingSpeak	42
4.3. Control y monitorización con multihilo	46
4.3.1. Multihilo	46
4.3.2. JavaScript Object Notation (JSON)	47
4.3.3. Microcontrolador ESP32	48
4.4. Aportaciones	52
5. Interfaz de control del Gemelo Digital	53
5.1. Herramientas	54
5.1.1. MATLAB	54
5.1.2. App Designer	54
5.1.3. Patrones de diseño	55
5.2. Implementación del modelo	57
5.2.1. Accesos a ThingSpeak	60
5.2.2. Accesos al servidor meteorológico	62
5.2.3. Herencia	63
5.3. Implementación de la vista	63
5.3.1. Diseño del sistema interactivo	72
5.4. Implementación del controlador	75
5.4.1. Temporizadores	75
5.5. Simulador	80
5.5.1. Desktop Real-Time	82
5.6. Aportaciones	84

6. Software integrado en el prototipo	85
6.1. Modelo del prototipo de turbina	85
6.1.1. Componentes	86
6.2. Implementación	87
6.2.1. Estructuras de datos	90
6.2.2. Monitor de generador	91
6.2.3. Monitor de IMU	93
6.2.4. Controlador de fase	94
6.2.5. Controlador de pitch	95
6.2.6. Controlador de carga	97
6.2.7. Turbina	100
6.2.8. Comunicaciones	102
6.3. Aportaciones	103
 III Análisis y discusión de resultados	 105
7. Casos de uso	107
7.1. Ciclo de funcionamiento automático completo	107
7.1.1. Análisis del prototipo de turbina	107
7.1.2. Análisis del Gemelo Digital	112
7.2. Funcionamiento manual	113
7.2.1. Análisis del prototipo de turbina y del Gemelo Digital . . .	114
7.2.2. Análisis del simulador y del Gemelo Digital	116
7.3. Aportaciones	117
 8. Conclusiones y trabajos futuros	 119
 8. Conclusions and future work	 121
 Bibliografía	 122

Índice de figuras

1.1. Diagrama de Gantt	10
1.1. Gantt diagram	20
2.1. Prototipo de aerogenerador	25
2.2. Tipos de turbinas eólicas flotantes.	26
2.3. Coeficiente de potencia según el <i>tip speed ratio</i> λ y el <i>pitch</i> β	28
2.4. Tramos de control	29
3.1. Primer concepto del Gemelo Digital de Grieves y Vickers.	33
4.1. Esquema de conexiones	40
4.2. Diagrama de flujo.	41
4.3. Esquema de funcionamiento [1]	43
4.4. Diagrama de flujo.	45
4.5. Esquema de conexiones con el microcontrolador ESP32	48
4.6. Diagrama de flujo.	50
4.7. Funcionamiento	51
5.1. Clases del Gemelo Digital	56
5.2. Clases del modelo del Gemelo Digital	59
5.3. Canales de ThingSpeak del Gemelo Digital	60
5.4. Versión alfa del Gemelo Digital	64
5.5. Versión modular del Gemelo Digital	66
5.6. Versión modo oscuro del Gemelo Digital	67
5.7. Versión granja del Gemelo Digital: vista principal	68
5.8. Versión granja del Gemelo Digital: vista de una turbina	69
5.9. Clases de la vista del Gemelo Digital	71
5.10. Versión final del Gemelo Digital: vista principal	74
5.11. Versión final del Gemelo Digital: vista de una turbina	74
5.12. Simulador	80
5.13. Simulador: turbina eólica	81

5.14. Simulador: control	81
5.15. Simulador: plataforma	81
6.1. Prototipo de aerogenerador <i>offshore</i>	86
6.2. Diagrama de flujo.	89
6.3. Clase del monitor del generador	92
6.4. Clase del monitor de la IMU	93
6.5. Clase del control de fase	94
6.6. Clases de control del <i>pitch</i>	96
6.7. Clases de control del load	99
6.8. Clases de la turbina	101
6.9. Clase de comunicación	102
7.1. Pantalla integrada de carga	108
7.2. Pantalla integrada sin viento	108
7.3. Pantalla integrada en distintas velocidades de viento	109
7.4. Pantalla integrada de parada	110
7.5. Prototipo de la turbina eólica flotante	111
7.6. Interfaz de la granja	112
7.7. Interfaz de la turbina mostrando un ciclo de funcionamiento completo	113
7.8. Interfaz de la turbina mostrando el cambio de carga manual	114
7.9. Interfaz de la turbina mostrando el cambio de <i>pitch</i> manual	115
7.10. Interfaz de la turbina mostrando la parada de emergencia	116
7.11. Interfaz del simulador mostrando el cambio de pitch manual	117

Parte I

Introducción

Capítulo 1

Preliminares

1.1. Motivación

Ante los perjuicios para la salud y el planeta, las dificultades de extracción y transporte, y la coacción de los productores de combustibles fósiles, Europa se enfrenta al desafío de transformar por completo su sistema energético. Hoy es más necesario que nunca que las energías renovables sustituyan al gas, petróleo o carbón y los conviertan en una fuente residual en el mercado eléctrico.

La energía eólica posee mucho potencial. El creciente interés en la expansión de las turbinas a aguas profundas responde a mayores rendimientos, así como a la reducción del impacto medioambiental. Sin embargo, también iría acompañado de nuevos retos: el medio marino aumenta los costes de las instalaciones, fatiga las estructuras y complica las operaciones de control.

Reducir los costes aumentaría las inversiones y aceleraría su implantación, diseñar nuevas estructuras especializadas mejoraría la estabilidad y la resistencia de las turbinas en mar abierto, y desarrollar estrategias de control eficaces incrementaría la seguridad y la eficiencia.

1.2. Objetivos

El trabajo pretende desarrollar un sistema asequible de monitorización y control de turbinas eólicas a través de la figura del Gemelo Digital. Está dirigido especialmente al ámbito de la eólica *offshore* o flotante, donde el acceso a la turbina no siempre es físicamente viable y la monitorización y control se deben realizar desde tierra firme.

La propuesta inicial surge de un primer encuentro entre los desarrolladores y los clientes. Los tutores, los profesores de la Universidad Complutense Matilde Santos Peñas y Segundo Esteban San Román, proponen a los autores un proyecto bífido: una parte de bajo nivel, de toma de datos y control local, y otra parte de alto nivel, de monitorización de una turbina eólica *offshore* en tiempo real.

Por un lado, la parte de bajo nivel se encargaría de controlar, con sensores de medición inercial o de tensión, variables como el ángulo de las palas (*pitch*) o la carga. Este diseño conformaría el lazo de control autónomo de la turbina. Por otro lado, la parte de alto nivel monitorizaría el estado de la turbina en tiempo real y permitiría enviar comandos de acción al lazo de control. Este diseño se materializaría en un modelo virtual o Gemelo Digital donde la visualización de los datos fuera clara e informativa para el usuario.

1.3. Asignaturas relacionadas

Los conocimientos adquiridos en las siguientes asignaturas del grado en Ingeniería Informática han sido especialmente útiles en el desarrollo de este trabajo.

- *Desarrollo de Sistemas Interactivos*
- *Estructuras de Datos y Algoritmos*
- *Ingeniería del Software*
- *Programación Concurrente*
- *Redes*
- *Robótica*
- *Sistemas Operativos*
- *Tecnología de la Programación*

1.4. Plan de trabajo

1.4.1. Metodología

Scrum es el patrón seguido en el desarrollo del proyecto [2]. Es un modelo de proceso ágil, de diseño iterativo e incremental.

Desarrollo ágil

La categoría de desarrollo ágil se basa en la evolución constante de requisitos y soluciones a través de la colaboración entre equipos autoorganizados e interfuncionales y sus clientes o usuarios finales. Valora los individuos sobre los procesos, el software sobre la documentación, la comunicación con el cliente sobre el contrato de negocio y la respuesta al cambio sobre el plan establecido. Además, este desarrollo se caracteriza por:

- La adaptabilidad ante los cambios en los requisitos y las prioridades del cliente.
- La difusión de la frontera entre diseño e implementación.
- La poca predictibilidad en la planificación.

Modelo de desarrollo Scrum

Uno de los principales métodos ágiles es el Scrum. Resulta especialmente eficaz para desarrollar, entregar y mantener productos complejos con plazos de entrega ajustados y requisitos cambiantes.

El principio clave del Scrum es la asunción de que el cliente cambiará los requisitos y de la existencia de desafíos inesperados que no pueden ser previstos con métodos de predicción. Aborda el proceso de desarrollo desde un enfoque empírico: acepta la dinámica cambiante y la utiliza en su beneficio.

Su funcionamiento se basa en tres términos:

Retraso Lista de objetivos susceptible de cambios ordenada por prioridad donde se recogen las peticiones del cliente.

Sprint Unidades de trabajo necesarias para completar un objetivo del Retraso.

Reunión Breves encuentros periódicos de coordinación.

¿Por qué Scrum?

El método organizativo del proyecto debía adecuarse fielmente a sus condiciones. A continuación, se muestra una selección de los aspectos que se han considerado fundamentales a la hora de elegir Scrum como modelo de desarrollo:

- Un equipo de desarrollo pequeño —de dos miembros— pero motivado y con conocimientos de la materia. La comunicación fluida y el reparto justo de los puntos del proyecto fueron clave para una correcta autoorganización.

- Gestión de entornos de alta incertidumbre y riesgo a variaciones en los requisitos iniciales. Las variaciones podrían producirse por restricciones técnicas o, simplemente, por decisión del cliente ante nuevas necesidades.
- Presentación constante de productos y servicios funcionales.

Estos, junto con otros motivos como los plazos ajustados, la buena colaboración entre las partes y el carácter innovador del proyecto, han servido para concluir en dicho modelo de desarrollo.

1.4.2. Contribuciones de los autores

La coordinación entre los autores ha sido máxima. Los desarrollos se han realizado de manera conjunta hasta alcanzar el objetivo básico de cada iteración. A continuación, los autores exponen concretamente su trabajo.

Belén Sánchez Centeno

En la primera reunión con los tutores del trabajo se decidió que mi compañero se iba a encargar de la parte a más bajo nivel y yo de la parte a más alto nivel.

Entre junio y septiembre me dediqué a recabar información sobre las turbinas eólicas *offshore*, a entender lo que era un Gemelo Digital para su posible futura implementación, y a instruirme en las tecnologías que iba a tener que utilizar. Por ejemplo, obtuve el certificado gratuito del curso MATLAB Onramp.

La primera fase del proyecto consistió en trabajar junto a mi compañero en los prototipos hardware iniciales. Para ello tuve que aprender a usar Arduino. Por querer abarcar la parte relacionada con la nube, me encargué de crear y configurar un canal en ThingSpeak dedicado a almacenar las lecturas de la IMU conectada al microcontrolador ESP8266.

Cuando dimos con la limitación en la frecuencia de subidas de datos a ThingSpeak, busqué la manera de atajarlo. Propuse el uso del formato JSON para subir paquetes de datos. Este método es más complejo y requiere de una librería en Arduino para realizar el encapsulamiento de la información.

Al alcanzar el hito de mover el *pitch* y subir los datos a ThingSpeak simultáneamente y en tiempo real, gracias a la implementación con dos hilos sugerida por mi compañero, la tutora Matilde Santos Peñas nos propuso presentar nuestro trabajo hasta la fecha al *III Workshop on Wind and Marine Energy*.

Habíamos estado acudiendo a conferencias relacionadas con la energía eólica y con sus diferentes metodologías de control, lo que nos fue muy útil a la hora de redactar la primera parte del artículo. En el desarrollo del mismo plasmamos lo

que habíamos conseguido hasta el momento, e indicamos en los trabajos futuros los siguientes pasos que íbamos a seguir, como el desarrollo de un Gemelo Digital. Para incluir algunas gráficas ilustrativas, implementé un *script* en MATLAB para descargar los datos de ThingSpeak y mostrarlos.

Tras recibir el visto bueno por parte de la Universidad del País Vasco, dedicamos varios días a la creación de una presentación sobre el artículo, que finalmente se mostró a los demás participantes y asistentes el día 16 de diciembre de 2021.

La segunda fase del proyecto comenzó después de que termináramos los exámenes de enero. Investigué diferentes herramientas de MATLAB para el desarrollo de interfaces gráficas hasta dar con la más adecuada: AppDesigner. En un primer acercamiento a la implementación del Gemelo Digital, diseñé una vista en MATLAB que permitía visualizar una gráfica con los datos actualizados de la aceleración de la IMU, almacenados en ThingSpeak. Les presentamos el resultado a los tutores, que nos indicaron la hoja de ruta a seguir a continuación con el Gemelo Digital, basada en las necesidades de control de una turbina eólica.

Decidimos que la mejor manera de enfocararlo era mediante el patrón de diseño Modelo-Vista-Controlador. Debido a que no es algo que se suele implementar en MATLAB, tuve que revisar profundamente su documentación. Entonces pude incorporar el código de la primera vista creada en AppDesigner al conjunto de clases requeridas por el patrón, separando la interfaz de usuario de los accesos a ThingSpeak. A partir de este punto, diseñé con mi compañero versiones del Gemelo Digital más complejas.

Al trabajar paralelamente a la implementación en el prototipo físico del código multihilo con la comunicación a ThingSpeak, hubo que encontrar otra manera de generar datos con los que suplir al Gemelo Digital. El tutor Segundo Esteban San Román nos proporcionó varias simulaciones de turbina eólica para el entorno de MATLAB Simulink. Sin embargo, nos encontramos con problemas a la hora de extraer los valores de sus variables en tiempo de simulación. Finalmente, con la herramienta *Desktop Real-Time*, conseguimos ejecutar las simulaciones en el kernel de nuestro sistema operativo en tiempo real. Desarrollé varios *Live Scripts* para iniciar y terminar la simulación desde MATLAB, así como para extraer o alterar los valores de sus variables. También se encargan de acceder a ThingSpeak. Todo ello con periodos de tiempo fijos, con temporizadores.

La implementación del Gemelo Digital se llevó a cabo de manera incremental. A medida que aumentaban las funcionalidades y la complejidad de las vistas, tuve que desarrollar más clases en MATLAB y trabajar con más eventos y temporizadores. La versión que me supuso un mayor reto fue en la que se pasó de tener una a varias vistas, de la granja y de cada turbina. Tuve que llevar a cabo algunos replanteamientos como la fragmentación de la parte del controlador en diferentes

instancias, asociadas a cada nueva vista. Cuando el prototipo físico pasó a ser funcional y a tener un flujo de datos con ThingSpeak, creé clases en la parte del modelo que se ajustasen más a la simulación o a la turbina real. Para poder almacenar los nuevos históricos de información abrí canales en ThingSpeak y reorganicé sus campos, siempre atendiendo a las limitaciones de la licencia gratuita.

La última versión aportada del Gemelo Digital permite monitorizar en tiempo real una granja eólica, conformada por el prototipo físico y por una simulación, así como enviarles comandos, ajustando por ejemplo el *pitch* y la carga eléctrica de forma remota. Es escalable a otros prototipos y a mayores proyectos.

La redacción de esta memoria se realizó de manera conjunta. En especial la primera y la tercera parte, correspondientes a la introducción y al análisis y discusión de resultados. En la Parte II, de desarrollo del sistema, mi mayor aportación recae sobre el Capítulo 5, dedicado a la interfaz de control.

Martín Fernández de Diego

Personalmente, en el ámbito técnico, siempre me ha interesado el desarrollo en microarquitecturas enfocado a pequeños proyectos funcionales. Antes de comenzar el trabajo, sabía que centraría gran parte de mi aportación en ese contexto pero, durante el desarrollo, se abrieron nuevos campos que me resultaban igual de atractivos. Entre ellos, el diseño gráfico.

La primera reunión con mi compañera y tutores del Trabajo de Fin de Grado fue previa al verano de 2021. Allí se esbozó una idea general del propósito del proyecto y se sugirieron herramientas hardware y software para llevarlo a cabo. Antes del comienzo del curso 2022, nos instruimos en el funcionamiento de MATLAB obteniendo el certificado gratuito del curso MATLAB Onramp que ofrece la propia marca. Adquirimos los microcontroladores ESP8266, unas protoboard y unos cables y codificamos pequeñas pruebas de contacto.

Durante los primeros tres meses de proyecto, acudí a conferencias online para conocer las técnicas de funcionamiento y el estado del arte de los controles inteligentes aplicados a turbinas eólicas, especialmente a turbinas *offshore*.

El método de trabajo decidido nos obligaba a presentar prototipos funcionales frecuentemente. Nuestra manera de actuar consistiría en desarrollar las estructuras principales del producto y ejecutar iteraciones o *sprints* para refinarlo hasta obtener el resultado deseado.

El primer gran objetivo fue el desarrollo del prototipo de baja fidelidad. Esta primera fase se realizó de manera conjunta, puesto que su propósito era también aprender a utilizar los entornos de desarrollo y conocer los problemas de funciona-

miento de primera mano. Tras investigar el comportamiento de los componentes hardware y del entorno de desarrollo Arduino, conseguimos implementar un primer lazo de control capaz de mover un servomotor en función de la inclinación de un acelerómetro. Cuando nos enfrentamos al problema de fluidez, investigué acerca de la posibilidad de incorporar un microcontrolador de dos núcleos en un proyecto tipo IoT¹ de bajo coste. El resultado conceptual de control y comunicación multihilo serviría como esqueleto para el desarrollo final.

Aunque este primer hito fue modesto, nos animó y nos ayudó a impulsar el proyecto. En noviembre de 2021, tras mostrar los resultados a los tutores, Matilde Santos Peñas nos invita a participar en la *III Workshop on Wind and Marine Energy*. A partir de este momento, el objetivo fue documentar el trabajo realizado en un artículo de investigación para presentarlo a la conferencia. También se expuso una presentación donde se explicaba de manera más ilustrativa el mecanismo. El resultado, *Arquitectura Hardware Asequible para Implementar Controles de Pitch en una Turbina Offshore*, puede verse publicado en la revista científica *Innovación y Docencia en la Ingeniería de Control para Energía Marina*.

El segundo gran objetivo, a la espera de relevar el prototipo sobre el que implementaríamos el software del proyecto, fue el desarrollo de un Gemelo Digital. Contribuí al proceso de modelado software basado en programación orientada a objetos (POO). Posteriormente y a través de varias etapas, realicé los diseños de la interfaz gráfica. Sin embargo, el Gemelo Digital final es producto de la coautora.

Cuando recibimos el prototipo de turbina y el programa de control que ya tenía implementado en febrero de 2022, establecemos el tercer gran objetivo: el desarrollo de un nuevo software integrado capaz de comunicarse bidireccionalmente con el Gemelo Digital. Para maximizar la consistencia y la compatibilidad entre los desarrollos también lo basamos en programación orientada a objetos. Una vez enfocado, me dediqué personalmente a la reestructuración y fusión del software recibido con el sistema de comunicación multihilo que ya habíamos esbozado en el prototipo de baja fidelidad. Fue necesaria una recalibración del algoritmo de control que el prototipo implementaba.

La documentación y la redacción de la memoria también se realizó de manera coordinada para asegurar la consistencia de formato y contenido. Ambos contribuímos a la redacción, las delimitaciones quedan difusas en este aspecto. Destaco mi preocupación por el formato y la edición de este texto a través del diseño de muchas de las figuras del documento y la creación y modificación de librerías L^AT_EX para mejorar la maquetación del texto, concretamente para el diseño de los diagramas de flujo, de clase y de Gantt.

¹*Internet of Things* describe la red de objetos con sensores incorporados capaces de realizar cálculos e intercambiar datos con otros sistemas a través de la red

1.4.3. Diagrama de Gantt

Véase la *Figura 1.1*.

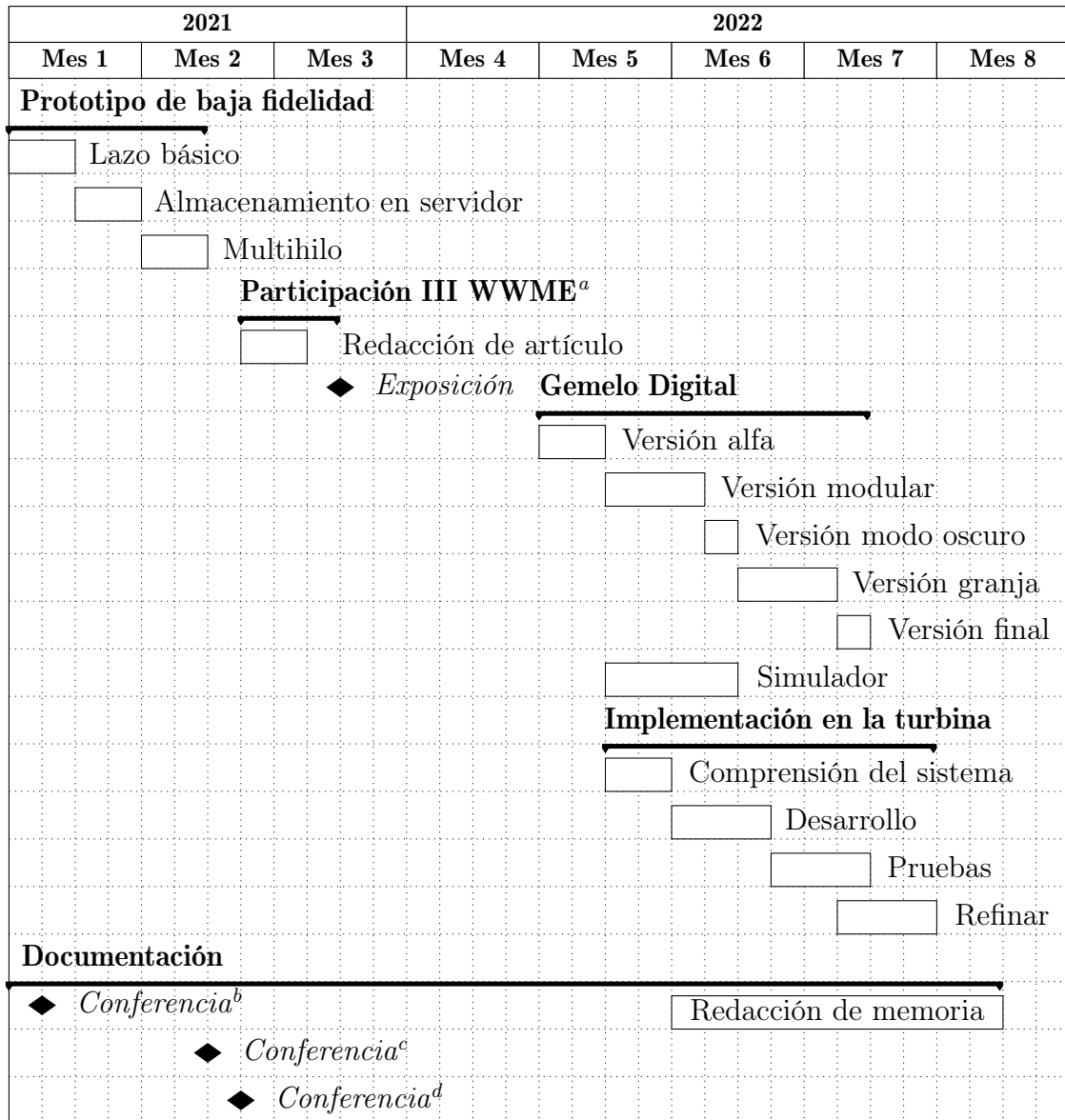


Figura 1.1: Diagrama de Gantt

^aIII Workshop on Wind and Marine Energy

^bControl inteligente de aerogeneradores [3]

^cChallenges and Opportunities for AI and Data analytics in Offshore wind [4]

^dRedes neuronales y reinforcement learning. Aplicación en energía eólica [5]

1.5. Estructura de la memoria

El trabajo queda estructurado como sigue.

La parte I introduce el trabajo y expone los conceptos claves de este proyecto: las turbinas eólicas y el Gemelo Digital.

La parte II recoge el desarrollo completo del sistema. Un primer capítulo, el aprendizaje de los autores a través del desarrollo de un prototipo de control y monitorización de baja fidelidad. Un segundo capítulo, la implementación completa y detallada del Gemelo Digital en MATLAB. Y un tercer capítulo, la implementación del software de control sobre un prototipo de turbina. Todos ellos haciendo hincapié en los conceptos clave.

La parte III muestra el resultado del proyecto mediante el análisis de diferentes casos de uso. Además, incluye una sección con las conclusiones y cuestiones a atajar en futuros trabajos.

Por último, la memoria se cierra con la bibliografía.

Chapter 1

Preliminaries

1.1. Motivation

Faced with the damage to health and the planet, the difficulties of extraction and transport, and the coercion of fossil fuel producers, Europe stands the challenge of transform its energy system. It is more necessary than ever for renewables to replace gas, oil and coal and make them a residual source in the electricity market.

Wind energy has a lot of potential. The growing interest in the expansion of deepwater turbines is driven by higher efficiencies as well as reduced environmental impact. However, it would also be accompanied by new challenges: the marine environment increases installation costs, fatigues structures and complicates control operations.

Reducing costs would increase investment and accelerate deployment, designing new specialised structures would improve the stability and resilience of offshore turbines, and developing effective control strategies would increase safety and efficiency.

1.2. Objectives

The work aims to develop an affordable system for monitoring and controlling wind turbines through the figure of the Digital Twin. It is especially aimed at the offshore or floating environment where access to the turbine is not always physically feasible and monitoring and controlling must be carried out from land.

The initial proposal comes up from a first meeting between developers and clients. Tutors, the professors of the Complutense University Matilde Santos Peñas and Segundo Esteban San Román, propose to the authors a bifid project: a low-level part, for data collection and local control, and a high-level part, for monitoring an offshore wind turbine in real time.

On the one hand, the low-level part would be responsible for controlling, with inertial or voltage measurement sensors, variables such as pitch or load. This design would form the autonomous turbine control loop. On the other hand, the high-level part would monitor the status of the turbine in real time and allow commands to be sent to the control loop. This design would materialise on a Digital Twin where the visualisation of data is clear and enough informative for the user.

1.3. Related subjects

The knowledge acquired in the following subjects of the degree in Computer Engineering has been especially useful in the development of this work.

- *Interactive Systems Development*
- *Data Structures and Algorithms*
- *Software Engineering*
- *Concurrent Programming*
- *Networks*
- *Robotics*
- *Operative Systems*
- *Programming Technology*

1.4. Work plan

1.4.1. Methodology

Scrum is the pattern followed in project development. It is an agile, iterative and incremental design process model.

Agile development

The agile development category is based on the constant evolution of requirements and solutions through collaboration between self-organised, cross-functional teams and their clients or end-users. It values individuals over processes, software over documentation, communication with the client over the business contract and capacity of change over the established plan. In addition, this work is characterised by:

- The adaptability to changing requirements and priorities.
- The diffused boundary between design and implementation.
- The lack of predictability in planning.

Scrum development model

One of the main agile methods is Scrum. It is particularly effective for developing, delivering and maintaining complex products with tight deadlines and changing requirements.

The key principle of Scrum is the assumption that the client will change requirements and that there are unexpected challenges that cannot be expected with predictive methods. It approaches the development process from an empirical way: it accepts the changing dynamics and uses them to its advantage.

It work is based on three terms:

Backlog A prioritised list of targets for change, including the client's requests.

Sprint Units of work required to complete a Backlog target.

Meeting Short daily coordination meetings.

Why Scrum?

Organisational method had to be closely tailored to its conditions. It follows a selection of the aspects that have been considered fundamental when choosing Scrum as a development model:

- A small —two-member— but motivated and knowledgeable development team. Smooth communication and fair distribution of project points were key to successful self-organisation.

- Management of environments of high uncertainty and risk to variations in initial requirements. Variations could be caused by technical constraints or simply by the client's decision in response to new requirements.
- Constant presentation of functional products and services.

These, beside with other reasons such as the tight deadlines, the good collaboration between the parties and the innovative nature of the project, have served to conclude on such a development model.

1.4.2. Author's contributions

Coordination between authors has been at its best. The development has been carried out mutually until the basic objective of each iteration has been reached. In the following, the authors present their work in concrete terms.

Belén Sánchez Centeno

In the first meeting with the tutors it was decided that my colleague would be in charge of the low-level part and I would be in charge of the high-level part.

Between June and September I spent time gathering information on offshore wind turbines, understanding what a Digital Twin was for possible future implementation and educating myself on the technologies I would need to use. For example, I obtained the free MATLAB Onramp certificate.

The first phase of the project consisted of working with my partner on the initial hardware prototypes. To do this I had to learn how to use Arduino. To cover the cloud-related part, I took care of creating and configuring a channel in ThingSpeak dedicated to storing the readings from the IMU connected to the ESP8266 microcontroller.

When we found the limitation on the frequency of uploads to ThingSpeak, I looked for a way to address it. I proposed the use of the JSON format for uploading data packets. This method is more complex and requires an Arduino library to encapsulate the information, which had to be found and learned to use.

When reaching the milestone of moving the pitch and uploading the data to ThingSpeak simultaneously and in real time, thanks to the two-threaded implementation suggested by my colleague, the tutor Matilde Santos Peñas proposed us to present our work to *III Workshop on Wind and Marine Energy*.

We had been attending conferences related to wind energy and its different control methodologies, which was very useful when writing the first part of the article. In the development of the project we captured what we had achieved so

far, and indicated in future work section the next steps we were going to take, such as the development of a Digital Twin. To include some illustrative graphs, I implemented a MATLAB script to download the data from ThingSpeak and display it.

After receiving the go-ahead from the University of the Basque Country, we spent several days creating a presentation on the content of the article, which was finally shown to the other participants and assistants on 16 December.

The second phase of the project started after we finished our exams in January. I investigated different MATLAB tools for the development of graphical interfaces until I found the most suitable one: AppDesigner. In a first approach to the implementation of the Digital Twin, I designed a view in MATLAB to display a graph of the updated IMU acceleration data stored in ThingSpeak. We presented the result to the tutors, who gave us the pathway to follow with the Digital Twin, based on the control needs of a wind turbine.

We decided that the best way to approach it was through the Model-View-Controller design pattern. Since this is not something that is usually implemented in MATLAB, I had to check the MATLAB documentation. I was then able to incorporate the code of the first view created in AppDesigner into the set of classes required by the pattern, separating the user interface from the accesses to ThingSpeak. From this point on, I designed with my partner more complex versions of the Digital Twin.

Working in parallel to the physical prototype implementation of the multi-threaded code with the communication to ThingSpeak, another way had to be found to generate data to supplement the Digital Twin. Tutor Segundo Esteban San Román provided us with several wind turbine simulations for the MATLAB Simulink environment. However, we encounter problems in extracting the values of its variables at simulation time. Finally, with the tool *Desktop Real-Time*, we can run the simulations on the kernel of our operating system in real time. I developed several *Live Scripts* to start and end the simulation from MATLAB, as well as to extract or alter the values of its variables. They are also responsible for accessing ThingSpeak. All with fixed time periods, with timers.

The implementation of the Digital Twin was carried out incrementally. As the functionality and complexity of the views increased, I had to develop more classes in MATLAB and work with more events and timers. The most challenging version for me was the one that went from having one to several views, of the farm and of each turbine. I had to carry out some restatements, such as fragmenting the controller part into different instances, associated with new views. When the physical prototype became functional and had a data flow with ThingSpeak, I created classes in the model part that match closely with the simulation or the

real turbine. In order to store the new historical information, I opened channels in ThingSpeak and reorganised their fields, always paying attention to the limitations of the free licence.

The latest version of the Digital Twin allows real-time monitoring of a wind farm as well as sending commands to them, for example, adjusting the pitch and the electrical load remotely. It is scalable to other prototypes and larger projects.

The writing of this report was carried out together. In particular, the first and third parts, corresponding to the introduction and the analysis and discussion of results. In Part II, the development of the system, my main contribution is in Chapter 5, dedicated to the control interface.

Martín Fernández de Diego

Personally, in the technical field, I have always been interested in microarchitecture development focused on small functional projects. Before I started the work, I knew that I would focus much of my contribution on this context but in the course of development, new fields opened up that were just as attractive to me. Among them, graphic design.

The first meeting with my colleague and my Final Degree Project tutors was before the summer of 2021. There was outlined a general idea of the purpose of the project and hardware and software tools to carry it out. Before the start of the 2022 course, we learnt how MATLAB works by obtaining the free MATLAB Onramp certificate offered by MathWorks. We acquired the ESP8266 microcontrollers, some breadboards and cables and coded small tests.

During the first three months of the project, I attended online conferences to learn about operating techniques and the state of the art of intelligent controls applied to wind turbines, especially offshore turbines.

The working method we decided on required us to present functional prototypes frequently. Our approach would be to develop the main structures of the product and run iterations or sprints to refine it until the desired result is achieved.

The first major objective was the development of the low-fidelity prototype. This first phase was carried out together as its purpose was also to learn how to use the development environments and to learn about the operational problems first hand. After investigating the behaviour of the hardware components and the Arduino development environment, we managed to implement a first control loop capable of moving a servomotor depending on the inclination of an accelerometer. When we faced with the problem of fluidity, I investigated the possibility of incor-

porating a dual-core microcontroller in an IoT¹ low cost project. The conceptual result of multi-threaded control and communication would serve as a sketch for the final development.

Although this first milestone was modest, it encouraged us and helped us to push the project forward. In November 2021, after showing the results to the tutors, Matilde Santos Peñas invites us to participate in the *III Workshop on Wind and Marine Energy*. From this point on, the aim was to document the work done in a research paper for presentation at the conference. A presentation was also given when explaining the mechanism in a more illustrative way. The result, *Arquitectura Hardware Asequible para Implementar Controles de Pitch en una Turbina Offshore*, can be found published in the scientific journal *Innovación y Docencia en la Ingeniería de Control para Energía Marina*.

The second major objective, pending the release of the prototype on which we would implement the project's software, was the development of a Digital Twin. At the beginning, I contributed to the software modelling process based on object-oriented programming (OOP). Furthermore, through several stages, I made the designs of the graphic interface. However, the final Digital Twin is a product of the co-author.

When we received the prototype turbine and the control programme it already had in February 2022, we set the third major target: the development of new embedded software capable of bi-directional communication with the Digital Twin. To maximise consistency and compatibility between developments we also base it on object-oriented programming. Once focused, I personally worked on restructuring and merging the incoming software with the multi-threaded communication system we had already sketched out in the low-fidelity prototype. A recalibration of the control algorithm implemented by the prototype was necessary. The result was greater modularity and symmetry between classes that are at the same level, for example between the class that models the pitch or the load. A modular software for the design and testing of control algorithms.

The documentation and drafting of the report was also coordinated to ensure consistency of format and content. We both contributed to the drafting, the boundaries are blurred in this respect. I emphasise my concern for the formatting and editing of this text through the design of most of the figures in the document and the creation and modification of libraries. $\text{\LaTeX}$ to improve the layout of the text, specifically for the design of flowcharts, class diagrams and Gantt charts.

¹*Internet of Things* describes the network of objects with embedded sensors capable of performing computations and exchanging data with other systems through the network.

1.4.3. Gantt diagram

See *Figure 1.1*.

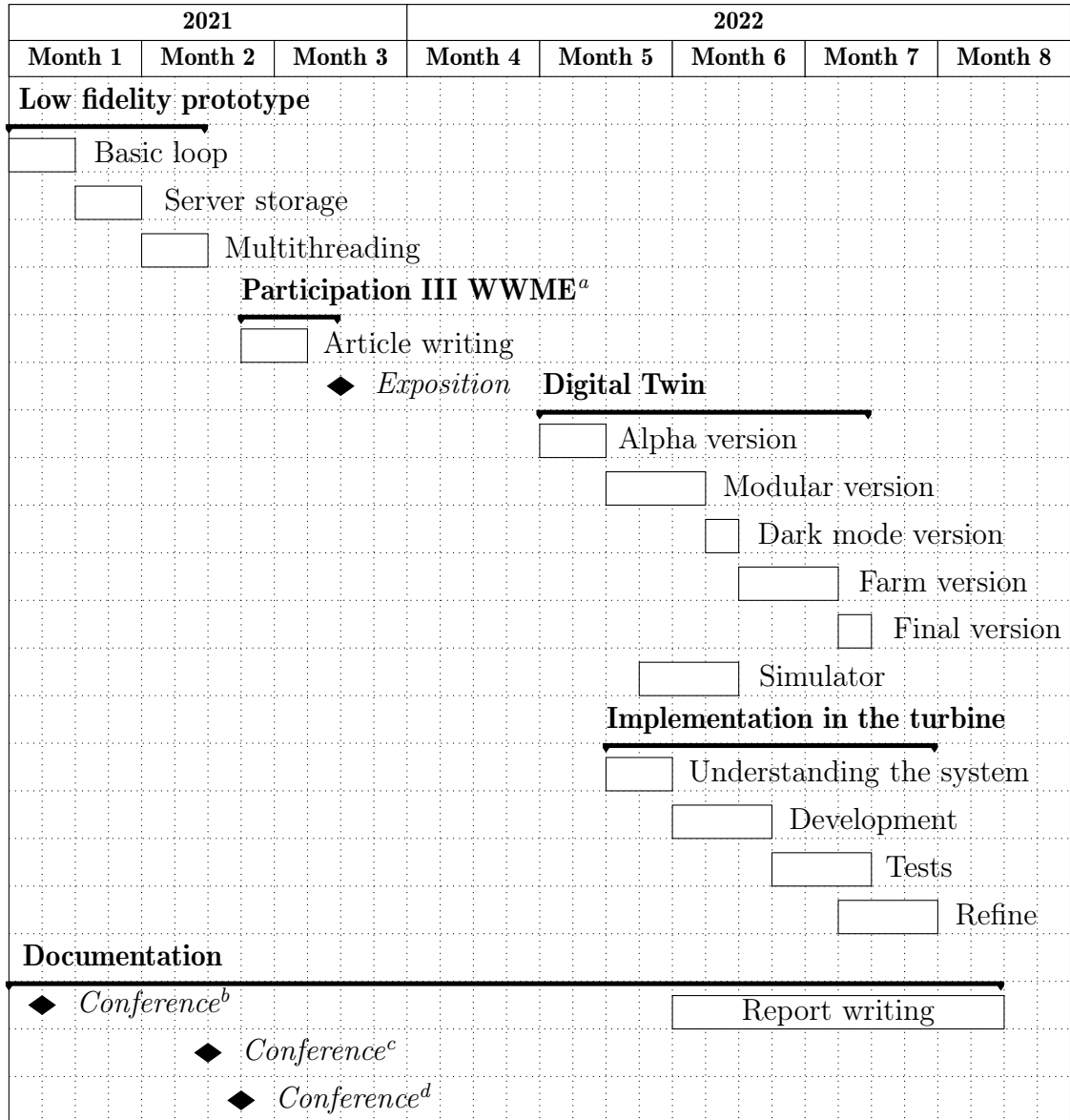


Figure 1.1: Gantt diagram

^aIII Workshop on Wind and Marine Energy

^bControl inteligente de aerogeneradores [3]

^cChallenges and Opportunities for AI and Data analytics in Offshore wind [4]

^dRedes neuronales y reinforcement learning. Aplicación en energía eólica [5]

1.5. Report structure

The work is structured as follows.

Part I introduces the work and sets out the key concepts of this project: wind turbines and the Digital Twin.

Part II covers the complete development of the system. A first chapter, the author's learning through the development of a low-fidelity control and monitoring prototype. A second chapter, the complete and detailed implementation of the Digital Twin in MATLAB. And a third chapter, the implementation of the control software on a prototype turbine. All of them emphasising key concepts.

Part III shows the outcome of the project by analysing different use cases. It also includes a section on issues to be addressed in future work.

Finally, the report ends with the bibliography.

Capítulo 2

Turbinas eólicas

El esfuerzo de instituciones públicas y empresas en materia de investigación e innovación ha dado lugar a un fuerte empuje en la evolución de los métodos de producción de energía limpia. Las energías renovables superaron a los combustibles fósiles como la principal fuente de energía de la UE en el año 2020 [6]. Los protagonistas de este hito han sido la energía eólica, la hidráulica y la solar fotovoltaica. En ese mismo año, la eólica y la hidráulica produjeron un 36 % y un 33 % del total de la energía renovable de la Unión, respectivamente, y la solar, aunque produjo un 14 % del total, es la que más ha crecido desde 2008 [7]. La renovable representó un 22 % de la energía total generada.

Además, el sector eólico impulsa el crecimiento económico y crea puestos de trabajo sostenibles a largo plazo. En 2020 dio trabajo a entre 240.000 y 300.000 personas de la eurozona, de los cuales unas 62.000 correspondían a la industria eólica marina [8].

La energía eólica es un tipo de energía cinética que se obtiene del viento y se puede transformar en electricidad a través de un generador eléctrico. Durante miles de años ha sido utilizada para propulsar los primeros veleros o para hacer girar las palas de pequeños molinos de viento utilizados, generalmente, para moler grano.

Una turbina eólica o aerogenerador convierte el viento en electricidad utilizando la forma aerodinámica de las palas del rotor. Cuando el viento sopla, la presión del aire en un lado de la pala disminuye. La diferencia de presión a través de los dos lados de la pala hace que el rotor gire. El eje del rotor se conecta al generador, directamente o a través de una serie de engranajes que aumentan el par angular, para producir energía eléctrica. Las turbinas se pueden agrupar en parques eólicos o granjas, y pueden cubrir varios kilómetros cuadrados de tierra o mar para aprovechar al máximo la energía de las zonas con más viento.

Dependiendo de la ubicación de la instalación, existen dos categorías: las turbinas eólicas *onshore* y *offshore*. El término *onshore* hace referencia a las turbinas localizadas en tierra o terrestres, generalmente posicionadas lejos de lugares poblados y fuera de espacios protegidos. El término *offshore*, hace referencia a las posicionadas en el agua, generalmente en el mar. A su vez, se pueden distinguir las costeras, ancladas al lecho marino, y las flotantes, que se instalan en alta mar y tienen gran capacidad de producción.

La primera granja *offshore* se construyó en 1991 [9], en la costa de Dinamarca, con once turbinas de 450kW cada una que suplían de electricidad a 2200 hogares. Su crecimiento posterior fue muy lento, hasta la última década. En todo el mundo la capacidad de la eólica marina pasó de 5GW en 2012 a 55 en 2022 [10], de los cuales 25 están sólo en Europa.

Se trata de una tecnología reciente con muchos retos aún por delante a los que hacer frente, pero tiene potencial para convertirse en el futuro protagonista de la transición ecológica que estamos viviendo. De hecho, la Agencia Internacional de la Energía [11] prevé que la capacidad de producción de energía eólica marina alcance en Europa los 130GW en 2040 —unos 180GW si se logran los objetivos en cuanto a neutralidad de carbono para ese año—.

Para aguas de más de 60 metros de profundidad las turbinas ancladas al lecho marino dejan de ser viables. En costas con pendientes pronunciadas, donde esas profundidades se alcanzan en poca distancia, se hace inviable el aprovechamiento de la energía eólica marina mediante esa vía. Las turbinas flotantes, en cambio, no cuentan con esa limitación, desbloqueando el 80 % estimado de los recursos eólicos marinos globales. Otras de sus ventajas son:

- Sencillez de fabricación. Se pueden montar en tierra y remolcar al mar.
- Turbinas más grandes.
- Mayor producción energética. En alta mar, el viento es más constante y alcanza mayores velocidades debido a la falta de obstáculos.
- Cero impacto visual y sonoro sobre las costas.

Las turbinas eólicas flotantes permitirán a países como Noruega, Portugal o España, donde el potencial de las ancladas al lecho marino es muy limitado, entrar en la industria *offshore*. El éxito en su desarrollo supondría satisfacer la demanda de electricidad de varios mercados clave, varias veces. Incluidos Europa, Estados Unidos y Japón, donde hoy en día se estima que su potencial llega a los 4.000GW, 2.450GW y 500GW, respectivamente [12].

A continuación se describe el modelo de una turbina eólica y su control.

2.1. Modelo

Una turbina eólica de eje horizontal se compone de las partes de la Figura 2.1.

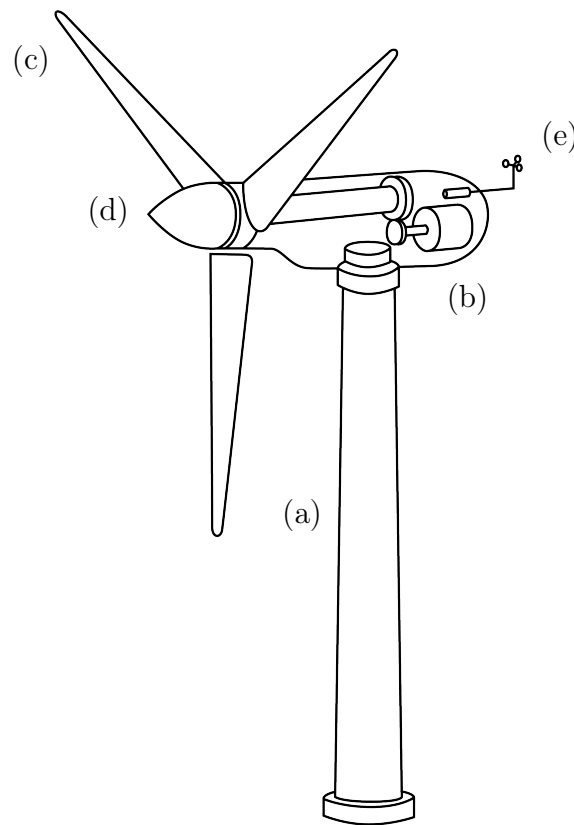


Figura 2.1: Prototipo de aerogenerador

Torre (a) Soporta la góndola y el rotor, formado por las palas y el buje.

Góndola (b) Protege a los componentes del mecanismo interno de la turbina, como al generador eléctrico.

Palas (c) Oponen resistencia al viento.

Buje (d) Sujeta las palas y transmite la potencia que capturan.

Anenómetro y veleta (e) Miden la velocidad y la dirección del viento.

En las turbinas *onshore* la base es un bloque de hormigón que está debajo del suelo y es lo suficientemente extenso como para soportar el peso de la turbina y las fuerzas a las que está sometida.

En las turbinas *offshore* la base se encuentra en el agua. La mayoría de las existentes en la actualidad tienen anclajes fijos al lecho marino, por lo que no son ni técnica ni económicamente recomendables para aguas profundas [13].

Las cimentaciones por gravedad, en las que una caja de hormigón o acero de unas 1.000 toneladas mantiene la turbina en posición vertical, se pueden instalar hasta los 10 metros de profundidad. En torno a los 20 metros se suelen usar monopilotes, compuestos por un único pilar de acero que penetra en el terreno. Hasta los 50 metros de profundidad son rentables los trípodos y los *jackets*, que son estructuras de acero de varios anclajes inspiradas en las plataformas de los pozos petrolíferos.

En el caso de las turbinas *offshore* flotantes no existen límites de profundidad pero, por tratarse de una tecnología en desarrollo, todavía no se han establecido tipologías de base definitivas. De hecho, se esperan nuevas propuestas de mejora ya que la estructura flotante de la base es su componente más caro, abarcando el 36 % del coste total [14].

En la actualidad, destacan los cuatro planteamientos de la *Figura 2.2*: las estructuras que se estabilizan por flotación o *barge*, las semisumergibles, las de boya y la *Tensión Leg Platform* (TLP). Para evitar desplazamientos, todas estas bases están sujetas por cables al fondo marino.

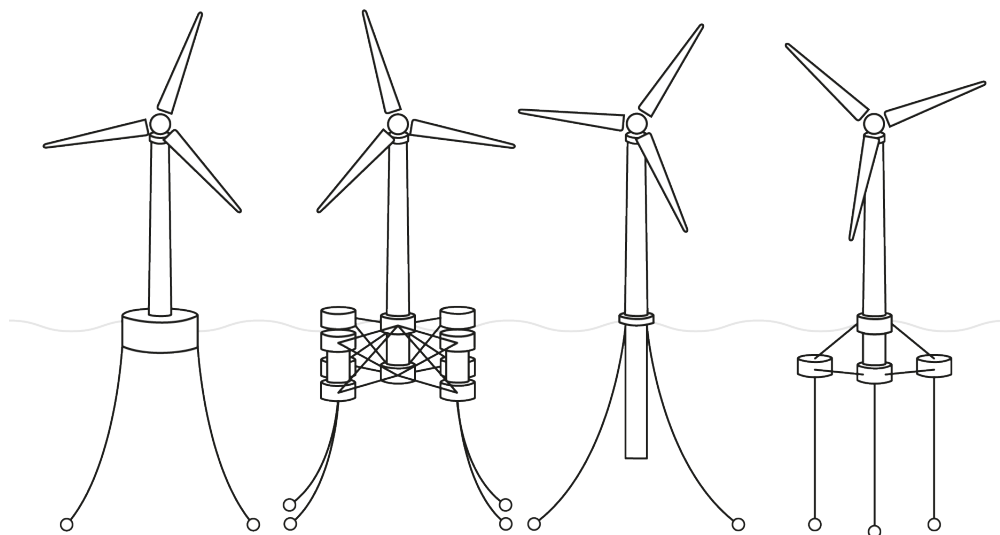


Figura 2.2: Tipos de turbinas eólicas flotantes.

La potencia disponible (W) del viento se expresa mediante la siguiente ecuación [15]

$$P_{viento} = \frac{1}{2} \rho A v^3 \quad (2.1)$$

donde

ρ es la densidad del aire (kg/m^3). Su incremento favorece la extracción de potencia. El aire frío es más denso que el cálido y el aire cercano a la superficie del mar es más denso que el aire en las montañas.

A es el área del rotor o superficie barrida por las palas (m^2). Suponiendo que estamos trabajando con turbinas de eje horizontal, corresponde a una circunferencia ($A = \pi * R^2$, siendo R el radio del rotor o la longitud de las palas).

v es la velocidad del viento media incidente en el rotor (m/s).

La potencia disponible del viento P_{viento} no puede ser capturada íntegramente por la turbina, sino que queda acotada por un máximo teórico de eficiencia aerodinámica, del 59,3 %, llamado coeficiente de Bertz. Este se explica por el arrastre generado por las turbulencias en los extremos de las palas. Teniendo en cuenta el resto de limitaciones de cada turbina en particular, la potencia real extraíble disminuye a en torno a un 40 % de la disponible del viento.

Entonces, la potencia extraíble (W) por la turbina se expresa mediante la ecuación [15]

$$P = \frac{1}{2} \rho A v^3 C_p(\beta, \lambda) \quad (2.2)$$

donde se denota como C_p a la eficiencia aerodinámica o coeficiente de potencia, que depende del ángulo de *pitch* de las palas β y del *tip speed ratio* λ . El *tip speed ratio* es la velocidad tangencial (w) en el extremo de las palas entre la velocidad del viento, es decir

$$\lambda = \frac{wR}{v} \quad (2.3)$$

El factor C_p es fundamental en el diseño del control de la turbina, ya que de él depende la obtención de la máxima potencia de la generada por el viento.

Al incrementar el ángulo de *pitch*, disminuye el coeficiente de potencia, ya que se produce una menor captación de la potencia disponible del viento. A mayor *tip speed ratio*, mayor es el coeficiente de potencia, hasta un punto en el que se revierte debido a las cargas de fatiga que surgen por turbulencias del viento y excesiva aceleración de la turbina. Véase en la *Figura 2.3*.

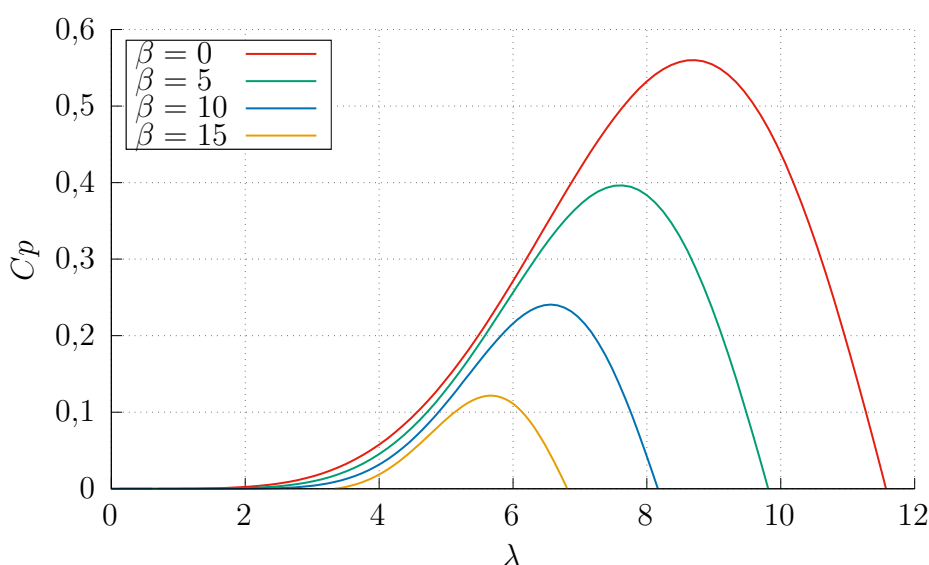


Figura 2.3: Coeficiente de potencia según el *tip speed ratio* λ y el *pitch* β

2.2. Esquema de control

Los primeros sistemas de control integrados eran mecánicos y su principal objetivo consistía en proteger el mecanismo ante condiciones meteorológicas adversas. A medida que las turbinas aumentaron en tamaño y potencia, el control se sofisticó. Los nuevos sistemas no sólo protegen la turbina sino que también mejoran su eficiencia.

Las turbinas *offshore* modernas utilizan controles de *pitch* y de carga. El *pitch* y la carga son los encargados de regular la velocidad angular del rotor y la producción del generador, respectivamente. Algunos tipos de turbinas, especialmente las onshore, también emplean el control de *yaw* para orientar las palas en dirección perpendicular al viento. Es decir, giran la góndola para que la corriente impacte de frente.

De una turbina eólica se espera obtener una transformación de energía mecánica en eléctrica de manera segura y eficiente. Ambos propósitos deben ir de la mano, maximizando los beneficios y manteniendo un rango de actuación suficientemente holgado para atajar complicaciones.

Se ven, a continuación, los objetivos de control en función de la velocidad del viento y la técnica para alcanzarlos usualmente aplicada en esta industria.

2.2.1. Objetivos de control

Los objetivos del aerogenerador varían para optimizar el rendimiento en función del tramo de operación. Estas etapas están definidas por la curva de potencia ideal del aerogenerador que se puede ver en la *Figura 2.4*. La zona operacional está delimitada por las velocidades del viento $v_{\min}$ y $v_{\max}$ [16]. Por debajo de la velocidad $v_{\min}$, la energía eólica es demasiado débil para compensar la resistencia del mecanismo de la turbina. Por encima de $v_{\max}$, el sistema debería detener su funcionamiento para evitar sobrecargas y fallos estructurales.

Estos límites dependen del diseño de la propia turbina. La amplitud de la zona operacional debe responder a un equilibrio entre coste y beneficio. Disminuir el valor de $v_{\min}$ o aumentar el valor de $v_{\max}$ reduciría la rentabilidad económica a causa de añadir nuevos valores de carga o de reforzar estructuralmente el aerogenerador, y no aumentarían significativamente la producción de potencia. La zona no operacional de viento suave no generaría apenas potencia y la de viento fuerte continuaría estando restringida a una potencia máxima.

Objetivos de cada tramo

Los tramos operacionales se encargan de adecuar la producción de potencia en función de la velocidad del viento siguiendo la forma ideal de la *Figura 2.4*.

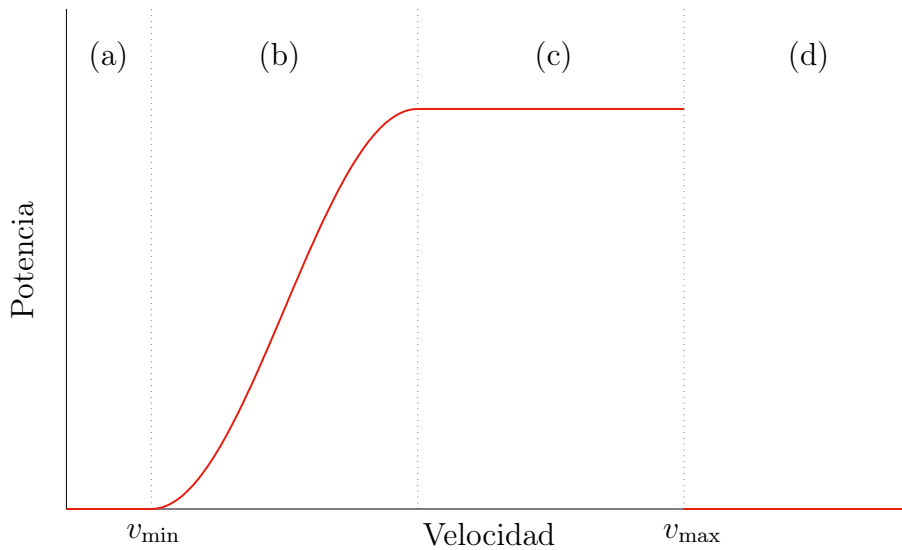


Figura 2.4: Tramos de control

Tramo en calma (a) La velocidad del viento es inferior a $v_{\min}$. Aún no es capaz de hacer girar el rotor de la turbina y no genera potencia.

Tramo de arranque (b) El objetivo de este tramo es maximizar la captura de energía eólica para alcanzar lo antes posible la potencia nominal.

Tramo de producción (c) Tras alcanzar cierto valor de potencia, el sistema debe reducir suavemente la producción para absorber la máxima energía posible mientras mantiene un comportamiento controlado. Una vez estabilizado, la producción de energía eléctrica es óptima y debe permanecer constante en régimen estacionario. De esta forma no se satura y asegura tanto la estructura como los componentes internos.

Tramo de parada (d) Cuando el viento supera la velocidad $v_{\max}$, la turbina coloca las palas en bandera para evitar que los sistemas eléctricos y estructurales puedan llegar a dañarse.

2.2.2. Controlador PID

Un controlador PID, en un lazo de control cerrado, permite regular la entrada de un sistema para que alcance la salida esperada. Es la técnica de control dinámico más utilizada [17].

El acrónimo PID hace referencia a las acciones que aplica el controlador: Proporcional, Integral y Derivativa. En algunos casos algunos términos se dejan fuera porque no son necesarios, por lo que es posible tener controladores PI, PD o simplemente P.

A partir de una señal de referencia, que indica la salida que se espera obtener del sistema, y de la señal real de salida, se calcula la de error, $e(t)$. Esta última señal es la que le indica al controlador lo lejos que se encuentra el estado actual del sistema del estado deseado.

Se ven a continuación, por separado, las acciones que aplica el controlador, siendo $u(t)$ su salida o acción de control:

Proporcional (K_p) Aplica un control proporcional al error actual:

$$u(t) = K_p e(t)$$

Esto aumenta la velocidad de respuesta y reduce el error estacionario, pero también aumenta la inestabilidad.

Integral (K_i) Aplica un control proporcional a la acumulación del error:

$$u(t) = K_i \int_0^t e(\tau) d\tau$$

Esto reduce el error estacionario, pero añade cierta inercia al sistema que lo hace más inestable.

Derivativa (K_d) Aplica un control proporcional a la tasa de variación del error:

$$u(t) = K_d \frac{de(t)}{dt}$$

Esto permite reconocer la velocidad a la que el sistema se acerca a la referencia, para poder frenarle con antelación y evitar que se sobrepase, aumentando la estabilidad del sistema. Disminuye ligeramente la velocidad de respuesta pero no altera el error estacionario.

2.3. Problemática

Las turbinas eólicas flotantes están expuestas a perturbaciones ambientales más extremas que las situadas cerca de la costa o en tierra. Los fuertes vientos generan un empuje en el rotor, que se transmite a toda la estructura, afectando a la estabilidad de la plataforma. Las olas y las corrientes también producen cargas y vibraciones indeseadas. Las fuerzas aerodinámicas e hidrodinámicas no deben despreciarse, lo que da lugar a modelos más complejos [18].

Los métodos de control clásicos como el PI o el PID ya no son apropiados debido al mayor número de variables a tener en cuenta [19]. Nuevas propuestas basadas, por ejemplo, en la inteligencia artificial o el control predictivo permitirían aumentar la fiabilidad de las turbinas eólicas flotantes, optimizar su producción energética y, posiblemente, el uso de estructuras más ligeras y económicas. La mejora de las tecnologías relativas a los sensores, así como la gestión de sus datos, facilitaría el uso de sistemas de control más complejos e inteligentes.

Capítulo 3

Gemelo Digital

3.1. Definición

Según Michael Grieves y John Vickers [20], precursores del concepto, un Gemelo Digital es un conjunto de construcciones de información virtual que describen completamente un sistema físico, potencial o real. En su punto óptimo, cualquier información que se pueda obtener de la inspección de un producto fabricado físicamente se puede obtener también de su Gemelo Digital.

La *Figura 3.1* distingue tres partes principales: el sistema físico, una representación virtual del mismo y las conexiones de datos bidireccionales entre ellos, que alimentan lo virtual con información de lo físico y de nuevo lo físico con los datos procesados.

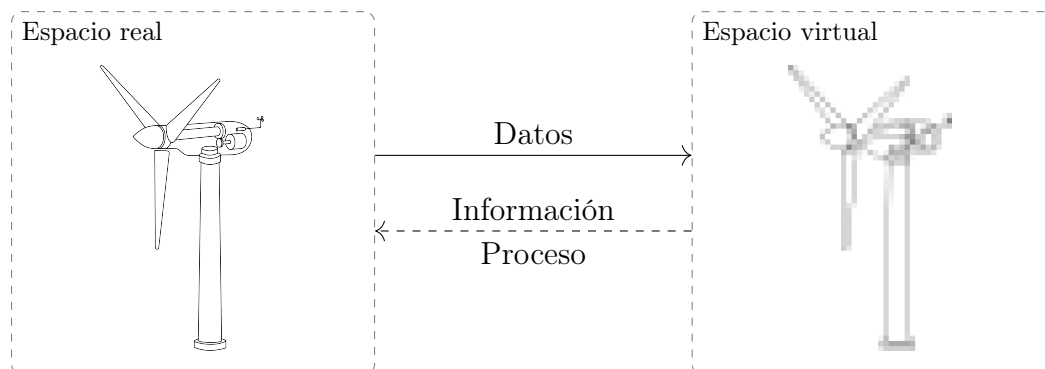


Figura 3.1: Primer concepto del Gemelo Digital de Grieves y Vickers.

Nótese que el Gemelo Digital es una construcción lógica, lo que significa que la información y los datos pueden estar contenidos en otras aplicaciones.

3.2. Origen

La idea de utilizar un Gemelo Digital como medio de estudio de un objeto físico se puede encontrar mucho antes de la primera aparición oficial del término. La NASA fue pionera en su uso durante las misiones de exploración espacial de la década de 1960, cuando cada nave era replicada con exactitud en una versión terrestre, con fines de análisis y simulación por parte del personal que prestaba servicios a los tripulantes [21].

En el año 1970 tuvieron una relevancia determinante cuando la misión tripulada Apolo 13 experimentó una explosión en un tanque de oxígeno del módulo de servicio de la nave. La NASA, frente a una nave espacial averiada y astronautas varados a 330.000km de distancia de la Tierra, empleó las 15 simulaciones de alta fidelidad con las que contaban para probar diferentes escenarios de reparación y así indicar a la tripulación cómo mantener los niveles de oxígeno y qué pasos llevar a cabo en la nave para poder regresar.

Por sí solo, un simulador no es un Gemelo Digital. Lo que distingue a la misión Apolo 13 es la forma en que se pudieron adaptar y modificar rápidamente las simulaciones para que coincidieran con las condiciones de la nave espacial averiada de la vida real. Eso permitió investigar, rechazar y perfeccionar las estrategias necesarias para traer a los astronautas de vuelta a casa [22].

Más de 30 años después, en una conferencia de la Universidad de Michigan sobre la gestión del ciclo de vida del producto, Grieves describía las representaciones virtuales como «relativamente nuevas e inmaduras» y la documentación sobre productos físicos como «limitada, recopilada manualmente y en su mayoría en papel». Es durante esa presentación cuando propone, por primera vez, el concepto de Gemelo Digital, que se le atribuye desde entonces. No obstante, no recibe ese nombre hasta que posteriormente el compañero de trabajo para la NASA de Grieves, John Vickers, empieza a referirse a él como tal [23].

En los últimos años los Gemelos Digitales se han utilizado en la NASA para el desarrollo de vehículos y aviones de próxima generación, así como en otras muchas áreas, desde la medicina hasta la producción de energías renovables y sostenibles. De hecho, desde 2010 han experimentado un crecimiento exponencial [24], paralelo al IoT, tecnología a la que están fuertemente ligados.

3.3. Tipos

En función de la madurez de la entidad de estudio, se pueden distinguir los siguientes tipos de Gemelo Digital [20]:

Prototipo (DTP) Describe la forma y el comportamiento del objeto físico antes de que haya sido creado.

Instancia (DTI) Describe un objeto físico concreto y permanece vinculado a él durante toda su vida útil. Almacena lo que el DTP junto con los resultados de cualquier medición y prueba, un registro de los componentes reemplazados y el histórico de estados operativos capturados, a partir de datos actuales, pasados y futuros predichos.

Agregado (DTA) Es la agregación de todos los DTI. Puede ser una construcción informática que tenga acceso a todos ellos y examine continuamente las lecturas de los sensores. Así podría hacer correlaciones con fallos y permitir diferentes pronósticos y aprendizajes. Por ejemplo, se le podría preguntar: «¿Cuál es el tiempo medio de deterioro del componente X?».

3.4. Aplicación y beneficios

Los Gemelos Digitales permanecen ligados a ciclos de desarrollo completos, desde el diseño de un producto hasta su implementación, e incluso su desmantelamiento.

En la fase de creación el objeto físico todavía no existe, pero empieza a tomar forma en un espacio virtual como un DTP. Esto permite a los desarrolladores probar y entender cómo se comportarán sus sistemas en una amplia variedad de entornos.

Una vez completado y validado el sistema virtual, la información recopilada se utiliza para crear su gemelo físico. Si se ha trabajado con modelos matemáticos correctos, se habrán reducido drásticamente el número de prototipos físicos fallidos.

En la fase de producción se establece el flujo de datos bidireccional entre el producto fabricado y su DTI. Si el objeto de estudio es, por ejemplo, una turbina eólica, esta se equipa con varios sensores relacionados con áreas vitales de su funcionalidad [21]. Estos sensores producen datos sobre diferentes aspectos del rendimiento del objeto físico, como la producción de energía, la temperatura o las condiciones climáticas. Luego, estos datos se transmiten a su versión digital. Además, a medida que el sistema físico sufre cambios, estos se capturan en la entidad virtual para conocer su configuración exacta.

El Gemelo Digital recopila y analiza los datos del producto en la vida real y proporciona sugerencias para hacerlo aún más eficiente. Por ejemplo, puede deter-

minar el impacto en la turbina si el viento soplara más fuerte, si soplara durante más tiempo o si se parase, ayudando en la toma de decisiones en cada caso.

Agrupando varias entidades físicas relacionadas en un DTA se puede llevar a cabo supervisión y mantenimiento preventivos. Tener constancia de cuando y bajo qué circunstancias ha tenido que ser reemplazado algún componente de una de las instancias permite pronosticar cuando va a tener que reemplazarse en las demás. Industrias grandes y poco accesibles se benefician enormemente de esto.

En la fase de desmantelamiento, el Gemelo Digital ayuda a decidir qué hacer con las piezas, al tener registrado el estado de cada una de ellas. Además, la información recopilada durante todo el ciclo de vida resulta de gran utilidad para perfeccionar las siguientes versiones del producto.

Parte II

Desarrollo del sistema

Capítulo 4

Aprendizaje y dominio de las tecnologías

En este primer capítulo de la parte II, se describe el desarrollo de un prototipo inicial de baja fidelidad que marcará la hoja de ruta del proyecto final. Es un lazo de control muy sencillo que varía el *pitch* en función de la inclinación de la turbina y envía los datos de su estado en tiempo real a un servidor.

A continuación se mostrará la evolución del prototipo haciendo hincapié en los problemas y restricciones y en las tecnologías empleadas para atajarlos. Se ejecutarán varios sprints para cumplir con objetivos iniciales o con nuevos propuestos por los clientes.

4.1. Desarrollo de un lazo de control básico

El primer paso será en el desarrollo del lazo de control. Debe mover un micro-servomotor atendiendo al ángulo de inclinación de un acelerómetro.

Dada una entrada, el sistema debe leerla, interpretarla y enviarla a la salida correspondiente en tiempo real con una frecuencia suficientemente sensible. Para materializarlo, la entrada será un sensor de medición inercial y la salida un microservomotor. El microcontrolador se encargará de seleccionar y transformar los datos.

Componentes

- *Microcontrolador NodeMCU ESP8266*
Alimentado vía nano USB con un ordenador o con un transformador de 5V.
- *Unidad de medición inercial (IMU) MPU6050*
Alimentado con los 3.3V de la placa ESP32 y conectado a la masa común.

Sus pines Serial Data (SDA) y Serial Clock (SCL) están enlazados con los respectivos pines del microcontrolador mediante el protocolo I2C¹.

■ *Microservomotor SG92R*

Alimentado con un transformador independiente de 5V y conectado a la masa común. Su pin de control está enlazado con un pin del microcontrolador compatible con Pulse Width Modulation (PWM²).

Véase el montaje en la *Figura 4.1*.

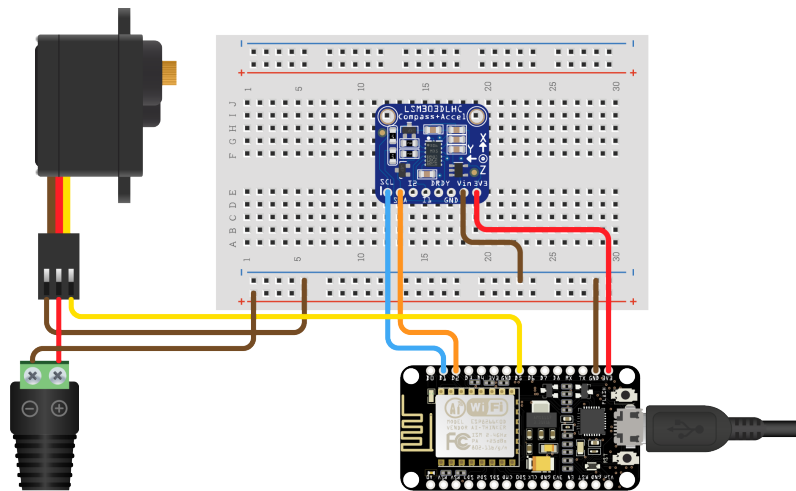


Figura 4.1: Esquema de conexiones

Entorno de desarrollo

Arduino IDE.

4.1.1. Arduino IDE

El entorno de desarrollo de Arduino es un programa multiplataforma que facilita y agiliza las tareas de codificación, compilación y subida de software a componentes hardware. Es compatible con los lenguajes de programación C y C++ y con muchos microcontroladores genéricos.

¹Es el protocolo de comunicación del bus de datos que permite la comunicación de los microcontroladores con sus múltiples periféricos en sistemas integrados mediante solamente dos cables, uno para el reloj (SCL) y otro para el dato (SDA).

²Es una técnica que modifica una señal periódica para transmitir información a través de un canal de comunicaciones o para controlar la cantidad de energía que se envía.

El fichero de código Arduino está formado por dos funciones: `setup()` y `loop()`. La función `setup()` se utiliza para inicializar componentes del sistema y se ejecuta una única vez. La función `loop()` es el ciclo principal del programa y se ejecuta indefinidamente. En la compilación, el entorno utiliza `avrdude` para convertir el código ejecutable en un archivo de texto hexadecimal que se carga en el firmware de la placa.

Esta versatilidad hace del IDE de Arduino un programa útil para el desarrollo en prototipos y versiones alfa y beta, donde los factores de sencillez y rapidez juegan papeles clave.

Implementación

Un fichero `imu_servo.ino` almacena el código del programa.

Configuración El programa comienza inicializando variables y módulos.

Bucle Realiza una lectura periódica del sensor de aceleración incorporado en la IMU a través de los pines I2C del microcontrolador. Los valores crudos³ se almacenan en tres variables, correspondientes a cada eje del espacio x , y , z , contenidas en el intervalo $[-32768, 32767]$ —equivalente a un número binario de 16 bits con signo— y se transforman en valores del intervalo $[0, 180]$. Los valores reescalados ya pueden ser interpretados por el servomotor y se envía directamente la inclinación correspondiente a la variable y .

La *Figura 4.2* recoge el orden de ejecución.

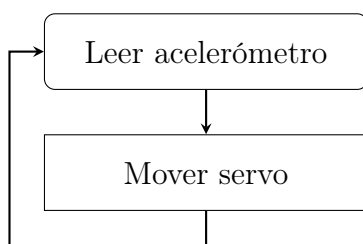


Figura 4.2: Diagrama de flujo.

³Valores que se leen directamente de los sensores sin ser manipulados.

Resultado

El servo se inclina de manera simultánea al acelerómetro en el eje y .

Problemas y restricciones

- Los módulos hardware más asequibles disponen de documentación muy pobre y es difícil conocer el funcionamiento a más bajo nivel del dispositivo.
- Los datos crudos leídos del acelerómetro presentan mucho ruido que se traslada directamente al movimiento del servo.

Conclusiones

El propósito más importante de este primer paso ha sido la interacción con todo el sistema, desde el entorno software al hardware. Un primer acercamiento que introdujo el uso del entorno, la inclusión de librerías, la compilación, la carga en placa, la depuración, los esquemas de conexiones y la traducción de los valores crudos de los componentes.

También ha servido para visualizar las limitaciones técnicas de los módulos.

4.2. Monitorización y almacenamiento en un servidor

Partiendo de la arquitectura e implementación previas, se ejecuta un nuevo sprint. La monitorización puede reducirse al muestreo vía puerto serial de los valores del acelerómetro. El objetivo es, sin embargo, almacenar en un servidor la información de la inclinación del sistema. Para facilitar su posterior integración, con la mirada puesta en MATLAB, se utilizará el servicio de almacenamiento online gratuito de ThingSpeak.

4.2.1. ThingSpeak

ThingSpeak es una plataforma de IoT que permite añadir, visualizar y analizar flujos de datos en directo en la nube. Proporciona visualizaciones instantáneas de los datos publicados por los dispositivos asociados. Gracias a su gran capacidad de integración con el entorno y código MATLAB, puede realizar el análisis y el procesamiento de los datos a medida que los recibe. ThingSpeak se utiliza a menudo en la creación de prototipos y sistemas IoT de prueba que requieren posteriores análisis.

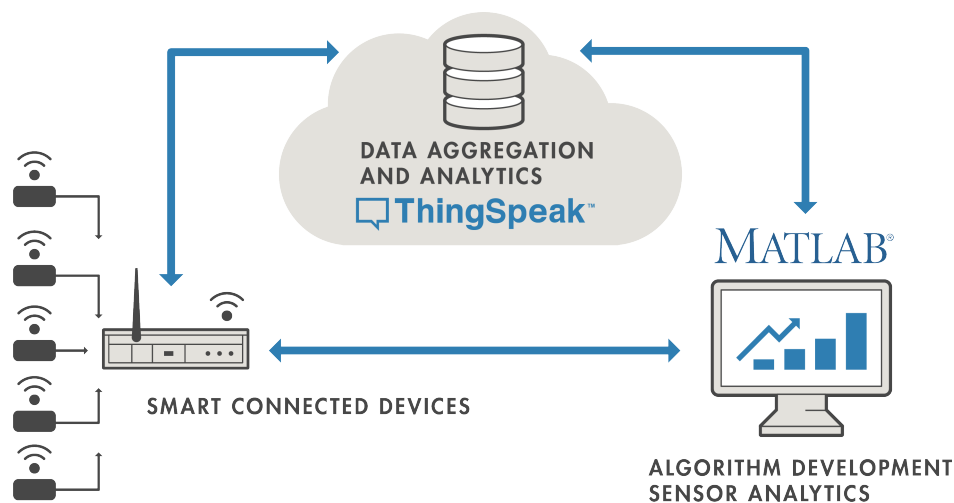


Figura 4.3: Esquema de funcionamiento [1]

La Figura 4.3 indica los tres actores principales implicados en el funcionamiento de ThingSpeak:

Los dispositivos IoT Recopilan datos por sus sensores y los envían a través de una conexión a Internet a la plataforma online de ThingSpeak. En este caso, el lazo de control ejecutado en el microcontrolador actúa como un dispositivo IoT.

La nube de ThingSpeak Almacena un histórico y es capaz de interpretar los datos recibidos en tiempo real.

El entorno de MATLAB Implementa código asociado al dispositivo IoT. En este caso, los datos se extraen de la plataforma de ThingSpeak al software de escritorio y, después, un ingeniero especializado realiza el análisis y diseña algoritmos para ser ejecutados local o remotamente en el prototipo.

Funcionamiento

La licencia gratuita de ThingSpeak ofrece la posibilidad de crear 4 canales de comunicación, cada uno con un identificador único, varios campos de metadatos, varios campos de almacenamiento y unos permisos de acceso.

Para realizar lecturas o escrituras sobre un campo de almacenamiento de un canal, el cliente debe conocer el identificador del canal y unas claves de acceso (API Keys) distintas para leer y escribir. Además, el cliente también puede leer los campos de metadatos. El protocolo de comunicación utilizado es HTTP, a través de los métodos GET y POST.

Características

Las principales ventajas del uso de esta plataforma respecto a un servidor convencional son:

- La sencillez en el prototipado de sistemas IoT sin configurar servidores ni desarrollar software web.
- El uso de protocolos estándares para la lectura y escritura de datos.
- La gran capacidad de integración con MATLAB para la implementación de algoritmos.

Limitaciones

La licencia gratuita para pequeños proyectos con fines no comerciales impone ciertas restricciones que afectan negativamente al desarrollo:

- La limitación en el número de envíos de datos es de 3 millones por año, unos 8.200 por día.
- El intervalo de tiempo mínimo entre envíos debe ser superior a 15 segundos.
- El número máximo de canales es 4.

Implementación

Un fichero `imu_thingspeak_servo.ino` almacena el código del programa.

Configuración El programa comienza inicializando los objetos y estableciendo conexión a Internet vía WiFi.

Proceso general Periódicamente, tras la lectura del sensor de aceleración, los datos crudos se envían a ThingSpeak escribiendo los valores de inclinación de cada eje en los 3 campos de almacenamiento habilitados. Para este envío, el programa hace uso de las librerías de ThingSpeak e incorpora un fichero `credentials.h` que almacena el identificador del canal y la clave de escritura. Dado que la versión gratuita de este servicio impide subir datos en intervalos de tiempo inferiores a 15 segundos, solo se almacenarán los datos obtenidos cada dicho periodo.

La *Figura 4.4* recoge el orden de ejecución.

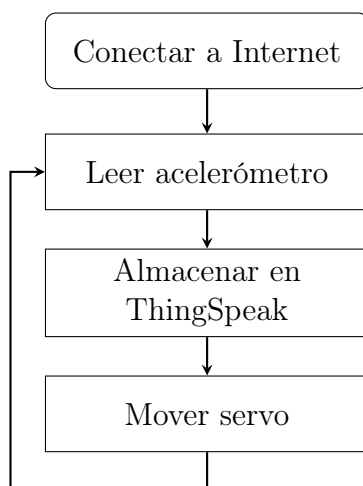


Figura 4.4: Diagrama de flujo.

Resultado

El lazo de control ahora almacena un histórico de los datos registrados por el acelerómetro cada 15 segundos. Sin embargo, el servo presenta pequeñas interrupciones en su actuación a causa de las subidas al servidor.

Problemas y restricciones

- El retraso de respuesta del lazo de control aumenta cuando las operaciones con los datos entre la lectura del acelerómetro y la escritura en el servo son computacionalmente más exigentes o incluso bloqueantes. En este caso, la conexión a Internet o el estado del servidor pueden interrumpirlo unos segundos.
- La limitación de ThingSpeak supone una pérdida significativa de datos en proyectos a pequeña escala donde los cambios se producen con mayor rapidez y frecuencia.

Conclusiones

Aunque las interrupciones en el lazo son breves, suponen una grieta en la seguridad del control autónomo de la turbina. Limitar el tiempo máximo de interrupción podría subsanar levemente el problema pero se perderían aún mayor cantidad de datos y no atajaría el fondo de la cuestión. Además, la frecuencia de muestreo de ThingSpeak es demasiado baja para pruebas en un prototipo a pequeña escala.

4.3. Control y monitorización con multihilo

Para evitar las interrupciones en el lazo de control, la solución conceptualmente más sencilla pasa por la implementación de un sistema concurrente donde los procesos de control y comunicación no se vean interrumpidos entre sí.

El aumento en la frecuencia de muestreo de ThingSpeak se realizará enviando paquetes de datos temporalmente equidistanciados en formato JSON cada vez que se vuelva a habilitar la subida.

4.3.1. Multihilo

La mayoría de proyectos IoT están desarrollados sobre microcontroladores de un solo núcleo. En ocasiones, esta limitación supone pequeñas suspensiones y pérdidas de control. A pesar de ello, no suponen un problema en la mayoría de los casos si se utilizan en sistemas no sensibles, donde los tiempos de actualización no suponen un problema de seguridad. Algunos ejemplos son las estaciones meteorológicas, los accesorios domóticos o los geolocalizadores de fauna.

A continuación, se expone el marco teórico de la programación concurrente sobre el que se sustenta la nueva estrategia capaz de evitar interrupciones no deseadas.

Programación concurrente

Un programa concurrente [25] está formado por dos o más procesos que trabajan en conjunto sobre uno o más procesadores para llevar a cabo una misma tarea. Cada proceso es individualmente un programa secuencial, donde las instrucciones se ejecutan unas detrás de otras en orden.

Los procesos de un programa concurrente trabajan juntos comunicándose entre ellos. La comunicación se realiza a través de variables compartidas o de paso de mensajes. En el método de variables compartidas, una variable perteneciente al ámbito de ambos procesos es modificada por uno de ellos y leída por el otro. En el de paso de mensajes, un proceso envía un mensaje al otro a través de una estructura de comunicación común.

Independientemente del modo de comunicación empleado, los procesos deben sincronizarse entre ellos siguiendo unas normas conocidas comunes. Esta sincronización también se puede realizar de dos formas, a través de exclusión mutua o a través de sincronización condicional. La exclusión mutua asegura que las secciones críticas de los procesos no sean ejecutadas simultáneamente. La sincronización condicional retrasa o interrumpe la ejecución hasta que cierta condición se cumpla.

Arquitectura hardware

Un monoprocesador actual contiene la unidad de procesamiento central (CPU), la memoria principal, uno o más niveles de memoria cache, almacenamiento externo y otros periféricos como pantalla, teclado...

Los procesadores con más de un núcleo pueden organizarse de otras formas. Los de memoria compartida disponen de una red de interconexión que les permite compartir la memoria principal aunque cada uno tenga su propia memoria cache. Los de memoria distribuida también disponen de una red de interconexión pero, en este caso, cada uno gestiona su propia memoria principal.

Programación paralela en arquitecturas multiprocesador

La programación paralela es una subcategoría de programación concurrente donde el número de procesos coincide con el número de procesadores. Un programa en paralelo puede comunicar sus procesos mediante variables compartidas o paso de variables, pero esto depende en gran medida del hardware.

El objetivo principal del paralelismo en la programación es la reducción de los tiempos de ejecución. Aunque también atiende, como es el caso, a procesos críticos que necesitan la disponibilidad absoluta de un procesador propio.

4.3.2. JavaScript Object Notation (JSON)

JSON es un formato de texto sencillo para el intercambio de datos. Este formato debe escribirse y leerse como un objeto nativo de JavaScript pero puede enviarse como cadena de caracteres a través de la red.

Estructura la información de manera jerarquizada, clara y natural de forma que es fácilmente interpretable por humanos y máquinas.

Los tipos de datos disponibles son números, cadenas, booleanos, *arrays* y colecciones no ordenadas de pares de la forma `<clave>:<valor>`.

Componentes

- *Microcontrolador ESP32 sustituye al microcontrolador NodeMCU ESP8266*
Alimentado vía nano USB con un ordenador o con un transformador de 5V.

Véase el montaje en la *Figura 4.5*.

4.3.3. Microcontrolador ESP32

El microcontrolador ESP32 está diseñado para ser escalable y adaptativo. Cuenta con dos procesadores que pueden ser controlados de manera autónoma, uno de ellos de baja potencia para tareas de menor coste computacional en periodos de escasez energética. El software nativo integrado es freeRTOS, un sistema operativo en tiempo real de código abierto.

Es compatible con protocolos de comunicación serial, con hasta 38 pines de entrada y salida y con los protocolos Wi-Fi y Bluetooth, que incorpora la tecnología de bajo consumo Bluetooth Low Energy (BLE). La conectividad inalámbrica que ofrece este hardware permite la conexión a Internet a través de Wi-Fi y, a su vez, la comunicación con dispositivos locales a través de Bluetooth.

Puede alimentarse con un voltaje de entre 3.3V y 5V. Para suministrar suficiente potencia a los periféricos es recomendable utilizar voltajes cercanos al límite superior y así evitar pérdidas de energía. En estado de reposo, el microcontrolador tiene un consumo de corriente inferior a $5\mu\text{A}$, idóneo para pequeñas baterías y dispositivos compactos.

Estas y otras características, como su reducido tamaño o su gran resistencia a temperaturas adversas, convierten al módulo ESP32 en un elemento central del desarrollo de la tecnología IoT.

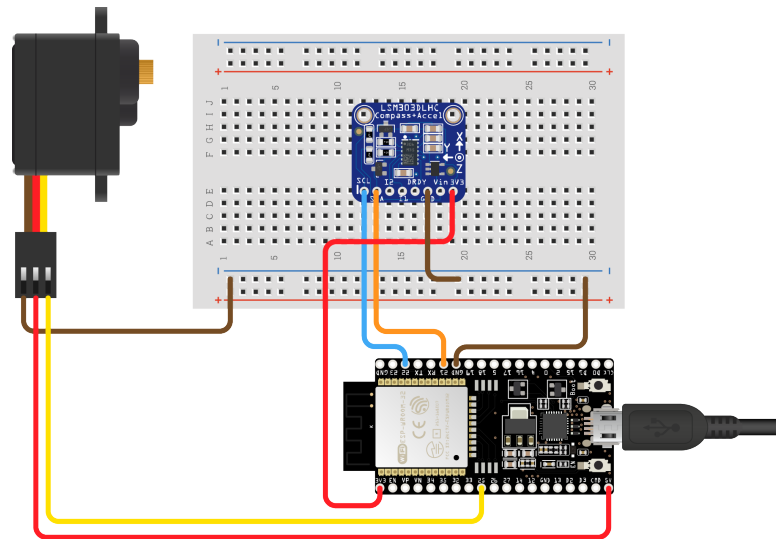


Figura 4.5: Esquema de conexiones con el microcontrolador ESP32

Implementación

El algoritmo alojado en el proceso general divide ahora sus tareas en dos procesos asignados a distintos núcleos del nuevo microcontrolador. Por un lado, el proceso de control con la lectura del acelerómetro y su posterior escritura en el servo y, por otro lado, el proceso de comunicación con la subida de los muestreos a ThingSpeak.

El microcontrolador ESP32 es un multiprocesador de dos núcleos con memoria compartida. En este caso, aplicando un enfoque estrictamente teórico, la comunicación entre procesos de distintos núcleos debería realizarse mediante el uso de variables compartidas. Sin embargo, la decisión de diseño más adecuada es el uso de paso de mensajes tipo cola o tubería de UNIX. Con el paso de mensajes se consigue mayor expresividad, generalización y modularidad. En primer lugar, las llamadas de envío y recepción son muy descriptivas y prácticamente atómicas. Permite que la misma lógica del programa sirva para implementaciones donde la memoria no sea compartida, donde el lazo de control esté en la góndola y el de comunicación no. Por último, aísla ambos procesos evitando que un eventual contratiempo en un proceso ralentice ambos.

Un fichero `pitch_controller.ino` almacena el código del programa.

Configuración Una vez establecida la conexión a Internet vía WiFi, el código asigna cada uno de los dos procesos, organizados por funciones, a procesadores diferentes gracias a las instrucciones de la librería de FreeRTOS. El proceso de comunicación se asigna al núcleo 0, que ya alberga labores internas relacionadas con protocolos de WiFi y Bluetooth, y el proceso de control al núcleo 1, reservado para las aplicaciones de usuario.

Proceso de control Lee el acelerómetro, mueve el servo y almacena los datos numéricos. Para enviar los datos al proceso de comunicación se hace la media aritmética de los datos numéricos almacenados en el último segundo y se empaquetan junto con la hora actual en formato EPOCH⁴. El sistema obtiene la hora EPOCH del servidor de consultas libre y colaborativo pool.ntp.org.

Proceso de comunicación Espera hasta recibir los datos correspondientes a los últimos 15 segundos y, una vez transcurridos, obtiene un vector de 15 muestreos equidistanciados 1 segundo entre sí. Este vector se transforma a objeto JSON con las correspondientes credenciales de ThingSpeak y se sube al servidor como cadena.

⁴El tiempo EPOCH es la hora actual medida en número de segundos desde el EPOCH UNIX, el 1 de enero de 1970 a las 00:00:00 GMT.

Cola compartida Una cola FIFO⁵ accesible por ambos procesos se encarga de comunicar internamente ambos procesos de manera segura. El proceso de control envía el dato a través de la cola aunque, si la cola está llena, omite el envío sin esperas. El proceso de comunicación, sin embargo, espera indefinidamente hasta que haya algún elemento disponible en la cola para extraerlo.

La *Figura 4.6* recoge el orden de ejecución.

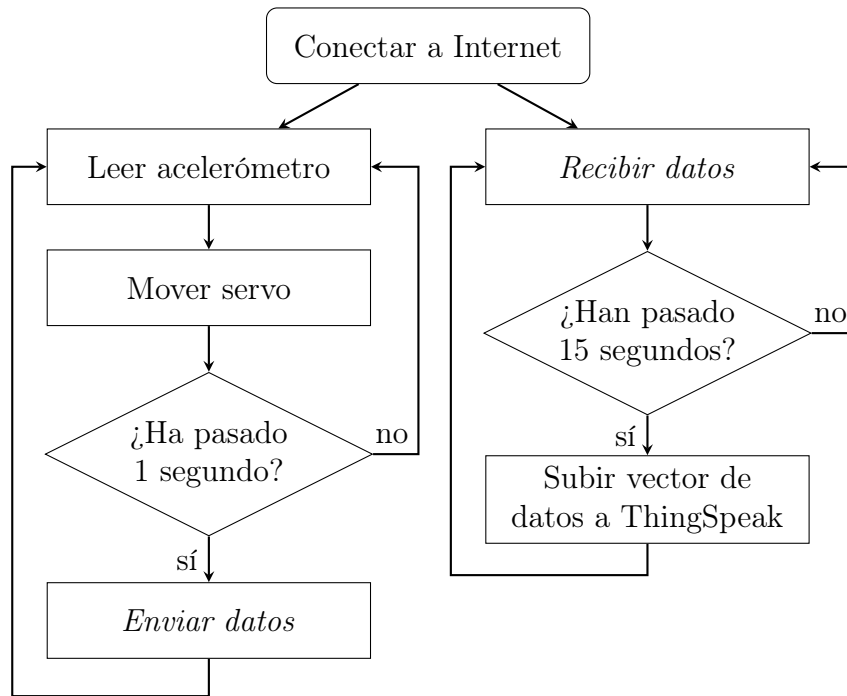


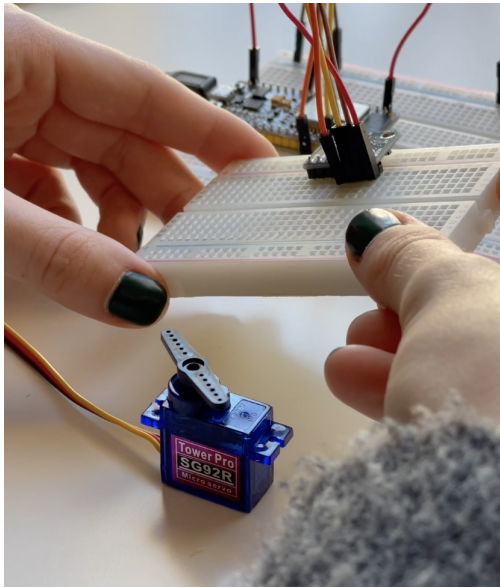
Figura 4.6: Diagrama de flujo.

Resultado

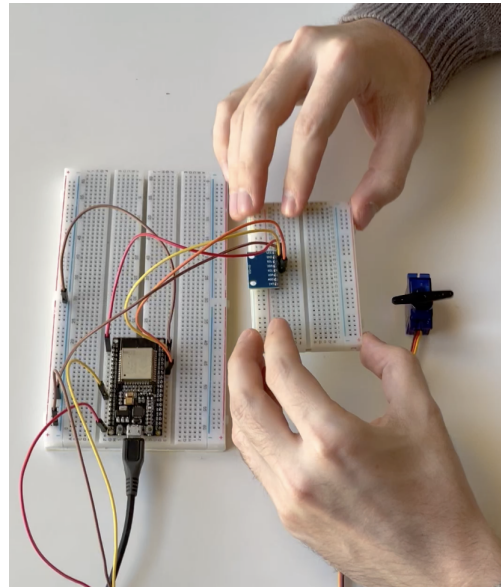
El resultado es un mecanismo básico de control y monitorización del *pitch* de la turbina a través de la posición de un microservomotor. Un lazo de control robusto con la fluidez de la implementación inicial, donde no existían más tareas que las implicadas en el control, y un proceso de comunicación capaz de almacenar los muestreos en ThingSpeak en intervalos de 1 segundo.

Véase el prototipo de baja fidelidad en la *Figura 4.7*.

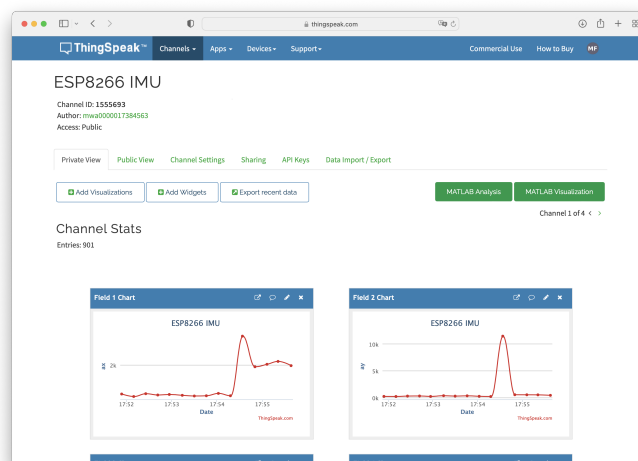
⁵Cola *First In First Out*.



(a) Con el acelerómetro inclinado



(b) Con el acelerómetro plano



(c) Servidor de ThingSpeak

Figura 4.7: Funcionamiento

Problemas y restricciones

- La cola de paso de mensajes puede llegar a llenarse si las instrucciones de subida tardan sustancialmente más de lo habitual. En ese caso, los envíos omitidos se perderían para siempre.

Conclusiones

El sistema es estable y seguro. Ante cualquier problema durante la subida de los datos, el lazo de control no se ve afectado y, una vez solucionado, recupera el funcionamiento normal.

La implementación puede requerir de calibraciones para adecuar la inclinación de la estructura y el ángulo del *pitch* a las necesidades de la turbina. Además, sería interesante establecer tiempos de ejecución fijos y acotados para robustecer aún más el sistema.

4.4. Aportaciones

A lo largo de este capítulo se han introducido las herramientas y las soluciones necesarias para el desarrollo de un prototipo de monitorización y control de una turbina eólica de baja fidelidad. La base del proyecto, un microcontrolador mononúcleo, componentes hardware sencillos, el entorno de desarrollo de Arduino y el servidor ThingSpeak, tuvo que ser modificada para solucionar limitaciones y alcanzar el resultado deseado. Para evitar los tiempos de espera en las instrucciones de comunicación con el servidor, se sustituye el microcontrolador mononúcleo por otro con dos procesadores; para sortear las limitaciones de la licencia gratuita de ThingSpeak se empaquetan y suben los datos en formato de archivo JSON.

Capítulo 5

Interfaz de control del Gemelo Digital

Una vez cumplido el objetivo de extraer los datos de un sistema en tiempo real, se plantea el desarrollo de la interfaz de control de una turbina eólica. En un trabajo a más alto nivel con el prototipo de baja fidelidad y el servidor con el que se comunica, se debe presentar y analizar la información alojada, así como permitir el envío de comandos al sistema. El concepto de Gemelo Digital, actualmente en auge, encapsula estas funciones y se adapta a la idea de aplicación que los clientes solicitaban.

Como se deduce del Capítulo 3, un Gemelo Digital debe otorgar una visión completa del estado de la entidad que refleja. Esto permite la toma de decisiones informadas y ayuda a visualizar, planificar y diseñar mejoras sobre esa entidad y los procesos que posibilita.

En el mundo real, las turbinas eólicas suelen agruparse en granjas, por lo que se propone un Gemelo Digital de tipo DTA, es decir, formado por un agregado de entidades diferentes. Por el momento está asociado al prototipo con el que se está trabajando y a una simulación, pero es completamente escalable.

En la vista principal, se muestran aspectos generales del conjunto de las turbinas. Se puede acceder a diferentes subvistas con los estados concretos de cada una de ellas. También se permite su control individual.

Debido a las características y la complejidad de la aplicación, resulta natural abordar su implementación a través de programación orientada a objetos (POO). De esta manera, el código resultará flexible, modular y seguro.

Alguno de sus identificativos clave son la herencia y la encapsulación, que se utilizarán para evitar repetir código —por ejemplo, en las clases de diferentes turbinas, cuyos atributos y funciones principales difieren en pequeños detalles— o para separar la parte gráfica del tratamiento de datos, respectivamente.

5.1. Herramientas

5.1.1. MATLAB

El entorno de desarrollo en el que se ha implementado el Gemelo Digital ha sido MATLAB, basado en el lenguaje de programación que lleva su mismo nombre. A continuación se expone por qué se ha escogido este software.

En primer lugar, por su implantación en el ámbito de la investigación. Los usuarios finales de este Gemelo Digital son científicos e ingenieros que van a probar nuevos algoritmos en diferentes prototipos. En este proyecto se proponen implementaciones tanto de un prototipo físico como de uno simulado, pero es importante que otras personas que vayan a utilizarlo en el futuro entiendan el lenguaje y puedan modificar el Gemelo Digital adecuándolo a sus propios sistemas y necesidades.

En segundo lugar, por su integración con la plataforma en la nube ThingSpeak y con el entorno de desarrollo de sistemas dinámicos Simulink. Esto facilita la comunicación bidireccional del Gemelo Digital con las turbinas eólicas sean del tipo que sean.

Los aspectos fundamentales de MATLAB incluyen operaciones básicas, como creación de variables, aritméticas y tipos de datos. Admite desarrollar programas tanto por procedimientos como orientados a objetos. Aunque difiere en algunos aspectos de otros lenguajes orientados a objetos como C++ o Java [26], de una manera u otra, permite aplicar las técnicas de este paradigma de programación.

Cabe destacar el tratamiento de los accesos a las variables privadas. Mientras que en lenguajes como Java se definen los métodos *get* para permitir la lectura de los atributos privados, en MATLAB basta caracterizar la propiedad¹ con `GetAcces` como público para acceder directamente a su valor [27]. De manera análoga, se utiliza `SetAcces` para permitir la escritura.

Otra distinción es que la instancia de la clase en la que se está ejecutando un método siempre viene dada como un parámetro explícito.

5.1.2. App Designer

Para la implementación de la GUI (*Graphic User Interface*) se ha partido de una clase muy básica generada en App Designer. Esta extensión de MATLAB permite crear interfaces gráficas de manera interactiva arrastrando y colocando los componentes visuales y consta de un editor integrado en el que aparece código generado automáticamente.

¹En MATLAB son los atributos usuales de una clase.

Tiene componentes estándar como botones y campos de texto; elementos de control como medidores, indicadores luminosos, controles y conmutadores que permiten replicar el aspecto y las acciones de los paneles de instrumentación. También se pueden utilizar componentes contenedores, como pestañas y paneles, y diseños de malla para organizar la interfaz de usuario.

El código generado es orientado a objetos, de forma que encaja adecuadamente con la aplicación que se pretende desarrollar. A pesar de que es limitado en cuanto a que la mayor parte no se puede editar, el código es fácilmente extraíble, ya que es accesible y puede ser copiado y pegado en un archivo `.m` de MATLAB como una nueva clase.

5.1.3. Patrones de diseño

El Gemelo Digital se materializa en una aplicación interactiva que muestra gran cantidad de información sobre el estado de varios sistemas y permite enviarles diferentes comandos, estableciendo así comunicaciones bidireccionales. Las interfaces de usuario son susceptibles de muchos cambios, y más si se pretende que sean escalables. Además, debe poder adaptarse a cualquier nuevo tipo de turbina, sean cuales sean sus características.

A continuación, se detalla la solución propuesta para su completo desarrollo.

Modelo-Vista-Controlador

El patrón de diseño de aplicaciones software Modelo-Vista-Controlador [28] separa la interfaz de usuario de la gestión de eventos y comunicaciones. A su vez, aísla estas dos de la lógica de negocio, es decir, del procesamiento y funcionalidad de los datos del sistema [29]. Esto resulta en una implementación modular, que permite modificar o reemplazar cualquiera de sus componentes sin necesidad de realizar cambios o siquiera entender las otras. Facilita además su ampliación, por un lado, añadiendo nuevas vistas y, por otro, sumando funcionalidades al modelo independientes del resto de componentes.

Modelo Encapsula la información y la funcionalidad. Cuando realiza cambios, avisa a los componentes que están escuchando. Es independiente de la interfaz de usuario, por lo que no tiene instancias de las clases de la vista ni del controlador.

Vista Toma la información del modelo y la muestra al usuario mediante una interfaz gráfica. En algunas variantes de este patrón de diseño, las interacciones del usuario se detectan directamente desde el controlador. En este caso, se atienden en la vista, que invoca a los métodos del controlador.

Controlador Gestiona las interacciones del usuario con la vista y las traduce en peticiones al modelo, en caso de que sea necesario. También lleva a cabo actualizaciones periódicas de los datos sin necesidad de que el usuario explícitamente lo solicite. Además, mantiene el comportamiento dinámico entre las diferentes vistas.

Véase este patrón de diseño sobre el Gemelo Digital en la *Figura 5.1*.

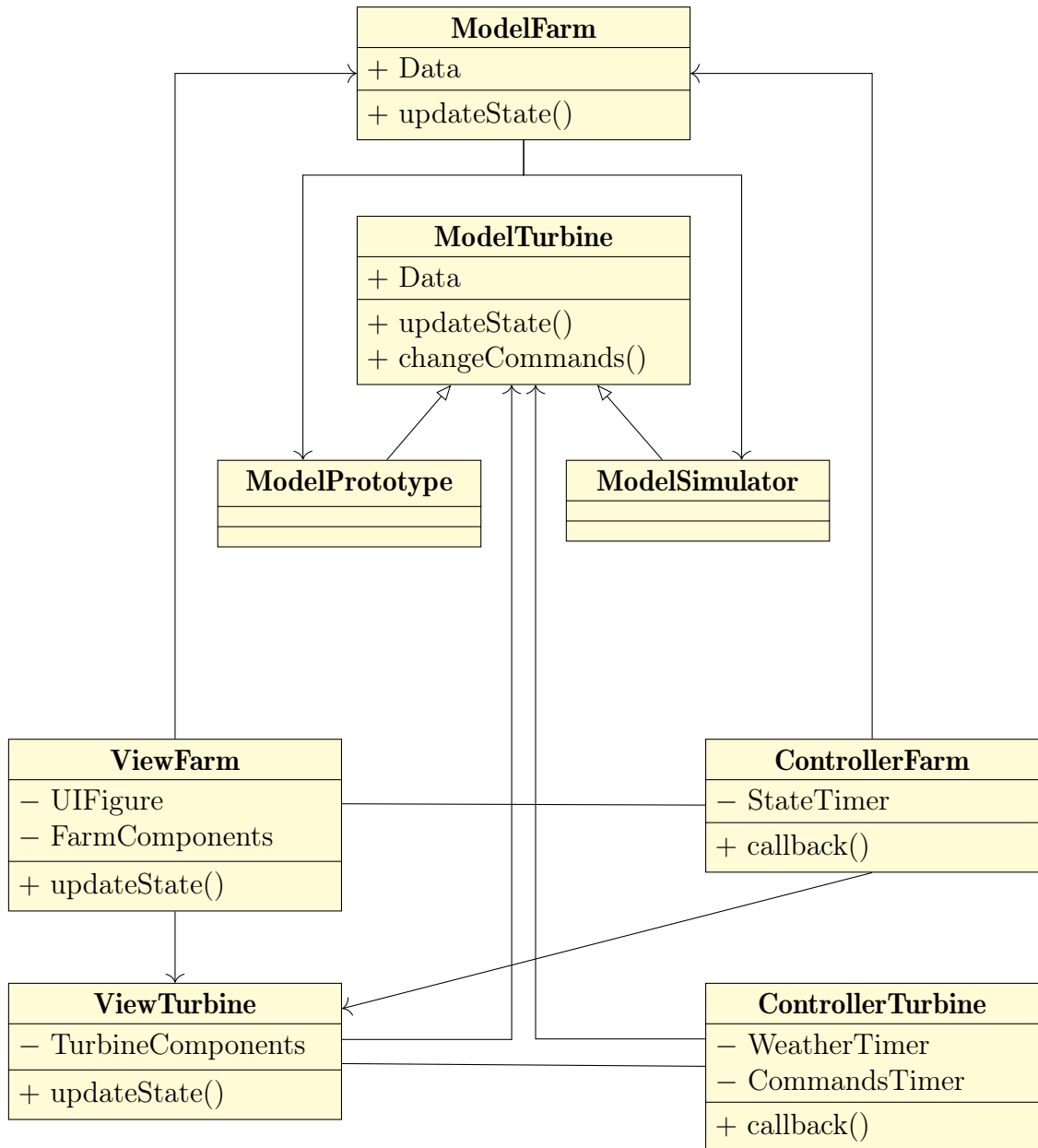


Figura 5.1: Clases del Gemelo Digital

Observer

Según el patrón anterior, el modelo debe notificar de los cambios que sufre, pero también debe ser independiente de la vista y del controlador, por lo que no tiene acceso a ellos. Para hacer esto posible, es común utilizar el patrón Observer.

Observer permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando. Se explica a continuación en qué consiste la variante implementada en el Gemelo Digital.

MATLAB cuenta con un objeto evento [30], que sirve para representar cambios o acciones que ocurren dentro de otro objeto. Para utilizarlo, la clase que manda el aviso debe tenerlo declarado y llamar al método `notify`. Desde la fuente no se puede definir qué objetos son notificados. Para escuchar el evento, otra clase debió haber añadido un oyente previamente, utilizando el método `addListener` con la clase origen y el evento, y la función que debe ejecutarse cuando este se dispara.

El modelo es el objeto observado. Sus clases cuentan con los eventos necesarios para describir cada cambio que pueda sufrir. Notifican del mismo cuando termina de llevarse a cabo, por si a cualquier otra parte le afecta y lo está escuchando.

El controlador posee a los objetos observadores. La vista no tiene oyentes al modelo ya que es el controlador el que se encarga de atender las notificaciones y de invocar sus métodos. Puesto que las instancias de la parte del controlador sí tienen acceso al modelo, al inicializarse pueden añadir oyentes de los eventos que les interesen y definir como van a reaccionar ante ellos. Por ejemplo, cuando el modelo notifica de que ha descargado nuevos datos de ThingSpeak, el controlador ordena a la vista que actualice sus gráficas.

5.2. Implementación del modelo

La clase más general del modelo es `ModelFarm`. Sus funciones son las que siguen: tener una mínima información sobre la granja, como nombre y descripción; almacenar un conjunto de instancias de la clase `ModelTurbine`, que en teoría son todas las turbinas de la granja que se están representando con este Gemelo Digital; y gestionar los datos de la potencia total generada por las turbinas activas de la granja en cada momento.

Las instancias de turbinas que almacena `ModelFarm` son de la clase `ModelTurbine` u otra que herede de ella. Sus funciones son acceder a ThingSpeak y, de manera secundaria, acceder a un servidor meteorológico.

Clase **ModelFarm**

Al inicio, declara unas constantes que homogeneizan las dimensiones temporales de la información almacenada en todo el modelo:

NumSeconds Se refiere el número de segundos de los que se descarga información de ThingSpeak en cada actualización. Cada dato allí está asociado a un segundo concreto, por lo que en realidad representa un número de puntos y de entradas en los conjuntos de marcas temporales. Afecta tanto al total de potencias al que nos referimos antes, como a las gráficas particulares de cada turbina. Evita mostrar información de diferentes periodos de tiempo mezclada, lo que puede confundir al usuario. Permite editarlo con facilidad dependiendo de las necesidades puntuales.

SecuritySeconds Marca el retraso de la información almacenada en el modelo, lo que da un margen que asegura que los datos de las turbinas que se le soliciten a ThingSpeak estén ya disponibles.

Además de las propiedades con la información descrita anteriormente, cuenta con el evento **StateUpdated**. Este sirve para avisar de que **ModelFarm** ha actualizado su estado, que en este caso se corresponde con los datos que representan el total de potencia generada en la granja.

ModelFarm(turbines) Establece el nombre y la descripción de la granja y guarda el conjunto de turbinas, que recibe como parámetro.

updateState(dateRange) Actualiza el total de potencia generada en la granja. Requiere del objeto de MATLAB *timetable* [31] y del método **retime** para rellenar con ceros las entradas que falten de los conjuntos de datos de potencia de las turbinas. Al terminar, la función dispara una notificación mediante el evento antes mencionado.

Clase **ModelTurbine**

Como puede observarse en la *Figura 5.2*, **ModelTurbine** tiene una estructura similar a **ModelFarm**, pero con algunas ampliaciones. Por ejemplo, junto con el nombre y la descripción, la turbina también guarda coordenadas de la latitud y la longitud de la ubicación en la que se encuentra.

Además del evento **StateUpdated**, cuenta con los eventos **WeatherUpdated**, **LastCommandsUpdated** y **CommandsChanged**. Análogamente, aparte de la constructora y del método **updateState**, cuenta con los métodos **updateWeather**, **updateLastCommands** y **changeCommands**. Sus implementaciones concretas se ven en los dos siguientes apartados, dependiendo de la funcionalidad que aporten.

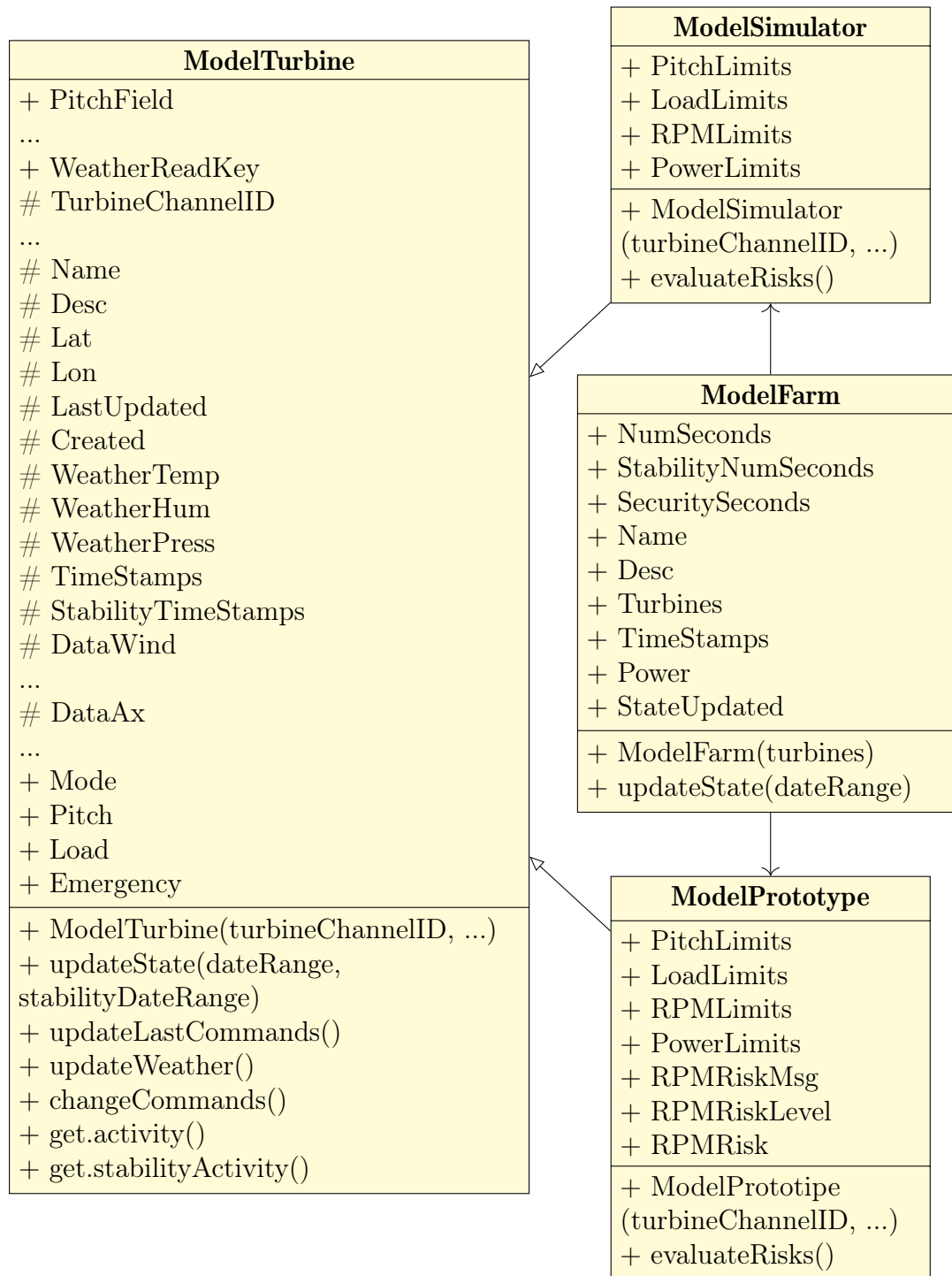


Figura 5.2: Clases del modelo del Gemelo Digital

5.2.1. Accesos a ThingSpeak

El modelo es el responsable de obtener y almacenar tanto la información general como un histórico de datos de cada turbina, así como de enviar comandos puntuales. Concretamente, la clase `ModelTurbine` es la única desde donde se accede a ThingSpeak.

Cada instancia de esta clase se inicializa con los identificadores y las claves de los canales de ThingSpeak a los que está asociada esa turbina como argumentos. Se ha establecido una estandarización para que haya una correspondencia clara entre el orden en que los datos están almacenados en ThingSpeak y en el que el Gemelo Digital espera obtenerlos.

El sistema está formado por 3 canales de comunicación que atienden a modularizar el sistema y a limitaciones de licencia, que pueden verse en la *Figura 5.3*. Por un lado, los canales de comunicación de estados *StateChannel* y comandos *CommandChannel* deben separarse por dirigirse en sentidos opuestos y ser asíncronos entre sí. Por otro lado, como la licencia gratuita de ThingSpeak limita a 8 campos por canal, para enviar la aceleración lineal y angular, que ocupan 3 ejes cada una, se debe habilitar un tercer canal específico *StabilityChannel*.

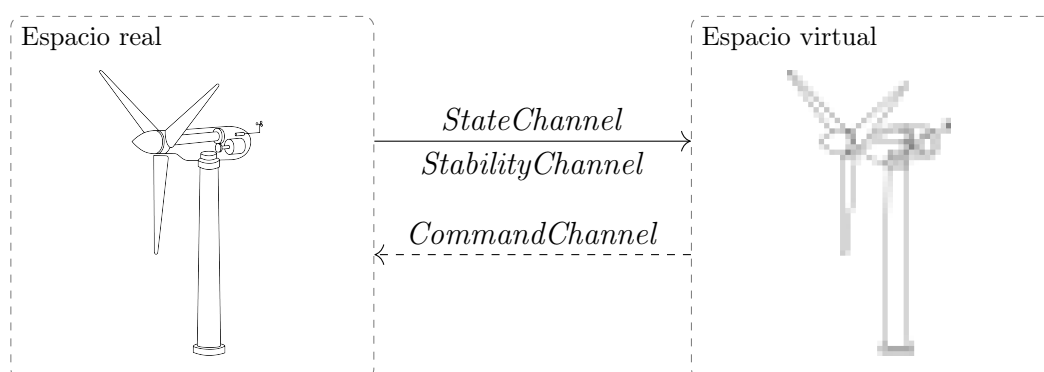


Figura 5.3: Canales de ThingSpeak del Gemelo Digital

De los canales *StateChannel* y *StabilityChannel* se descargan el estado y la inclinación de la turbina, respectivamente. Al canal *CommandChannel* se suben los comandos.

Además, cada vez que el usuario accede a una turbina concreta, puede ver en qué modo se quedó y cuáles fueron los últimos comandos enviados para partir de ellos al darle unos nuevos. Esta funcionalidad aumenta la consistencia y la sensación de libertad de la interfaz gráfica. Para que esta información prevalezca entre distintas ejecuciones de la aplicación, se descarga de *CommandChannel*.

La numeración de los campos queda indicada con propiedades constantes en `ModelTurbine`, lo que se recomienda editar si se hacen cambios en la configuración de los canales de ThingSpeak, ya que la interfaz podría mostrar datos equivocados.

MATLAB cuenta con funciones específicas para interactuar con un canal de ThingSpeak: `thingSpeakRead` [32] y `thingSpeakWrite` [33], para leer información almacenada y para escribir nueva, respectivamente. Deben indicarse como argumentos de entrada el número del canal y la clave.

En el caso de la lectura, puede configurarse la cantidad, los campos o el momento de procedencia de los datos que se descargan. El método `thingSpeakWrite` admite hasta 3 argumentos de salida.

data Matriz de datos numéricos de cada campo del canal con todas las entradas hechas en un fecha y hora dentro del intervalo temporal indicado.

timeStamps Conjunto de marcas temporales para cada entrada en **Data**. En el caso de que esté vacío se deduce que la turbina no está encendida, lo que se refleja en la propiedad dependiente² `Activity` de la clase `ModelTurbine`.

channelInfo Estructurado con diferentes informaciones sobre el canal. El creador de *StateChannel* debe haber puesto un nombre y escrito una descripción, con lo que considere oportuno, e indicado su localización geográfica para ubicarla en el mapa de la pantalla principal. Algunos de sus campos contienen información aportada por ThingSpeak, como la fecha de creación del canal.

En el caso de la escritura, pueden pasarse como argumentos de entrada tanto datos individuales, como conjuntos, como tablas de históricos con sus marcas temporales.

Los métodos de `ModelTurbine` dedicados a acceder a ThingSpeak son los siguientes:

`ModelTurbine(turbineChannelID,...)` Almacena en sus respectivas propiedades los identificadores y claves de los canales de ThingSpeak asignados a la turbina. Descarga de *StateChannel* información general de la turbina que no cambia durante la ejecución de la aplicación. En esta primera lectura sólo se queda con los campos del estructurado `channelInfo` *Name*, *Description*, *Latitude*, *Longitude* y *Created*.

²Propiedad que no almacena información. Su valor depende de otras propiedades.

`updateState(dateRange)` Se ejecuta periódicamente para cada instancia de una turbina. Es el encargado de descargar el histórico de datos, en el intervalo de tiempo `dateRange`. La información del estado y de la estabilidad de la turbina se disecciona por campos y se guarda en diferentes propiedades de la clase hasta que vuelve a ser actualizada. Se guardan también las marcas temporales. Se toma el campo *Updated* de `channelInfo`, que indica cuando se subieron datos por última vez. Al terminar, ejecuta el método que evalúa los riesgos de la turbina y dispara una notificación mediante el evento `StateUpdated`.

`updateLastCommands()` Descarga de *CommandChannel* el último valor de los campos `ModeCmd`, `PitchCmd` y `LoadCmd`. Al terminar, dispara una notificación mediante el evento `LastCommandsUpdated`.

`changeCommands()` Sube a *CommandChannel* todos los comandos que hay almacenados en el modelo en el momento de la llamada. No se envían uno a uno porque requiere el mismo tiempo y así se evita que se almacenen valores no numéricos en el resto de campos, dando errores de lectura en el prototipo. Al terminar, dispara una notificación mediante el evento `CommandsChanged`.

5.2.2. Accesos al servidor meteorológico

Es interesante que el Gemelo Digital almacene también factores externos que afectan a la entidad física, como es la meteorología. Esto puede ayudar al usuario a decidir qué comandos envía, se puede tener en cuenta para los algoritmos de control, o puede estudiarse para ver cómo afecta al deterioro de la turbina.

Las instancias de las turbinas ya cuentan con datos de geolocalización para mostrar su situación en un mapa, que ahora se aprovechan para obtener información meteorológica en ese mismo emplazamiento. En la implementación actual, el Gemelo Digital accede al servidor *OpenWeather* [34] y descarga datos en tiempo real de temperatura, humedad, presión y dirección del viento. Gracias a la estructura modular, esto se puede modificar o ampliar fácilmente.

La clave única para leer datos del servidor se almacena como constante en `ModelTurbine`. El método de `ModelTurbine` dedicado a acceder al servidor meteorológico es el siguiente:

`updateWeather()` Se ejecuta periódicamente para cada instancia de una turbina. Mediante la función `webread` y la latitud y la longitud descarga un JSON, del que extrae los datos y los normaliza. Al terminar, dispara una notificación mediante el evento `WeatherUpdated`.

5.2.3. Herencia

Para que el Gemelo Digital sirva para estudiar diferentes tipos de turbinas, se reutiliza el código de `ModelTurbine` y se crean clases más específicas, que únicamente contienen las diferencias que caracterizan a cada una de ellas.

En este proyecto se ha trabajado con un prototipo físico y con una simulación, por lo que se crea una nueva clase dedicada a cada uno de ellos, que heredan de `ModelTurbine`.

Para poder tener conjuntos de turbinas de diferentes tipos, `ModelTurbine` debe heredar de la clase abstracta de MATLAB `matlab.mixin.Heterogeneous` [35], que permite la formación de matrices heterogéneas.

Las clases `ModelPrototype` y `ModelSimulator` contienen, por un lado, constantes que indican los límites en los que trabajan sus variables de estado, lo que permite a la vista mostrar gráficas e indicadores personalizados para cada tipo de turbina.

Por otro lado, encapsulan la funcionalidad del indicador de riesgos del Gemelo Digital, ya que se trata de un aspecto fuertemente ligado a cada turbina. Se propone a modo de ejemplo el riesgo producido por unas revoluciones por minuto demasiado elevadas en el prototipo, implementado en el método `evaluateRisks()` de `ModelPrototype`.

5.3. Implementación de la vista

Se ha visto en secciones precedentes cómo App Designer ayuda a desarrollar interfaces de usuario en MATLAB, en código orientado a objetos, y que el patrón de diseño más conveniente para implementar un Gemelo Digital es el Modelo-Vista-Controlador, por su capacidad de ampliación y fácil mantenimiento.

En esta parte se evidencia el carácter incremental del desarrollo de este Gemelo Digital.

Versión alfa

La primera versión del Gemelo Digital de una turbina eólica puede verse en la *Figura 5.4*. Únicamente muestra los últimos puntos de información almacenados en ThingSpeak sobre la aceleración de la IMU, que se descargan cada vez que se pulsa el botón *Actualizar*.

Sus componentes gráficos son la ventana principal, una gráfica y un botón.

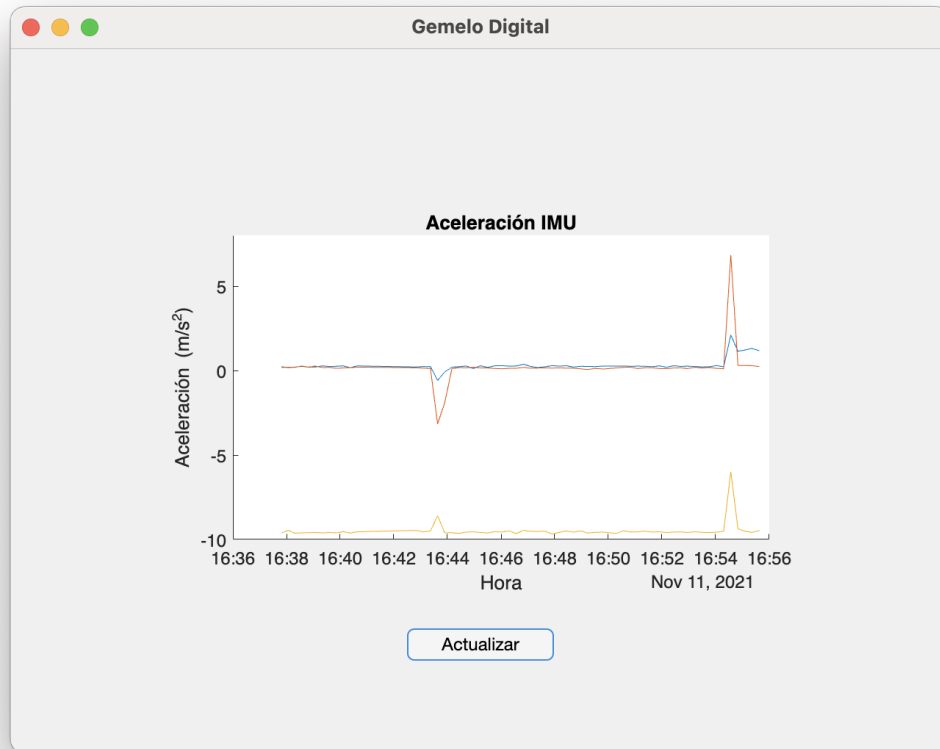


Figura 5.4: Versión alfa del Gemelo Digital

El código generado en App Designer al diseñar esta vista se caracteriza por lo siguiente:

- Hereda de `matlab.apps.AppBase`.
- Las componentes visuales están declaradas como propiedades de acceso privado. La más importante es `UIFigure`, que contiene a las demás.
- Tiene el método privado `createComponents`, que modifica propiedades de cada variable de una componente visual para configurarla adecuadamente, por ejemplo, dándole una posición o un tipo de letra. Al final, hace `UIFigure` visible.
- Tiene el método privado `startupFcn`, que se ejecuta antes de recibir ninguna interacción por parte del usuario.

- Tiene el método público `delete`, que permite añadir código que se ejecuta antes de que se borre la aplicación. Al final, borra `UIFigure` y la GUI desaparece.
- La constructora crea los componentes ejecutando `createComponents`. Registra la aplicación, con el método `registerApp` y `UIFigure` como argumento, y ejecuta `startupFcn` con el método `runStartupFcn`.

Se presentan a continuación los cambios llevados a cabo en la clase `View`, generada por App Designer para la vista de la *Figura 5.4*, al incorporarla al Gemelo Digital:

- De acuerdo con el patrón Modelo-Vista-Controlador, se añaden instancias del modelo y del controlador a la vista.
- Se modifica la constructora de `View` añadiendo el modelo de la granja como parámetro y asignándolo a la propiedad correspondiente.
- Se inicializa la instancia del controlador en `startupFcn`, asignándolo también a la propiedad correspondiente. Se pasa la instancia de la propia vista como argumento.
- Se vincula el controlador a la vista. Mediante el método `addListeners`, se añaden oyentes a los eventos provocados por el usuario en los objetos gráficos de la GUI que invocan a los métodos correspondientes del controlador, de acuerdo con el patrón Modelo-Vista-Controlador.
- Se implementa el método `updateState`, que actualiza la vista tomando información del modelo. Este método se llama desde el controlador, que en este caso es quien gestiona los eventos del modelo y se encarga de actualizar la vista, puntual y periódicamente.
- En el método `delete` se añade una llamada al controlador para que detenga a su vez los componentes que tiene en ejecución, previa a borrar `UIFigure`, la ventana principal.

Fruto de esta fusión ha sido posible el diseño de versiones del Gemelo Digital más complejas.

Versión modular

En el siguiente sprint se añaden datos de la turbina como nombre y descripción, un mapa con su localización e información sobre su estado a través de diferentes gráficas. Además, se ha dejado espacio para la futura implementación de una lista de diferentes turbinas, que conformarían la granja eólica. En la *Figura 5.5* puede observarse la modularidad el diseño.

Sus componentes gráficos quedan organizados de la manera que sigue. De nuevo, cuenta con una ventana principal, de tipo `matlab.ui.Figure`. Dentro se pone una cuadrícula, de tipo `matlab.ui.container.GridLayout`, permitiendo organizar al resto de componentes de manera más sencilla, sin necesidad de especificar sus posiciones exactas. Cada hueco de la cuadrícula se rellena con un panel, de tipo `matlab.ui.container.Panel`, dedicado a encapsular un apartado del gemelo. Algunos paneles pueden contener a su vez otra cuadrícula para organizar sus componentes.

Se han utilizado gráficas, etiquetas de texto, un mapa y una lista.

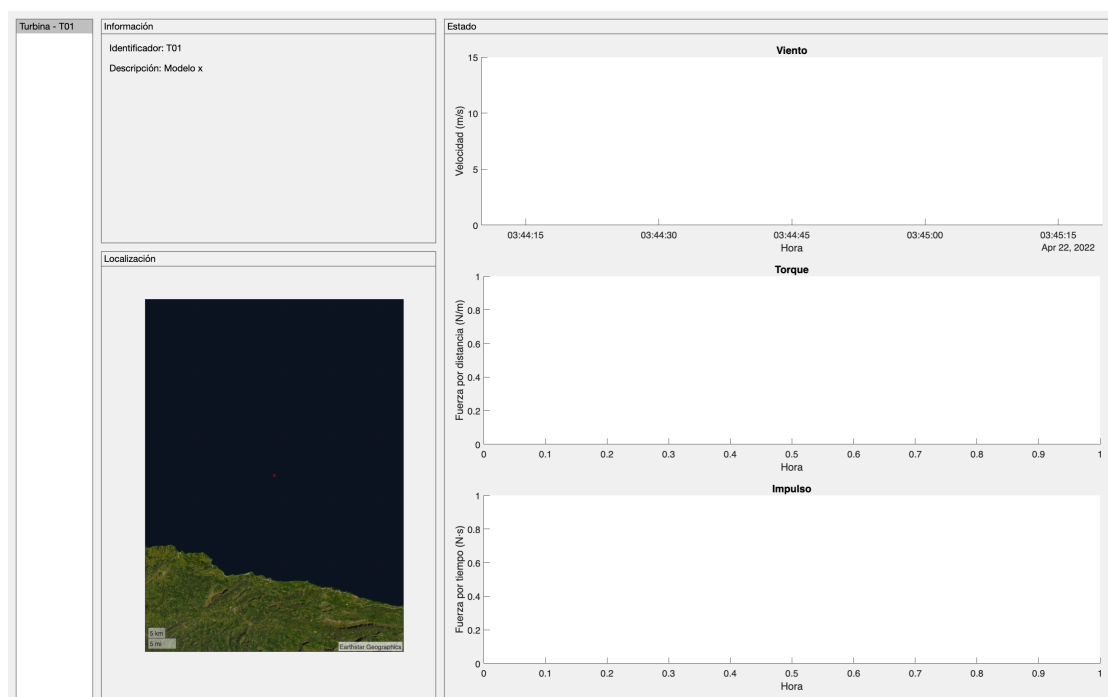


Figura 5.5: Versión modular del Gemelo Digital

Versión modo oscuro

En versiones posteriores se opta por una interfaz en modo oscuro. Es beneficioso para la vista de las personas que vayan a pasar varias horas seguidas monitorizando una turbina eólica en el Gemelo Digital, ya que reduce el esfuerzo ocular respecto al que se hace al mirar un fondo blanco. Además, aporta un aspecto más profesional.

Véase el resultado en la *Figura 5.6*.

Se sigue evolucionando el diseño atendiendo a las necesidades manifestadas por los clientes durante las reuniones. Se incorpora un dibujo de la turbina, un panel dedicado a los comandos que puede enviar el usuario y paneles para albergar avisos, dando la posibilidad de implementar una infinitud de algoritmos que interpreten y se anticipen a sucesos en la entidad física, por ejemplo, utilizando inteligencia artificial.

Se han añadido un campo de texto, que solo permite ángulos entre 0 y 90, un botón, otra lista, indicadores de colores y medidores.

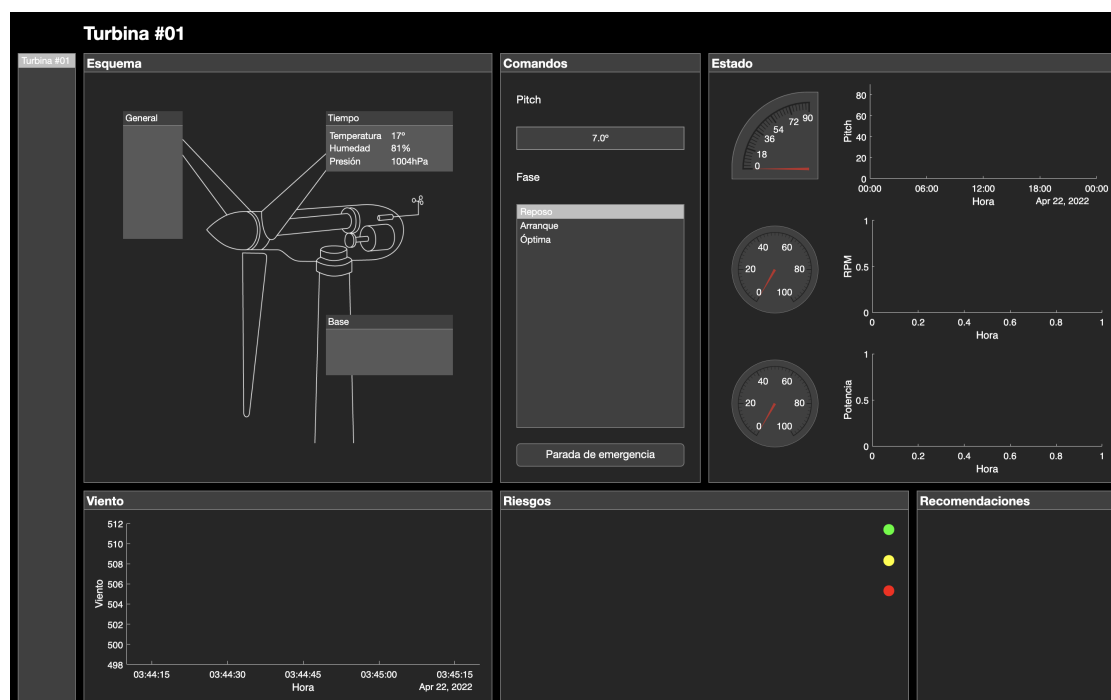


Figura 5.6: Versión modo oscuro del Gemelo Digital

Versión granja

El siguiente sprint consiste en generalizar el Gemelo Digital para que pase de reflejar una única turbina a una granja eólica.

En primer lugar, se diseña la vista general, que aparecerá al iniciar la aplicación. Consta de un mapa en el que se señala la posición de cada turbina, un panel informativo y la gráfica con la potencia total generada en la granja en cada momento. Véase la *Figura 5.7*.

Para la vista de la turbina se continúa con los diseños previos, que siguen evolucionando. Véase la *Figura 5.8*.

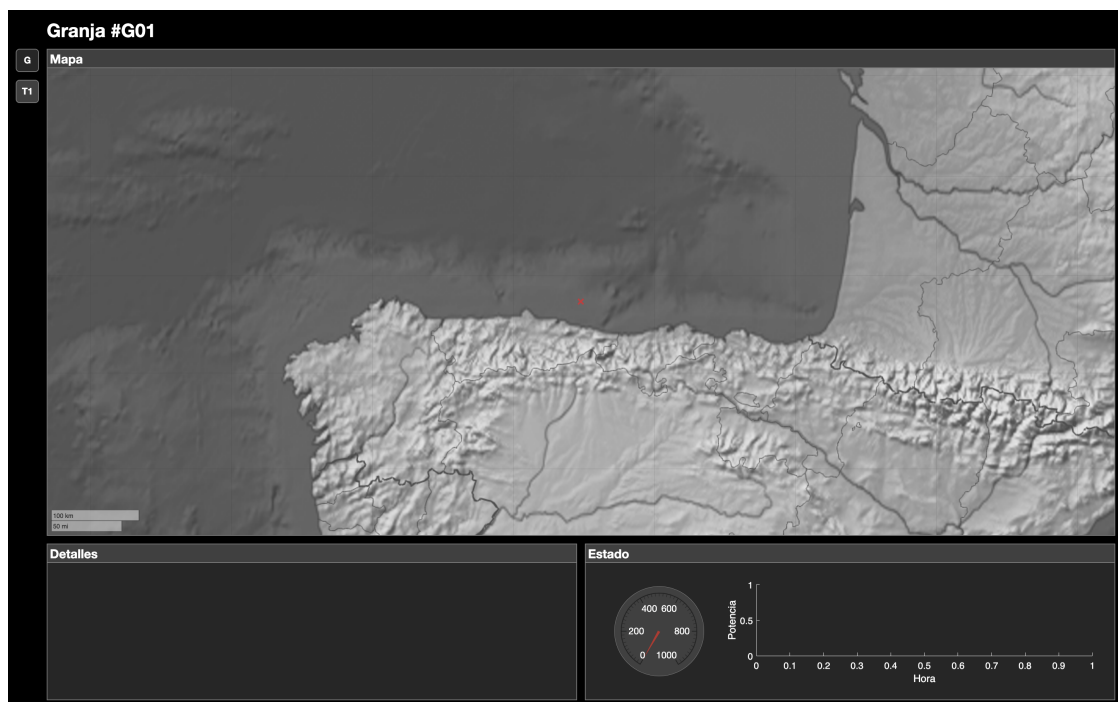


Figura 5.7: Versión granja del Gemelo Digital: vista principal

En la parte izquierda de la interfaz están los botones que el usuario puede presionar para acceder a una de las turbinas o volver a la granja. Se implemente un diseño conformado por subvistas. La ventana principal no se cierra al cambiar de unas vistas a otras, sino que solo varía la visibilidad de parte de los componentes existentes en la interfaz. Para facilitar el cambio entre granja y turbinas, se encapsulan cada uno de sus componentes en una misma cuadrícula, la cual cuenta con la propiedad `Visible`, que permite ocultarla junto con todo lo que contiene.

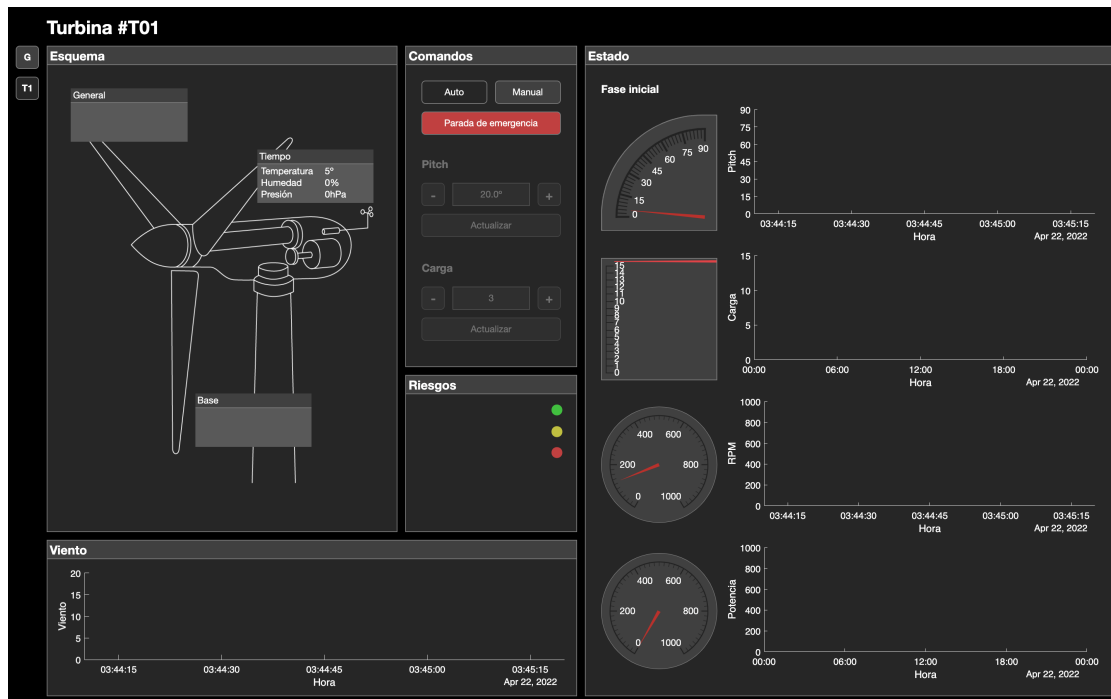


Figura 5.8: Versión granja del Gemelo Digital: vista de una turbina

Esta nueva implementación requiere de los siguientes cambios:

- La clase `View` pasa a llamarse `ViewFarm`.
- Se crea una nueva clase, `ViewTurbine`, que no hereda de `matlab.apps.AppBase`, ya que no es una nueva vista como tal sino una encapsulación de los componentes visuales de una turbina.
- En `ViewFarm` se añade la implementación de los botones para cambiar de vista: tanto propiedades como oyentes que avisan al controlador.
- En ambas clases se define la cuadrícula `ComponentsGridLayout`.
- Las propiedades y métodos específicos de la turbina se trasladan de `ViewFarm` a `ViewTurbine`. Para ello es necesario definir los métodos `createComponents` y `addListeners` también en `ViewTurbine`.
- Se define la propiedad `ControlObj` en `ViewTurbine`, en la que se guarda una instancia del controlador para vistas de turbinas, inicializado en la constructora.

- Se añade a la constructora de **ViewTurbine** como parámetro una cuadrícula en la que va a inicializar sus componentes, y se asigna a la propiedad correspondiente.
- Se añade a la constructora de **ViewTurbine** como parámetro la instancia del modelo correspondiente a la turbina a la que está asociada, y se asigna también a la propiedad correspondiente.
- En **startupFcn**, de **ViewFarm**, se inicializa una instancia de **ViewTurbine** para cada elemento del conjunto de turbinas almacenado en el modelo. Se les pasa como argumentos la cuadrícula principal contenida directamente en **UIFigure** y la instancia de la turbina que les corresponda. Se guardan en un conjunto de vistas que **ViewFarm** almacena en su nueva propiedad, **ViewTurbineObjs**. Además, **ViewFarm** le pasa esta propiedad a su controlador como argumento al inicializarlo.
- Se crean los métodos **initView** y **endView** en **ViewTurbine**, que informan a su controlador y establecen la visibilidad de la cuadrícula con sus componentes, cuando el usuario abra a cierre la vista de esa turbina, respectivamente.
- Se crea el método **delete** en **ViewTurbine**, que no borra ninguna ventana pero sí informa a su controlador de que el usuario ha cerrado la aplicación. Se añaden sus llamadas al método **delete** de **ViewFarm** para cada turbina.

Puede verse el resultado de la implementación de las clases de la vista en la *Figura 5.9*.

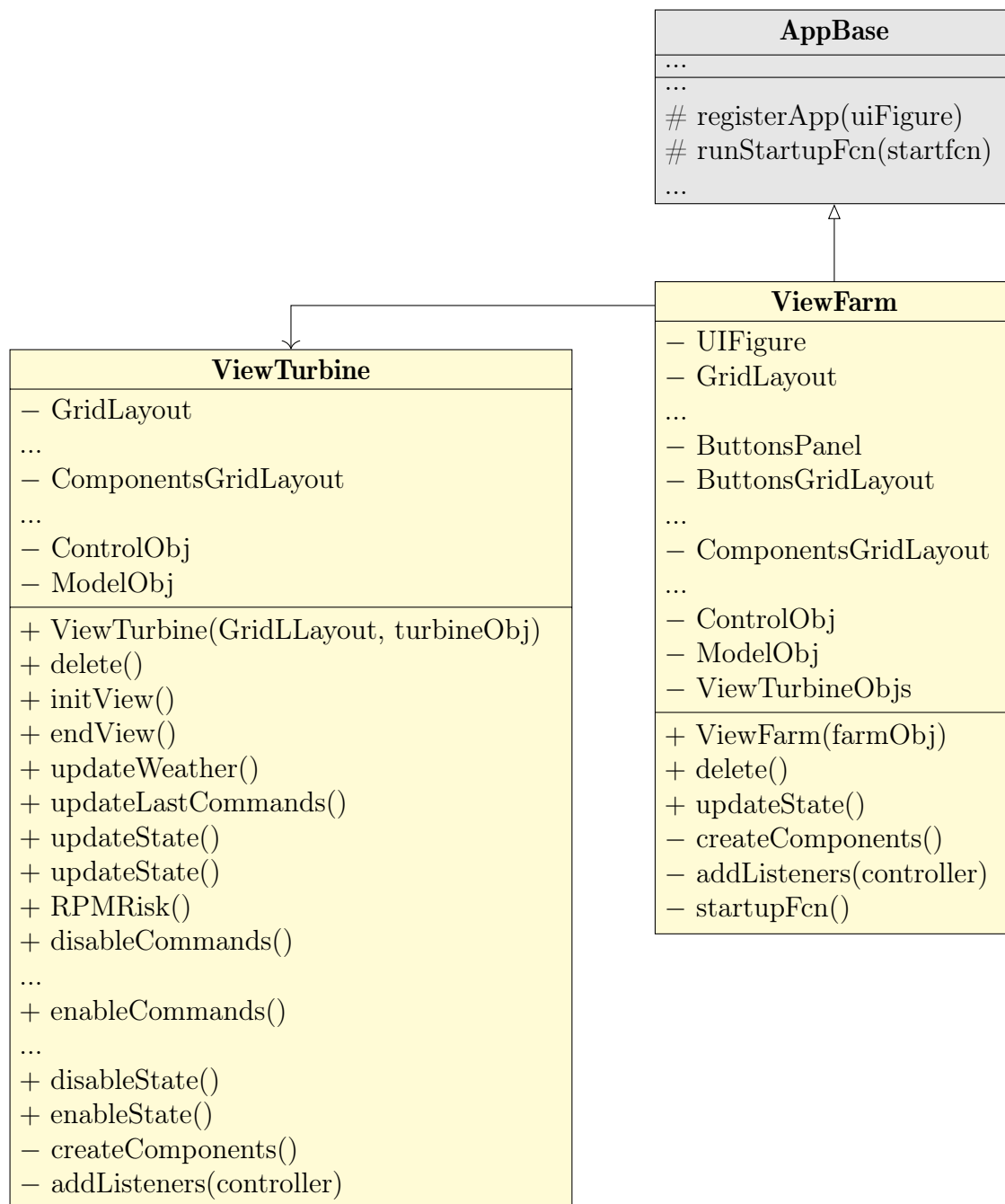


Figura 5.9: Clases de la vista del Gemelo Digital

Versión final

En siguientes *sprints* se afina el diseño y se añaden paneles requeridos por los clientes, como el de la estabilidad, e información sobre el estado de la mar.

A continuación, se justifican las decisiones de diseño tomadas durante todo el desarrollo, que afectan sobre todo a las turbinas.

5.3.1. Diseño del sistema interactivo

El espacio de la interfaz de la turbina está dividido en marcos. Su disposición se reparte en una cuadrícula de dos filas y tres columnas visibles. La primera columna muestra información externa al sistema. La segunda contiene un marco de comandos y otro de recomendaciones. La última representa el estado de la turbina.

Como puede observarse en las *Figuras 5.10 y 5.11*, se han aplicado los siguientes principios de diseño:

Proximidad

La selección de turbinas se encuentra en el margen izquierdo de la interfaz.

Las variables externas se agrupan en la primera columna. Entre otras, se muestran la información meteorológica o el oleaje.

Los elementos relacionados con el control se encuentran en el cuadro de comandos. Aquí se agrupan la selección de modo, del ángulo de *pitch*, de la carga eléctrica aplicada y la parada de emergencia. Además, un marco inferior sugiere órdenes de acción en función del estado del sistema.

Los elementos relacionados con el muestreo se encuentran en el cuadro de estado. Aquí se agrupan las salidas analógicas del sistema: *pitch*, carga eléctrica, revoluciones por minuto y potencia extraída.

Cierre

El proceso de acción y reacción sigue un orden natural.

Las entradas, correspondientes a los comandos, se encuentran en la segunda columna. Las salidas, correspondientes al estado de la turbina, se encuentra en la tercera columna. Así, una vez enviadas las órdenes se puede observar su repercusión en la turbina más a la derecha.

Ley de Fitts

La actualización de los comandos de control puede realizarse ágilmente gracias a la proximidad de sus campos de ajuste.

Consistencia interna

Los tamaños de los cuadros de texto establecen una jerarquía al sistema. El título principal a 36 puntos, el título de los marcos a 24, información importante a 16 e información genérica a 12.

Todos los botones del sistema poseen una apariencia similar, presionados son concretamente un 10 % más oscuros que sin presionar.

La paleta de colores es consistente entre componentes. Por ejemplo, todos los rojos del sistema son el #BF4040.

Las gráficas de estado muestran un mismo periodo de tiempo en toda la interfaz.

Consistencia externa

La modularidad de la interfaz, los medidores y el botón de parada de emergencia pretenden imitar un panel de control físico convencional.

Cuando no se detecta actividad en la turbina, se indica con una etiqueta de texto de color rojo, lo que se relaciona con el estado apagado.

Visibilidad y feedback

Los botones de modo son de estado, es decir, se mantienen presionados para indicar la opción en la que se encuentran. De forma análoga funciona el botón de parada de emergencia, indicando si su funcionamiento se encuentra interrumpido.

Los valores de *pitch* y carga eléctrica admisibles dependen del tipo de la turbina y no se permite la escritura de comandos fuera esos límites.

Gestión del estado visible

Es sencillo ubicarse dentro de la aplicación observando qué botón del panel selector de granja o turbina, situado en el margen izquierdo, se encuentra presionado.

Tras el envío de un comando se deshabilitan todos los botones y cuadros de escritura del marco de comandos durante unos segundos. Esto impide que el usuario incurra en escrituras en intervalos de tiempo inferiores al de la limitación de ThingSpeak.

Los medidores del marco de estado también permanecen deshabilitados cuando la turbina no presenta actividad.

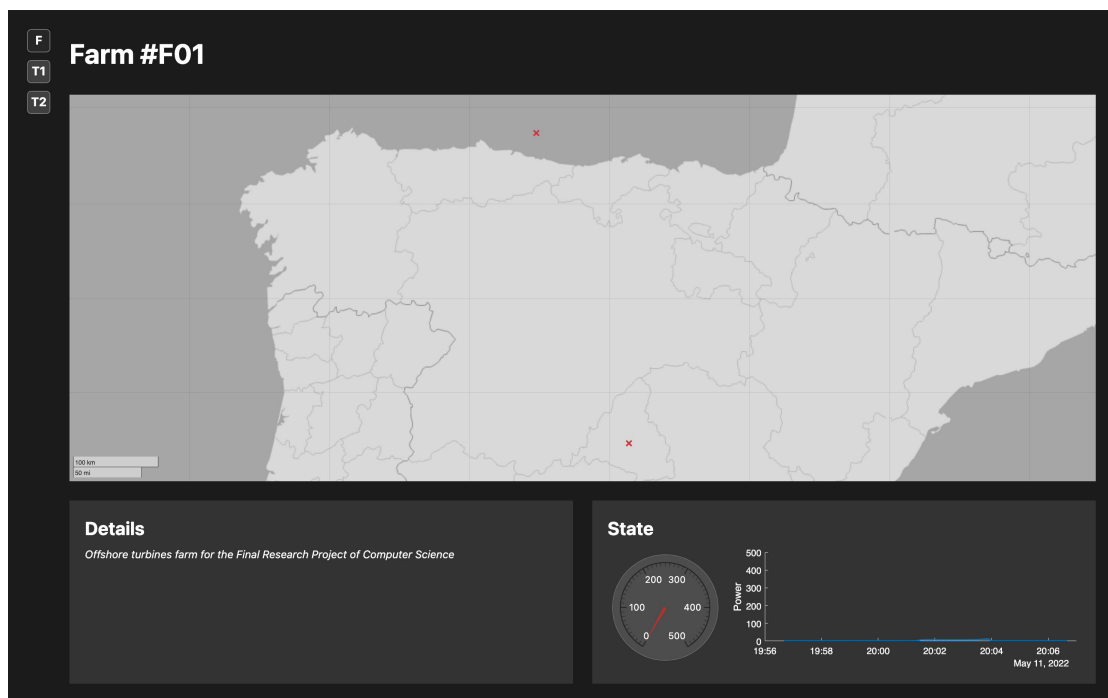


Figura 5.10: Versión final del Gemelo Digital: vista principal

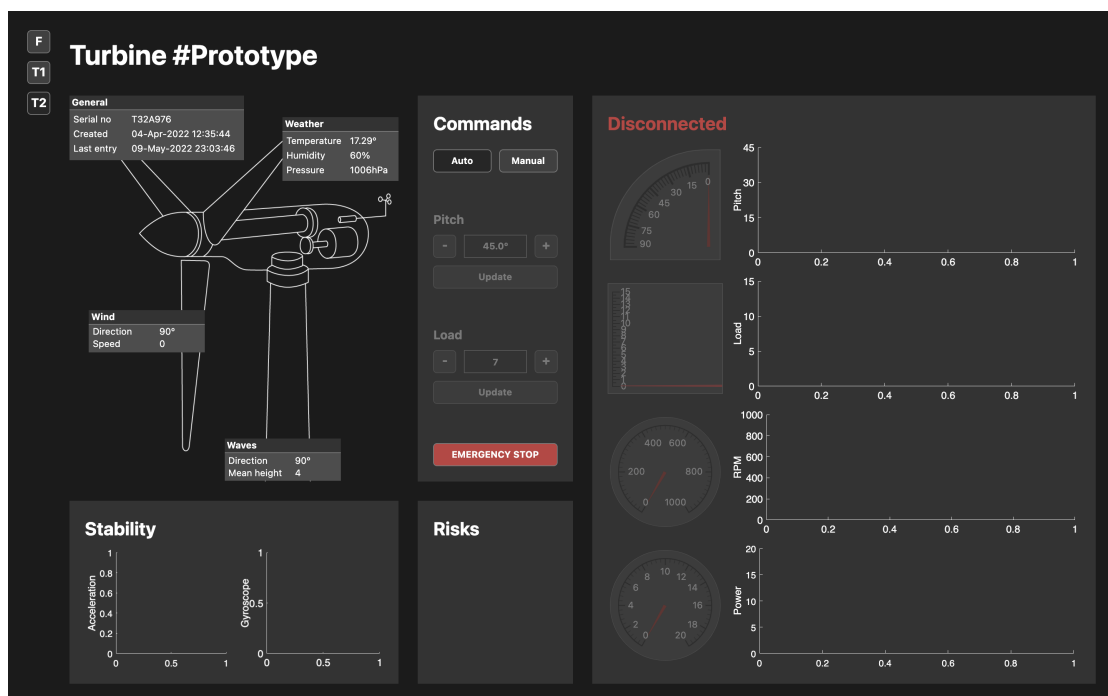


Figura 5.11: Versión final del Gemelo Digital: vista de una turbina

5.4. Implementación del controlador

Cada vista inicializa su propio controlador cuando es creada. Su función principal es recibir y gestionar las interacciones en la GUI por parte del usuario, lo que en la mayoría de los casos requiere de llamadas al modelo. Alberga también los oyentes de los eventos del modelo, que repercuten por lo general en invocaciones directas a métodos de las vistas, excepto en el caso de las actualizaciones de la potencia total generada en la granja, para las que su controlador debe asegurarse previamente de que todas las turbinas han descargado ya nuevos datos de ThingSpeak.

Adicionalmente, debido a que el Gemelo Digital debe mostrar información en tiempo real, se encarga de mantener la GUI actualizada ordenando al modelo descargar nuevos datos de ThingSpeak periódicamente, sin necesidad de una petición previa por parte de la vista.

5.4.1. Temporizadores

El software de MATLAB incluye un objeto temporizador [36] que se puede usar para programar la ejecución de comandos o funciones.

Para crearlos se utiliza la función `timer`, que consta de varias propiedades que establecen sus comportamientos. Es obligatorio definir en `TimerFcn` lo que se ejecuta al dispararse el temporizador, que en este caso van a ser funciones. Sus dos primeros argumentos son siempre una referencia al temporizador y un estructurado del evento, pero pueden ignorarse. También pueden definirse funciones que se ejecutan al iniciarse, si ocurre un error o cuando se para: `TimerFcn`, `ErrorFcn` y `StoptFcn`, respectivamente.

Existen varios tipos de temporizadores según su modo de ejecución, que se elige en la propiedad `ExecutionMode`. Por defecto está seleccionado *singleShot*, con el que `TimerFcn` se ejecuta una única vez. En ese caso suele indicarse el tiempo que tiene que pasar entre que se inicia el temporizador y se dispara, en `StartDelay`. En la implementación del Gemelo Digital se ha utilizado también el modo *fixedRate* que permite una ejecución periódica de `TimerFcn`, cada vez que pasa el tiempo establecido en `Period`, y hasta que se para al temporizador o durante las especificadas en `TasksToExecute`.

El tiempo especificado y el real pueden variar, ya que los temporizadores funcionan en el entorno de ejecución de un solo proceso de MATLAB, y puede haber otras tareas en la cola. Sin embargo, esto no afecta negativamente por el régimen de funcionamiento de las turbinas eólicas, ya que los segundos de retraso son despreciables

Una vez creado el objeto de temporizador, se inicia con la función `start`. Para pararlo y que deje de dispararse, se usa la función `stop`. Antes de que se cierre la aplicación deben eliminarse de la memoria, con la función `delete` y previamente parados.

En la implementación del Gemelo Digital se han utilizado los siguientes temporizadores:

StateTimer Actualiza la información del estado de todas las turbinas, así como del total de potencia de la granja, periódicamente. Está establecido que se ejecute cada 20 segundos, periodo con el que se está subiendo información a ThingSpeak desde simuladores y prototipos.

WeatherTimer Actualiza la información meteorológica de cada turbina, periódicamente. Está establecido que se ejecute cada 10 minutos, ya que no suele haber grandes variaciones en ese tiempo en la climatología.

CommandsTimer Impide al usuario que mande varios comandos a la turbina en menos segundos de los que permite ThingSpeak. Cuando se envía un comando, se deshabilita la escritura en el panel de comandos y se inicia el temporizador que, tras el tiempo establecido, en este caso 15 segundos, la vuelve a habilitar y no vuelve a dispararse. Así se asegura que al menos ha pasado el tiempo mínimo para que ThingSpeak no devuelva un error.

Implementación en clases separadas

Según el patrón Modelo-Vista-Controlador, en una aplicación que tiene varias vistas distintas puede ser necesario implementar varios controladores. En las primeras fases del desarrollo sólo había uno, que respondía a la interacciones en la granja o en la turbina que se estuviera mostrando, lo que se indicaba en la propiedad `ActiveView`, y que contenía instancias de cada tipo de temporizador.

El temporizador **CommandsTimer** estaba asociado a la vista activa y se paraba al cerrarse esta, en caso de estar iniciado. Un problema que surge es que el usuario puede enviar un comando a una turbina, cambiar a otra vista, volver a la turbina anterior y enviar otro comando en menos de 15 segundos, haciendo que ThingSpeak devuelva un error de escritura.

Dicho problema podría atajarse impidiendo al usuario cambiar de vista hasta que **CommandsTimer** se dispare o con un conjunto de instancias de temporizadores.

Sin embargo, se ha optado por separar las implementaciones del controlador por vistas, que además favorece la modularidad y claridad del código.

Así cada turbina cuenta con su propio controlador y, concretamente, con su temporizador `CommandsTimer`, que no se tiene que parar al cambiar de una vista a otra. Así se aporta fluidez a la aplicación y sensación de libertad al usuario, que puede navegar entre las diferentes interfaces enviando comandos sin peligro de provocar errores de escritura en ThingSpeak, ya que no se le permitirá.

El temporizador `WeatherTimer` se cambia también de controlador, para acceder al servidor meteorológico únicamente para las turbinas que se estén observando en detalle. En cambio, `StateTimer` se mantiene en el controlador de la granja, ya que para poder mostrar la potencia total generada es necesario actualizar el estado de todas las turbinas.

Se expone a continuación la implementación particular de cada controlador.

Clase `ControllerFarm`

Al inicio, declara una constante que establece la frecuencia de actualización del temporizador `StateTimer`. Cuenta con instancias de la vista de la granja, del conjunto de vistas de las turbinas y del modelo, y con un temporizador. Instancias adicionales almacenan en cada momento los rangos temporales en los que se descarga información de ThingSpeak, el número de turbinas que ya han actualizado su estado y la vista activa.

`ControllerFarm(viewObj, viewTurbineObjs, farmObj)` Guarda en las propiedades correspondientes las instancias de las vistas y el modelo, establece que se está mostrando la vista de la granja, inicializando la propiedad que indica la vista activa a 0, llama a los métodos que `addListener(viewFarm)` e `initializeTimers()` y, por último, inicia el temporizador `StateTimer`.

`deleteTimers()` Detiene y elimina el temporizador `StateTimer`.

`callback_changeView(val)` Se ejecuta cuando el usuario pulsa uno de los botones que sirven para cambiar de vista. Tiene como parámetro la vista que quiere visualizar. En el caso de que no sea la que está visible en ese momento, procede a hacer que se cambie una por otra. Si estaba activa la vista de una turbina, llama a su método `endView`. Después, actualiza la propiedad que indica la vista activa. De nuevo, si la nueva vista es una turbina, llama a su método `initView`.

addListener(viewFarm) Añade oyentes del evento **StateUpdated** de la instancia del modelo de cada turbina y de la granja. Llamam al método **updateState** de esta clase y al de la vista de la granja, respectivamente.

initializeTimers() Inicializa el temporizador **StateTimer** con las propiedades descritas en 5.4.1.

updateTurbineState() Es el método que se ejecuta cada 20 segundos por el temporizador **StateTimer**. A partir de la fecha y hora del momento actual y las constantes **numSeconds** y **securitySeconds**, almacenadas en el modelo, calcula los rangos temporales de los que se van obtener nuevos datos para la siguiente actualización del estado de las turbinas. A continuación, reinicia el contador de instancias de turbinas del modelo ya actualizadas a 0 y ordena a cada una de ellas que actualice su estado, llamando a sus respectivos métodos **updateState** indicándoles los rangos calculados.

updateState(turbineNum) Se ejecuta cada vez que una turbina del modelo notifica que ha actualizado su estado. Llama al método **updateState** de la vista correspondiente, contabiliza la actualización sumando uno al número de turbinas actualizadas y, en caso de que esa propiedad sea ya igual al número total de turbinas de la granja, ordena al modelo que se actualice también, llamando a su método **updateState**.

Clase **ControllerTurbine**

Al inicio, declara unas constantes que establecen la frecuencia de actualización del temporizador **WeatherTimer** y el tiempo para el disparo de **CommandsTimer**. Cuenta con instancias de la vista y el modelo de la turbina y con dos temporizadores.

ControllerTurbine(viewObj, turbineObj) Guarda en las propiedades correspondientes las instancias de la vista y el modelo y llama a los métodos **addListener(viewTurbine)** e **initializeTimers()**

deleteTimers() Detiene y elimina los temporizadores **WeatherTimer** y **CommandsTimer**.

initTurbineComponents() Se ejecuta cuando el usuario quiere acceder a la vista de la turbina. Inicia el temporizador **WeatherTimer**. Llama al método del modelo **updateLastCommands()** para que cuando se inicie se vean los últimos comandos enviados y habilita o no la escritura del *pitch* y de la carga eléctrica, en función del modo en que se haya quedado la turbina.

`endTurbineComponents()` Se ejecuta cuando el usuario quiere cambiar a otra vista que no es la asociada a este controlador. Detiene el temporizador `WeatherTimer`.

`callback_modeCommandButton(val)` Se ejecuta cuando el usuario interactúa con la vista para enviar el comando a la turbina de cambio de modo. Recibe como parámetro el nuevo valor que el usuario ha indicado, directa o indirectamente. Deshabilita la escritura de nuevos comandos en la vista, llamando al método `disableCommands()`, asigna a la propiedad que corresponda del modelo el nuevo valor y, por último, le ordena que suba a ThingSpeak el comando, llamando al método `changeCommands()`.

`callback_pitchCommandButton(val)` Análogo al anterior para el comando de nuevo valor de *pitch*.

`callback_loadCommandButton(val)` Análogo al anterior para el comando de nuevo valor de carga eléctrica.

`callback_emergencyCommandButton(val)` Análogo al anterior para el comando botón de emergencia.

`addListener(viewTurbine)` Añade oyentes de los eventos `WeatherUpdated`, `LastCommandsUpdated`, `CommandsChanged` y `RPMRisk`, en caso de contar con él, del modelo de la turbina. Llamam a los métodos de actualización correspondientes de la vista, excepto el de `LastCommandsUpdated`, que llama al método de esta clase `delayTurbineCommands()` para iniciar la cuenta atrás que permite al usuario volver a enviar comandos.

`initializeTimers()` Inicializa los temporizadores `WeatherTimer` y `CommandsTimer` con las propiedades descritas en 5.4.1.

`updateWeather()` Es el método que se ejecuta cada 10 minutos por el temporizador `WeatherTimer`. Ordena a la turbina que actualice su estado meteorológico, llamando a su respectivo método `updateWeather()`.

`enableTurbineCommands()` Es el método que se ejecuta cuando se dispara el temporizador `CommandsTimer`. Ordena a la vista que habilite la escritura de comandos.

`delayTurbineCommands(obj)` Se ejecuta cuando el modelo notifica de que ya ha escrito en ThingSpeak el comando enviado por el usuario. Inicia el temporizador `CommandsTimer`.

5.5. Simulador

El Gemelo Digital puede mostrar información tanto de turbinas eólicas reales como simuladas. Durante su implementación, ha sido especialmente útil el poder trabajar con simulaciones. Para crearlas, se ha utilizado la herramienta de MATLAB, Simulink.

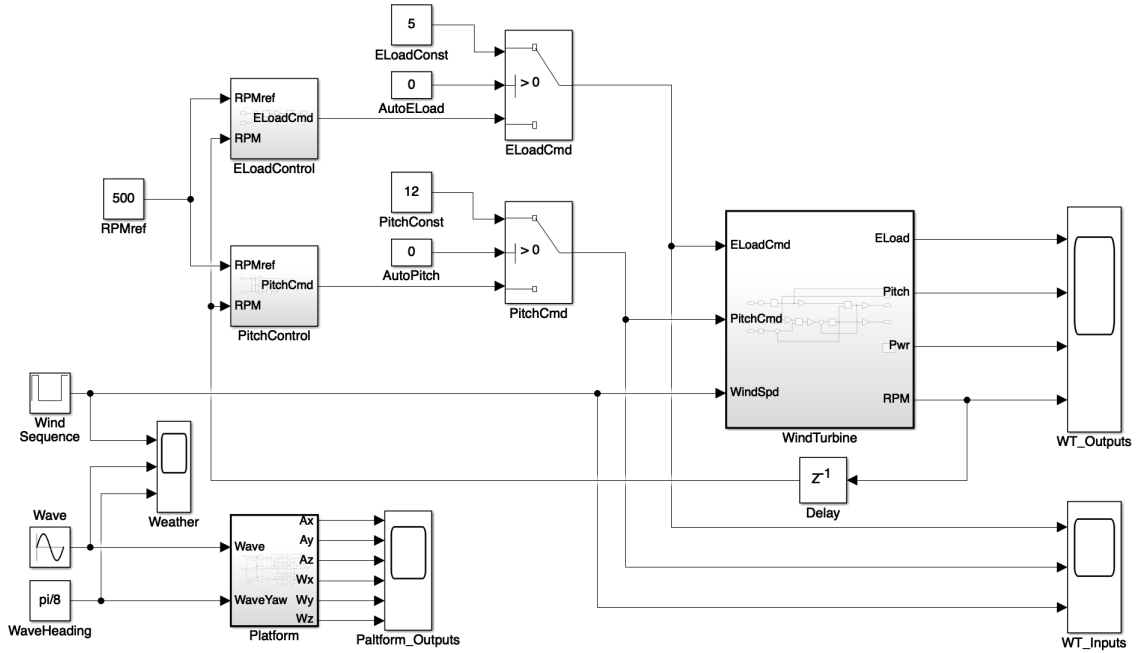


Figura 5.12: Simulador

Se propone un modelo de turbina *offshore* flotante que trata de asemejarse lo máximo posible al prototipo físico con el que se ha trabajado en este proyecto. Como puede verse en la Figura 5.12, consiste en un sistema en el que se simulan viento y olas, y en el que el subsistema principal es la turbina. Además, cuenta con dos modos de funcionamiento relacionados con el ajuste de *pitch* y carga eléctrica: el automático, en el que se aplican lazos de control, y el manual, con el que no varían pero que está pensado para recibir comandos externos.

El bloque *WindTurbine* recibe comandos de *pitch* y carga eléctrica y una velocidad del viento. Aplicando el modelo físico de turbina eólica visto en la Sección 2.1, devuelve las revoluciones por minuto y la potencia generada. Devuelve también los comandos de *pitch* y carga eléctrica según se van aplicando de manera escalonada. Véase la Figura 5.13.

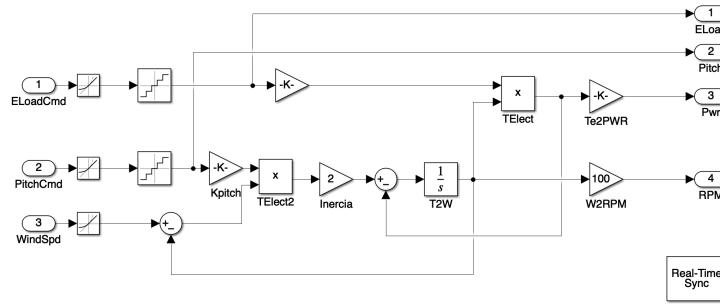


Figura 5.13: Simulador: turbina eólica

Los bloques *ELoadControl* y *PitchControl* son los encargados de los lazos de control de la turbina. Se aplica un controlador PI a la carga eléctrica y uno PID al *pitch*, los cuales se han explicado en la Subsección 2.2.2. Véase la Figura 5.14.

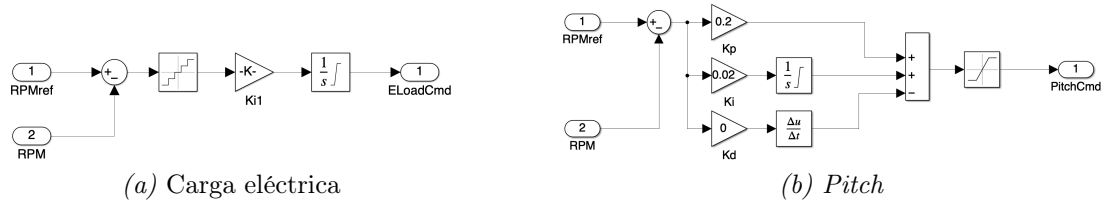


Figura 5.14: Simulador: control

El bloque *Platform* simula los datos que se obtendrían con una IMU colocada en la base de una turbina *offshore* flotante. Recibe la amplitud y dirección de la ola y devuelve valores de aceleración y aceleración angular, del giroscopio, en los 3 ejes. Véase la Figura 5.15.

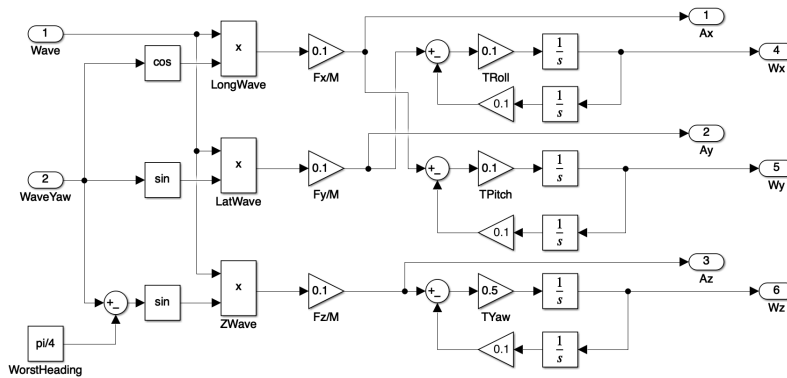


Figura 5.15: Simulador: plataforma

Para trabajar con el Gemelo Digital es necesaria una simulación en tiempo real que permita recibir y extraer información.

Se han desarrollado varios *Live Scripts*³, que permiten controlarla mediante código MATLAB. Con la función `load_system` se carga el sistema en memoria, para trabajar con él sin necesidad de abrir el editor Simulink, y con `set_param` se puede establecer el comando que inicia la simulación.

Sin embargo, la simulación resultante no cumple con los requisitos.

El primer problema encontrado fue que el sistema no se ejecutaba en tiempo real, lo cual impedía la inserción y extracción de variables durante la simulación. Es decir, una simulación de 15 segundos puede tardar 2 en llevarse a cabo y devolver los resultados finales. Además, no se puede predecir el tiempo que tardará cada nueva simulación.

La solución que se propone entonces es realizar simulaciones fragmentadas de 15 segundos. Para subir un conjunto de 15 datos a ThingSpeak, la simulación almacena en la variable de salida al Workspace de MATLAB listas de 15 elementos equidistanciados correspondientes a cada segundo de simulación. Además, como no hay restricciones en el número de lecturas en ThingSpeak, cada segundo se lee y actualiza el comando enviado a la simulación de la turbina.

El problema derivado consiste en que cada vez que se inicia un nuevo fragmento de simulación los valores de las variables internas se restauran y el gráfico global no sigue el flujo natural de un comportamiento continuo.

Una posible solución a este último problema pasaba por reservar el estado de la turbina en el último momento del fragmento de simulación anterior y configurarlo en el fragmento siguiente. Esta propuesta parecía debilitar el sistema y podría presentar problemas a largo plazo.

5.5.1. Desktop Real-Time

El componente adicional de Simulink, *Desktop Real-Time*, incluye un kernel en tiempo real que se ejecuta con la máxima prioridad en el sistema operativo del desarrollador. Requiere a su vez del componente *Simulink Coder*, que genera el código en C de la simulación que va a ejecutarse fuera el entorno de MATLAB.

De esta manera, la simulación está completamente sincronizada con el reloj en tiempo real. La función principal de Simulink queda restringida a leer y mostrar los resultados de la simulación devueltos por el ejecutable, a través de una interfaz de memoria compartida.

³Archivos de programa de MATLAB que contienen el código, la salida y el texto con formato en un solo entorno interactivo.

Cambios en el sistema de Simulink

Para incorporar estos dos nuevos componentes, simplemente se pone el bloque *Real-Time Synchronization* dentro de *WindTurbine* y se selecciona el modo *Run in Kernel*, en *Desktop Real-Time*. En Simulink se establece un tiempo de simulación infinito, ya que va a ser con código MATLAB como se ordene su inicio o su final.

Previamente establecido en los parámetros de un bloque que debe tratarse como una unidad atómica, se pueden usar las funciones `set_param` y `get_param` para establecer y obtener los valores de sus variables, con la simulación ejecutándose en tiempo real.

Nótese que, en los casos en los que una señal de salida de un sistema es también una señal de entrada, se pueden formar bucles algebraicos. Puede verse en la *Figura 5.12* que las revoluciones por minuto de la turbina retroalimentan los controles de carga eléctrica y *pitch*. Al convertir algunos de los bloques intermedios en unidades atómicas se detectó este problema. Para solucionarlo, bastó con agregar un bloque de retardo, como el que puede verse en la misma figura debajo del bloque de la turbina.

Cambios en la implementación de MATLAB

Los *Live Scripts* que se proponen para trabajar con esta última versión ya plenamente funcional de la simulación utilizan temporizadores, que permiten realizar todas las lecturas y escritura a Simulink y ThingSpeak temporalmente equidistantes entre sí.

simulator Contiene declaraciones de identificadores y claves de ThingSpeak y de bloques de Simulink. Inicializa el estructurado `data` con listas vacías para guardar datos de la simulación y con un contador de las lecturas que se van almacenando. Permite seleccionar la frecuencia de subida de datos, en este caso 20 segundos, y lo que va a durar la simulación, siendo infinito un posible valor. Inicializa el temporizador, `TimerLoop`, para que ejecute a los otros dos Live Scripts: `loop` con frecuencia de 1 segundo y `stop` al terminar o si se produce un error. Con el sistema de Simulink cargado en memoria, establece el comando que inicia la simulación, deja pasar unos segundos de margen e inicia también el temporizador.

loop Lee datos de la simulación y los añade a `data`. Si `data` ya cuenta con 20 lecturas, las encapsula con sus marcas temporales y las escribe en ThingSpeak, resturando `data` a su estado inicial antes de terminar. Por último, lee comandos de ThingSpeak y los escribe en la simulación.

stop Establece en la simulación el comando que la termina.

5.6. Aportaciones

En este capítulo se ha descrito la implementación completa del Gemelo Digital de una granja eólica. Expone el novedoso proceso de cómo aplicar el patrón de diseño Modelo-Vista-Controlador en MATLAB y cuál ha sido el resultado de cada una de sus partes. En primer lugar, el modelo se encarga de acceder a ThingSpeak y de almacenar la información del estado y la estabilidad de la turbina, así como de gestionar el envío de comandos. En segundo lugar, la vista muestra dicha información al usuario y le permite interactuar con las turbinas mediante una interfaz que cumple con los principios de diseño. En tercer lugar, el controlador gestiona los eventos de la vista y del modelo y fuerza a que se muestren datos actualizados mediante el uso de temporizadores. Por último, se ha presentado el simulador de turbina eólica flotante creado para realizar pruebas en el Gemelo Digital y se ha visto cómo atajar los problemas de su funcionamiento en tiempo real.

Capítulo 6

Software integrado en el prototipo

La implementación en un prototipo real consiste en una fusión del trabajo de los autores con el trabajo de modelado y control realizado por el ingeniero electrónico Giordy Alexander Andrade Aimara. En su Trabajo de Fin de Grado de Ingeniería Electrónica, *Modelo a escala de aerogenerador para pruebas de control* [37], Giordy modela y construye un prototipo a escala de un aerogenerador implementando un algoritmo concreto de los controles de *pitch* y carga.

El software producto de este capítulo es libre y se encuentra hospedado en un repositorio público de GitHub [38].

6.1. Modelo del prototipo de turbina

El prototipo sirve para probar el software desarrollado en un sistema que trata de simular el comportamiento real de un aerogenerador flotante. Inicialmente fue desarrollado por Giordy Alexander Andrade Aimara; posteriormente ampliado por el tutor Segundo Esteban San Román y finalmente por los autores, al sustituir su microcontrolador por otro que soportase multihilo.

La mayor diferencia entre el prototipo y la realidad reside en las velocidades de acción del mecanismo. Controles como el *pitch* son mucho más rápidos en el prototipo que en la realidad, donde deben moverse palas muy pesadas. También son mayores las revoluciones por minuto del rotor para una misma velocidad del viento donde la turbina real emplea trenes de engranajes multiplicadores para aumentar la velocidad angular. Por ello, la importancia de los tiempos varía. La limitación del servidor de ThingSpeak no supone un problema en grandes turbinas. Los tiempos de actualización pueden ser más holgados ya que los cambios son mucho más lentos y los sistemas más suaves.

6.1.1. Componentes

Generador (a) Un motor sin escobillas de señal continua devuelve una señal trifásica cuya amplitud y frecuencia determina su velocidad de rotación.

Accionador del pitch (b) Un microservomotor establece el ángulo de *pitch*.

Accionador de la carga (c) Un sistema compuesto por un array de 4 relés que provee 16 estados determina la resistencia eléctrica del motor del generador.

Sensor de medición inercial (d) Una IMU anclada a la base flotante recoge las aceleraciones lineales y angulares de las vibraciones y del oleaje simulado.

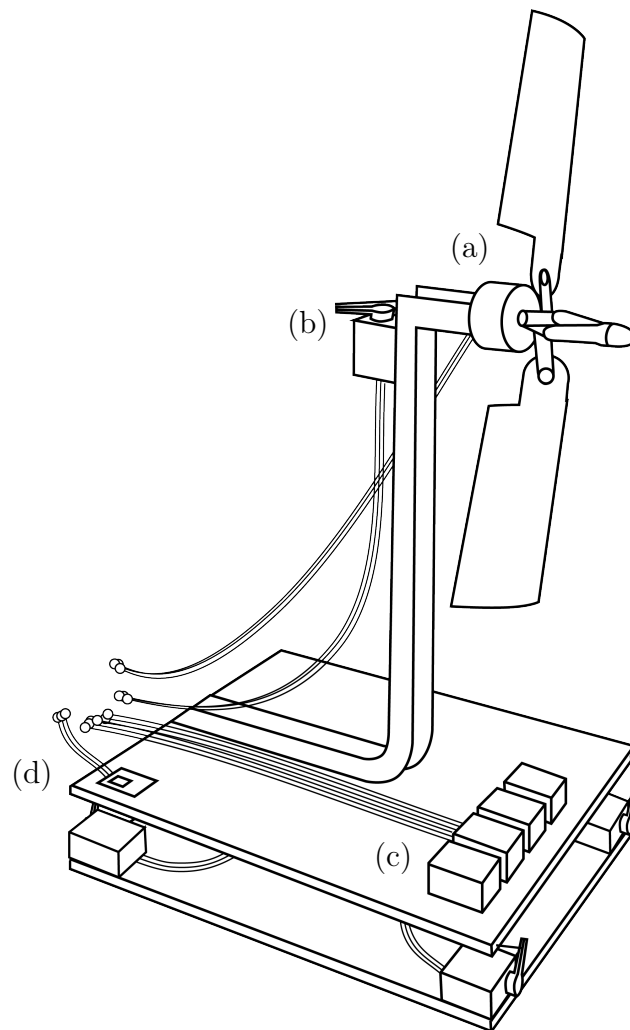


Figura 6.1: Prototipo de aerogenerador *offshore*

Además, el prototipo cuenta con una pequeña pantalla integrada dedicada a la monitorización local en tiempo real. De esta forma, se puede obtener información básica rápidamente.

Todos estos componentes están unidos mediante una placa PCB¹ que incorpora el nuevo microcontrolador de dos núcleos ESP32 y se encarga de organizar las interconexiones y la alimentación.

6.2. Implementación

El diseño concreto de los algoritmos de control de *pitch* y carga corresponde a Giordy Alexander Andrade Aimara. El trabajo de los autores en este campo consiste en la reestructuración íntegra del software bajo un estricto marco teórico y un formato formalizado, la incorporación del sistema multihilo, la implementación de comunicación bidireccional basada en ThingSpeak y la extensión de nuevas estrategias de control.

El código está cargado en el microcontrolador incorporado en el prototipo de la turbina y lo ejecuta indefinidamente desde que se conecta a la alimentación. Una aerogenerador real podría ejecutarlo continuamente desde la instalación interponiendo reinicios de seguridad programados.

El diagrama de flujo de la *Figura 6.2* pretende mostrar el funcionamiento del esqueleto del programa de manera simplificada para conseguir mayor expresividad. Hace uso directo de las clases *Turbine* y *Communication*, de las estructuras de datos de *Data* y de las colas compartidas que comunican ambos procesos bidireccionalmente por paso de mensajes.

Antes de comenzar, cabe recordar que la licencia de ThingSpeak limita a 15 segundos el periodo entre subidas de datos y a 1 segundo la granularidad del muestreo. El periodo se ampliará de 15 a 20 segundos para asegurar las subidas y, como en la implementación del prototipo de baja fidelidad, estas limitaciones se sortean subiendo conjuntamente, en formato de texto JSON, conjuntos de 20 datos cada 20 segundos.

Haciendo uso de las librerías de FreeRTOS, el programa comienza declarando las dos colas encargadas de comunicar los procesos de ambos hilos a través de la función `xQueueCreate(...)`. Corresponden a dos canales unidireccionales accedidos constantemente por ambos procesos.

¹*Printed Circuit Board* o Placa de Circuito Impreso.

statusQueue El canal de estados parte del proceso de control hasta el de comunicación. Tiene capacidad para albergar un número de instancias de Status suficiente como para no llenarse en el tiempo habitual que demora la escritura del estado en el servidor de ThingSpeak. En este caso, se hace coincidir con el periodo de limitación entre subidas para que si ocurriese una larga demora en la subida y el canal se llenara, el programa deseche los datos correspondientes a lapsos concretos.

commandQueue El canal de comandos parte del proceso de comunicación hasta el de control. Tiene capacidad para almacenar una única instancia de Command y, cualquier escritura, la sobrescribe. De esta forma, si eventualmente dos comandos distintos se reciben antes de que el control los lea, el segundo prevalece al interpretarse como una corrección del primero.

A continuación, de igual forma que en la implementación resultante de la primera fase de desarrollo, asocia las dos funciones a los dos procesadores de la placa a través de `xTaskCreatePinnedToCore(...)`.

controlProcess() Asociada al procesador 1 de la placa. Posee una instancia de la clase Turbine y ejecuta directamente sus métodos. La implementación fuerza a que cada iteración dure exactamente 1 segundo.

communicationProcess() Asociada al procesador 0 de la placa. Posee una instancia de la clase Communication y ejecuta directamente sus métodos. Los tiempos de este proceso son asíncronos para asegurar que las instrucciones de bajada y subida no se vean corrompidas.

La funcionalidad queda recogida en las clases cuyas instancias son las protagonistas de los anteriores procesos y queda detallada en lo que sigue. Sin embargo, el paso de mensajes entre procesos implementa detalles a reseñar.

El envío al canal de estados se realiza en el proceso de control con frecuencia de un envío por segundo. Si el canal no está saturado, añade el elemento a la cola; sino, lo vacía y desecha un intervalo completo. La recepción se realiza en el proceso de comunicación durante las iteraciones necesarias hasta completar un intervalo de subida. Si el canal está vacío, el proceso queda esperando indefinidamente por un estado.

El envío al canal de comandos se realiza en el proceso de comunicación, en el mismo bucle en el que se reciben estados, pero tras al menos 1 segundo desde el anterior envío. No importa si el canal está vacío o lleno, el comando siempre se sobrescribe sobre la única posición de la cola. La recepción se realiza en el proceso de control cada segundo si la cola no está vacía.

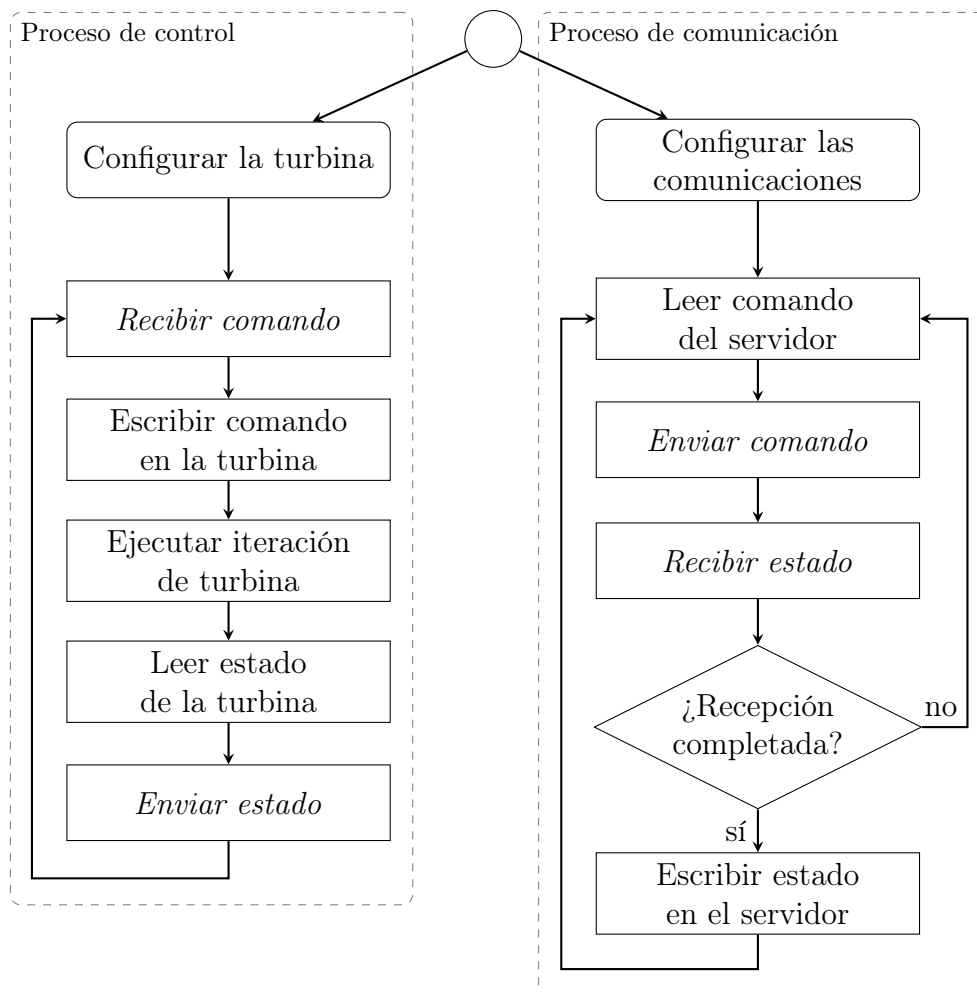


Figura 6.2: Diagrama de flujo.

6.2.1. Estructuras de datos

Fichero `Data.h`

El fichero de encabezado `Data.h` almacena estructuras de datos, funciones de conversión y otras genéricas. Provee a todo el programa de las estructuras necesarias para el almacenamiento y manipulación de los datos. Su diseño puede estar sujeto a cambios si se realizan ampliaciones en la funcionalidad del sistema.

enum Mode Define los modos de uso AUTO o MANUAL del sistema.

enum Phase Define las fases o regiones de trabajo en las que se divide el control. Este diseño considera las fases INIT, START, PITCH_CONTROL, LOAD_CONTROL y STOP.

struct Status Define una estructura de datos que almacena el estado actual de todas las variables de salida de la turbina. Guarda el tiempo, el ángulo del *pitch*, la carga, las revoluciones por minuto, la potencia generada, la resistencia, la aceleración lineal y angular de la estructura y la fase de trabajo en la que se encuentra.

struct Command Define una estructura de datos que almacena variables que el usuario puede ordenar a la turbina. Guarda el modo, el ángulo del *pitch*, la carga o el comando de parada de emergencia.

Las funciones de conversión se encargan de transformar conjuntos de datos a formato de cadena de caracteres con la sintaxis de JSON y de transformar unidades.

El fichero también implementa la función para obtener el tiempo actual en formato EPOCH del servidor `ntp.pool.org`.

6.2.2. Monitor de generador

La *Figura 6.3* recoge el subdiagrama UML de las relaciones entre las clases descritas a continuación.

Clase BrushlessSignal

Las constantes definidas permiten interpretar las señal analógica recibida por el motor sin escobillas (brushless) que hace de generador.

Las variables privadas `real_voltage[]`, `imaginary_voltage[]` y `peak_voltage` guardarán los voltajes registrados en cierto número de muestreos y su pico máximo. Además, `gain` servirá para indicar la ganancia de lectura deseada a los optoacopladores.

`BrushlessSignal()` Inicializa el valor de `gain`.

`void attach(pin)` Vincula las instancias del motor brushless y de los optoacopladores.

`void run()` Ejecuta varias iteraciones para obtener un muestreo amplio de voltajes y poder eliminar picos de tensión indeseados. Reserva los datos recogidos en las variables privadas de `real_voltage[]`, `imaginary_voltage[]` y `peak_voltage`.

`int readFrequency()` Calcula y devuelve las revoluciones por minuto (rpm) del rotor de la turbina en un instante del tiempo.

`float readPower()` Calcula y devuelve la potencia en milivatios (mW) que suministra el generador en un instante del tiempo.

Clase Brushless

La variable privada `pin` sirve para configurar el pin de comunicación de la placa con el motor brushless y `value` para almacenar el valor crudo de la tensión leída en dicho pin.

`void attach(pin)` Asocia el motor brushless al pin de comunicación serial `pin` y lo configura como de lectura.

`int read()` Lee y devuelve el valor analógico de la tensión presente en el pin del generador.

Clase Optocoupler

Las variables privadas `pin1` y `pin2` guardarán los pines de control de los optoacopladores y `value` el valor actual de la ganancia.

`void attach(pin)` Asocia los dos optoacopladores a los pines `pin1` y `pin2` de la placa y los configura de salida.

`void write(float value)` Configura los optoacopladores para establecer la ganancia deseada `value`.

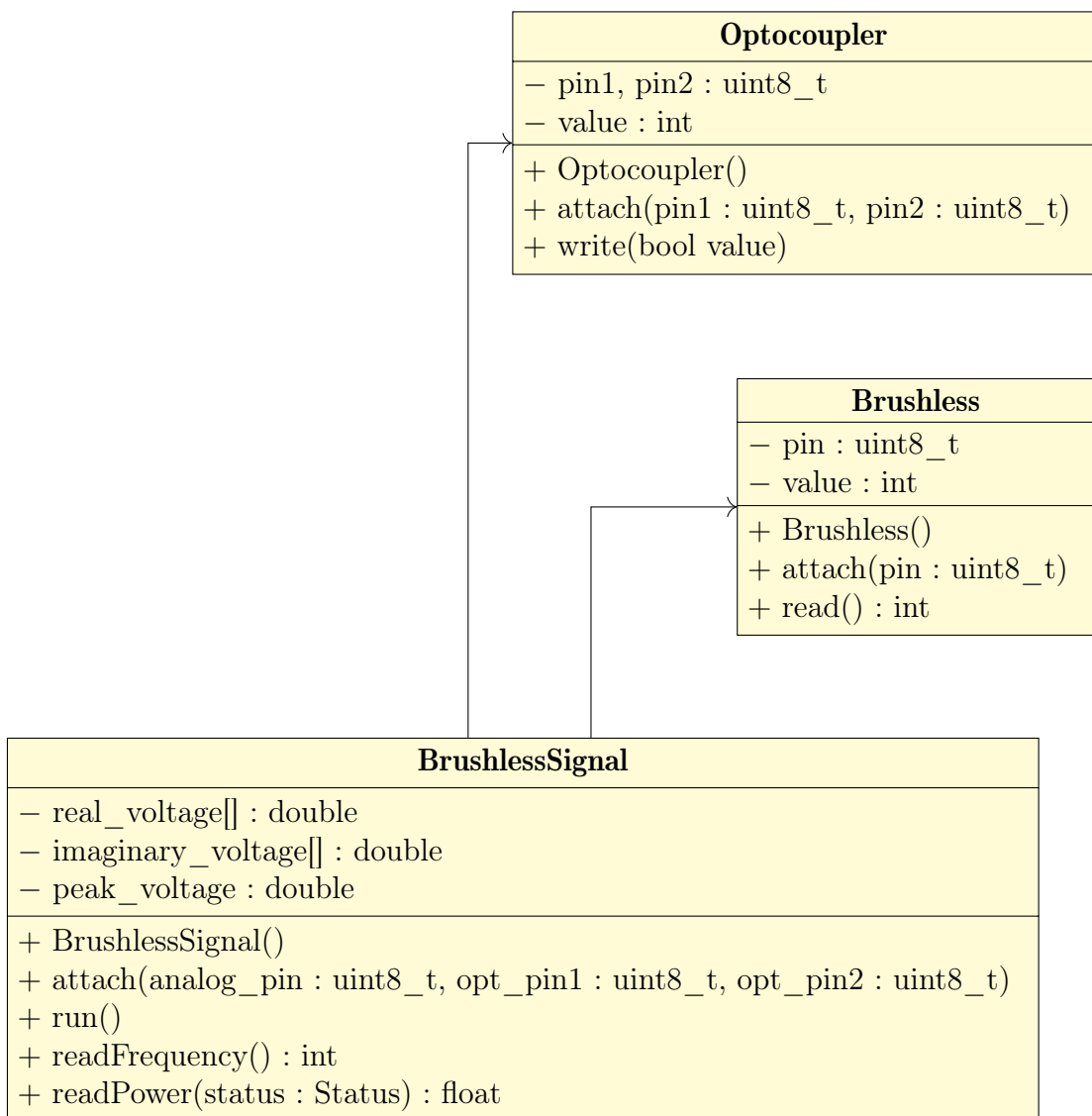


Figura 6.3: Clase del monitor del generador

6.2.3. Monitor de IMU

La *Figura 6.4* recoge el subdiagrama UML de las relaciones entre las clases descritas a continuación. La clase MPU6050 aparece sombreada para indicar que su autoría es de un tercero.

Clase ImuSignal

Las constantes definidas permiten transformar los valores crudos, leídos por la unidad de medición inercial, a unidades del Sistema Internacional.

`void attach(address_pin)` Vincula e inicializa una instancia MPU6050 en la dirección de pines `address_pin`.

`Point readAcceleration() / readRotation()` Lee, convierte al sistema internacional y devuelve los valores de aceleración lineal y angular de cada eje.

Clase MPU6050

Esta clase corresponde a una librería de muestreo del sensor de medición inercial desarrollada por Electronic Cats. La clase `ImuSignal` posee una instancia de la misma para realizar lecturas sencillas de las aceleraciones lineales y angulares.

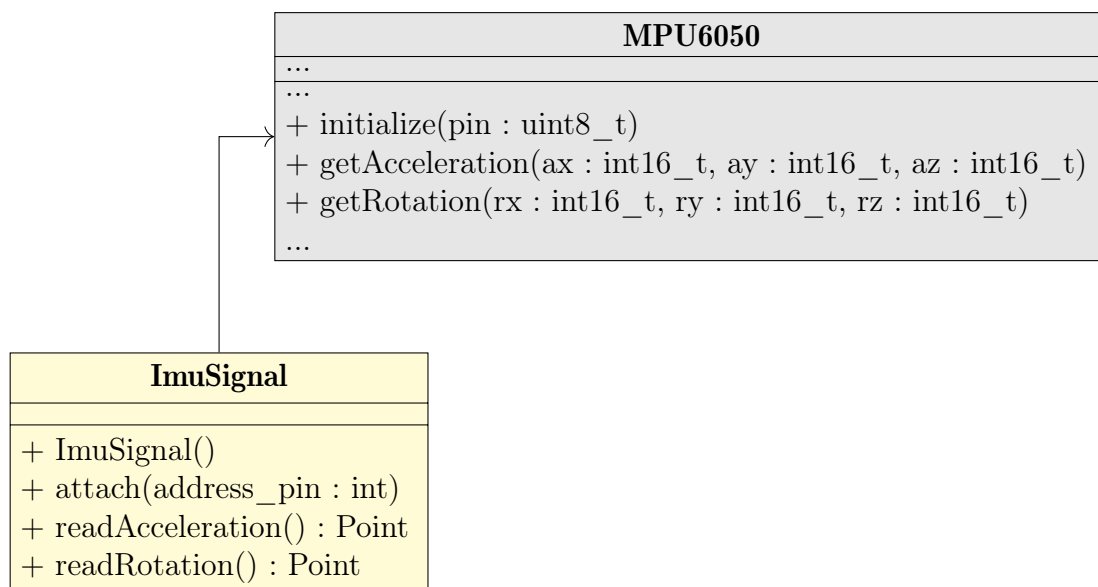


Figura 6.4: Clase del monitor de la IMU

6.2.4. Controlador de fase

La *Figura 6.5* recoge el subdiagrama UML de la clase descrita a continuación.

Clase Control

El algoritmo de control de fase responde a un diseño concreto. Por eso, el desarrollo de esta clase corresponde al diseñador del modelo de control.

Las constantes MAX_RPM, MIN_RPM y OPT_RPM almacenan las revoluciones máximas, mínimas y óptimas deseadas. Además, también define ERR_RPM para ampliar intervalos de trabajo.

nextPhase(command, status) Indica en qué fase se encuentra el sistema en cada iteración a partir de los comandos ordenados por el usuario y del estado de la turbina. El diseño implementado se basa en las revoluciones por minuto del aerogenerador para determinar la fase.

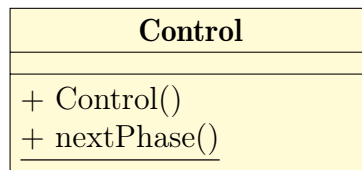


Figura 6.5: Clase del control de fase

6.2.5. Controlador de pitch

La *Figura 6.6* recoge el subdiagrama UML de las relaciones entre las clases descritas a continuación. La clase ESP32Servo aparece sombreada para indicar que su autoría es de un tercero.

Clase Pitch

Las constantes MAX_PITCH, MIN_PITCH y OPT_PITCH almacenan los ángulos máximo, mínimo y óptimo del *pitch* de las palas.

La variable protegida `pitch` guarda la inclinación actual de las palas.

`Pitch()` Inicializa el valor de `pitch` a MIN_PITCH.

`void attach(pin)` Vincula la instancia del servo ESP32Servo a `pin` y establece el ángulo `pitch` al arranque, antes de que comience el proceso de control.

`void run(command, status)` Es un método virtual vacío que debe ser sobreescrito por los hijos de la clase. Está reservada para implementar el comportamiento de las palas en función de los comandos o del estado de la turbina.

`int read()` Devuelve el valor de `pitch`, el ángulo de inclinación de las palas.

Clase PitchControl

Esta clase corresponde por completo al diseñador del algoritmo de control del *pitch*. La siguiente descripción solo representa un ejemplo. Como hereda de la clase Pitch, está obligada a implementar el funcionamiento del método `void run(command, status)`.

`void run(command, status)` Sobreescribe el método homónimo de la clase padre donde implementa la actuación del *pitch*. En este caso, primero diferencia entre control automático o manual. El automático comprueba la fase de `status` y ejecuta el método correspondiente a cada una. El manual comprueba si la parada de emergencia está activa y, sino, aplica el *pitch* de `command` con métodos de la clase padre Pitch.

Los siguientes métodos se encargan de implementar el comportamiento del *pitch* en cada fase del modo automático.

`void init(status)` En la fase INIT se establece el valor del *pitch* a MIN_PITCH para favorecer el arranque del sistema.

void start(status) En la fase START se establece el *pitch* de manera que experimente un crecimiento exponencial y aumente rápidamente las revoluciones para alcanzar cuanto antes la región de trabajo.

void pid(status) En la fase PITCH_CONTROL se establece el *pitch* mediante un control PID para mantener las revoluciones en la región de trabajo.

void stop(status) En la fase STOP se establece el *pitch* a MAX_PITCH para detener la rotación de las palas lo antes posible.

Clase ESP32Servo

Esta clase corresponde a una librería de control del microservomotor para el microcontrolador ESP32 desarrollada por John K. Bennett. La clase Pitch posee una instancia de la misma para facilitar el envío de comandos al servo.

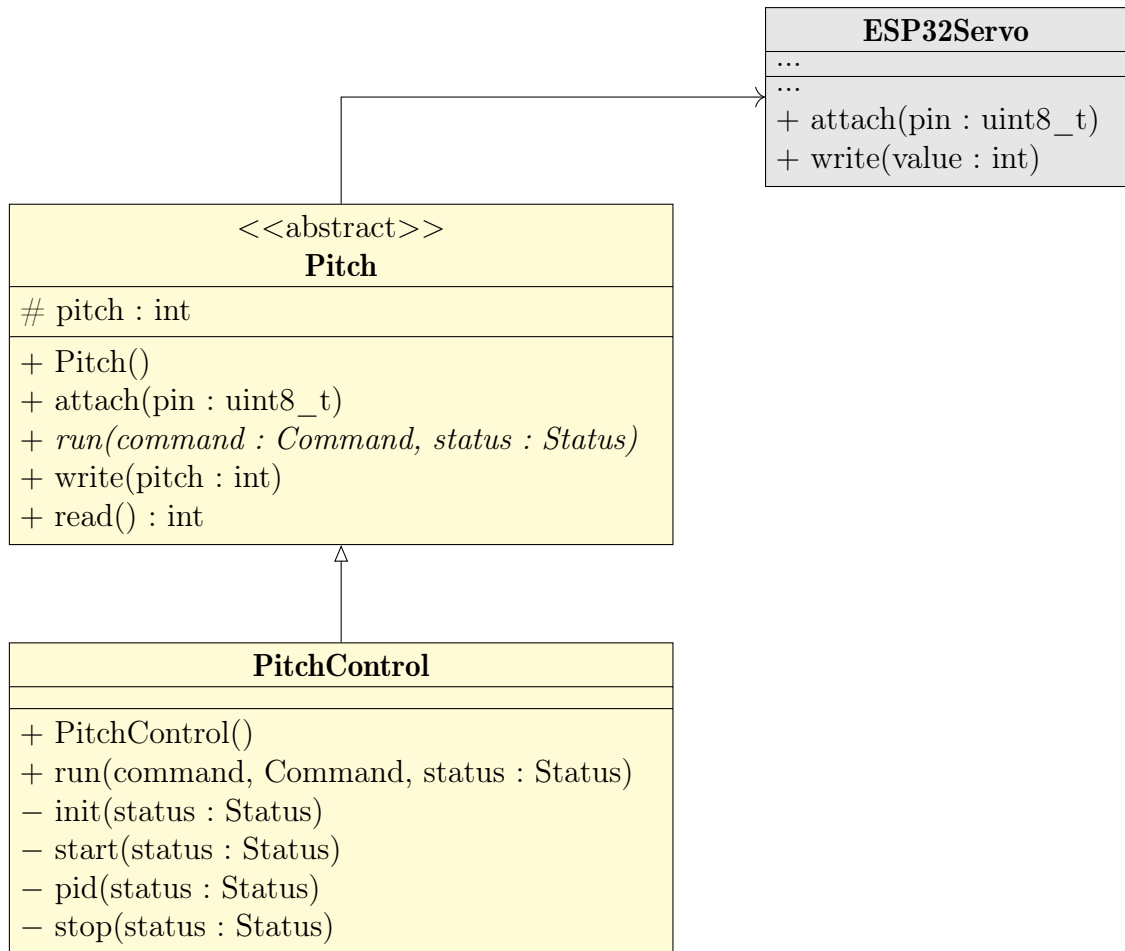


Figura 6.6: Clases de control del *pitch*

6.2.6. Controlador de carga

La *Figura 6.7* recoge el subdiagrama UML de las relaciones entre las clases descritas a continuación.

Clase Load

Las constantes `MAX_LOAD` y `MIN_LOAD` almacenan los estados de carga máxima y mínima de la resistencia que el sistema de relés aplica en el generador.

La variable protegida `load` está destinada a guardar el estado de la carga actualmente aplicada al motor.

`Load()` Inicializa el valor de `load` a `MIN_LOAD`.

`void attach(pin1, pin2, pin3, pin4)` Vincula la instancia del sistema de relés Relay a `pin1`, `pin2`, `pin3` y `pin4` y establece el estado de carga `load` al arranque, antes de que comience el proceso de control.

`void run(command, status)` Es un método virtual vacío que debe ser sobrescrito por los hijos de la clase. Está reservada para implementar el comportamiento de la resistencia, a través de los relés, en función de los comandos enviados o del estado de la turbina.

`int read()` Devuelve el valor de `load`, el estado de la carga del generador.

`float readResistance()` Realiza una llamada a la clase Relay para que le informe de la resistencia que actualmente está aplicada.

Clase LoadControl

Al igual que `PitchControl`, esta clase corresponde por completo al diseñador del algoritmo de control de la carga. La siguiente descripción solo representa un ejemplo. Como hereda de la clase `Load`, está obligada a implementar el funcionamiento del método `void run(command, status)`.

`void run(command, status)` Sobreescibe el método homónimo de la clase padre donde implementa la actuación de la carga. En este caso, primero diferencia entre control automático o manual. El automático comprueba la fase de `status` y ejecuta el método correspondiente a cada una. El manual comprueba si la parada de emergencia está activa y, sino, aplica la carga de `command` con métodos de la clase padre `Load`.

Los siguientes métodos se encargan de almacenar el comportamiento de la carga en cada fase del modo automático.

void start(status) En la fase START se establece el valor de la carga con un control típico D para favorecer el aumento de y alcanzar rápido la región de trabajo.

void pid(status) En la fase LOAD_CONTROL se establece el valor de la carga con un control típico PD para mantener las revoluciones en la región de trabajo.

void stop(status) En la fase STOP se establece el valor de la carga a MAX_LOAD para detener la rotación de las palas lo antes posible.

Clase Relay

Esta clase está desarrollada por los autores y mantiene un funcionamiento simétrico al de la clase ESP32Servo en los métodos exclusivamente necesarios.

Las variables privadas **pin1**, **pin2**, **pin3** y **pin4** sirven para guardar los pines de conexión del microcontrolador con el módulo de relés, **value** para el estado de carga y **resistance** para la resistencia asociada al estado de la carga.

void attach(pin1, pin2, pin3, pin4) Vincula el array de relés a los pines de comunicación serial **pin1**, **pin2**, **pin3** y **pin4** y los configura de salida.

void write(value) Transforma el estado de la carga dado por **value** en un array binario de 4 posiciones que activa los relés que aplican la resistencia correspondiente a dicho estado.

float readResistance() Devuelve la resistencia que actualmente está aplicada por el estado de **load** en el sistema de relés.

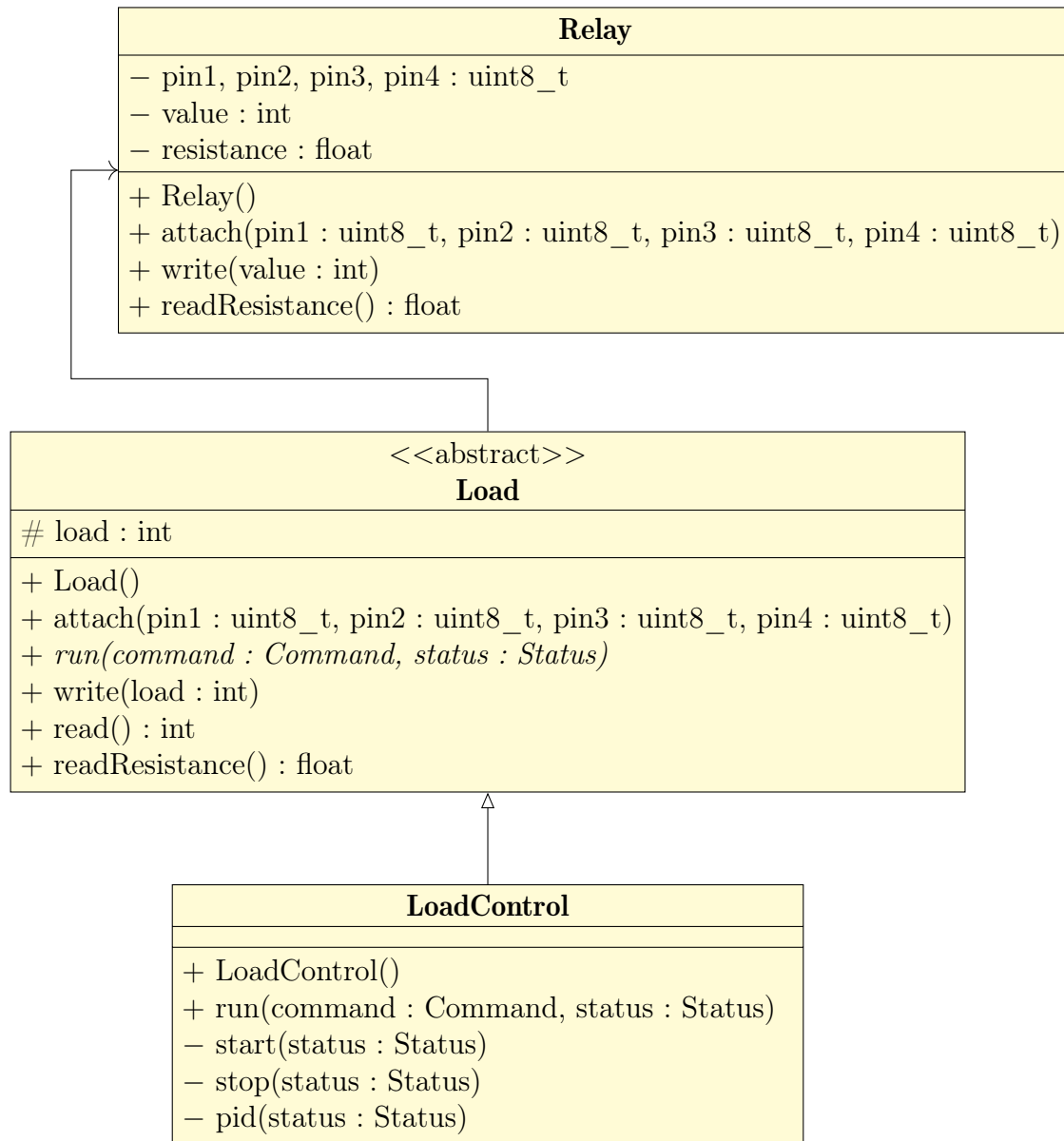


Figura 6.7: Clases de control del load

6.2.7. Turbina

La *Figura 6.8* recoge el subdiagrama UML de las relaciones entre las clases descritas a continuación.

Clase Turbine

Las constantes públicas almacenan los pines necesarios de todos módulos hardware implicados en el funcionamiento del control de la turbina.

Las variables privadas `status` y `command` estarán actualizadas con la información de la turbina. Solo pueden ser accedidas por el resto del programa a través de los métodos `Status read()` y `void write(command)`. La variable `status` solo puede ser leída por el resto del programa a través del método `Status read()`. La variable `command` puede ser modificada por el resto del programa a través del método `void write(command)`.

`void setup()` Configura cada componente en los pines correspondientes. Vincula el controlador del *pitch*, el controlador de la carga, el generador y la unidad de medición inercial.

`void run()` Ejecuta una iteración de monitorización y control de la turbina. Este método es llamado periódicamente con una frecuencia suficiente para que la turbina actualice sus variables de estado y ejecute las acciones de control sin riesgos. Primero, lee y analiza las señales que recibe del generador y de la unidad de medición inercial con los métodos `void run()` y `void read()` correspondientes de las clases `BrushlessSignal` e `ImuSignal`. A continuación, llama a los controladores de *pitch* y carga para que ejecuten las modificaciones pertinentes con el método `void run()` de respectivas clases `PitchControl` y `LoadControl`. Por último, actualiza la fase en la que se encuentra el sistema a través del método estático `Phase nextPhase(command, status)` de la clase `Control`.

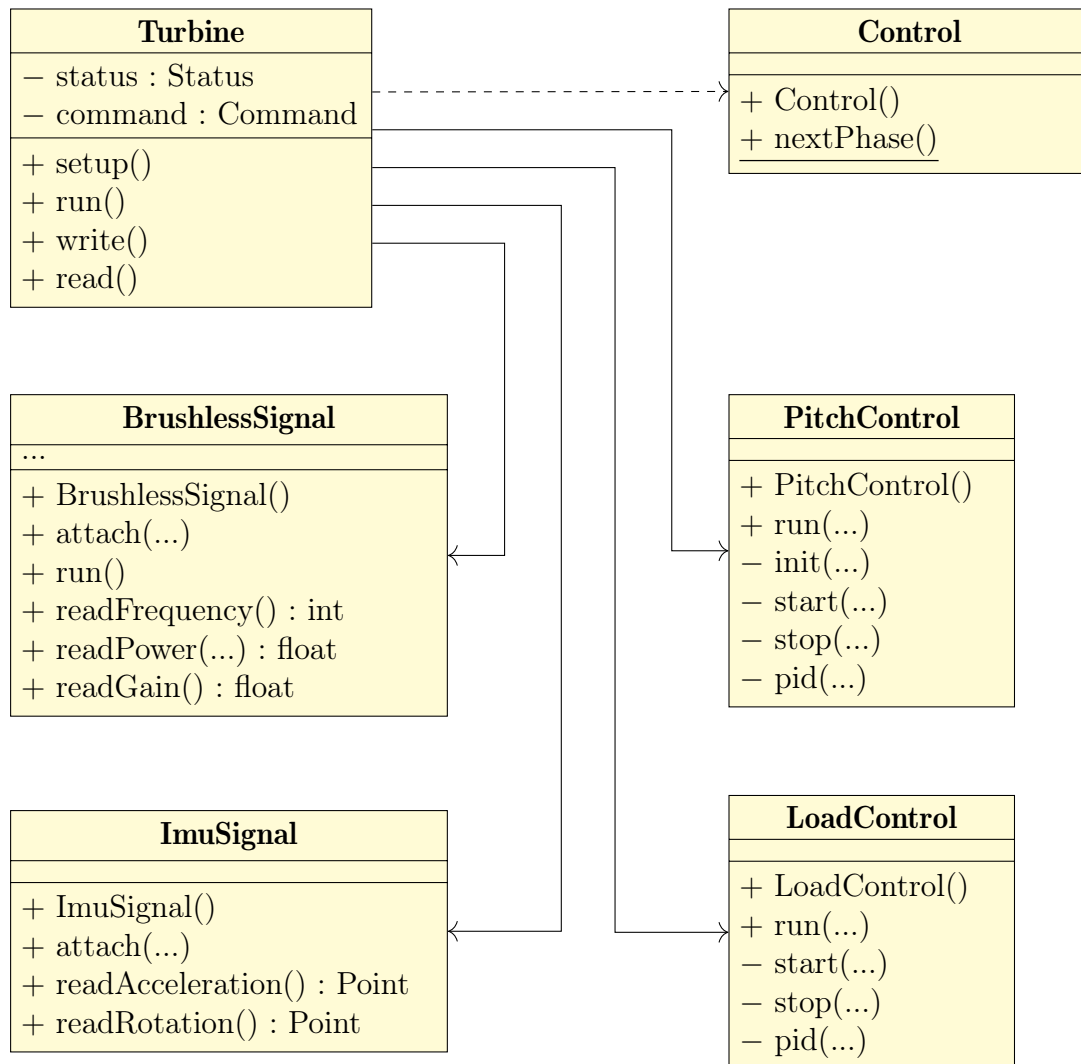


Figura 6.8: Clases de la turbina

6.2.8. Comunicaciones

La *Figura 6.9* recoge el subdiagrama UML de la clase descrita a continuación.

Clase Communication

Las constantes públicas almacenan parámetros de configuración de la pantalla. Aparte, un fichero `credentials.h` guarda claves de acceso a los servidores.

La variable privada `wifi_client` almacenará una conexión con determinados IP y puerto y las variables `state_http_client` y `acc_rot_http_client` guardarán la configuración de acceso a servidores HTTP, a ambos canales de ThingSpeak.

`void setupWiFi()` Conecta la placa a la red WiFi con la SSID y contraseña recogidas en el fichero `credentials.h`. También realiza configuraciones auxiliares, como establecer conexión con el servidor de consulta de la hora EPOCH.

`void setupThingSpeak()` Inicia la comunicación con ThingSpeak y asocia el servidor HTTP de cada canal a las instancias correspondientes.

`void writeThingSpeak(status[])` Realiza una solicitud POST a cada servidor HTTP y escribe la información del estado de la turbina.

`Command readThingSpeak()` Realiza solicitudes GET al canal de comunicación de comandos y lo devuelve para escribirlo en la turbina.

`void setupScreen()` Configura la pantalla integrada y muestra en ella una pantalla de carga hasta el fin de la configuración del proceso de comunicación.

`void writeScreen(command, status)` Muestra en la pantalla integrada la información relativa al parámetro `status`.

Communication
– <code>wifi_client</code> : WiFiClient – <code>state_http_client</code>, <code>acc_rot_http_client</code> : HTTPClient – <code>screen</code> : Adafruit_SSD1306
+ <code>setupWiFi()</code> + <code>setupThingSpeak()</code> + <code>writeThingSpeak(status_array[] : Status)</code> + <code>readThingSpeak() : Command</code> + <code>setupScreen()</code> + <code>writeScreen(command : Command, status : Status)</code>

Figura 6.9: Clase de comunicación

6.3. Aportaciones

Este capítulo recoge el desarrollo del software integrado en el microcontrolador ESP32 del prototipo de turbina eólica. Antes de proceder al desarrollo, muestra el prototipo del que se dispone y sus componentes. Comienza esbozando el software de manera general e introduciendo las estructuras internas de datos. También hace especial énfasis en las estructuras encargadas de la comunicación entre hilos mediante paso de mensajes. A continuación, describe las estructuras partiendo de las más concretas para llegar a las más abstractas: primero, los monitores y los controladores, y después, la clase de la tarea de la turbina y la clase de la tarea de las comunicaciones.

Aunque no se recoge un diagrama UML global, se presenta de forma mucho más clara dividido en subdiagramas. Resulta más sencillo para una interpretación a primera vista y permite una inmersión posterior en aspectos más concretos si fuera necesario.

Parte III

Análisis y discusión de resultados

Capítulo 7

Casos de uso

Este capítulo analizará el funcionamiento del sistema a través de casos de uso.

Las pruebas de funcionamiento en el prototipo se realizaron con un ventilador casero de tres velocidades ascendentes, de la 1 a la 3, siendo 0 el botón de apagado.

7.1. Ciclo de funcionamiento automático completo

Este apartado describe el caso de uso del funcionamiento del control automático y de la monitorización cuando la velocidad del viento aumenta progresivamente y atraviesa las tres etapas vistas en la Subsección 2.2.1. Solo se estudiará el funcionamiento de este caso para el prototipo de turbina real.

La descripción del funcionamiento queda dividida en dos partes. Por un lado, el comportamiento del prototipo y la respuesta de sus algoritmos de control. Por otro, el comportamiento del Gemelo Digital y la visualización de su interfaz.

7.1.1. Análisis del prototipo de turbina

Configuración del sistema

Casi inmediatamente después de alimentar el sistema, las palas se colocan en posición de parada con un ángulo de *pitch* de 0° a la espera de que se terminen de cargar y configurar los componentes del hilo de control.

Al mismo tiempo, el hilo de comunicaciones también carga y configura sus componentes. Entre ellos la pantalla integrada, la conexión a Internet y ThingSpeak. La pantalla integrada será especialmente útil para ilustrar el funcionamiento. Muestra el estado de la carga como en la *Figura 7.1* a la espera de que, cuando termine la configuración, comience a recibir datos de muestreo del hilo de control.



Figura 7.1: Pantalla integrada de carga

Unos milisegundos después de conectar la alimentación, el proceso de control autónomo habrá comenzado con la estrategia implementada. Gracias a esta rapidez y a su estado inicial de parada, la turbina no sufre riesgos por pérdida de control al comienzo del ciclo.

El proceso de comunicación comenzará unos segundos más tarde ya que la configuración de conexiones implica mayores tiempos de espera. La velocidad de carga y descarga de la red WiFi es clave a la hora de determinar estos tiempos.

Tramo en calma

El ventilador aún está apagado.

El lazo de control autónomo comienza colocando el *pitch* de las palas a 45° y el indicador de la carga a 0 —con resistencia de 500Ω entre los terminales del generador¹— para favorecer el arranque. Este primer empuje es necesario para vencer al rozamiento mecánico de los componentes.

El proceso de comunicación comienza a recibir datos del control. La pantalla integrada muestra ahora su estado como en la *Figura 7.2* actualizado con frecuencia de 1Hz. ThingSpeak recibe los primeros paquetes de datos.

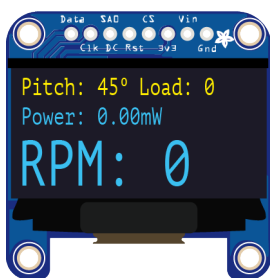


Figura 7.2: Pantalla integrada sin viento

¹Los valores de *pitch*, carga y ganancias han sido reajustados respecto al trabajo de *Modelo a escala de aerogenerador para pruebas de control* [37] y se pueden encontrar en los ficheros de cabecera del software en el repositorio [38].

Tramo de arranque

Tras pulsar el botón de velocidad 1 del ventilador, las palas comienzan a girar.

En este punto, a medida que aumenta la velocidad, el controlador de *pitch* acerca el ángulo de las palas al óptimo —establecido en 10° — y el controlador de carga incrementa el indicador que reduce la resistencia eléctrica. Este control consigue que, incluso con la velocidad más baja del ventilador, la turbina alcance la velocidad angular nominal o de producción.

Tramo de producción

Las revoluciones han llegado al umbral de producción, establecido con un margen del 20 % respecto de las revoluciones nominales. El algoritmo fija la carga y activa el control PID del *pitch* para mantener las revoluciones dentro de dicho umbral. El control de *pitch* consigue estabilizarlas con un error inferior al 3 % a pesar de las pequeñas variaciones en el flujo del viento. Véase el estado de trabajo normal en la *Figura 7.3a*.

Al pulsar el botón de velocidad 2 del ventilador, las revoluciones aumentan lentamente.

El controlador PID del *pitch* distancia el ángulo de ataque del óptimo y lo acerca a los 45° . Las revoluciones disminuyen y el sistema, con su nuevo ángulo de *pitch*, consigue estabilizarse en el umbral de producción. En la *Figura 7.3b* se observa cómo el *pitch* ha tenido que aumentar el ángulo de ataque para mantener revoluciones óptimas.

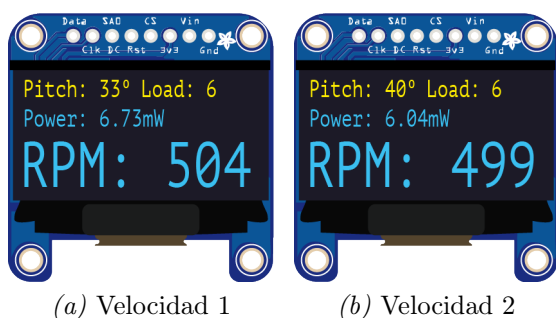


Figura 7.3: Pantalla integrada en distintas velocidades de viento

Tramo de parada de emergencia

Por último, se activa el botón 3 del ventilador. Dado que el control de *pitch* ya había reducido su margen de actuación al estabilizarse en la velocidad 2, las revoluciones aumentan rápidamente y se produce un pico que supera el umbral de producción.

El sistema desactiva el controlador de *pitch* y activa el controlador de carga eléctrica del generador. A pesar de que este método es capaz de parar el aumento de las revoluciones, no es suficiente como para disminuirlas y devolverlas al rango de trabajo. En este punto, la estrategia consiste en realizar una parada de emergencia: coloca las palas en bandera y aumenta al máximo la resistencia de carga. Esta estrategia hace que las palas frenen por completo la rotación del generador ayudadas por la resistencia que ofrece el generador al absorber mayor cantidad de energía.

La pantalla muestra una alerta para indicar la parada. Para resaltar esta fase, simula un parpadeo en la primera línea. Este efecto se puede ver en las *Figuras 7.4a* y *7.4b*.

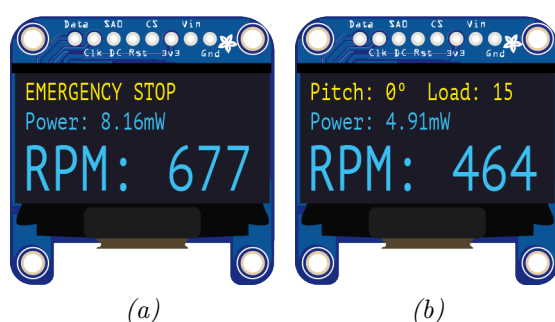


Figura 7.4: Pantalla integrada de parada

Aquí termina el caso de uso. Sin embargo, cuando la turbina queda parada por completo, vuelve a intentar el arranque para tratar de estabilizar de nuevo sus revoluciones en el rango de trabajo.

El prototipo

En la *Figura 7.5* puede verse el prototipo físico en funcionamiento en el laboratorio del departamento de Arquitectura de Computadores y Automática de la Facultad de Física de la Universidad Complutense.

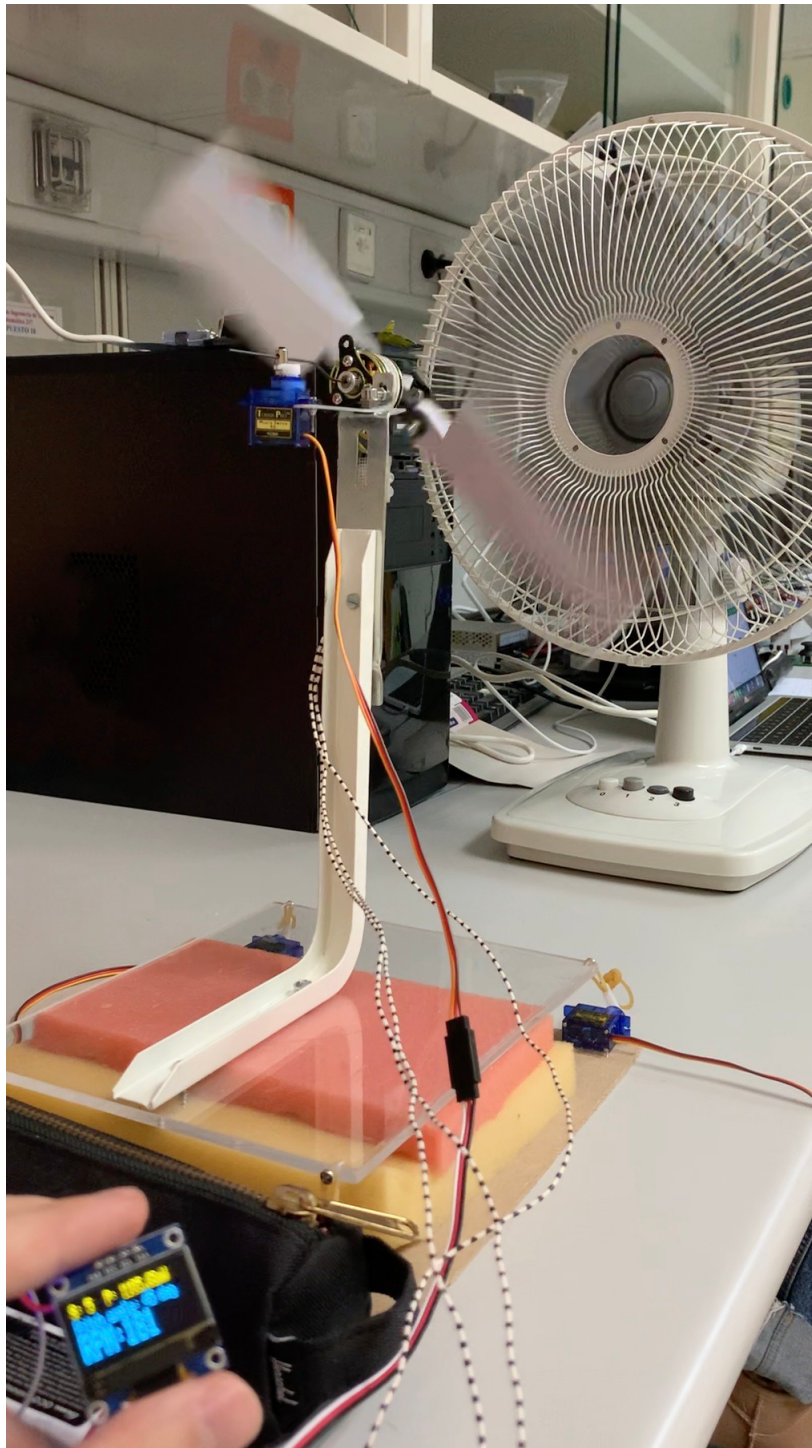


Figura 7.5: Prototipo de la turbina eólica flotante

7.1.2. Análisis del Gemelo Digital

Pantalla de la granja

Al iniciar el entorno del Gemelo Digital, aparece la pantalla de la *Figura 7.6* donde se recoge información de la granja formada por las turbinas asociadas. Muestra detalles generales, la localización precisa de las turbinas y la producción total de la granja completa. En este caso, la granja está formada por dos turbinas: el prototipo ubicado en Madrid, en la Facultad de Ciencias Físicas de la Universidad Complutense; y el simulador ubicado virtualmente en el mar Cantábrico, enfrente de las costas de Ribadesella.

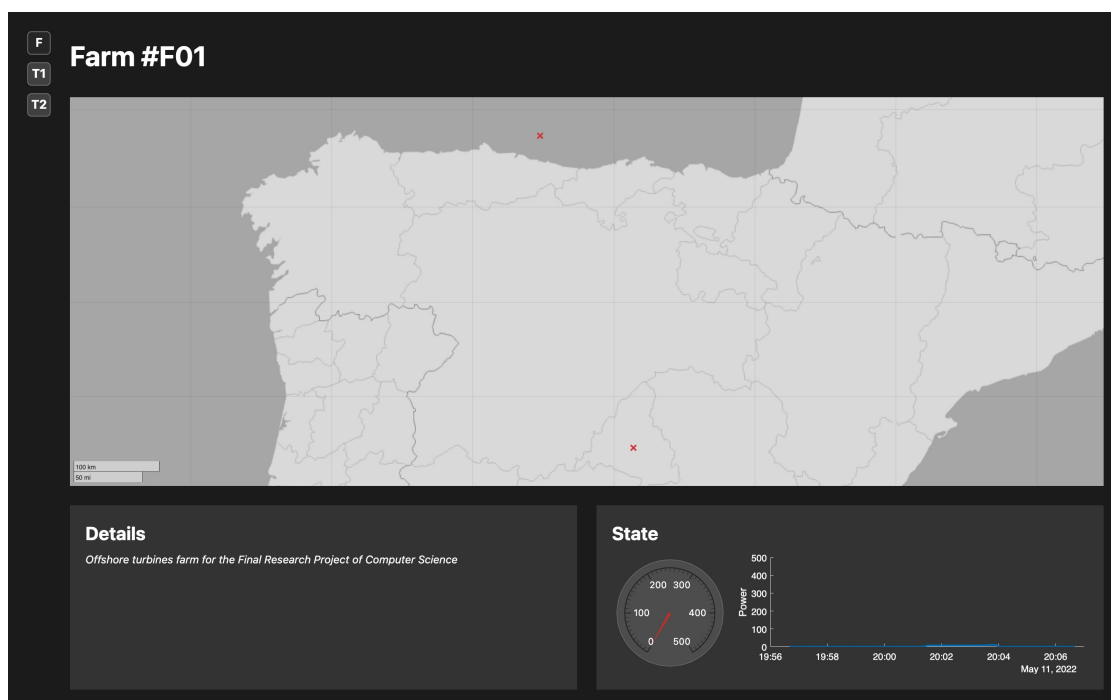


Figura 7.6: Interfaz de la granja

Pantalla del prototipo

Mientras el prototipo de turbina eólica esté desconectado de la alimentación o de Internet, el Gemelo Digital mantiene deshabilitados los componentes encargados de la monitorización. Deja activos, sin embargo, los componentes de control para que, en cuanto el prototipo de turbina se conecte, adopte los parámetros ordenados. También se puede visualizar la información básica del modelo y el estado meteorológico actual de la localización de la turbina.

Cuando el prototipo se enciende o establece conexión, el Gemelo Digital comienza a leer y graficar los paquetes de datos de muestreo. También aparecen los primeros indicadores de riesgos. En este caso, el indicador de ejemplo implementado monitorea las revoluciones para advertir de rangos no deseados.

La *Figura 7.7* muestra claramente el comportamiento de las regiones de trabajo de la turbina a lo largo de todo el ciclo expuesto anteriormente. Además, incluye un módulo en la esquina inferior izquierda dedicado al análisis de la estabilidad de la estructura flotante. Estos datos, proporcionados por el módulo de medición inercial incorporado, serán de utilidad en posteriores implementaciones.

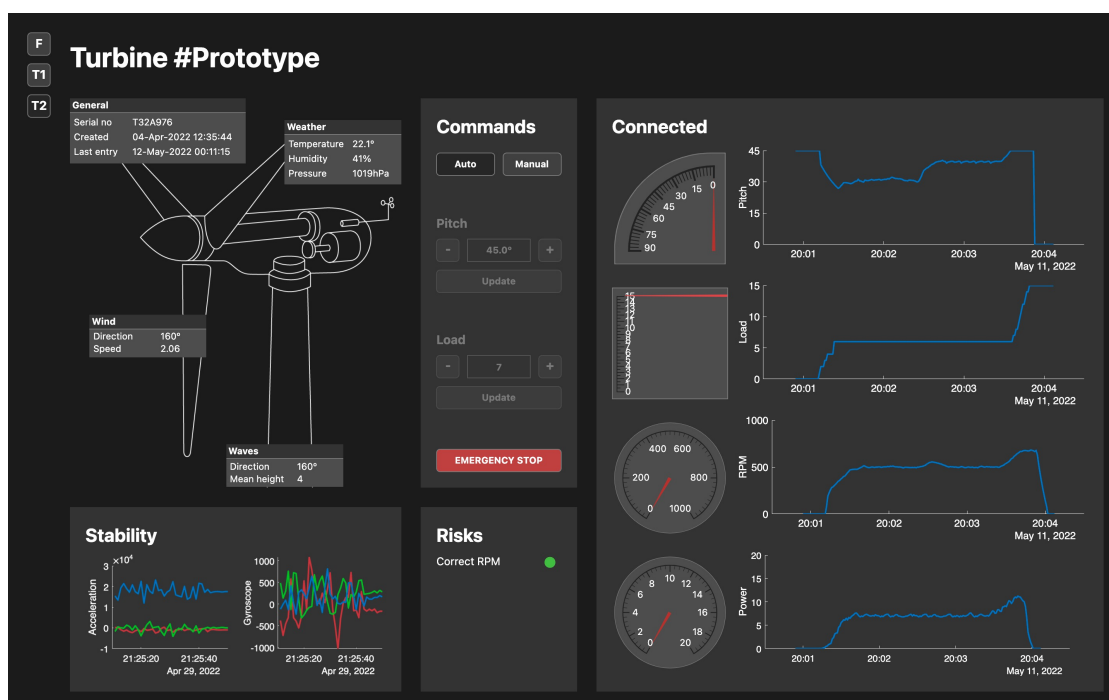


Figura 7.7: Interfaz de la turbina mostrando un ciclo de funcionamiento completo

7.2. Funcionamiento manual

Este apartado describe el caso de uso del funcionamiento del control manual y de la monitorización para una velocidad de viento estable. Aunque el prototipo puede ser controlado por completo desde el inicio de su actividad, se realizarán pruebas partiendo del tramo de producción.

Dado que el Gemelo Digital juega un papel activo en el comportamiento del prototipo y del simulador, el funcionamiento se describe de manera conjunta.

7.2.1. Análisis del prototipo de turbina y del Gemelo Digital

Cuando las revoluciones han llegado al umbral nominal y los parámetros se estabilizan, el usuario activa el modo manual con los mismos parámetros. El controlador PID deja de actuar y el manejo queda por completo en manos del usuario.

Modificación de la carga

Incrementar el indicador de la carga puede reducir levemente las revoluciones. A pesar de ello, resulta interesante si se quiere aumentar la producción de potencia del generador de la turbina. Cuanto mayor sea el indicador, mayor será la potencia extraída. En la *Figura 7.8* se muestra el resultado.

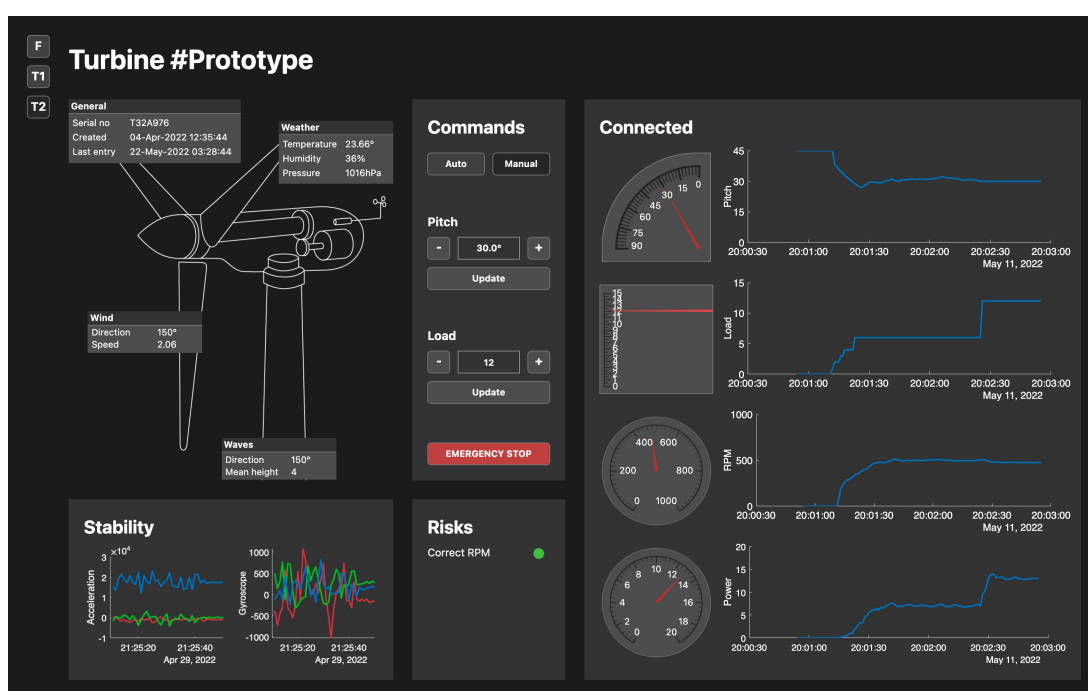


Figura 7.8: Interfaz de la turbina mostrando el cambio de carga manual

Modificación del pitch

Variar el ángulo de *pitch* determina de forma mucho más relevante los cambios en las revoluciones. Partiendo de la velocidad angular nominal, posicionar el *pitch* en el ángulo óptimo puede llegar a aumentar las revoluciones en más de un 200 % y posicionar el *pitch* en el ángulo mínimo detiene el funcionamiento. En la *Figura 7.9*, tras un periodo de estabilidad, se muestra el resultado de haber establecido el *pitch* en su ángulo óptimo.

En este momento, la señal de riesgo de revoluciones cambia y ahora indica que están fuera de rango. El color rojo insta al usuario a realizar una parada de emergencia.

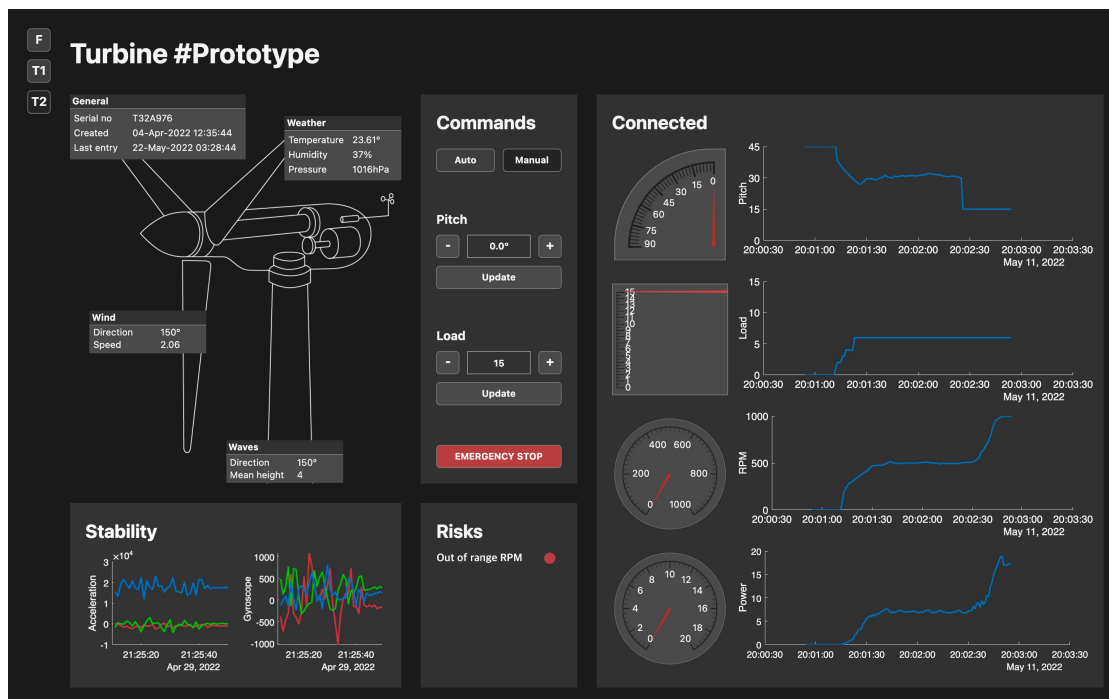


Figura 7.9: Interfaz de la turbina mostrando el cambio de *pitch* manual

Parada de emergencia

Cuando las revoluciones se han disparado y han salido fuera de los rangos normales, ejecutar controles de carga o *pitch* puede suponer riesgos eléctricos o estructurales para el sistema. El usuario debe presionar, aconsejado por el Gemelo, el pulsador de parada de emergencia que posiciona las palas en posición de bandera y el indicador de carga al máximo. Véase el comienzo del descenso de revoluciones y potencia en la Figura 7.10, tras el pico de revoluciones ocasionado con el comando anterior.

Gracias al diseño del software integrado y del Gemelo Digital y a sus afinidades, el comando de emergencia se ve reflejado en el prototipo con un retraso mínimo. La limitación es la frecuencia con la que el software integrado lee los comandos de ThingSpeak, una lectura por segundo.

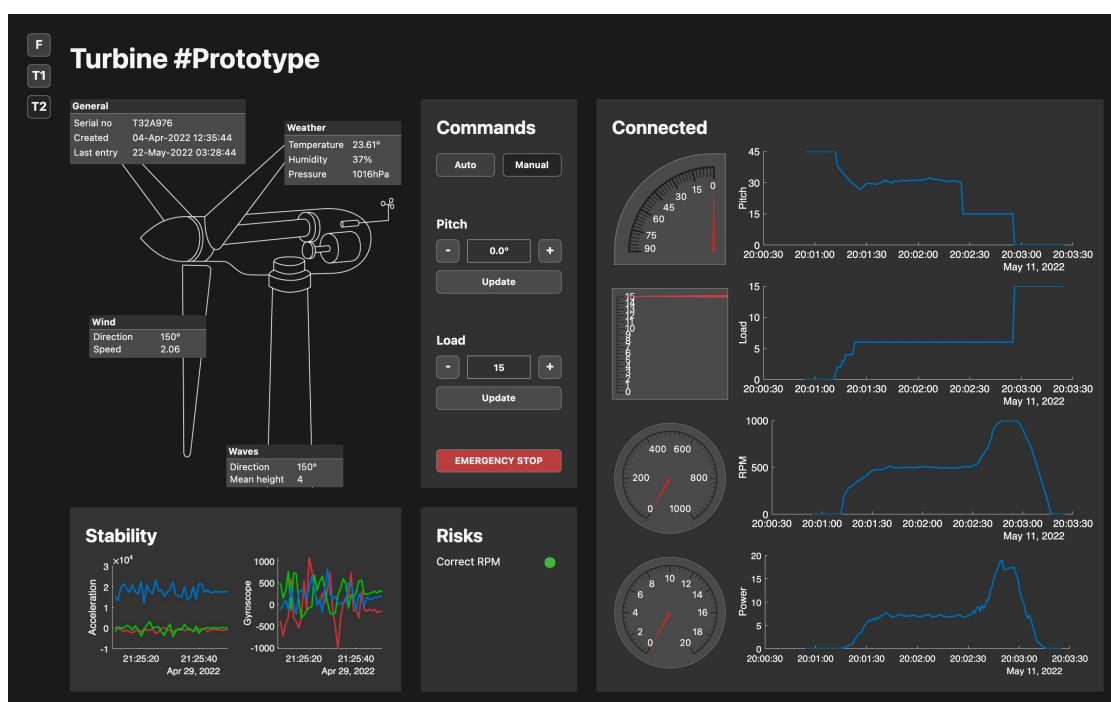


Figura 7.10: Interfaz de la turbina mostrando la parada de emergencia

7.2.2. Análisis del simulador y del Gemelo Digital

Dado que el resultado gráfico es igual al anterior para cada comando de control, solo se mostrará el comportamiento del simulador tras sufrir una modificación en el *pitch*.

El comportamiento de la turbina implementada en el simulador no coincide con exactitud con la del prototipo, por ejemplo, el ángulo óptimo es distinto. El simulador presenta gráficas mucho más suaves pero permite conocer el resultado de aplicar ciertos controles sin correr el riesgo de dañar componentes en un prototipo real. Además, ejecutando muchas simulaciones paralelas, también podría servir para estudiar el comportamiento de grandes granjas.

Como se observa en la Figura 7.11, tras un periodo de arranque en modo de control automático, se aumenta el ángulo de las palas a 30° y las revoluciones y la potencia generada también crecen.

Modificación del pitch

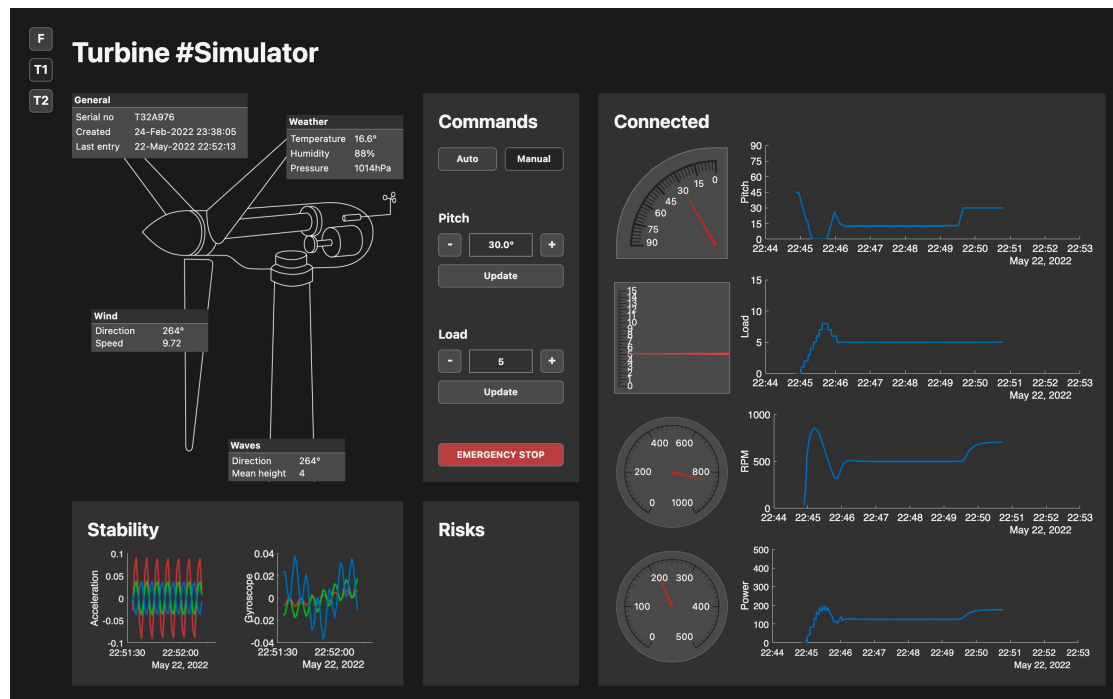


Figura 7.11: Interfaz del simulador mostrando el cambio de pitch manual

7.3. Aportaciones

A lo largo de este capítulo se han expuesto dos casos de uso representativos que pretenden dar una imagen nítida del funcionamiento real de todo el sistema desarrollado para este trabajo. En primer lugar, para mostrar el funcionamiento del control automático del prototipo de turbina y; en segundo lugar, para mostrar la capacidad de control total del Gemelo Digital sobre los componentes del prototipo. También se ha visto el funcionamiento del control manual aplicado al simulador.

Capítulo 8

Conclusiones y trabajos futuros

Este trabajo ha abordado el problema de generar una arquitectura software sólida capaz de dirigir de manera autónoma la estrategia de control del prototipo de una turbina eólica y de mostrar su estado y enviar órdenes de actuación en base al comportamiento observado en tiempo real a través de la figura de un Gemelo Digital. El resultado se ha materializado en lo que sigue.

El software multihilo integrado en el microcontrolador del prototipo se encarga de gestionar las variables de control de la turbina: el *pitch* y la carga eléctrica, y, al mismo tiempo, de muestrear y almacenar en un servidor las variables de seguimiento: las revoluciones por minuto, la potencia generada y las aceleraciones lineal y angular de la base flotante. La estrategia queda determinada por el modo de funcionamiento, automático o manual. El automático es un ejemplo particular de actuación, funciona como una máquina de estados y maneja los componentes de control utilizando técnicas de tipo PID. El manual activa el campo de comandos del Gemelo Digital y las variables de control dependen estrictamente del usuario.

El Gemelo Digital es el programa desarrollado en MATLAB encargado de recopilar y visualizar sobre la interfaz de usuario en tiempo real la información almacenada en el servidor de ThingSpeak y de seleccionar el modo y los parámetros manuales del prototipo de manera remota. Dado su diseño modular, puede albergar varios prototipos o simulaciones.

Cabe destacar algunos aspectos clave por su innovación en la materia o por la originalidad en su implementación:

- El uso de un microcontrolador con varios procesadores para implementar un programa multihilo capaz de ejecutar el control autónomo y de atender comunicaciones de manera bidireccional al mismo tiempo. Esta arquitectura es poco habitual en sistemas IoT económicos de prototipos a pequeña escala.

- La utilización de una cola tipo tubería de UNIX para la comunicación entre procesos de distintos hilos en lugar de variables compartidas. Aparte de conseguir mayor expresividad, generaliza aún más el sistema permitiendo el uso de procesadores que no compartan memoria.
- El desarrollo de un Gemelo Digital basado en programación orientada a objetos con un patrón de diseño Modelo-Vista-Controlador en MATLAB, dirigido a ingenieros de otras ramas familiarizados con la herramienta.
- El diseño de una interfaz fluida y profesional en el software MATLAB, muy poco especializado en ese ámbito.
- El uso del formato de texto JSON para encapsular los datos de un lapso de tiempo y almacenarlos con tan solo un envío en el servidor de ThingSpeak, conservando sus marcas temporales.
- Un simulador Simulink de turbina eólica flotante capaz de ejecutarse y comunicarse en tiempo real con el servidor de ThingSpeak subiendo paquetes de datos encapsulados, de igual forma que el software integrado en el prototipo.

El horizonte del proyecto es conseguir un sistema formado por el prototipo, su software integrado y el Gemelo Digital para realizar pruebas de algoritmos de control. Los trabajos futuros podrían dividirse en distintas ramas dependiendo de las necesidades del cliente.

Una vía de desarrollo podría ser la ampliación del sistema basada en la estructura lógica del resultado actual. Dada la modularidad y ampliabilidad del diseño, añadir nuevos modos de actuación y nuevas variables de control o de monitorización al software integrado y al Gemelo Digital sería una tarea sencilla. Podría incorporarse al Gemelo Digital la posibilidad de modificar el sistema a través de la propia interfaz de usuario sin necesidad de acceder al código fuente.

Otra vía podría reconsiderar la lógica del programa para permitir la carga de los algoritmos a probar a través del propio Gemelo Digital. Profundizando en este caso, y volviendo a hacer uso de todo el entorno de MathWorks, el usuario técnico podría desarrollar el algoritmo de control en Simulink y, apoyados en su integración con MATLAB, enviarlo directamente al prototipo de turbina a través del Gemelo Digital y ThingSpeak, sin necesidad de conocimientos básicos de programación. Por supuesto, esta posibilidad no estaría contemplada de manera oficial por el software de MathWorks pero podría considerarse visto cómo se han sorteado otras limitaciones a lo largo de este desarrollo.

Por último, enfocar el proyecto de forma más específica en la eólica *offshore*. Mejorar las mediciones acelerométricas a través de transformaciones matemáticas como la *Fast Fourier Transform* o los *Filtros de Kalman*, y adaptar el prototipo para realizar las pruebas de control en una piscina con viento y oleaje real.

Chapter 8

Conclusions and future work

This work has addressed the problem of generating a robust software architecture capable of autonomously directing the control strategy of a wind turbine prototype and of displaying its status and sending commands based on the behaviour observed in real time through the figure of a Digital Twin. The result has materialised as follows.

The multithreaded software embedded in the microcontroller of the wind turbine prototype is responsible for managing the control variables of the turbine: pitch and electric charge, and, at the same time, to sample and store the tracking variables on a server: the revolutions per minute, the generated power and the linear and angular acceleration of the floating base. Strategy is determined by the mode of operation, automatic or manual. Auto mode is a particular example of actuation, it operates as a state machine and commands control to the components using PID techniques. Manual mode activates the command fields of the Digital Twin and the control variables are strictly user-dependent.

The Digital Twin is the MATLAB-developed program responsible for collecting and printing on the user interface the sampling information stored on the ThingSpeak server in real time and for selecting the mode and parameters remotely. Due to its modular and scalable design, it can host several prototypes or simulations and it is easy to maintain.

Some key aspects should be highlighted for their innovation in the field or for the originality of their implementation:

- The adoption of a microcontroller with multiple processors to implement a multi-threaded program capable of executing autonomous control and handling bi-directional communications at the same time. This architecture is barely used for cheap, small-scale prototype IoT systems.

- The use of a UNIX pipe-like queue for communication between multi-threaded processes instead of shared variables. Not also for achieving greater expressiveness, but only generalises the system by allowing the use of processors that do not share memory.
- The development of a Digital Twin based on object-oriented programming with a Model-View-Controller design pattern in MATLAB, aimed at engineers from other branches used to the tool.
- The design of a smooth and professional interface in MATLAB software, which is not specialised in this field.
- The use of the JSON text format to encapsulate time lapse data and store it with a single push on the ThingSpeak server, preserving its timestamps.
- A floating wind turbine Simulink simulator capable of running and communicating in real time with the ThingSpeak server by uploading encapsulated data packets, just like the software embedded in the prototype.

The project horizon is to achieve a system with the prototype, its embedded software and the Digital Twin for testing control algorithms. Future work could be divided into different branches depending on the client's needs.

A development pathway would be an extension of the system based on the logical structure of the current result. Given the modularity and expandability of the design, adding new actuation modes and new control or monitoring variables to the embedded software and Digital Twin would be a simple task. It would be interesting to incorporate the possibility of modifying the system of the Digital Twin through the user interface itself without accessing source code.

Another pathway could be to reconsider the program logic and allow to upload the algorithms to be tested through the Digital Twin. Diving deeper into this case, and again using the entire MathWorks environment, the technical user could develop the control algorithm in Simulink and, supported by its integration with MATLAB, send it directly to the turbine prototype through the Digital Twin and ThingSpeak, without even the need for basic programming skills. Of course, this possibility is not officially contemplated by the MathWorks software but could be considered based on how other limitations have been solved throughout this development.

Finally, to focus the project more specifically on offshore wind. Por último, enfocar el proyecto de forma más específica en la eólica offshore. Improving accelerometer measurements through mathematical transformations such as the Fast Fourier Transform or the *Kalman Filters*, and adapt the prototype to carry out control tests in a pool with real wind and waves.

Bibliografía

- [1] «Learn More About ThingSpeak,» ThingSpeak, 2022, última vez accedido 10 de abril 2022. [En línea]. Disponible en: https://thingspeak.com/pages/learn_more
- [2] R. S. Pressman, *Ingeniería del Software. Un Enfoque Práctico*, 2010, p. 777.
- [3] J. E. Sierra García, «Control inteligente de aerogeneradores.» Comité Español de Automática, 10 2021.
- [4] R. Pandit, «Challenges and Opportunities for AI and Data analytics in Offshore wind.» Universidad Complutense de Madrid, 11 2021.
- [5] J. E. Sierra García, «Redes neuronales y reinforcement learning. Aplicación en energía eólica.» Universidad de Burgos, 11 2021.
- [6] «Las energías renovables superan a los combustibles fósiles y pasan a ser la principal fuente de energía de la UE,» *Estado de la Unión de la Energía 2021*, última vez accedido 2 de mayo 2022. [En línea]. Disponible en: https://ec.europa.eu/commission/presscorner/detail/es/ip_21_5554
- [7] «Las energías renovables superan a los combustibles fósiles y pasan a ser la principal fuente de energía de la UE,» *Estado de la Unión de la Energía 2021*, última vez accedido 2 de mayo 2022. [En línea]. Disponible en: https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Renewable_energy_statistics
- [8] «Onshore and offshore wind,» *European Comission*, última vez accedido 2 de mayo 2022. [En línea]. Disponible en: https://energy.ec.europa.eu/topics/renewable-energy/onshore-and-offshore-wind_en
- [9] «Making green energy affordable,» Orsted, última vez accedido 7 de mayo 2022. [En línea]. Disponible en: <https://orsted.com/-/media/WWW/Docs/Corp/COM/explore/Making-green-energy-affordable-June-2019.pdf>

- [10] «Wind energy,» International Renewable Energy Agency, última vez accedido 8 de mayo 2022. [En línea]. Disponible en: <https://www.irena.org/wind>
- [11] «Offshore wind to become a \$1 trillion industry,» International Energy Agency, 10 2019, última vez accedido 7 de mayo 2022. [En línea]. Disponible en: <https://www.iea.org/news/offshore-wind-to-become-a-1-trillion-industry>
- [12] «Floating offshore wind vision statement,» Wind Europe, p. 6, 6 2017, última vez accedido 7 de mayo 2022. [En línea]. Disponible en: <https://windeurope.org/wp-content/uploads/files/about-wind/reports/Floating-offshore-statement.pdf>
- [13] L. Romero Lozano, *Gestión del montaje de parques eólicos*. Paraninfo, 2017.
- [14] X. Wu, Y. Hu, Y. Li, J. Yang, L. Duan, T. Wang, T. Adcock, Z. Jiang, Z. Gao, Z. Lin, A. Borthwick, y S. Liao, «Foundations of offshore wind turbines: A review,» *Renewable and Sustainable Energy Reviews*, vol. 104, pp. 379–393, 2019.
- [15] A. L. Wai Hou, *Blade-Pitch Control for Wind Turbine Load Reductions*. Springer, 2018.
- [16] F. D. Bianchi, H. De Battista, y R. J. Mantz, *Wind Turbine Control Systems*. Springer, 2007.
- [17] R. Paz, «The Design of the PID Controller,» 01 2001.
- [18] M. Tomas-Rodriguez y M. Santos, «Modelado y control de turbinas eólicas marinas flotantes,» *Revista Iberoamericana de Automática e Informática industrial*, vol. 16, p. 381, 09 2019.
- [19] T. Salic, J. Charpentier, M. Benbouzid, y M. Boulluec, «Control Strategies for Floating Offshore Wind Turbine: Challenges and Trends,» *Electronics*, vol. 8, 10 2019.
- [20] M. Grieves y J. Vickers, *Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems*, 08 2017, pp. 85–113.
- [21] «¿Qué es un Gemelo Digital?» IBM, última vez accedido 15 de abril 2022. [En línea]. Disponible en: <https://www.ibm.com/es-es/topics/what-is-a-digital-twin>
- [22] S. Ferguson, «Apollo 13: The First Digital Twin,» Siemens, 04 2020, última vez accedido 15 de abril 2022. [En línea]. Disponible en: <https://blogs.sw.siemens.com/simcenter/apollo-13-the-first-digital-twin/>

- [23] M. Shafto, M. Conroy, R. Doyle, E. Glaessgen, C. Kemp, J. LeMoigne, y L. Wang, «Modeling, simulation, information technology & processing road-map,» *National Aeronautics and Space Administration*, vol. 32, no. 2012, pp. 1–38, 2012.
- [24] «Gartner Survey Reveals Digital Twins Are Entering Mainstream Use,» Gartner, 02 2019, última vez accedido 17 de abril 2022. [En línea]. Disponible en: <https://www.gartner.com/en/newsroom/press-releases/2019-02-20-gartner-survey-reveals-digital-twins-are-entering-mainstream>
- [25] G. R. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming*, 1999, p. 665.
- [26] «Comparison of MATLAB and Other OO Languages,» MathWorks, 2022, última vez accedido 3 de mayo 2022. [En línea]. Disponible en: https://es.mathworks.com/help/matlab/matlab_oop/matlab-vs-other-oo-languages.html
- [27] «Property Access Methods,» MathWorks, 2022, última vez accedido 4 de mayo 2022. [En línea]. Disponible en: https://es.mathworks.com/help/matlab/matlab_oop/property-access-methods.html
- [28] J. Correas, «El patrón Modelo-Vista-Controlador,» en *Tecnología de la Programación*. Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2018.
- [29] M. Kalelkar, P. Churi, y D. Kalelkar, «Implementation of Model-View-Controller Architecture Pattern for Business Intelligence Architecture,» *International Journal of Computer Applications*, vol. 102, pp. 16–21, 09 2014.
- [30] «Events,» MathWorks, 2022, última vez accedido 18 de marzo 2022. [En línea]. Disponible en: <https://es.mathworks.com/help/matlab/events-sending-and-responding-to-messages.html?lang=en>
- [31] «Timetable,» MathWorks, 2022, última vez accedido 20 de marzo 2022. [En línea]. Disponible en: <https://es.mathworks.com/help/matlab/ref/timetable.html>
- [32] «ThingSpeakRead,» MathWorks, 2022, última vez accedido 1 de abril 2022. [En línea]. Disponible en: <https://es.mathworks.com/help/thingspeak/thingspeakread.html>
- [33] «ThingSpeakWrite,» MathWorks, 2022, última vez accedido 1 de abril 2022. [En línea]. Disponible en: <https://es.mathworks.com/help/thingspeak/thingspeakwrite.html>

- [34] «Current weather data,» OpenWeather, 2022, última vez accedido 5 de abril 2022. [En línea]. Disponible en: <https://openweathermap.org/current>
- [35] «Matlab.mixin.Heterogeneous class,» MathWorks, 2022, última vez accedido 19 de abril 2022. [En línea]. Disponible en: <https://es.mathworks.com/help/matlab/ref/matlab.mixin.heterogeneous-class.html>
- [36] «Timer,» MathWorks, 2022, última vez accedido 2 de abril 2022. [En línea]. Disponible en: https://es.mathworks.com/help/matlab/ref/timer.html?s_tid=srchtitle_timer_1
- [37] G. A. Andrade Aimara, «Modelo a escala de aerogenerador para pruebas de control,» Trabajo de Fin de Grado, Dto. de Arquitectura de Computadores y Automática, Facultad de Ciencias Físicas, Universidad Complutense de Madrid, Madrid, 2022.
- [38] M. Fernández de Diego y B. Sánchez Centeno, «Turbine controller software,» <https://github.com/martinfdezdg/Affordable-Architecture-to-Monitor-and-Control-Offshore-Turbines>, 2022.