Procesamiento de señales de encefalograma para la detección de ataques epilépticos.

Trabajo fin de grado

Autor: José Antonio Verde Jiménez
Tutor: José Daniel García Sánchez

13 de octubre de 2025

Grado en Ingeniería Informática – Universidad Carlos III de Madrid

1. Introducción

- Colaboración entre dos grupos de investigación (PPMC y ARCOS) de dos universidades (UMA y UC3M).
- Tema interesante y relevante.
  - Optimización.
  - Paralelismo.
  - Sistema en tiempo real.
- Un reto personal.
  - Aplicar lo aprendido.
  - Resolver un problema difícil.
  - Investigar.
- Aportar algo.

Figura 1: PPCM y ARCOS

1.2 Objetivos del proyecto

Detectar en una señal de encefalograma qué épocas pertenecen a un ataque epiléptico y optimizar el código que lo hace.

1. Módulo de Python 3 escrito en C++ para minimizar el tiempo de entrenamiento.
2. Sistema en tiempo real duro en un dispositivo empotrado.
    1. RISC-V
    2. Bajo consumo
    3. **ESP32C3**
3. Verificación formal.

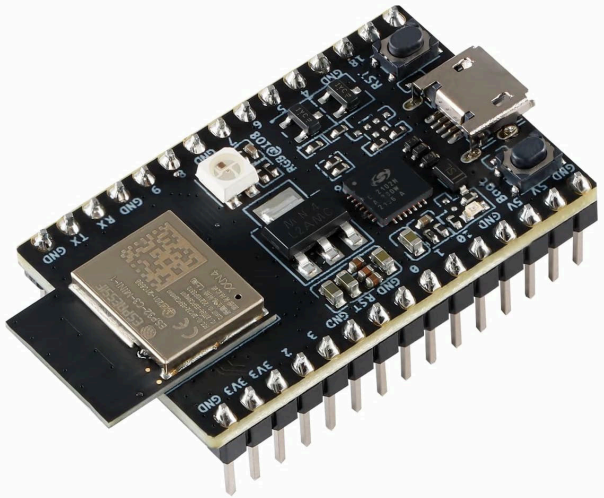

Figura 2: ESP32C3

2. Estado de la cuestión

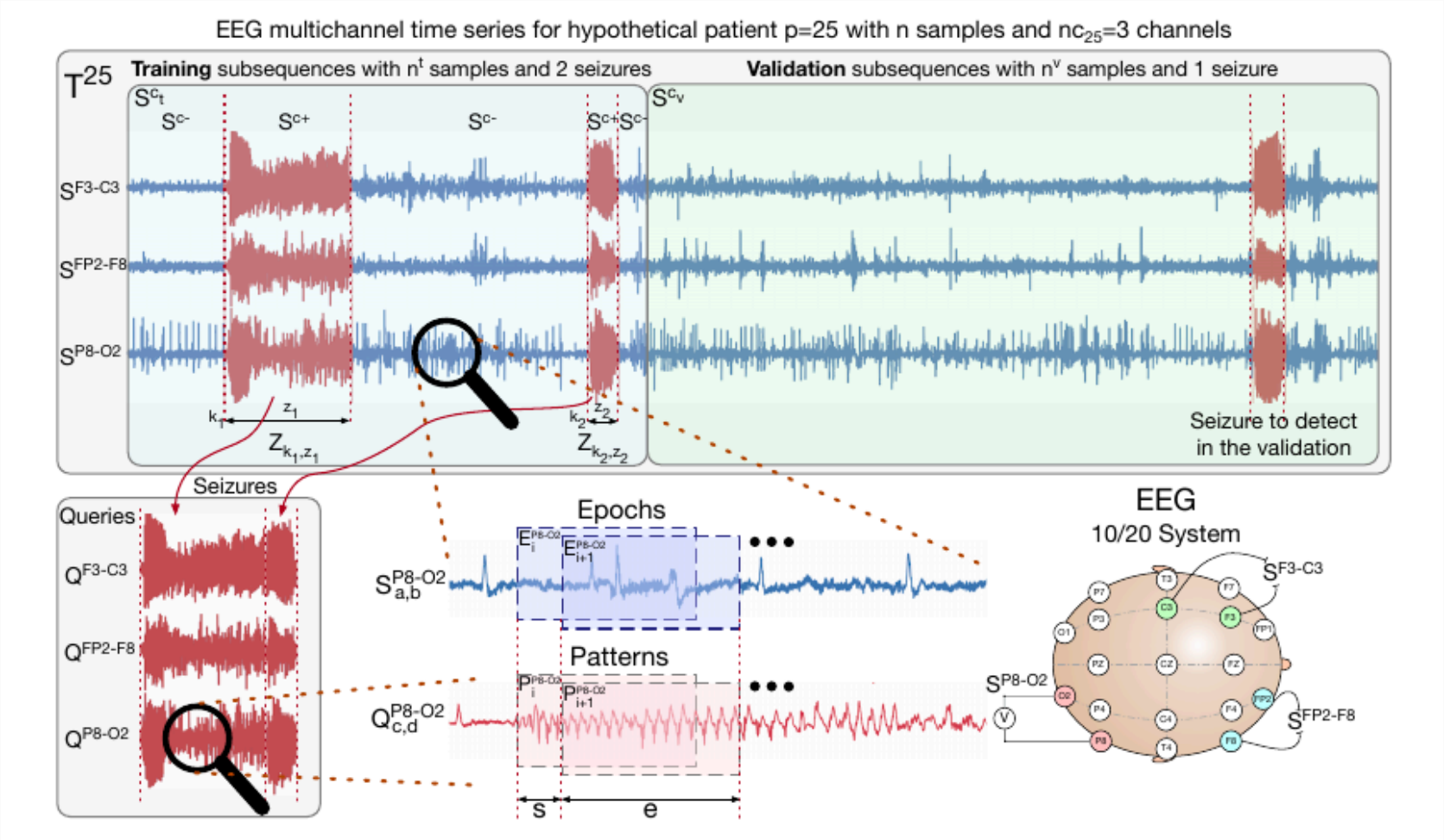

Figura 3: PaFESD: Patterns augmented by Features Epileptic Seizure Detection

- **Rangos**
  - `range-v3`
  - `flux`
  - Estándar
- **Paralelismo**:
  - Hilos (C++ y Python 3) y tareas (Ada).
  - OneTBB.
  - OpenMP.
  - C++ y políticas de ejecución.
- **Diseño por contrato**
  - Ada y SPARK
  - Eiffel
  - * C++ 26
- **Demostración interactiva de teoremas**
  - Rocq
  - Lean
  - SPARK

3. Análisis

3.1 Casos de uso y requisitos

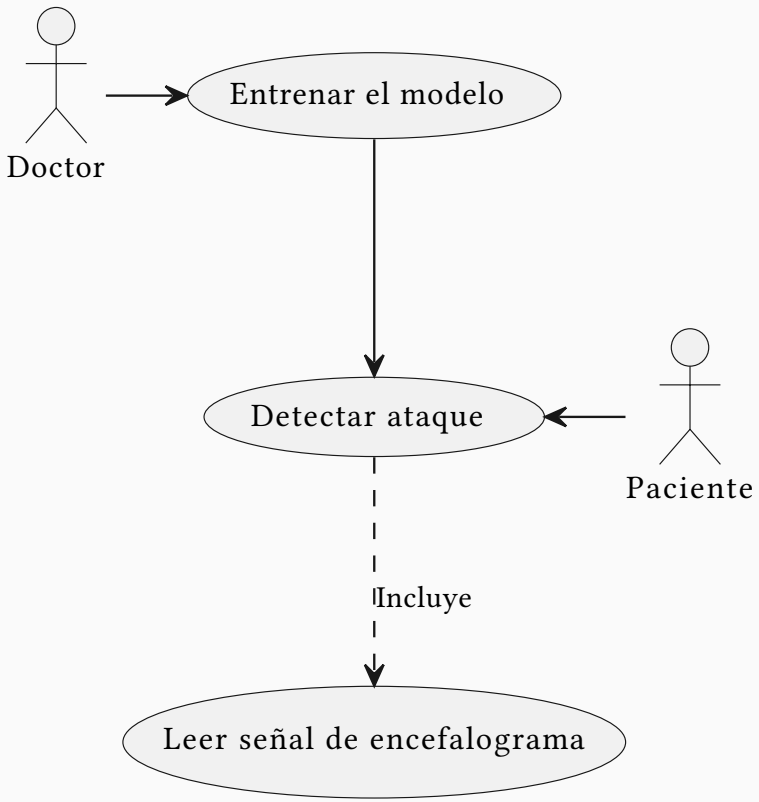

Figura 4: Casos de uso

	Leer señal de encefalograma	Detectar ataque	Entrenar el modelo
RF-01			✓
RF-02		✓	
RF-03		✓	
RF-04	✓		
RF-05			✓

	Leer señal de encefalograma	Detectar ataque	Entrenar el modelo
RN-01	✓		
RN-02	✓		
RN-03		✓	
RN-04	✓	✓	
RN-05			✓
RN-06			✓
RN-07			✓
RN-08			✓
RN-09			✓
RN-10			✓
RN-11			✓
RN-12		✓	
RN-13	✓	✓	

Figura 5: Requisitos funcionales y requisitos no funcionales

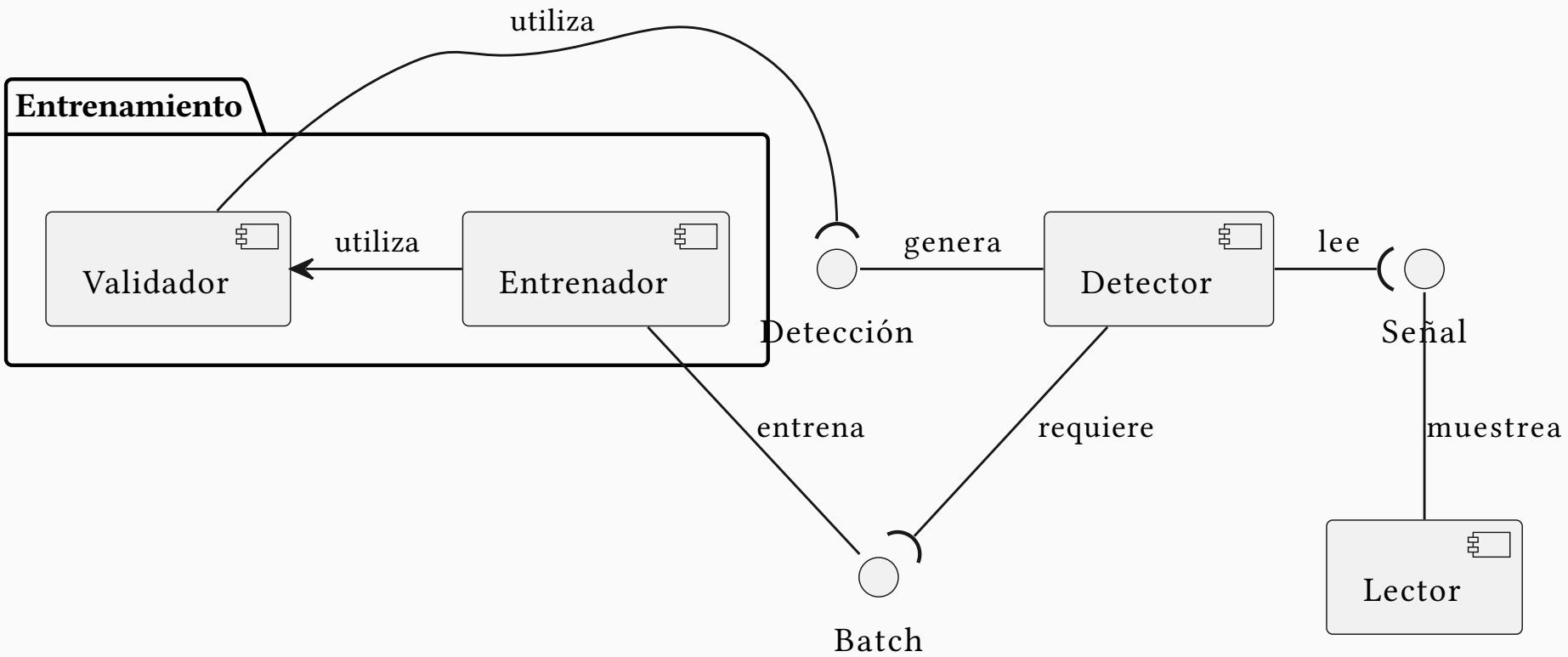

Figura 6: Arquitectura

4. Implementación en C++

```C++
PYBIND11_MODULE(signals, m) {
  m.doc() = "Signals library for Seizure Algorithm";
  m.def("simpson", simpson, pybind11::arg("y"), pybind11::arg("dx"));
  m.def("welch", welch);
  m.def("call_psd", call_psd);
  m.def("call_max_dist", call_max_dist);
  m.def("call_energy", call_energy);
  m.def("begin", optimization_step_begin, pybind11::arg("data_all"),
        pybind11::arg("stride"), pybind11::arg("ops"),
        pybind11::arg("query_all"), pybind11::arg("qstride"),
        pybind11::arg("qops"), pybind11::arg("mw"));
```

Listado 1: Integración de C++ en Python 3 con `pybind11`.

```C++
#include <range/v3/all.hpp>
constexpr auto sliding_window_view(auto size, auto stride) {
  return ranges::views::sliding(size) | ranges::views::stride(stride);
}
constexpr auto chunk_split_view(auto && view, auto size) {
  return view | ranges::views::chunk(view.size() / size + 1);
}

// Extracto de validator.hpp
auto window_view     = scv_view | sliding_window_view(epoch, stride);
auto indices         = std::views::iota(from, to);
for (auto && [epoch, idx] : ranges::views::zip(window_view, indices)) {
  // ...
}
```

Listado 2: Vistas y rangos en C++.

Función	**Muestras**	**Python 3**		**C++**		***Speed-up***
		μ	σ	μ	σ	
Simpson	1159	0.1335 s	0.0109 s	0.0219 s	0.0018 s	$\times$ **6.1**
Welch	109	1.398 s	0.0323 s	0.2598 s	0.0049 s	$\times$ **5.4**
PSD	18	9.9002 s	0.1139 s	0.1596 s	0.0049 s	$\times$ **62**
Energy	172	1.0355 s	0.0298 s	0.0152 s	0.0024 s	$\times$ **68.1**
Max Distance	41	4.4274 s	0.0728 s	0.0304 s	0.0044 s	$\times$ **145.6**
Subrutina	12	15.1082 s	0.1109 s	0.2664 s	0.0398 s	$\times$ **56.7**

Tabla 1: *Slimbook*: 20 hilos

12th Gen Intel i7-12700H (20) @ 4.600GHz con 20 hilos lógicos.

Función	**Muestras**	**Python 3**		**C++**		***Speed-up***
		μ	σ	μ	σ	
Simpson	831	0.1845 s	0.0024 s	0.0321 s	0.0008 s	$\times$ **5.7**
Welch	75	2.0053 s	0.0048 s	0.4095 s	0.0022 s	$\times$ **4.9**
PSD	53	3.3504 s	0.2755 s	0.0704 s	0.0069 s	$\times$ **47.6**
Energy	56	3.2378 s	0.6921 s	0.0129 s	0.0021 s	$\times$ **251**
Max Distance	60	3.0195 s	0.7865 s	0.0148 s	0.0053 s	$\times$ **204**
Subrutina	21	8.5647 s	1.4619 s	0.0997 s	0.0192 s	$\times$ **85.9**

Tabla 2: Servidor: 64 hilos

Intel Xeon Gold 6326 (64) @ 3.500GHz con 64 hilos lógicos.

Lenguaje	Compilador	Máquina	épocas/s
C++	GCC 14	ESP32C3	1.64
C++	Clang 18	RPi 4 (32 bits)	470
C++	GCC 12	RPi 3 (64 bits)	217
C++	Clang 18	Slimbook	3036

Tabla 3: Épocas por segundo con tipo flotante IEEE de 32 bits en C++.
Peor caso con 3 patrones

- **1.64** épocas por segundo da tiempo real, pero se puede hacer mejor.
- En el futuro habría que limpiar la señal, filtrar artefactos.

5. Implementación en Ada

5.1 ¿Por qué?

- Pese a ser en general más lento que C++.
- Prototipo rápido.
- Punto fijo en el estándar.
  - GNAT pertenece al juego de compiladores de GCC
  - Intrínsecos del compilador para punto fijo.
- **Verificación formal del algoritmo.**
  - Comportamiento no definido en C++ es comportamiento erróneo en Ada.
  - Contratos en todas las operaciones.
  - **SPARK**.

Lenguaje	***Checks***	**Compilador**	**Máquina**	**épocas/s**
Ada	Desactivados	GNAT 14	ESP32C3	0.96
Ada	Activados	GNAT 14	ESP32C3	0.94
Ada	Desactivados	GNAT 10	RPi 4 (32 bits)	286
Ada	Activados	GNAT 10	RPi 4 (32 bits)	280
Ada	Desactivados	GNAT 12	RPi 3 (64 bits)	199
Ada	Activados	GNAT 12	RPi 3 (64 bits)	147
Ada	Desactivados	GNAT 14	Slimbook	1725
Ada	Activados	GNAT 14	Slimbook	1433

Tabla 4: Épocas por segundo con tipo flotante IEEE de 32 bits en Ada. Peor caso con 3 patrones

5.3 Pruebas preliminares (punto fijo)

Checks	Tipo	épocas/s
Activados	$\mathbb{X}_{32,-8}$	`Constraint_Error`
Activados	$\mathbb{X}_{64,-16}$	1.08
Desactivados	$\mathbb{X}_{32,-8}$	**21.04**
Desactivados	$\mathbb{X}_{64,-16}$	1.12

Tabla 5: Prueba preliminares de rendimiento de punto fijo en Ada.

Técnicamente se podrían procesar alrededor de
21 épocas por segundo

Técnicamente se podrían procesar alrededor de 21 épocas por segundo

- *Después de eliminar los errores*

5.4 Problemas

- Desbordamiento y subdesbordamiento de enteros con signo.
  - En C++ es comportamiento no definido (*undefined behaviour*).
  - En Ada es `Constraint_Error`.
- Precisión
  - 32 bits: `float` $(23 + 1 + 1)$ y `double` $(52 + 1 + 1)$.
  - Variabilidad del exponente.
  - Productos y divisiones usan el doble de bits.

5.4 Problemas

- Desbordamiento y subdesbordamiento de enteros con signo.
  - En C++ es comportamiento no definido (*undefined behaviour*).
  - En Ada es `Constraint_Error`.
- Precisión
  - 32 bits: `float` $(23 + 1 + 1)$ y `double` $(52 + 1 + 1)$.
  - Variabilidad del exponente.
  - Productos y divisiones usan el doble de bits.
  - **Divisiones (49.2% del tiempo total en punto fijo de 64 bits).**

5.4 Problemas

- Desbordamiento y subdesbordamiento de enteros con signo.
  - En C++ es comportamiento no definido (*undefined behaviour*).
  - En Ada es `Constraint_Error`.
- Precisión
  - 32 bits: `float` $(23 + 1 + 1)$ y `double` $(52 + 1 + 1)$.
  - Variabilidad del exponente.
  - Productos y divisiones usan el doble de bits.
  - **Divisiones (49.2% del tiempo total en punto fijo de 64 bits).**
- Errores
  - En C++ son silenciosos.
  - En Ada lanzan excepciones. No capturables con la configuración de la biblioteca de tiempo de ejecución (*runtime*) del dispositivo empotrado (`No_Exception_Propagation`).

SPARK

5.5 Soluciones

- Subconjunto de Ada con anotaciones adicionales.
- Probadores de teoremas: `Alt-Ergo`, `Colibri`, `cvc5`, `Z3`, `Coq`...
- Uniformizar valores, porque: $x, y \in (-1, 1) \rightarrow xy \in (-1, 1)$.
- Hacer iterativa la transformada de Fourier.
- Punto fijo de 64 bits de manera dispersa.
- SPARK *Gold*: ***Division***, ***Index***, ***Length***, ***Overflow***, ***Range***, *Tag, Elaboration and* ***Flow*** *checks.*

Compilador	***Checks***	**Máquina**	**épocas/s**
GNAT 14	Desactivados	ESP32C3	**10.36**
GNAT 14	Activados	ESP32C3	7.68
GNAT 10	Desactivados	RPi 4 (32 bits)	154
GNAT 10	Activados	RPi 4 (32 bits)	104
GNAT 12	Desactivados	RPi 3 (64 bits)	222
GNAT 12	Activados	RPi 3 (64 bits)	127
GNAT 14	Activados	Slimbook	2139
GNAT 14	Desactivados	Slimbook	2195

Tabla 6: Pruebas finales de rendimiento de punto fijo en SPARK + Ada.

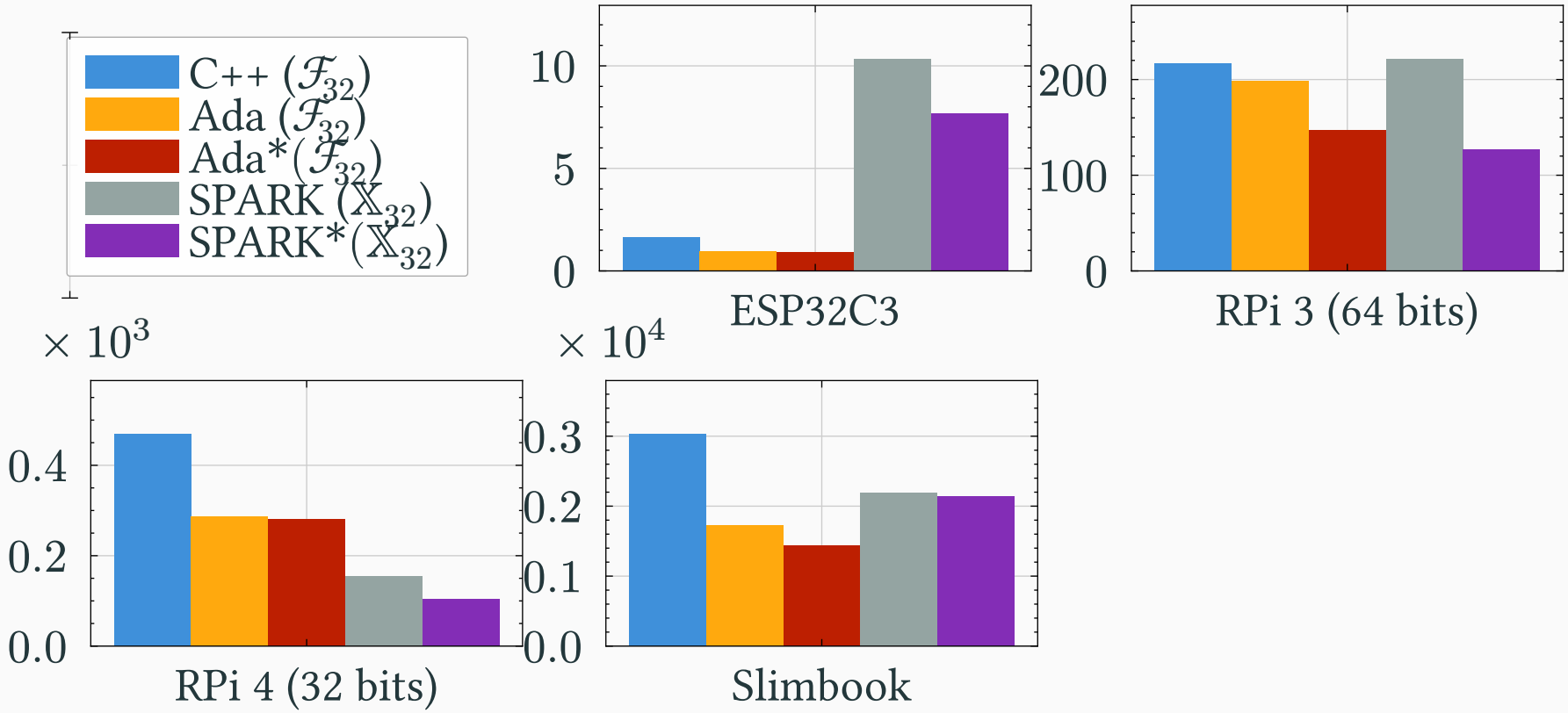

Figura 7: Comparación final de implementaciones. El eje de ordenadas indica el número de épocas por segundo. El asterisco indica que se ha compilado con comprobaciones en tiempo de ejecución activadas.

SPARK Analysis results	Total	Flow	Provers	Justified	Unproved
Data Dependencies	41	41	.	.	.
Flow Dependencies	.	.	.	.	.
Initialization	34	34	.	.	.
Non-Aliasing	3	3	.	.	.
Run-time Checks	618	.	608	4	6
Assertions	125	.	123	1	1
Functional Contracts	68	.	57	8	3
LSP Verification	.	.	.	.	.
Termination	44	41	3 (CVC5)	.	.
Concurrency	.	.	.	.	.
Total	933	119 (13%)	791 (85%)	13 (1%)	10 (1%)

Tabla 7: Estadísticas del análisis de SPARK

6. Entorno socioeconómico

Concepto	**Coste**
Recursos humanos	6908.46 €
Recursos materiales	369.20 €
Costes indirectos	5175.00 €
Total del proyecto	**12452.66 €**
Beneficio industrial (16%)	1992.43€
IVA (21%)	2615.06€
Importe final	**17478.56 €**

Tabla 8: Coste total

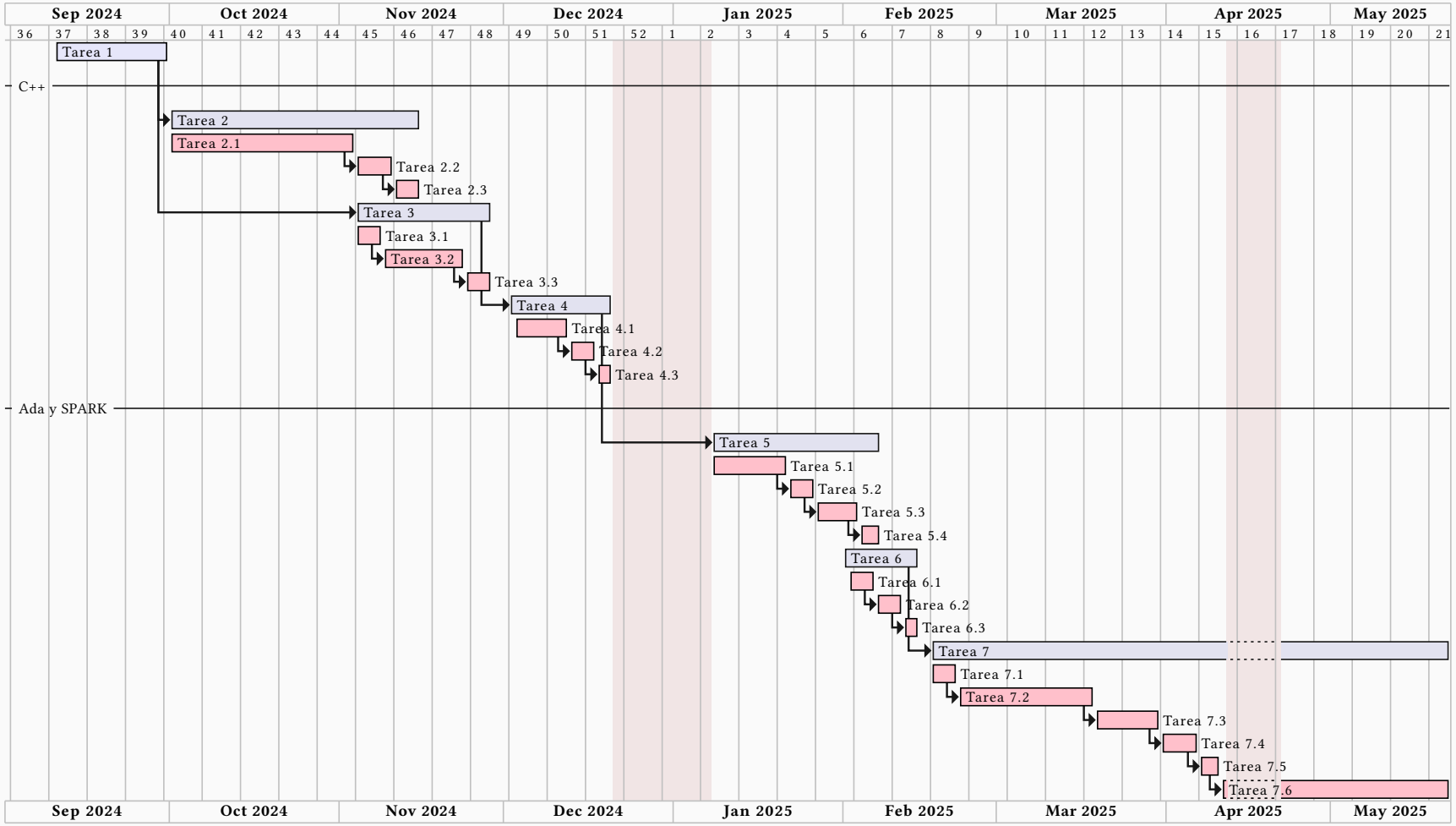

Figura 8: Diagrama Gantt del proyecto

7. Conclusiones

7.1 Conclusiones del proyecto

Objetivos cumplidos:

1. Entrenamiento más rápido. Visto bueno del equipo de investigación de la UMA.
2. Tiempo real.
   - ESP32C3
     - C++: 1.64 épocas por segundo
     - Ada: 10.36 épocas por segundo
   - Dependiendo de la calidad de la FPU mejora el rendimiento de punto flotante (vectorización, optimizaciones, ...).
   - A falta de FPU, punto fijo puede llegar a ser más rápido que punto flotante.
   - Trabajar con punto fijo es tedioso y requiere demostraciones.
3. Exceptuando Welch y la transformada de Fourier, se ha verificado formalmente todo el proyecto con un probador de teoremas.

7.2.1 Conclusiones personales

- Varios lenguajes de programación.
- SPARK en un proyecto real.
- Aprendido
  - Compilación cruzada
  - Probadores de teoremas
  - Dispositivos empotrados

7.2.2 Trabajo futuro

- Terminar de demostrar funciones.
- Reentrenamiento en tiempo de ejecución.
- Limpiado de señal y artefactos.
- Publicación de resultados.

¡Muchas gracias por su atención!