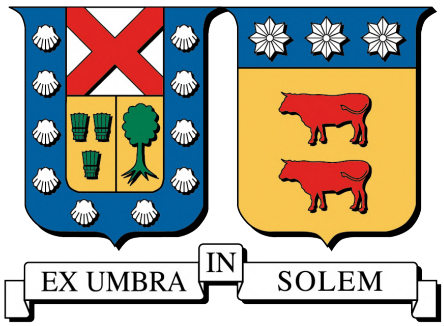
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE ELECTRÓNICA
VALPARAÍSO - CHILE



EVALUACIÓN DE HERRAMIENTAS DE GENERACIÓN DE CÓDIGO PARA PLATAFORMAS EMBEBIDAS UTILIZANDO MATLAB

JARED ISRAEL SOTO PEREZ

MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN ELECTRÓNICA

PROFESOR GUÍA	:	GONZALO CARVAJAL
PROFESOR CORREFERENTE	:	CÉSAR SILVA
PROFESOR CORREFERENTE	:	JUAN CARLOS AGÜERO

NOVIEMBRE 2022

Resumen

La utilización de descripciones de alto nivel para el prototipado rápido de algoritmos es una práctica recurrente tanto en la industria como academia, debido a la abstracción y herramientas proporcionadas que facilitan el desarrollo, depuración y validación de algoritmos. Sin embargo, la gran mayoría de prototipos precisan ser implementados en sistemas físicos que operan con plataformas embebidas que no cuentan con los recursos para ejecutar descripciones de alto nivel. En consecuencia, una herramienta que facilite y reduzca la brecha que existe entre el traspaso de descripciones de alto nivel a sistemas embebidos, trae mejoras en los tiempos de desarrollo y productividad de quien las utiliza, permitiendo además reducir al error humano al sistematizar este proceso.

En particular, dado que el enfoque de esta memoria está en ser un aporte en proyectos del ámbito académico, es que se centra en explorar la utilidad efectiva de Embedded Coder, una de las herramientas que entrega Matlab para la generación automática de código en C a partir de descripciones realizadas en scripts de código o diagramas de bloques. En específico, en esta memoria se propone y valida una metodología sistemática para la utilización e implementación de los códigos generados por Embedded Coder, utilizando como ejemplos distintos casos de estudio que son implementados y ejecutados en un computador de escritorio y una gama de plataformas embebidas con diferentes sistemas operativos. Para uno de los casos de estudio, particularmente relevante para proyectos en curso en el Departamento de Electrónica, se evalúan los tiempos de ejecución obtenidos para la ejecución del código generado en cada plataforma. Los códigos fuente e información necesaria para reproducir y extender los experimentos reportados en este informe se encuentran disponibles en un repositorio de código.

Dado los resultados experimentales obtenidos a lo largo del desarrollo de esta memoria, es posible concluir que Embedded Coder es una herramienta útil para la generación de código C optimizado para plataformas embebidas con una equivalencia funcional a descripciones descritas en Matlab.

Índice general

1. Introducción **5**
1.1. Motivación y contexto 5
1.2. Planteamiento del problema 8
1.3. Alcances y contribuciones 8
1.4. Organización del informe 9

2. Antecedentes **11**
2.1. Diseño basado en modelos y herramientas de generación de código en la industria 11
2.2. Herramientas de MATLAB para generación de código 13
2.2.1. Matlab Coder 13
2.2.2. Simulink coder 15
2.2.3. Embedded Coder 15
2.2.4. Discusión 16

3. Metodología y desarrollo **18**
3.1. Aplicaciones de ejemplo 18
3.2. Flujo de trabajo 19
3.3. Plataformas objetivo y configuraciones 21
3.4. Implementaciones 22

4. Resultados experimentales **24**
4.1. Validación funcional de códigos generados 24
4.2. Evaluación de ADMM en distintas plataformas. 25
4.2.1. Errores numéricos 26
4.2.2. Caracterización tiempos de ejecución 28

5. Conclusiones y trabajo futuro **34**

6. Anexo **38**

Índice de tablas

3.1. Información de los distintos entornos de prueba. 22

4.1. Estadísticas de error realtivo y tiempo de ejecución del código generado a partir de distintos algoritmos. 25
4.2. Estadísticas de error relativo un computador de escritorio con Windows. 28
4.3. Resultados de la caracterización temporal de las plataformas. 30

Índice de figuras

2.1. Flujo en la generación de código. 17

3.1. Flujo de trabajo. 23

4.1. Tiempos de ejecución a partir de la ejecución del algoritmo ADMM(8x8) en una Raspberry 4B. 31
4.2. Tiempos de ejecución a partir de la ejecución del algoritmo ADMM(8x8)ADMM(8x8) en una BeagleBone AI. 31
4.3. Tiempos de ejecución a partir de la ejecución del algoritmo ADMM(4x4) en una tarjeta STM32f7xx con FreeRTOS 32
4.4. Tiempos de ejecución a partir de la ejecución del algoritmo ADMM(4x4) en una tarjeta STM32f7xx con Bare Metal. 33

6.1. Tiempos de ejecución a partir de la ejecución del algoritmo ADMM(4x4) en un Computador de escritorio con Linux. 38
6.2. Tiempos de ejecución a partir de la ejecución del algoritmo ADMM(4x4) en un Computador de escritorio con Windows. 39
6.3. Tiempos de ejecución a partir de la ejecución del algoritmo ADMM(4x4) en una Raspberry Pi 3B. 40
6.4. Tiempos de ejecución a partir de la ejecución del algoritmo ADMM(4x4) en una Raspberry Pi 4B. 41
6.5. Tiempos de ejecución a partir de la ejecución del algoritmo ADMM(4x4) en una BeagleBone AI. 42
6.6. Tiempos de ejecución a partir de la ejecución del algoritmo ADMM(4x4) en una STM32f7xx y stm32f4xx con freeRTOS. 43
6.7. Tiempos de ejecución a partir de la ejecución del algoritmo ADMM en una STM32f4xx Bare Metal. 44
6.8. Tiempos de ejecución a partir de la ejecución del algoritmo ADMM(4x4) en una STM32f7xx Bare Metal. 45

1 Introducción

El trabajo realizado en esta memoria de título tiene como finalidad entregar resultados cualitativos y cuantitativos que validen la usabilidad de la herramienta Embedded Coder de Matlab para la generación de código automático en C a partir de descripciones de alto nivel en scripts .m o en bloques de Simulink [1]. En específico, este trabajo apunta a complementar la documentación base entregada por Matlab, definiendo y validando un flujo de trabajo y metodología para la utilización de Embedded Coder, utilizando ejemplos concretos que ilustren el proceso de pasar algoritmos descritos a alto nivel a descripciones equivalentes en lenguaje C que permitan su ejecución, validación, y caracterización de tiempo de ejecución en plataformas embebidas de distinta gama.

Se espera que este trabajo sea de utilidad para trabajos de investigación y desarrollo actualmente en ejecución en el Departamento de Electrónica que requieren simplificar el proceso de prototipado rápido de algoritmos avanzados para distintas áreas de aplicación, considerando entornos de ejecución realistas asociados a su implementación en plataformas embebidas.

1.1. Motivación y contexto

Matlab es una plataforma de programación y modelamiento numérico ampliamente utilizada en la academia para la creación de modelos, desarrollo de algoritmos y análisis de datos [2]. Matlab provee distintos entornos de programación que permite la implementación y verificación de algoritmos y sistemas a distintos niveles de abstracción que incluyen la descripción de código en lenguaje propietario (.m) hasta entornos gráficos que permiten la programación por bloques (Simulink, StateFlow, etc.). Además, Matlab provee una amplia disponibilidad de *toolboxes* con herramientas especializadas para distintos campos de aplicación, lo cual provee a ingenieros y científicos una vía de prototipado rápido para sus algoritmos, enfocándose en la aplicación en lugar de los detalles de la programación.

Si bien las herramientas que ofrece Matlab resultan bastante útiles para la verificación funcional de algoritmos a alto nivel, en general las capas de abstracción superior vienen con el costo de ofrecer poco control sobre el uso de recursos computacionales que permitan aumentar la eficiencia de ejecución del código. Por ejemplo, Matlab ofrece herramientas que facilitan el modelamiento, descripción, y verificación de sistemas de control automático, lo cual permite evaluar y sintonizar técnicas de control avanzado en forma rápida [3, 4]. Sin embargo, los códigos desarrollados en Matlab no son directamente portables a plataformas embebidas típicamente utilizadas en los sistemas físicos donde las tareas de control son aplicadas. En general, los sistemas computacionales embebidos se diferencian de los entornos de escritorio en que cuentan

con recursos computacionales limitados (en términos de memoria y capacidad del procesador), operan con sistemas operativos restringidos o incluso sin sistema operativo (configuración *bare metal*), y requieren garantías temporales para las acciones de control.

Hoy en día, el lenguaje dominante en el desarrollo de sistemas embebidos es C, siendo algunas de las razones para esto las siguientes:

- C provee un nivel de abstracción que está bastante cercano al hardware del procesador, entregando un control fino sobre los recursos del procesador sin necesidad de programar directamente en assembly. Esto es beneficioso tanto en términos de reducir los tiempos de ejecución como para el uso determinístico de recursos, siendo esto último critico para dispositivos que suelen estar limitados en este aspecto.
- El tamaño de los archivos ejecutables obtenidos a partir de C suele ser menor que la descripción equivalente que se tendría en lenguajes de más alto nivel, por lo que los requerimientos para almacenamiento en memoria suelen ser bajos. Además, los programas obtenidos a partir de C pueden ejecutarse directamente en el procesador, requiriendo nulo o muy poco soporte de un sistema operativo en tiempo de ejecución, a diferencia de lenguajes de alto nivel que suelen requerir soporte de sistema operativo y/o intérpretes.
- Existe una completa y madura infraestructura en torno al desarrollo en C (compiladores, depuradores, analizadores de código, entornos de desarrollo integrado, etc.) para plataformas embebidas de distinta gama y con distintas arquitecturas de procesador. Prácticamente cada fabricante de microprocesadores, incluyendo los de muy bajo costo, provee un entorno de desarrollo para códigos en C. En el caso de que un proyecto este escrito en C y sea necesario cambiar de microprocesador, es altamente probable que el código pueda ser utilizado en el nuevo hardware con muy pocas modificaciones.
- C es actualmente el lenguaje por defecto en la industria de sistemas embebidos, y prácticamente cualquier ingeniero que trabaje en plataformas embebidas tendrá dominio de C, no así para lenguajes de alto nivel.

En consecuencia, el uso de C en plataformas embebidas trae potencialmente significativas ventajas en comparación a lenguajes de alto nivel interpretados como es el caso de Matlab, principalmente en términos de eficiencia en el uso de recursos, bajo overhead en tiempo de ejecución, incremento en el determinismo de los tiempos de ejecución de los programas, etc. Sin embargo, estas ventajas están asociadas al acceso directo a las capas de abstracción más bajas que entregan un control fino al programador, a costa de hacer más complicada la programación. Por ejemplo, C no provee abstracciones ni funcionalidades para el manejo automático de memoria dinámica, y el programador debe configurar explícitamente la solicitud y liberación de memoria en caso de operar con datos cuya dimensión cambia en tiempo de ejecución. El uso de manejo dinámico de memoria se realiza por medio de punteros, los cuales entregan gran versatilidad y control fino sobre la memoria, pero, si no se manejan adecuadamente, pueden corromper datos de sistema y generar fallos en los dispositivos físicos conectados al procesador. Por otro lado, Matlab incorpora abstracciones de alto nivel que automatizan el manejo de memoria dinámica en forma segura, y el programador no necesita preocuparse de estos aspectos, simplificando la programación pero agregando significativo overhead durante la ejecución de los programas.

Además del uso de lenguajes de bajo nivel, el uso o no uso de sistemas operativos es también una parte importante a considerar en el desarrollo orientado a plataformas embebidas en

general. En general, dependiendo de la aplicación y entorno de operación se vuelve necesario la utilización de diferentes tipos de sistemas operativos que se ajusten a las necesidades de cada proyecto. Por esto también se hace necesario considerar las características del sistema operativo al escribir un código en bajo nivel para un sistema embebido.

En el contexto previamente planteado, un problema práctico radica en que no todos los ingenieros o investigadores cuentan con la experticia para portar proyectos desarrollados en descripciones de alto nivel a plataformas embebidas, lo cual limita y ralentiza desarrollos o investigaciones que tengan relación con este tipo de implementaciones. Entonces, la existencia de una herramienta que pueda facilitar este paso de descripciones de alto nivel a un código que pueda ser implementado directamente en una plataforma embebida, resulta atractiva para facilitar el prototipado rápido que permita evaluar la utilidad efectiva de algoritmos avanzados en aplicaciones reales. Este el pricipal propósito de Embedded Coder, una herramienta entregada por Matlab que facilita el traspaso de un algoritmo escrito en Matlab a código C estructurado considerando características específicas de plataformas embebidas.

Matlab entrega documentación oficial acerca del uso de Embedded Coder, una herramienta que permite la generación de código en C a partir de descripciones de Matlab, además de ejemplos que documentan los pasos a seguir para generar código equivalente en C a partir de descripción de alto nivel en scripts .m o modelos de simulink. Sin embargo, a partir de las evaluaciones realizadas durante la asignatura ELO307, previo a esta memoria de título, se determinó que la documentación base no es exhaustiva y está orientada a la reproducción de ejemplo predefinidos y con scripts previamente adaptados para ello, sin mayores indicaciones sobre como extender estos scripts para nuevos proyectos. A partir de evaluaciones prácticas, se determinó que al tratar de procesar un código customizado para aplicaciones específicas que se alejan de las especificaciones y estructura de los ejemplos, se deben tener en cuenta varios aspectos asociados a las estructuras de datos, restricciones de memoria, y aspectos de arquitectura de la plataforma objetivo que no son descritos ni tratados en la documentación base. En síntesis, por medio de una evaluación práctica se determinó que, si bien Embedded Coder efectivamente simplifica el proceso de prototipado de algoritmos en plataformas embebidas, no cualquier código de alto nivel es directamente portable a código de bajo nivel, y la herramienta de generación de código no exime al usuario de conocer aspectos de programación de bajo nivel que pueden afectar la validez, funcionalidad, y eficiencia de los códigos generados. Descuidar o directamente omitir estructuras de codificación adecuadas o configuraciones de la herramienta puede llevar, en un buen caso, a que el código generado no se pueda compilar o ejecutar en la plataforma objetivo (lo cual se detecta a tiempo de diseño y es posible corregir previo a la ejecución), y en el peor caso, a generar un código válido pero con funcionalidad distinta a la intención de diseño, lo cual puede generar errores numéricos que, si no se sigue una metodología de verificación, pueden pasar desapercibidos y solo manifestarse en tiempo de ejecución con distinto tipo de consecuencias.

En base a lo anteriormente expuesto, se espera que los resultados de este trabajo ayuden en aumentar la productividad de ingenieros e investigadores en general, y en particular a proyectos actualmente en ejecución en el Departamento de Electrónica asociados a sistemas de control y electrónica de potencia, los cuales requieren acelerar el proceso de validación y verificación de la utilidad efectiva de algoritmos avanzados descritos en alto nivel cuando se consideran entornos de ejecución realistas que se encontrarían en sistemas físicos. Además, se espera que este trabajo también sirva para fomentar actividades de docencia en torno a sistemas de ingeniería aplicados en el mundo real.

1.2. Planteamiento del problema

El trabajo desarrollado en esta memoria de título se orienta a diseñar, validar y documentar una metodología para el uso de la herramienta Embedded Coder para la generación automática de código en C a partir de descripción de alto nivel en scripts .m o modelos de Simulink. El objetivo principal de este trabajo se orienta a proveer directrices que permitan a ingenieros e investigadores que no necesariamente cuentan con experiencia en programación de sistemas a bajo nivel, prototipar y validar sus algoritmos en plataformas embebidas de una forma confiable y rápida, disminuyendo en gran medida tanto el tiempo invertido en implementación como el arrastre de errores humanos en el proceso de codificación en bajo nivel.

El trabajo desarrollado consideró las siguientes etapas:

- **Obtención de una versión en C funcionalmente equivalente a un código de referencia en Matlab utilizando Embedded Coder.** Este paso consiste en utilizar Embedded Coder para la generación automática de código en C. Se apuntó a establecer una metodología que permita realizar una validación preliminar de los códigos generados en un computador de escritorio para cerciorarse que la funcionalidad de la versión de los algoritmos en C sea equivalente a la de Matlab.
- **Validación funcional de código generado en diferentes plataformas y sistemas operativos.** Esta etapa consideró la ejecución del código generado previamente en distintas plataformas embebidas considerando distintas configuraciones de hardware y sistemas operativos de uso común en entornos académicos e industriales. Siguiendo la meotdología establecida en etapa anterior, se realiza para cada caso una verificación funcional para verificar que los códigos generados entregan resultados equivalente al original en alto nivel al ejecutarse en los entornos objetivo.
- **Caracterización de tiempos de ejecución de los códigos generados.** Luego de verificar la equivbalencia funcional, se caracterizó el tiempo de ejecución en cada plataforma, contrastando con el tiempo obtenido en Matlab. El objetivo de este paso es la obtención de estimaciones cuantitativas que permitan determinar líneas base sobre el desempeño esperado al ejecutar los códigos generados en las plataformas objetivo.
- **Generación de un repositorio con documentación y ejemplos.** Se creó un repositorio GIT [5] en donde se encuentra la documentación y códigos utilizados a lo largo del trabajo, para facilitar replicar los resultados reportados en este documento, además, dar directrices sobre el uso de Embedded Coder para futuros proyectos.

1.3. Alcances y contribuciones

Sobre la base de evaluaciones preliminares previas a la definición del proyecto de memoria, para este trabajo se consideraron los siguientes alcances y restricciones:

- Este trabajo se limita a explorar y evaluar la utilidad efectiva de la herramienta Embedded Coder de Matlab para la generación de código C a partir de descripciones en scripts .m o modelos de Simulink. Si bien existen otras alternativas de uso académico e industrial que se analizaron durante el curso ELO307 antes del trabajo realizado en esta memoria de título, se determinó que Embedded Coder ofrecía las mayores ventajas en términos

de accesibilidad y facilidad de uso en el contexto de las actividades desarrolladas en el Departamento de Electrónica.

- Para orientar el trabajo a desarrollar, se definió una estructura para el flujo de trabajo para el uso de Embedded Coder utilizando cuatro algoritmos como casos de estudio orientados a distintos campos de aplicación: método de Broyden, factorización de Cholesky, operador Sobel, Alternating Direction Method of Multipliers (ADMM). Estos algoritmos son ampliamente utilizados en distintos campos de aplicación y presentan distinto grado de complejidad para ilustrar distintos aspectos y recomendaciones asociados a la correcta generación de código. Si bien estos ejemplos permiten expandir la documentación base de Matlab, en ningún caso cubren todos los aspectos de la herramienta y solo deben considerarse como guías.
- La validación y verificación experimental de los códigos generados se realizó utilizando un computador de escritorio y un número acotado de plataformas embebidas de distinta gamma de uso común en tareas académicas y de hobbys, incluyendo Raspberry Pi, Beaglebone, y STM32. Cada una de estas plataformas se probó con la configuración adecuada de sistema operativo, incluyendo bare-metal, FreeRTOS, y Linux. Una descripción detallada de las configuraciones evaluadas se entrega en la Sección 3.
- La validación experimental en las distintas plataformas se enfocó principalmente en evaluar la equivalencia funcional entre la versión original en alto nivel y la versión generada. Si bien se reportan tiempos de ejecución, estos solo deben usarse como una base de referencia, ya que no se explotaron aspectos específicos a la aceleración o ejecución eficiente explotando los recursos específicos de cada plataforma. En síntesis, los resultados de tiempo de ejecución tienen el potencial de ser mejorados, en caso de ser necesario.

Finalmente, las contribuciones de esta memoria de título consisten en el reporte de evidencia empírica sobre las capacidades y limitaciones del uso de Embedded Coder para la generación de código en C para su ejecución en plataformas embebidas de distinta gama. Adicionalmente, se entrega documentación y ejemplos con una metodología sistemática del uso y configuración de Embedded Coder, incluyendo pasos detallados para su uso mediante ejemplos, complementando la documentación base entregada por Matlab.

1.4. Organización del informe

Este trabajo tiene un enfoque experimental, y los entregables principales consisten en códigos y documentación asociada a la utilización de Embedded Coder con la respectiva implementación de los códigos generados en distintas plataformas embebidas. Todos estos códigos y detalles de como utilizar Embedded Coder se encuentran disponibles en un repositorio GIT [5] para su validación.

El resto de este documento sirve como complemento al entregable principal, y se limita a entregar información complementaria sobre aspectos técnicos y resultados que no se encuentran en el repositorio.

Los siguientes capítulos se estructuran de la siguiente manera:

- El **Capítulo 2** describe brevemente algunas herramientas de generación de código automático a partir de descripciones de alto nivel utilizadas en la industria y aquellas

que ofrece Matlab.

- El **Capítulo 3** presenta las pautas a seguir para la implementación y evaluación de los códigos generados en las diferentes plataformas embebidas.
- El **Capítulo 4** presenta la evaluación final de las implementaciones de los códigos generados por Embedded Coder en las distintas plataformas embebidas.
- El **Capítulo 5** resume las principales conclusiones de este trabajo y entrega las pautas para los trabajos futuros.

2 Antecedentes

Este capítulo describe un contexto general de los detalles técnicos de herramientas de generación de código automático. Se presentan brevemente algunas herramientas utilizadas en la academia e industria para generación de código a partir de descripciones de alto nivel. El objetivo de este capítulo es dar a conocer la importancia de las herramientas de generación de código, nombrando algunas de sus aplicaciones y presencia en la industria.

2.1. Diseño basado en modelos y herramientas de generación de código en la industria

El diseño basado en modelos (Model-based design, MBD) es un paradigma orientado al diseño de sistemas de ingeniería utilizando abstracciones matemáticas y herramientas visuales e intuitivas que permiten describir funcionalidad a alto nivel, permitiendo al diseñador abstraerse de los detalles de implementación y restricciones físicas [6, 7]. MBD facilita el diseño y verificación de funcionalidad del sistema diseñado en etapas tempranas del diseño, sin necesidad de considerar restricciones asociadas a la operación de las componentes internas y la interacción con componentes externas que son requeridas para la implementación física del sistema. Este paradigma es utilizado en distintos campos de aplicacion, incluyendo el diseño de sistemas de control complejos, procesamiento de señales, sistemas de comunicación, etc.

La metodología MBD es actualmente ampliamente utilizada en diversas industrias, incluyendo sistemas automotrices, aeroespaciales, y equipamiento industrial. Una aplicación común de MBD en estas industrias es la generación automática de código especializado que se ejecuta en plataformas embebidas. A diferencia del diseño tradicional de software que requiere el uso de estructuras de datos especializadas y optimización manual de código, MBD permite definir funcionalidad avanzada usando bloques estandarizados y la interconexión entre ellos, lo cual se entrega a una herramienta que genera el código para su implementación final, facilitando asi el prototipado y testeo rápido, además de reducción de costos no recurrentes. Además, el uso de diagramas de alto nivel beneficia directamente la auto-documentación de los sistemas, consistencia en la información, y comunicación entre equipos de trabajo. Este enfoque es también una parte fundamental de las simulaciones del tipo *hardware-in-the-loop*, la cual permite realizar testeo del comportamiento dinámico de un sistema, incluyendo condiciones de estrés y anomalías, utilizando un entorno virtual representativo del sistema físico. Este tipo de pruebas permiten prever escenarios que puedan ocasionar fallas como señales erróneas provenientes de sensores o caídas de voltaje y facilitan la detección de potenciales errores a tiempo de diseño.

En la industria existen herramientas que automatizan el proceso de traducir modelos descritos en diagramas de alto nivel a código de bajo nivel que puede ser compilado y ejecutado en plataformas embebidas como las que están comúnmente disponibles en sistemas prácticos. Una de las herramientas conocidas dentro de la industria es "SCADE suite KCG Code Generator", la cual permite la generación de código en C o Ada a partir de modelos SCADE suite [8]. Esta herramienta está enfocada en aplicaciones relacionadas con *safety critical systems* y sistemas de tiempo real, siendo utilizada en entornos industriales como automotriz, aeronáutica y aeroespacial [9]. De forma similar existe "Gene-auto Code Generator", una herramienta open source utilizada en la industria para la generación de código en C en aplicaciones similares a SCADE; sin embargo, se diferencian en la integración directa que tiene Gene-auto con las herramientas que entrega Matlab para el modelado y simulación (Simulink y StateFlow) [10] y la accesibilidad a la documentación y ejemplos de uso.

Es importante mencionar que herramientas enfocadas en safety critical systems aseguran un código listo para producción y certificado. Un código listo para producción se caracteriza por cumplir con todos los requisitos funcionales propuestos en la etapa de diseño y que se encuentra libre de bugs. Por otra parte, un código certificado hace referencia a que cumple con los estándares de alguna industria en particular. Por ejemplo, un código que este certificado en aplicaciones aeroespaciales garantiza que no incumple restricciones propias impuestas por la aplicación. Cabe destacar que la certificación de un código de bajo nivel generado automáticamente por una herramienta computacional solo indica que el código generado es funcionalmente equivalente al original y que sigue ciertos criterios en términos de uso de recursos; sin embargo, no garantiza que el código opere correctamente desde el punto de vista de la aplicación, lo cual corresponde a un problema de diseño. Si el diseño original está errado, el código generado seguirá estando errado aun cuando cuente con la certificación. Las herramientas de generación de código no pueden detectar problemas de diseño automáticamente, y el usuario debe aplicar procedimientos adicionales para la depuración y verificación del comportamiento correcto tanto del código original como del generado.

Por otro lado, existen herramientas como Labview [11] y Dspace Target Link [12], que son alternativas para la generación de código. Labview permite la generación de código en C y Verilog HDL [13] a partir de modelos de autoría propia (VI), diferenciándose de herramientas similares al facilitar la portabilidad de modelos a FPGAs. Target Link, por su parte, tiene integración directa con Matlab e incluye bloques especiales dentro de los modelos implementados que permiten la optimización del código generado para cumplir con las limitaciones de procesadores de plataformas embebidas en donde estos códigos serán ejecutados.

Si bien el uso de herramientas de generación de código C a partir de MBD se han establecido como una práctica estandarizada en la industria, las herramientas orientadas a aplicaciones industriales, como las mencionadas previamente, presentan algunas desventajas. Una de las principales complejidades es que no existe un enfoque de desarrollo integrado estandarizado, lo que resulta en que las herramientas disponibles utilicen entornos propietarios y de código cerrado que requieren capacitar al usuario en nuevas herramientas propias de cada entorno (normalmente necesitando entrenamiento específico para ello). Además, estas herramientas no son fácilmente accesibles debido al alto costo de licencias y requerimientos específicos de hardware (como es el caso de LabView y DSpace). Estas limitantes determinaron los alcances de este trabajo para enfocarse en el uso de las herramientas ofrecidas por Matlab, ya que si bien Matlab también tiene costos de licencias asociados y entrega un entorno de desarrollo cerrado, tiene una amplia presencia en los entornos académicos debido a que los programas universitarios

reducen sus costos, lo que deriva a que un mayor número de usuarios esté familiarizado con los lenguajes y el entorno de desarrollo que ofrece.

2.2. Herramientas de MATLAB para generación de código

Dado que este trabajo tiene como finalidad contribuir a proyectos e investigaciones que actualmente se encuentran en desarrollo en el Departamento de Electrónica, es que orienta el uso de generadores de código en un ámbito de carácter académico, siendo las herramientas que proporciona Matlab una alternativa que se ajusta más para este propósito, ya que, Matlab tiene mayor popularidad en ámbitos académicos, ya se cuentan con licencias facilitando accesibilidad, existe mayor familiaridad con el uso de este software, entre otras. Matlab proporciona tres herramientas para la generación de código automático en C, cada una con un enfoque distinto según sea la necesidad. Este tipo de herramientas se diferencian de las que son utilizadas en la industria, puesto que no están pensadas para cumplir estándares industriales y en consecuencia no deberían utilizarse para generar código orientado a aplicaciones críticas. Sin embargo, en las evaluaciones previas a este trabajo de título, se observó que las herramientas que entrega Matlab son suficientes para facilitar el prototipado rápido de modelos y algoritmos en un entorno académico.

A continuación, se presenta una breve descripción de Matlab Coder, Simulink Coder y Embedded Coder como las herramientas que ofrece Matlab para la generación de código automático en C, entregando características técnicas relevantes de cada una.

2.2.1. Matlab Coder

Matlab Coder es una herramienta de generación de código automático que permite a partir de descripciones de alto nivel en scripts .m obtener un código en C funcionalmente equivalente [14]. Según la documentación oficial de Matlab Coder, la utilización de la herramienta puede realizarse mediante dos modalidades, la primera modalidad consta de su utilización vía línea de comandos, mediante el comando *codegen* y la segunda consta de su utilización vía interfaz gráfica, proporcionada por la aplicación llamada Matlab Coder disponible en los *toolboxes* de Matlab. Cabe destacar que ambas formas de utilizar Matlab Coder son equivalentes y varían solo en la forma de ingreso de los datos de entrada e interacción del usuario.

Una de las características adicionales que tiene Matlab Coder a la generación de código automático en C, es la generación de archivos especiales con extensión .MEX. Los archivos MEX compilan y enlazan uno o más archivos fuente en lenguajes de más bajo nivel (C/C++ Fortran) en una biblioteca compartida llamada archivo binario MEX, el cual es interpretado y ejecutado como un script .m en Matlab. En consecuencia, la utilización de archivos MEX generados por Matlab Coder permiten reducir el tiempo de ejecución de los códigos escritos directamente en el lenguaje de Matlab [15, 16].

Algunas de las características principales de Matlab Coder se listan a continuación:

- Generación de archivos .MEX de punto flotante y punto fijo. La elección del tipo de la precisión a utilizar tiene un impacto en el uso de memoria y tiempo de ejecución del programa final.

- Generación de código con bibliotecas dinámicas o estáticas. La elección del tipo de biblioteca en nuestro código puede traer un impacto en el tiempo de ejecución o memoria utilizada. Cuando se utilizan bibliotecas dinámicas los ejecutables son más ligeros pero más lentos en tiempos de ejecución, en cambio, cuando se utilizan bibliotecas estáticas, los ejecutables son más pesados pero más rápidos en tiempo de ejecución.
- Prototipado para plataformas embebidas. Pese a que su enfoque no se encuentra en la generación de código para plataformas embebidas, ofrece configuraciones que facilitan la portabilidad de código a una gama muy reducida de ellas, siendo soportadas solo modelos específicos de tarjetas de la familia Raspberry y Jetson.
- Optimización del código generado. Esta característica se ve traducida con mejores tiempos de ejecución mediante la optimización de ejecución del código dentro del procesador. Sin embargo, esta optimización no ofrece control fino sobre la arquitectura donde el código finalmente será ejecutado, ni tampoco la cantidad de recursos extra (principalmente memoria) que está utilizando para obtener dichos resultados.

Adicionalmente, consideraciones para el uso correcto de la herramienta se relacionan con el uso dinámico de memoria dentro de un código. Matlab provee un nivel de abstracción alto que permite realizar el prototipado de algoritmos sin preocuparse demasiado en detalles técnicos de programación que tienen lenguajes de más bajo nivel. Por ejemplo, dentro de las abstracciones que brinda Matlab se encuentra el manejo automático de memoria, permitiendo no tener que solicitar o liberar memoria explícitamente, simplificando operaciones relacionadas con el cambio de dimensión de datos dentro de un algoritmo o modelo. En consecuencia, es común ver variables que tengan un uso de memoria dinámico a lo largo de un script, sin embargo, Matlab Coder requiere que el tipo y tamaño de variables este explícitamente definido desde el inicio de la ejecución de nuestro programa hasta el final de este mismo (similar como sería en un código en C).

Por otro lado, Matlab Coder no tiene soporte para todas las funciones de alto nivel que nos entrega Matlab [17]. Debido a esto cierto tipo de aplicaciones podrían verse afectadas, ya que al no contar con la totalidad funciones es posible que deban de confeccionarse desde cero algunas de ellas, invirtiendo tiempo que en principio no sería necesario en una implementación que utilice solo Matlab para su desarrollo. Además, hay que tener en consideración que pese a que los archivos resultantes de la generación de código entregan una versión funcionalmente equivalente, cubriendo todos los aspectos de bajo nivel que pueden ser fuente de errores, para poder compilarlos se necesita confeccionar manualmente un archivo main que haga el llamado individual a cada una de las funciones necesarias, siendo especialmente difícil cuando se está iniciando en el uso de la herramienta debido a la poca legibilidad del código generado.

En síntesis, en base a la documentación entregada por Matlab y a nuestras propias evaluaciones preliminares, determinamos que Matlab Coder efectivamente permite la generación de código en C a partir de scripts en formato .m presentando características interesantes, como lo son la generación de archivos MEX o la optimización de ejecución del código generado en el procesador. Sin embargo, el objetivo de esta herramienta está orientado a mejorar el tiempo de ejecución de códigos en computadores de escritorio x86 y versiones específicas de platformas Nvidia's Jetson y Raspberry Pi.

2.2.2. Simulink coder

Simulink Coder, es una herramienta que puede generar el código en C a partir de modelos realizados en Simulink (.slx) y diagramas StateFlow, no así directamente de archivos .m [18]. Esta herramienta está pensada para el prototipado rápido de modelos, algoritmos o simulaciones de sistemas físicos, dando soporte para aquellos proyectos que estén pensados para aplicaciones de tiempo real. Dentro de las características de Simulink Coder, se encuentra la optimización de ejecución del código generado, integración con *toolboxes*, generación de reportes, entre otros.

Debido a que los modelos en Simulink tienen asociado un tiempo de simulación, es que se vuelve relevante tomar en consideración aspectos propios que trae trabajar en este tipo de entornos. Una de los principales aspectos que trae el trabajar con tiempo de simulación, es la incorporación del tipo de *solver* a utilizar, debido a que este es el que se encarga de calcular el *siguiente paso* dentro de una simulación, siendo especialmente importante en la generación de código. Para Simulink, existen dos tipos de solvers a implementar, los solvers de *paso fijo* o *fixed step*, que consisten en utilizar un intervalo regular de tiempo entre cada paso de la simulación, y los solvers de *paso variable* o *variable step*, que consisten justamente en lo contrario, es decir, toman intervalos irregulares tiempo para cada paso de simulación. Dadas las características de Simulink Coder, esta solo soporta la generación de código para aquellos modelos que utilicen solvers de paso fijo para el cálculo entre cada paso de simulación.

Otra de las características importantes de Simulink Coder es la configuración que trae para la plataforma objetivo en donde se implementarán los códigos generados. La configuración por defecto de Simulink Coder es *generic real time target* (grt), que para efectos de la generación de código significa que no está pensado para ninguna plataforma como objetivo, es decir, el formato del código generado no está optimizado para ninguna implementación en particular, y por lo tanto el uso de memoria, tamaño del código o utilización de ciertas estructuras de datos para el mejor manejo de los recursos disponibles no son tomados en cuenta a la hora de la generación de código. El tipo de configuración *grt* se utiliza normalmente para correr los códigos generados en un computador de escritorio, ya que generalmente no se encuentran limitados en recursos como memoria o potencia de cómputo.

Es importante mencionar, que además de las características mencionadas durante esta sección, Simulink Coder posee, al igual que Matlab Coder ciertas consideraciones al momento de generar código, como las restricciones con el tipo de funciones a utilizar [17] y el uso de memoria dinámica, siendo análogas para ambas herramientas. Finalmente, es pertinente añadir que Simulink Coder es una herramienta que permite la generación de código en C a partir de modelos Simulink para el prototipado rápido de modelos que tengan su implementación en un computador de escritorio.

2.2.3. Embedded Coder

Embedded coder es una herramienta para generación automática de código en C a partir de funciones escritas en Matlab (.m) y modelos realizados en Simulink (.slx) [1]. Se diferencia de las herramientas descritas anteriormente (Matlab Coder y Simulink Coder) porque estas están pensadas para una generación de código de propósito más general, sin tener en consideración cuál será la plataforma donde finalmente se correrán los códigos generados. Por otro lado, Embedded Coder está enfocado a la plataforma objetivo en donde se implementarán los códigos

generados, específicamente apunta a la implementación en sistemas de bajo nivel, entregando un código listo para producción, agregando configuraciones extras para la optimización del código generado, apuntando a obtener un mejor manejo de recursos en el procesador objetivo:

- Optimización del código generado según la arquitectura del procesador objetivo.
- Optimización para minimizar la memoria RAM y/o ROM que utilizará el código generado.
- Equilibrio entre optimización de ejecución y memoria RAM y/o ROM utilizada.
- Decidir como guardar la información de las matrices en memoria, ya sea con las filas contiguas en memoria (major row layout) o con las columnas contiguas en memoria (major column layout). Este tipo de configuración puede llegar a tener un impacto en el tiempo de ejecución en el código generado, ya que si nuestro algoritmo tiene en su mayoría operaciones con las filas de matrices y tenemos la configuración major column layout, nos veremos penalizados en tiempo de acceso a memoria.
- Precisión máxima o mínima de las variables.

Embedded Coder también permite la personalización de los comentarios que tendrá el código generado, la indentación, nombre de variables, comentarios por bloques de código, entre otros. Este tipo de configuraciones ayuda a la legibilidad del código generado, permitiendo realizar modificaciones si se encontrasen pertinentes o necesarias. Por ejemplo, es posible incluir comentarios que describan la funcionalidad de bloques de código dentro del código generado, entonces para alguien con experiencia trabajando y programando en plataformas embebidas puede serle de utilidad para utilizar dicho código como base y sintonizarlo posteriormente. Embedded Coder está pensada para la generación de código para plataformas embebidas, entregando un código más compacto y adecuado para las restricciones que normalmente se tienen que tener en cuenta para implementaciones de hardware con recursos limitados.

Cabe destacar que Embedded Coder es una herramienta que extiende tanto a Matlab Coder como Simulink Coder, como puede observarse en el flujo de ejecución de estas herramientas de la Figura 2.1. Embedded Coder pasa a ser una versión mejorada de las otras dos herramientas, puesto que para su utilización Matlab internamente hace un llamado, ya sea a Matlab Coder o Simulink Coder según corresponda. Dada esta información, es correcto concluir que las limitaciones y/o restricciones impuestas por Matlab Coder o Simulink Coder se vean reflejadas en el uso de Embedded Coder.

2.2.4. Discusión

Dadas las tres herramientas para la generación de código automático que nos ofrece Matlab, es posible decir que dado los alcances y enfoque de este trabajo, el uso de plataformas embebidas es fundamental para obtener resultados que sean de utilidad para trabajos futuros. En consecuencia es factible concluir, basándose en la información presentada, que Embedded Coder es la herramienta que mejor satisface las restricciones que conlleva utilizar plataformas embebidas, ya que dadas las configuraciones existentes en la herramienta es posible adaptarse a las restricciones comúnmente presentadas al utilizar sistemas que tengan una cantidad de recursos limitados.

En estricto rigor, si la utilización de plataformas embebidas no fuese necesario, la elección de qué herramienta utilizar dependería exclusivamente del tipo de aplicación. Por ejemplo,

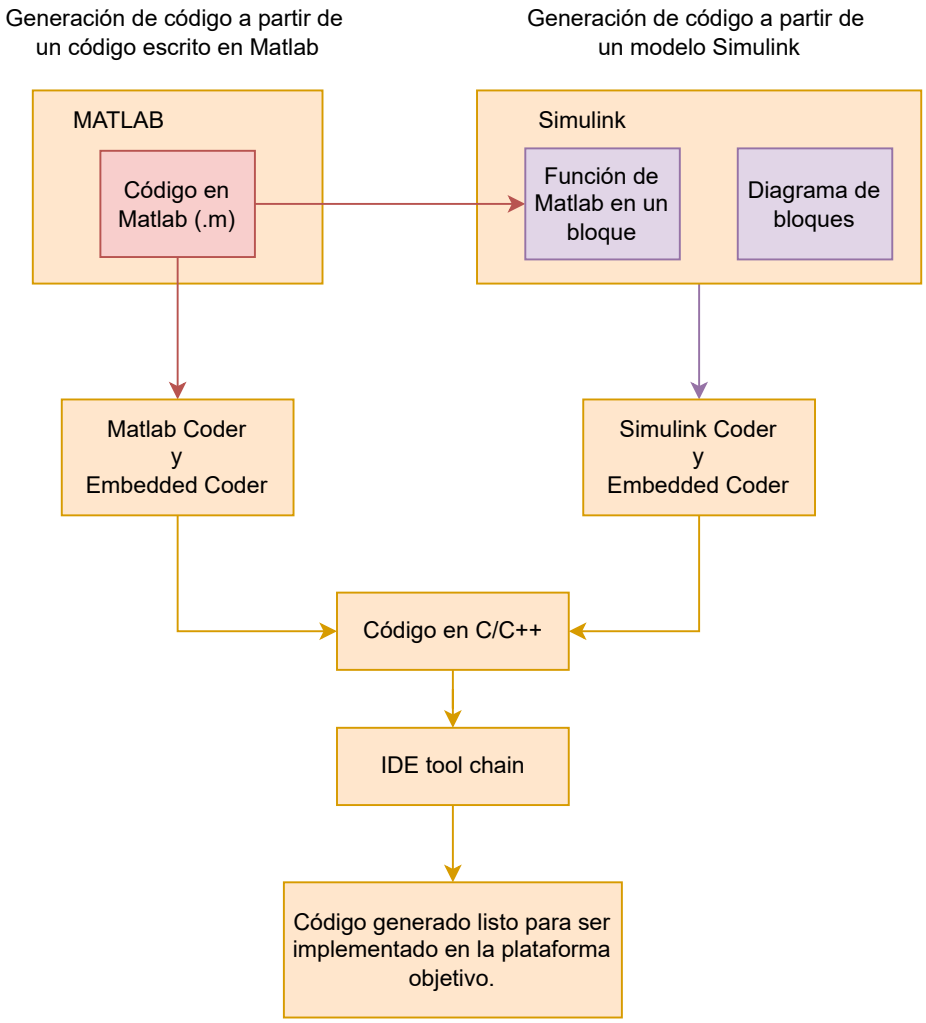


Figura 2.1: Flujo en la generación de código.

si lo que buscamos es únicamente mejorar el tiempo de ejecución de un código de Matlab que finalmente correrá en un computador de escritorio, bastaría con utilizar Matlab Coder en lugar de Embedded Coder. De igual forma, si estamos trabajando con aplicaciones de tiempo real o simulaciones y disponemos de un computador de escritorio donde finalmente será implementando el proyecto, la alternativa más adecuada es Simulink Coder en vez de Embedded Coder.

3 | Metodología y desarrollo

Este capítulo describe el flujo de trabajo empleado para la utilización de Embedded Coder, los casos de estudio escogidos, y la metodología empleada para la validación de los resultados obtenidos, detallando los entornos utilizados para realizar todas las pruebas. Cabe destacar que este capítulo es complementario al material incluido en el repositorio de código disponible en [5] que concentra los códigos desarrollados y la información necesaria para reproducir y expandir los resultados, entregando un contexto general sobre antecedentes y decisiones de diseño que se tomaron para el desarrollo de este trabajo.

3.1. Aplicaciones de ejemplo

Para poder realizar pruebas de la utilización de Embedded Coder se consideraron los siguientes algoritmos como casos de estudio:

- **Método de Broyden.** El método de Broyden se define como un algoritmo que permite la resolución de sistemas no lineales de ecuaciones, mediante la aproximación numérica, es ampliamente utilizado en diferentes disciplinas como en ingeniería, matemáticas, medicina y robótica [19]. Algunos ejemplos de su uso son en la meteorología, reacciones químicas, criptografía, entre muchos otros.
- **Factorización Cholesky.** La factorización de Cholesky es un método de álgebra lineal que permite factorizar o descomponer una matriz en dos matrices tal que su multiplicación sea igual a la matriz antes de su descomposición. Este método es ampliamente utilizado en diferentes campos de aplicación como geometría analítica, ingeniería, física, ciencias de la computación, etc [20]. Algunos de ejemplos de su uso son la resolución de sistemas lineales, problemas de programación lineal, cálculo de matriz hessiana en problemas de funciones convexas en optimización, entre otros.
- **Operador Sobel.** El Operador Sobel es un algoritmo de procesamiento de imágenes que permite la detección de bordes en elementos de una imagen. Este algoritmo es ampliamente utilizado en el campo de visión por computador, siendo utilizado en aplicaciones como reconocimiento de objetos o personas y algoritmos de procesamiento de imágenes en general.
- **Alternating direction method of multipliers (ADMM).** Es un algoritmo que se utiliza para resolver problemas de optimización cuadrática, los cuales son típicos en modelos de control predictivo (MPC). Este algoritmo se identificó como de especial interés para proyectos de investigación en curso dentro del Departamento de Electrónica,

por lo cual se utiliza como ejemplo para realizar pruebas más exhaustivas que permitan validar el funcionamiento y reportar tiempos de ejecución de los códigos generados por Embedded Coder.

3.2. Flujo de trabajo

El procedimiento que se siguió se muestra en la Figura 3.1 y detalles de cada paso se describen a continuación:

1. **Algoritmo Matlab.** Este paso consiste en definir apropiadamente el caso de estudio en Matlab, adaptando el código de alto nivel para cumplir con las restricciones y requerimientos de la herramienta Embedded Coder planteadas en la Sección 2.2.3. Estas restricciones incluyen evitar elementos con dimensiones dinámicas, uso de funciones de Matlab no soportadas para generación [17], etc.

2. **Crear Golden Reference.** Este paso consiste en utilizar Matlab para la creación de dos archivos, uno que contengan las entradas y otro las respectivas salidas del algoritmo en el que posteriormente se utilizará Embedded Coder. Cada archivo contiene una variable de entrada o salida según corresponda separadas por saltos de línea, además, la cantidad de set de entradas contenidas en un archivo queda a elección del usuario, para el caso de este trabajo fueron consideradas cien mil set de entradas diferentes. La finalidad de la creación de este tipo de archivo es poder realizar una validación funcional de los códigos generados.

3. **Modelo Simulink.** Este paso consiste en tomar el algoritmo escrito en Matlab y empaquetarlo a un diagrama en Simulink. Con la implementación del bloque *MATLAB function* en Simulink es posible portar directamente el algoritmo que tenemos en forma de script .m sin complicaciones. Ahora bien, es de especial importancia definir explícitamente la dimensión de todas las entradas del bloque MATLAB function en Simulink, de lo contrario no podremos realizar la generación de código.

4. **Utilización Embedded Coder.** Este paso consiste en utilizar Embedded Coder a partir del modelo Simulink obtenido en el paso anterior. Mediante la aplicación Embedded Coder disponible en los toolboxes de Simulink se genera un código en C con procesador objetivo, el computador de escritorio que estamos utilizando. Para la utilización de la herramienta se deben seguir las instrucciones mostradas en el repositorio disponible [5].

5. **Creación Main incluyendo wrapper.** Esta etapa consiste en la modificación del archivo main creado por Embedded Coder que permita compilar y utilizar los códigos generados (Detalles [5]). Adicionalmente, para la validación de resultados obtenidos, se modifica el archivo main creado para que realice múltiples llamados a las funciones generadas por Embedded Coder (consultar [5]), creando de esta forma un wrapper de nuestras funciones. Para cada llamado de las funciones, es necesario utilizar los datos de entradas posibles contenidos en nuestra golden reference.

6. **Validación de códigos generados.** Mediante la implementación del wrapper creado, es que se comparan en un computador de escritorio los resultados de la golden reference con los obtenidos por el código en C. Para verificar que los resultados obtenidos por el código en C son equivalente a los obtenidos por Matlab, es que se define una tolerancia

arbitraria para los resultados obtenidos. El resultado se considerará como correcto o válido si y solo si, la diferencia entre el resultado obtenido con respecto al de la golden reference es menor a la tolerancia indicada. La forma en que se calcula el error, está dado por la siguiente fórmula:

$$Error = |\frac{dato_C - dato_M}{dato_M}| * 100\,\% \tag{3.1}$$

donde:

$dato_C$: Representa el resultado obtenido a partir del código en C.

$dato_M$: Representa el resultado de la golden reference, obtenido con Matlab.

$Error$: Representa la diferencia numérica obtenida entre el resultado del código en C y Matlab, se encuentra expresado de forma porcentual.

7. **Configuración de plataformas embebidas.** Se preparan las plataformas embebidas con los diferentes sistemas operativos. Para el caso de las tarjetas de la familia Raspberry y Beaglebone se instala Linux. Por otro lado, para las tarjetas pertenecientes a la familia de STM32 se configura su IDE para utilizarla con RTOS y Bare Metal. Adicionalmente, se generan los códigos para todas las plataformas embebidas a utilizar utilizando Embedded Coder.

8. **Validación resultados en plataformas embebidas.** Utilizando el archivo Main elaborado en el paso 5 y la golden reference es que se validan los resultados en las diferentes plataformas embebidas comparando los resultados obtenidos con los de la golden reference.

9. **Obtención tiempos de ejecución.** Se usa el profiler de Matlab para medir tiempo de ejecución del algoritmo asociado al caso de estudio. Con el uso de la biblioteca time.h de C, se obtienen los tiempos de ejecución de los códigos generados. Sin embargo, para la familia de tarjetas STM32 no es posible utilizar esta biblioteca, en consecuencia los fabricantes entregan funcionalidades similares en su biblioteca HAL, permitiendo utilizar contadores de ciclos de reloj para la obtención de información equivalente.

Cabe destacar que la potencia computacional que poseen los microcontroladores utilizados difieren considerablemente a la que tiene el computador de escritorio que corre Matlab. El objetivo en este caso no es realizar una comparación directa entre el desempeño del computador de escritorio corriendo Matlab y las plataformas embebidas, sino que evaluar la versatilidad que entrega Embedded Coder para generar código ejecutable en plataformas de distintas familias y gamas.

3.3. Plataformas objetivo y configuraciones

Las plataformas embebidas utilizadas para la implementación de los casos de estudio planteados, son las nombradas en la Tabla 3.1 en donde se pueden ver detalles técnicos de cada una de ellas conjunto el sistema operativo utilizado. La elección de los diferentes microcontroladores para la implementación de los códigos generados por Embedded Coder, viene dada por la exploración de diferentes arquitecturas y potencia de procesadores para de esta forma expandir la gama de sistemas en los que es posible implementar directamente código generado por Embedded Coder. Por otro lado, se definen los siguientes entornos de trabajo para las diferentes plataformas embebidas:

- **Linux.** Una de las razones por la que se eligió este sistema operativo, es debido a los pocos recursos que necesita para poder ser utilizado manteniendo las ventajas de utilizar sistemas operativos, como una interfaz visual, manejo de multitareas, recursos compartidos, modularidad, entre otros. Dado que por lo general los sistemas embebidos tienen restricciones debido a la cantidad de recursos, es especialmente importante contar con un sistema operativo que le permita al usuario tener ciertas facilidades cuando se está programando en estos sistemas. Sin embargo, no hay que dejar de lado que para aplicaciones en donde las restricciones temporales sean muy estrictas, el overhead que conlleva utilizar Linux podría tomarse como una desventaja.
- **FreeRTOS.** Este sistema operativo es utilizado para sistemas de tiempo real que requieran cumplimiento de deadlines. La elección de este tipo de sistema operativo es debido a que se espera que tenga un impacto en el tiempo de ejecución de los códigos generados, obteniendo resultados más determinísticos en comparación a Linux.
- **Configuración *bare-metal*.** Bare Metal es la configuración que tiene una plataforma embebida cuando no tiene ningún sistema operativo y el nivel de abstracción es muy cercano al hardware. Dado que no hay sistema operativo, el manejo de recursos o periféricos queda a total disposición del desarrollador permitiendo tener un control más fino de los recursos disponibles, a costa de complicar la programación debido a la abstracción perdida al no usar sistema operativo. Se decide probar en este tipo de entorno, ya que al poseer un overhead menor que utilizar sistemas operativos, se espera que haya un impacto en los tiempos de ejecución de los códigos generados.

Es importante mencionar que no todas las plataformas embebidas utilizadas para correr los códigos generados implementan todos los sistemas operativos mencionados. Para el caso de las tarjetas de la familia STM32 utilizadas, su limitada capacidad en memoria se volvía una barrera que impedía poder instalar alguna distribución de Linux. Por otro lado, las tarjetas Raspberry y BeagleBone no tuvieron una implementación en freeRTOS, ya que a la fecha de realización de este trabajo no existía un soporte oficial por parte de freeRTOS para la instalación de este sistema operativo en su hardware. Cabe destacar que se probó un proyecto open source en donde se portaba freeRTOS para una Raspberry pi 3B, sin embargo, no se tuvo éxito en su utilización. Entonces, esto genera que en un futuro los resultados de tiempo de ejecución no sean directamente comparables entre plataformas, sin embargo, dado que el objetivo de estas pruebas es validar los resultados obtenidos por los códigos generados por Embedded Coder, esta decisión no tiene un gran impacto en la veracidad de los resultados.

Tabla 3.1: Información de los distintos entornos de prueba.

Plataforma	Procesador	Nro. Núcleos	Memoria	Sistema Operativo
Computador escritorio	Intel Core i7-8550U @1.8GHz	4	16 GB	Windows 11
Computador escritorio	Intel Core i7-8550U @1.8GHz	4	16 GB	Linux Mint
Raspberry 3B	ARM Cortex-A53 @900MHz	4	1 GB	Raspbian
Raspberry 4B	ARM Cortex-A72 @1.5GHz	4	2 GB	Rapsbian
BeagleBone AI	ARM Cortex-A15 @1.5GHz	2	1 GB	Debian
STM32f7xx	ARM Cortex-M7 @216MHz	1	512KB	FreeRTOS, Bare Metal
STM32f4xx	ARM Cortex-M4 @100MHz	1	128KB	FreeRTOS, Bare Metal

3.4. Implementaciones

Todos los códigos utilizados a lo largo del desarrollo de este trabajo relacionados con la validación y verificación de los algoritmos propuestos, se encuentran disponibles en un repositorio creado para facilitar la réplica de resultados obtenido. Además, se encuentra tutoriales de como utilizar Embedded Coder, configurar freeRTOS y la comunicación con aquellos microcontroladores que fueron utilizados en Bare Metal [5].

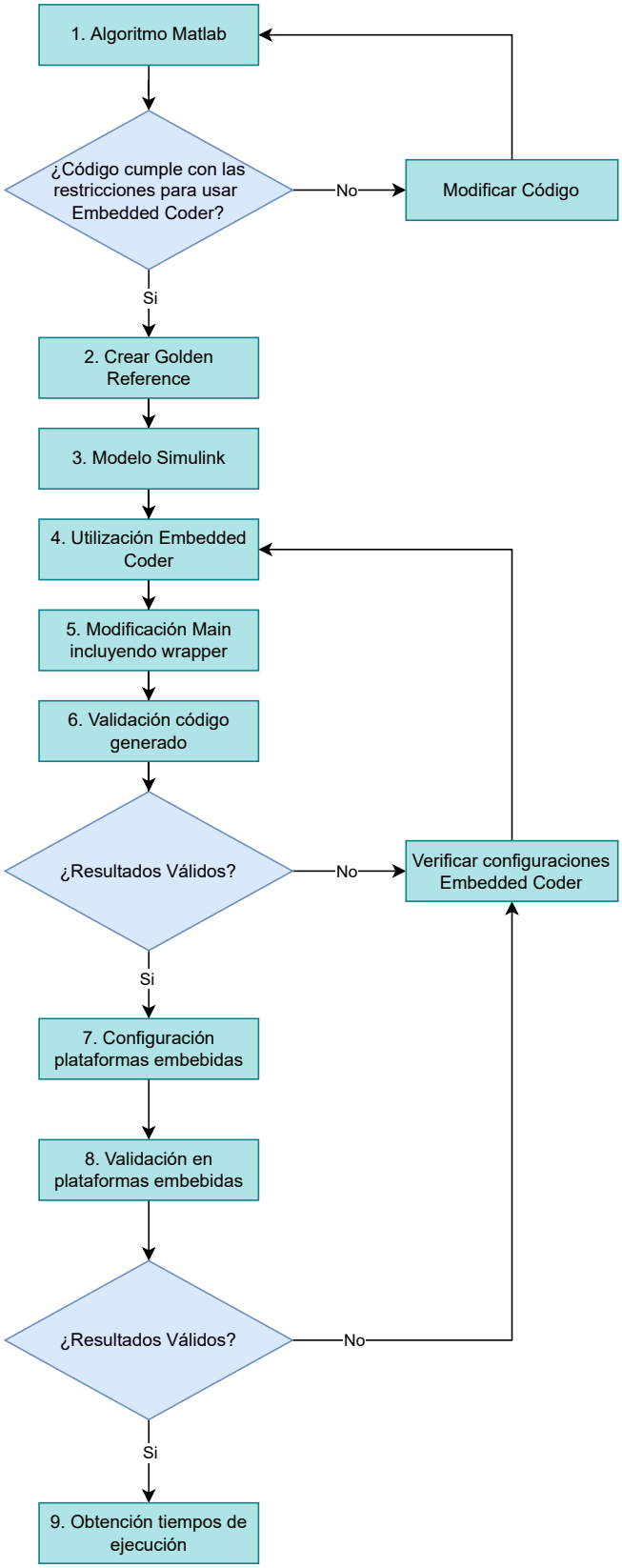


Figura 3.1: Flujo de trabajo.

4 | Resultados experimentales

Este capítulo reporta los resultados obtenidos en tiempo de ejecución y diferencias numéricas que existen entre los códigos de Matlab con los códigos generados por Embedded Coder a partir de los casos de estudio planteados. El objetivo de realizar estas pruebas es obtener resultados cuantitativos que justifiquen la validez de Embedded Coder para la generación de código en C. Además, la implementación en diferentes plataformas embebidas y sistemas operativos entrega información relevante acerca del impacto que tiene el uso de diferentes arquitecturas de procesadores y sistemas operativos en el tiempo de ejecución de los códigos generados.

4.1. Validación funcional de códigos generados

Como primera parte del trabajo se realizaron evaluaciones preliminares enfocadas en la validación de la documentación base entregada por Matlab para la utilización de Embedded Coder, y su posible extensión en base a la implementación de nuevos ejemplos y casos de estudio distintos a los entregados en la documentación. El enfoque de estas pruebas está principalmente en la equivalencia funcional, centrándose en los resultados numéricos, sin realizar una evaluación exhaustiva de los tiempos de ejecución ni optimizaciones específicas para cada plataforma.

Para estas pruebas se consideraron los casos de estudio **Método Broyden**, **Factorización Cholesky** y **Operador Sobel**. Estas pruebas fueron realizadas en un set reducido de plataformas que permitieran cubrir diferentes familias de microcontroladores:

- Un computador de escritorio con Windows.
- Una BeagleBone AI con Debian.
- Una STM32f7xx en Bare Metal.

Por otro lado, se tuvieron en cuenta las siguientes consideraciones:

- El método Broyden fue implementado en un sistema de ecuaciones de dos dimensiones para cada una de las plataformas.
- La factorización Cholesky implementa una matriz cuadrada de dimensión 5×5 para cada una de las plataformas.
- El operador o filtro Sobel considera una imagen cuadrada de dimensión 500×500 pixeles para el caso de computador de escritorio y BeagleBone. Sin embargo, de dimensión 25×25 para el caso de la STM32f7xx debido a limitaciones en memoria para almacenar una imagen de dimensión superior.

Para cada caso se probaron los algoritmos con distintos set de entradas, realizándose siempre una comparación con una golden reference. Todos los códigos e indicaciones para la reproducción correcta de los resultados mostrados se encuentran disponibles en el repositorio [5].

La Tabla 4.1 muestra los resultados de tiempo de ejecución, error máximo en los resultados y plataformas utilizadas para realizar cada prueba, entendiéndose por prueba el correr código generado por Embedded Coder midiendo el tiempo en que demora y diferencias numéricas obtenidas con su equivalente en Matlab. Cada una de las pruebas presentadas fueron realizadas una vez en cada una de las plataformas especificadas Para el caso del método de Broyden, el error máximo supera el 2 %; sin embargo, hay que tener en cuenta que este error es relativo a los valores obtenidos, es decir, hay que tener en consideración que este error es relativo a los valores obtenidos teniendo una diferencia en magnitud absoluta que ronda valores de 10^{-3} lo que puede considerarse como despreciable dependiendo del contexto de la aplicación. Por otro lado, las magnitudes de los errores de la factorización Cholesky y el operador Sobel se consideran despreciables para efectos prácticos.

Los tiempos de ejecución obtenidos por parte de los códigos generados mejoran con respecto a los obtenidos por Matlab al menos un 10 % en la mayoría de algoritmos, siendo lo esperado por el overhead que trae consigo el uso de Matlab, sin embargo, al realizarse tan solo una prueba del algoritmo en cada plataforma, los resultados temporales obtenidos no pueden considerarse como representativos. En síntesis, la utilización y posterior implementación de los códigos generados por Embedded Coder presentan un comportamiento equivalente al de los códigos escritos en Matlab. Es importante recalcar que para el caso de la STM32F7xx se utilizó una dimensión de imagen más pequeña que en las demás plataformas embebidas, ya que, no era posible computar una imagen utilizada en los otros entornos.

Tabla 4.1: Estadísticas de error realtivo y tiempo de ejecución del código generado a partir de distintos algoritmos.

Algoritmo	Plataforma	% Error máx	Tiempo Matlab [μs]	Tiempo C [μs]
Broyden	PC Windows	2.313	6316	0.7
	BeagleBone AI	2.3586		9.76
	STM32F7xx	2.446		38
Cholesky	PC Windows	0.000446	2000	5.4
	BeagleBone AI	0.000446		122.163
	STM32F7xx	0.0564		654.611
Sobel	PC Windows	0.00044	580821	20792.5
	BeagleBone AI	0.000311	580821	530897.638
	STM32F7xx	0.000212	1930	6039

4.2. Evaluación de ADMM en distintas plataformas.

Para el caso de estudio ADMM fueron realizadas pruebas en todos los entornos descritos por la Tabla 3.1. Cada prueba consistió el correr código generado por Embedded Coder para un set de cien mil entradas en particular. Con esto, se mide el tiempo demorado y se generan cien mil resultados con los cuales se calcula el error relativo entre el valor obtenido con el codigo C y lo indicado en la golden reference de Matlab. Cada resultado se consideran como un set de salidas obtenidas por la ejecución de un set de entradas en particular, además, se

consideran los valores como cada número en particular de este set de resultados. La cantidad de ejecuciones del código generado realizadas para todas las plataformas a excepción de las tarjetas de la familia STM32 fueron un total diez millones, donde se consideraron cien mil set de posibles entradas al algoritmo y estas se repitieron cien veces cada una. El objetivo de tener esta cantidad de pruebas es poder identificar y filtrar distorsiones asociadas a un set de entradas en particular y así poder ver el impacto de los entornos de prueba (sistema operativo y procesador) en el tiempo de ejecución.

Adicionalmente, fueron probadas tres versiones del algoritmo, entendiéndose por versión una diferencia en la dimensión de las matrices de entrada, las dimensiones probadas fueron de 4, 8 y 16. Si bien se realizaron algunas pruebas con matrices de mayor dimensión y se verificó la equivalencia funcional, el tiempo necesario para los experimentos exhaustivos con millones de vectores de prueba crecía significativamente en ordenes de horas con la dimensión, por lo que se omitieron en el análisis. Por la misma razón, las pruebas realizadas en las tarjetas de la familia STM32 solo consideraron diez mil sets de entradas.

Por último, la implementación de freeRTOS solo fue posible la dimensión cuatro del problema, ya que de lo contrario existían problemas de memoria que no pudieron ser resueltos en el tiempo en que se realizó esta memoria.

4.2.1. Errores numéricos

Para la evaluación de los errores numéricos obtenidos por el código generado, se comparan los resultados obtenidos por el algoritmo a partir de los cien mil sets de entradas diferentes con su valor especificado en la golden reference generada a partir del código en Matlab. Debido a que considerar repeticiones de resultados obtenidos a partir de las mismas entradas es redundante, no se toman como muestra todos los resultados provenientes de los diez millones de ejecuciones del código generado, sino que solo una repetición del set de entradas de cien mil. En este caso, solo se comparan los resultados obtenidos al ejecutar el código C generado en un computador de escritorio ejecutando Windows, ya que en pruebas preliminares se determinó que el error al ejecutar el código generado en distintas plataformas era despreciable.

La Tabla 4.2 resume las métricas obtenidas para la evaluación del error numérico,donde cada métrica representa lo siguiente:

- **Media:** Representa la media aritmética de la diferencia numérica de los valores obtenidos a partir de los códigos generados con respecto a su versión en Matlab. Su cálculo consiste en el promedio de los errores obtenidos a partir de la ecuación (3.1).
- **SD:** Representa la desviación estándar de la diferencia numérica de los valores obtenidos a partir de los códigos generados con respecto a su versión en Matlab. Su cálculo consiste en la desviación estándar de los errores obtenidos a partir de la ecuación (3.1).
- **Min:** Representa la mínima diferencia numérica de los valores obtenidos a partir de los códigos generados con respecto a su versión en Matlab.
- **Max:** Representa la máxima diferencia numérica de los valores obtenidos a partir de los códigos generados con respecto a su versión en Matlab.
- **Valid:** Representa el porcentaje de la cantidad resultados válidos. Estos corresponden al número de resultados cuya diferencia numérica entre todos sus valores difieren en menos

de un 1 % con respecto a los valores obtenidos por Matlab. Su cálculo es mediante la ecuación (3.1) descrita previamente.

- **Tamaño muestra:** Representa la cantidad de sets de entrada probados.

Se hace notar que el umbral de error para calcular el porcentaje de resultados válidos es arbitrario y dependerá del contexto de la aplicación. En el contexto de esta evaluación se definió arbitrariamente que un error de 1 % podía considerarse despreciable, pero si hay errores mayores a eso no necesariamente implica que el resultado sea inválido, como se explica a continuación.

Basándose en los resultados presentados en la Tabla 4.2 es posible afirmar que las diferencias numéricas máximas alcanzan cifras de casi un 500 % para todas las dimensiones del problema. Entrando en detalle, estas diferencias tienen su origen en resultados cuyo valor absoluto es cercano a cero, lo cual hace que ante una leve diferencia absoluta en los valores comparados, el valor del error relativo según fórmula utilizada en la ecuación (3.1) se vea amplificado. En la práctica, el error o diferencia en magnitud absoluta ronda valores de $4{,}954 \cdot 10^{-4}$. Cabe mencionar que el impacto e importancia del error de los valores obtenidos está directamente relacionado con la magnitud de referencia del valor con el que se esté comparando. No es lo mismo tener un error de 500 % en donde el valor de referencia sea del orden de 10^4, que el valor de referencia sea 10^{-4}. En ambos casos el error es el mismo, pero el posible impacto que tendría este error solo se vería reflejado en valores de referencia del orden de 10^{-4} si usamos nuestro caso como ejemplo.

Entonces, se considera que errores como los mencionados pueden tratarse de outliers, ya que, si consideramos la totalidad de valores que alcanzan errores por sobre el 200 %, obtenemos que para ninguna dimensión del problema estos representan más de un 0.05 % de la totalidad de valores obtenidos.

Por otro lado, tenemos que a medida que se aumenta la dimensión del problema aumentan también los errores, en principio esto es algo esperable debido a que al aumentar la dimensión la cantidad de resultados también aumenta, existiendo más posibilidad de encontrarse con errores. Sin embargo, si tenemos en consideración todos los valores obtenidos a partir de ADMM sin hacer una distinción al agruparlos por set de salidas, obtenemos que la proporción de valores que no cumplen con el umbral del 1 % establecido son muy similares, siendo 0.6633 %, 0.6647 % y 0.6642 % para las dimensiones 4,8 y 16 respectivamente.

Para el caso de la dimensión más alta probada se obtienen diferencias numéricas en promedio de al menos tres veces superior a los casos con menor dimensión. Además, la desviación estándar aumenta aproximadamente dos veces con respecto a los otros casos y Valid disminuye en casi un 5 %. Estos valores indican que es probable que al aumentar la dimensión del problema, la dispersión de la magnitud de los errores también lo haga y en consecuencia puede verse reflejado en mayor cantidad de errores (más errores superan el umbral establecido en diferentes sets de salidas) disminuyendo el valor de Valid y aumentando el valor de la desviación estándar.

Finalmente, es posible concluir que a partir los datos de la Tabla 4.2 es razonable decir que los resultados obtenidos por el código generado por Embedded Coder son válidos, confirmando los resultados preliminares obtenidos, siendo un código funcionalmente equivalente al de Matlab.

Tabla 4.2: Estadísticas de error relativo un computador de escritorio con Windows.

Dimensión	% Media	% SD	% Max	% Min	Cantidad	% Valid	Tamaño muestra
4x4	0.2872	6.7477	499.870	0	1198	98.802	100000
8x8	0.456	8.9554	499.12	0	1530	98.478	100000
16x16	1.4971	16.077	499.96	0	6392	93.608	100000

4.2.2. Caracterización tiempos de ejecución

Para la obtención de los tiempos de ejecución en los diferentes entornos fue necesario hacer uso de las diferentes herramientas de código que estaban a disposición. Tal como se mencionó en la Sección 3.2, la biblioteca time.h fue de especial importancia para la adquisición del tiempo de ejecución del algoritmo en la mayoría de plataformas, exceptuando los microcontroladores de la familia STM32, los que requirieron la utilización de herramientas propias del microcontrolador fue posible la obtención de una información equivalente. En la Tabla 4.3 pueden verse todos los resultados de tiempo de ejecución para cada una de las plataformas con sus respectivos sistemas operativos, en donde las plataformas de la familia STM32 consideran la realización de diez mil pruebas. Lo primero a resaltar es la clara correlación existente entre el sistema operativo utilizado y el tiempo de ejecución obtenido. Para el caso del computador de escritorio puede verse que pese a correr el mismo código en un mismo procesador, los resultados obtenidos para las medidas estadísticas consideradas difieren cuando se está utilizando distinto sistema operativo, por ejemplo cuando se utiliza Linux los tiempos de ejecución máximos disminuyen aproximadamente nueve veces en relación con Windows. Para el caso de las tarjetas de la familia STM32 puede verse un fenómeno similar, al ejecutar el código generado en un mismo procesador, el uso de freeRTOS o Bare Metal permite observar que los resultados de las medidas estadísticas difieren, la implementación de freeRTOS se ve reflejada como una disminución en el tiempo de ejecución en promedio hasta en un 36 %. Además, la diferencia entre el valor de tiempo máximo y mínimo es hasta un 27 % menor para el caso de utilizar freeRTOS por sobre Bare Metal, entonces es posible decir que los resultados de tiempo de ejecución obtenidos por este sistema operativo tienen un mayor determinismo que los obtenidos en Bare Metal.

Con respecto a los tiempos máximos en la ejecución del algoritmo, podemos ver que para el caso del computador de escritorio los resultados obtenidos son al menos nueve veces menor en todas las dimensiones de este problema cuando se utiliza el sistema operativo Linux por sobre Windows; sin embargo, los valores mínimos tienden a ser similares para las dimensiones 8 y 16 del problema, teniendo menos de un 5 % de diferencia. Por otro lado, para el caso de los microcontroladores Raspberry 3B, Raspberry 4B y BeagleBone AI los resultados de tiempo de ejecución tienden a respetar la correlación que existe entre la potencia de un procesador al ser comparados entre sí. De antemano se sabe que la Rapberry 4B posee una capacidad de cómputo superior a la Raspberry 3B y BeagleBone AI disponibles, debido a la frecuencia, núcleos y arquitectura de su procesador, esto puede verse reflejado en los tiempos de ejecución máximos y promedio siendo menores para todas las dimensiones del problema alcanzando razones de hasta cinco veces menor. Por otra parte, Raspberry 3B y BeagleBone AI, tienen potencia de cómputo en sus procesadores más semejante que al compararlos con una Raspberry 4B. Por lo tanto, puede verse que al comparar sus resultados de tiempo de ejecución promedio existe una diferencia menor que al compararlos con la Raspberry 4B. Por ejemplo, en la dimensión dieciséis del problema, la razón de los tiempos de ejecución promedio entre la Raspberry 3B y

BeagleBone AI es de aproximadamente 1.5, sin embargo, al compararlos con la Raspberry 4B esta razón asciende hasta cinco.

Por su parte, las tarjetas de la familia STM32 presentan menores máximos tiempos de ejecución cuando se utiliza freeRTOS en vez de Bare Metal, teniendo al menos un 25 % menos de tiempo. De forma análoga, es posible decir que debido a la potencia de cómputo por parte de estas tarjetas con respecto a los demás microcontroladores utilizados, los resultados de tiempo de ejecución promedio son al menos dos veces mayor. Sin embargo, es posible notar que los tiempos máximos de ejecución pueden llegar a ser comparables en todas las dimensiones del problema con los microcontroladores, incluso siendo hasta cuatro veces menor para el caso de la máxima dimensión del problema. Cabe resaltar que la comparación realizada no es del todo "justa", ya que la cantidad de pruebas realizadas en esta familia de tarjetas es mil veces menor que en los demás microcontroladores.

Otro aspecto interesante a resaltar de las pruebas realizadas a la familia de tarjetas STM32 es la desviación estándar obtenida a partir las pruebas. La desviación estándar es hasta 10 % menor cuando se está utilizando la tarjeta en Bare Metal. En general, el uso de recursos cuando se utiliza una tarjeta en Bare Metal es a selección del usuario, es por esto que el overhead es prácticamente nulo al compararlo con el que trae consigo el uso de un sistema operativo como Linux o freeRTOS. Para el caso de las pruebas realizadas en las tarjetas STM32, solo se estaba ejecutando el algoritmo ADMM tanto para freeRTOS como para Bare Metal, entonces, es lógico pensar que como solo se estaba ejecutando un algoritmo, la variación obtenida entre los tiempos de ejecución cuando se estaba usando Bare Metal sea más homogéneo a cualquier sistema operativo, ya que, solo se ejecuta la tarea en particular sin ningún proceso o tarea adicional en segundo plano, a diferencia de freeRTOS que pese a estar ejecutando un único algoritmo, aun así existe el overhead del propio sistema operativo. Es importante mencionar que no por esto Bare Metal es mejor que freeRTOS en todos los casos, siempre dependerá del contexto que tipo de configuración sea la más conveniente. Para el caso de freeRTOS su utilización tiene un enfoque más en la multitarea y deadlines estrictas, es decir, un proyecto que necesite que el microcontrolador ejecute múltiples algoritmos, manejo de periféricos, comunicación y que además la aplicación sea de tiempo real, implica que sea altamente probable que freeRTOS sea una mejor opción por sobre Bare Metal u otro sistema operativo.

Es importante mostrar como es el comportamiento del tiempo de ejecución a lo largo de las pruebas realizadas. Como ejemplo, se observa en las Figuras 4.1y 4.2 los resultados obtenidos en una Rapsberry 4B y una BeagleBone AI, en donde cada color representa una de las cien pruebas realizadas. Es posible notar que existen resultados que destacan por demorarse más que los demás, ya que al estar corriendo un sistema operativo, el overhead y manejo en la ejecución de procesos son factores que ocasionan este tipo de resultados. El propósito de las figuras es visualizar y comprobar si los datos extremos de tiempo están sujetos a alguna muestra o conjunto de estas en particular, si existe alguna tendencia o simplemente son aleatorios. Aparentemente, no existe alguna tendencia dado que los datos de mayor magnitud solo se observan muestras asociadas a una ejecución en particular, esto indica que no se repiten en los demás casos por lo que no corresponde a un tema algorítmico, sino, como ya se ha mencionado, a un aspecto computacional debido a la organización de ejecución de procesos, utilización y características de hardware, disponibilidad de datos en memoria y acceso a esta.

En contraste, tomando como ejemplo los resultados obtenidos en una STM32f7xx mostrados en las Figuras 4.3y 4.4, los gráficos muestran los tiempos de ejecución obtenidos a partir de diez mil sets de entradas. Es posible notar que no hay resultados que se alejen en las proporciones

que lo hacían en la Rapsberry 4B o la BeagleBone AI del conjunto en general. Las razones principales son dos; la primera, la posible falta de pruebas en esta plataforma para poder ver evidenciada esta característica; y la segunda, por el tipo de sistema operativo utilizado. Tal como se ha mencionado con anterioridad, el uso de RTOS facilita el cumplimiento de deadlines estrictas, ya que el manejo a nivel de sistema operativo está concentrado en asegurar el determinismo de la latencia en las tareas o procesos que está ejecutando, debido aquello, el gráfico es consistente pese a no parecerse a aquellos que pertenecen a plataformas que lleven sistemas operativos como Linux. Por otro lado, Bare Metal al no tener un overhead adicional, no es raro suponer que la ejecución de algoritmos siempre tome una cantidad similar de tiempo, ya que no hay tareas adicionales que esté realizando en segundo plano. Cabe destacar que los gráficos de los resultados de las demás plataformas se encuentran disponibles en el Anexo de este documento.

Tabla 4.3: Resultados de la caracterización temporal de las plataformas.

Dimensión	Plataforma	Media [μs]	SD [μs]	Max [μs]	Min [μs]
4x4	PC Windows	6.2593	5.8793	1140.9	4.1
	PC Linux	1.1364	0.4644	120.443	7.58
	Raspberry 3B	28.7235	6.3463	5973.289	15.051
	Raspberry 4B	22.4981	6.7648	5360.209	21.166
	BeagleBone AI	120.2993	21.1647	16240.64	116.632
	STM32f7xx	170.8498	0.6513	173.4444	168.9398
	STM32f7xx(RTOS)	124.9757	0.6924	127.3241	124.0556
	STM32f4xx	1007.4494	13.915	1065.99	982.41
	STM32f4xx(RTOS)	737.3292	15.6635	750.8942	731.6727
8x8	PC Windows	41.9760	30.2451	5729.9	31.90
	PC Linux	41.0751	2.96904	261.972	30.929
	Raspberry 3B	934.0544	310.5293	14909.25	606.0550
	Raspberry 4B	182.7730	31.6746	10366.92	174.831
	BeagleBone AI	911.3078	61.8564	12983.89	885.882
	STM32f7xx	1189.8534	2.5159	1198.3055	1182.0648
	STM32f7xx(RTOS)	-	-	-	-
	STM32f4xx	5791.2115	60.7892	5998.98	5634.41
	STM32f4xx(RTOS)	-	-	-	-
16x16	PC Windows	193.443	142.9419	30310.4	137.4
	PC Linux	182.185	7.2387	2263.278	138
	Raspberry 3B	2745.237	639.838	725317.9	250.395
	Raspberry 4B	817.0843	120.0437	25116.89	790.429
	BeagleBone AI	4139.684	109.222	41124.74	4089.928
	STM32f7xx	9623.4828	4.3562	9636.0787	9610.5602
	STM32f7xx(RTOS)	-	-	-	-
	STM32f4xx	41582.5721	252.429	42457.4	40656.25
	STM32f4xx(RTOS)	-	-	-	-

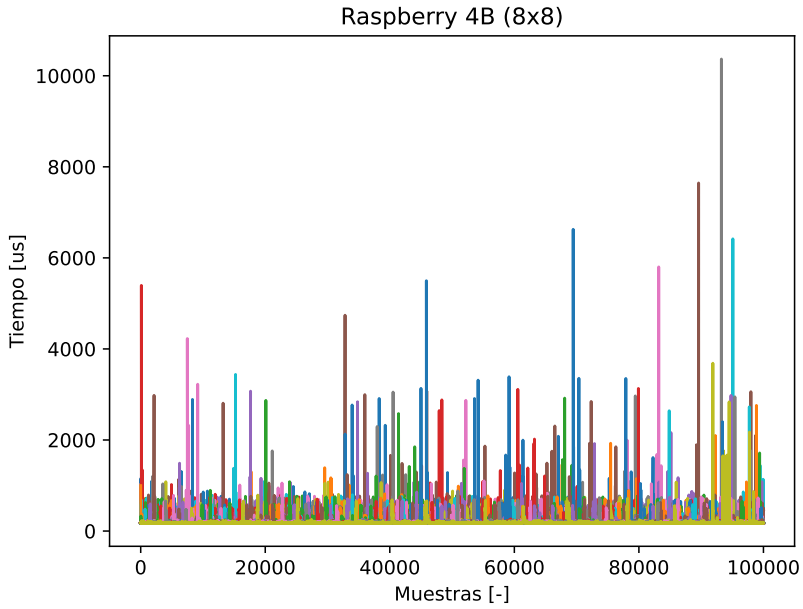


Figura 4.1: Tiempos de ejecución a partir de la ejecución del algoritmo ADMM(8x8) en una Raspberry 4B.

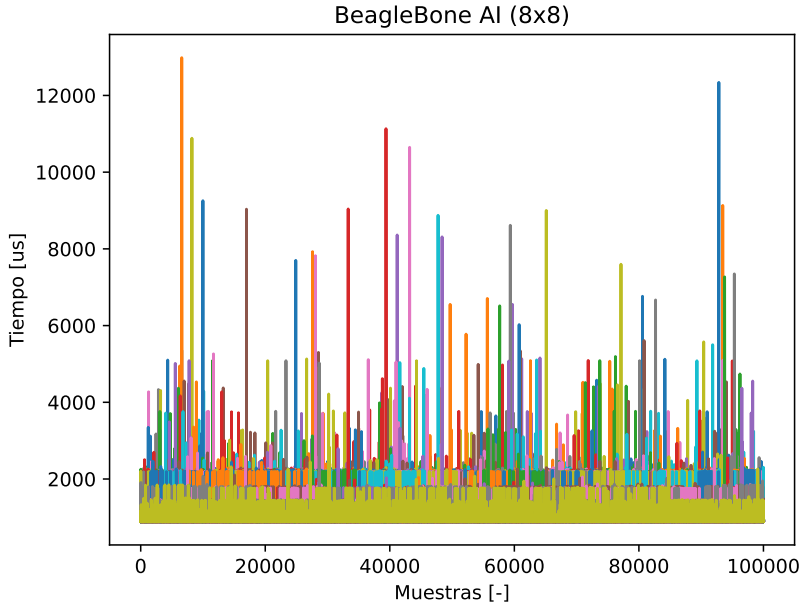


Figura 4.2: Tiempos de ejecución a partir de la ejecución del algoritmo ADMM(8x8)ADMM(8x8) en una BeagleBone AI.

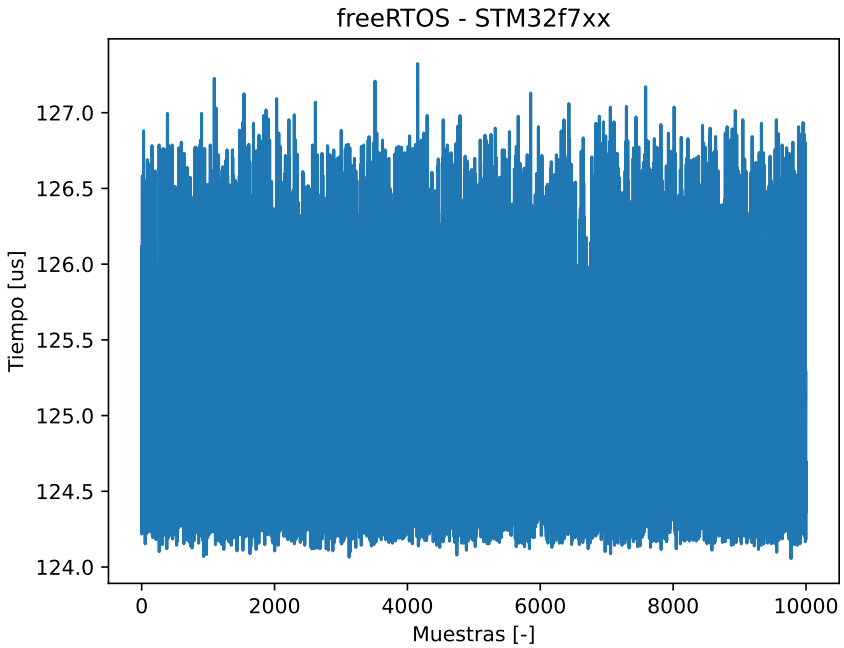


Figura 4.3: Tiempos de ejecución a partir de la ejecución del algoritmo ADMM(4x4) en una tarjeta STM32f7xx con FreeRTOS

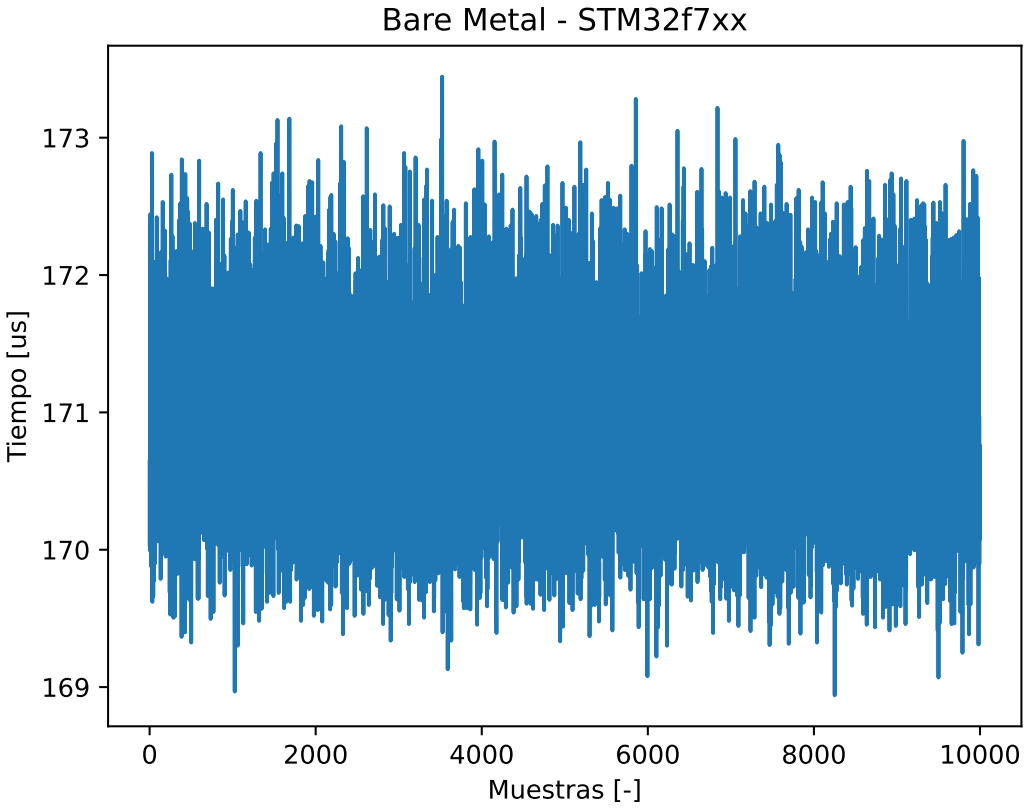


Figura 4.4: Tiempos de ejecución a partir de la ejecución del algoritmo ADMM(4x4) en una tarjeta STM32f7xx con Bare Metal.

5 | Conclusiones y trabajo futuro

Una vez concluidas las pruebas, es posible afirmar que el impacto que tiene un sistema operativo en el tiempo de ejecución de un algoritmo es evidente, viéndose evidenciado en las pruebas realizadas en las tarjetas de la familia STM32. FreeRTOS demuestra ser un sistema operativo ideal para aplicaciones en tiempo real, debido al determinismo de los tiempos de ejecución mostrados, dejando en vista el potencial para cumplir deadlines muy estrictas. La configuración Bare Metal por su parte, plantea una vertiente interesante para su uso en diversas aplicaciones, ya que pese a obtener tiempos de ejecución mayores que los de freeRTOS, se presenta como una alternativa viable al momento de utilizar microcontroladores, cuando las restricciones temporales no son tan estrictas. Por otro lado, el uso de sistemas operativos de propósito general como lo fue Linux y Windows, plantean una clara problemática que no tiene las alternativas previamente mencionadas (freeRTOS y Bare Metal), la cual es el poco determinismo de los resultados. Dada la naturaleza presentada por este tipo de sistemas operativos (maximización de throughput) pueden presentarse peores casos que se alejen mucho de la media de resultados, dificultando su uso para aplicaciones que requieran ser ejecutadas en tiempo real. Por otro lado, fue posible observar que los códigos generados pudieron correr en distintas plataformas embebidas con diferentes arquitecturas y recursos disponibles, demostrando así la generalidad de los códigos generados.

A raíz de lo anterior, se concluye que el uso de Embedded Coder para la generación de código automático en C es una alternativa bastante sólida a la hora de necesitar una versión de código de bajo nivel de algún modelo o algoritmo escrito en Matlab. Si bien hay que tener en cuenta una serie de factores a la hora de escribir o preparar un código en Matlab para poder utilizar Embedded Coder de una forma correcta, tener una alternativa viable que facilite el traspaso de modelos implementados en Matlab a un código directamente portable a distintas plataformas embebidas, facilita significativamente el prototipado en Matlab. Es importante resaltar que los resultados obtenidos a lo largo de este trabajo son altamente mejorables, ya que no consideran aspectos más finos de optimización más allá del un uso básico de Embedded Coder.

Este trabajo de título es la primera aproximación al uso de Embedded Coder, plantando bases y directrices iniciales para el flujo de trabajo que trae consigo el uso de esta herramienta. Dadas las limitaciones temporales presentes en la realización de este trabajo, existen temas tratados de una forma superficial o tópicos no abordados. Debido a lo anterior, la posibilidad de la continuidad de este trabajo, tomando como piso inicial, el trabajo realizado hasta la fecha (este mismo documento y el repositorio [5]) se vuelve una alternativa viable para profundizar en configuraciones que tengan impacto el rendimiento del código generado. Entonces, respecto al trabajo futuro, existen 3 claras vertientes para focalizar. La primera está estrictamente relacionada con la utilización de distintos sistemas operativos para la implementación de los códigos generados por Embedded Coder. Encargarse de ver a mayor profundidad las

herramientas que nos entrega freeRTOS como sistema operativo idóneo para aplicaciones en tiempo real, realizando pruebas que aprovechen características propias del sistema operativo y realizando un contraste más adecuado Linux, para de esta forma tener más evidencia del potencial de freeRTOS. Por otro lado, la segunda vertiente tiene que ver con el mismo uso de la herramienta Embedded Coder. Si bien aquí se propone un flujo de trabajo para esta, incluyendo configuraciones y forma de implementar el código generado, es claro que la herramienta trae consigo una alta personalización que no fue aprovechada en su totalidad. Adicionalmente, el flujo de trabajo no es más que una propuesta, dejando las puertas abiertas para ser mejorado y optimizado para futuros trabajos. Y por último, la vertiente relacionada con una implementación en un sistema físico. Pese a que los códigos generados fueron probados en distintos microcontroladores, nunca se probó estos mismos códigos en sistemas físicos como tal. Esta implementación sería una validación funcional robusta, ya que habría interacción con periféricos, y restricciones que normalmente no pueden ser simuladas.

Finalmente, es posible aseverar basándose en la evidencia empírica de las pruebas realizadas, que Embedded Coder es una herramienta funcional y efectiva para la obtención de un código en C. Facilitando a desarrolladores el prototipado de algoritmos desarrollados en Matlab.

Bibliografía

[1] "Embedded coder, generación de código c y c++ optimizado para sistemas embebidos." `https://la.mathworks.com/products/embedded-coder.html`. Accedido: 2022-07-30.

[2] "What is matlab?." `https://la.mathworks.com/discovery/what-is-matlab.html`. Accedido: 2022-07-30.

[3] J. Wang, Y. Zhao, T. Cui, and K. Varahramyan, "Synthesis of the modeling and control systems of a tunneling accelerometer using the matlab simulation," *Journal of Micromechanics and Microengineering*, vol. 12, no. 6, p. 730, 2002.

[4] G. J. Balas, A. Packard, J. C. Doyle, K. Glover, and R. Smith, "Development of advanced control design software for researchers and engineers," 1991.

[5] "Embedded coder." `https://gitlab.com/Jared677/embedded-coder`.

[6] G. Fortney, "Model based systems engineering using validated executable specifications as an enabler for cost and risk reduction," in *Proceedings of the 2014 Ground Vehicle Systems Engineering and Technology Symposium (GVSETS)*, 2014.

[7] A. Bergmann, "Benefits and drawbacks of model-based design," *KMUTNB International Journal of Applied Science and Technology*, vol. 7, pp. 15–19, 09 2014.

[8] C. Ferdinand, R. Heckmann, T. Le Sergent, D. Lopes, B. Martin, X. Fornari, and F. Martin, "Combining a high-level design tool for safety-critical systems with a tool for wcet analysis of executables," in *Proc. of the 4th European Congress on Embedded Real Time Software (ERTS), Toulouse*, p. 10, 2008.

[9] "Ansys scade suite." `https://www.mistralsolutions.com/defence_products/scade-suite/`. Accedido: 2022-08-02.

[10] A. Riid, J. Preden, R. Pahtma, R. Serg, and T. Lints, "Automatic code generation for embedded systems from high-level models," *Elektronika ir Elektrotechnika*, vol. 95, no. 7, pp. 33–36, 2009.

[11]

[12] "Targetlink." `https://www.dspace.com/en/pub/home/products/sw/pcgs/targetlink.cfm`. Accedido: 2022-09-25.

[13] G. Dhanabalan and S. Selvi, "Software design of veriloghdl code generation for ladder diagram and data acquisition using labview," *Wireless Personal Communications*, pp. 1–29, 2022.

[14] "Matlab coder, generación de código c y c++ a partir de código de matlab." https://la.mathworks.com/products/matlab-coder.html. Accedido: 2022-07-30.

[15] "Call mex functions." https://la.mathworks.com/help/matlab/call-mex-functions.html?lang=en. Accedido: 2022-08-02.

[16] N. Van Riel, "Speeding up simulations of ode models in matlab using cvode and mex files," *Eindhoven University of Technology*, 2012.

[17] "Functions and objects supported for c/c++ code generation." https://la.mathworks.com/help/simulink/ug/functions-and-objects-supported-for-cc-code-generation.html. Accedido: 2022-08-02.

[18] "Simulink coder genere código c y c++ desde modelos de simulink y stateflow." https://la.mathworks.com/products/simulink-coder.html. Accedido: 2022-08-02.

[19] A. Ramli, M. L. Abdullah, and M. Mamat, "Broyden's method for solving fuzzy nonlinear equations," *Advances in Fuzzy Systems*, vol. 2010, 2010.

[20] S. Kadry, "3 - linear algebra," in *Mathematical Formulas for Industrial and Mechanical Engineering* (S. Kadry, ed.), pp. 31–51, Oxford: Elsevier, 2014.

6 Anexo

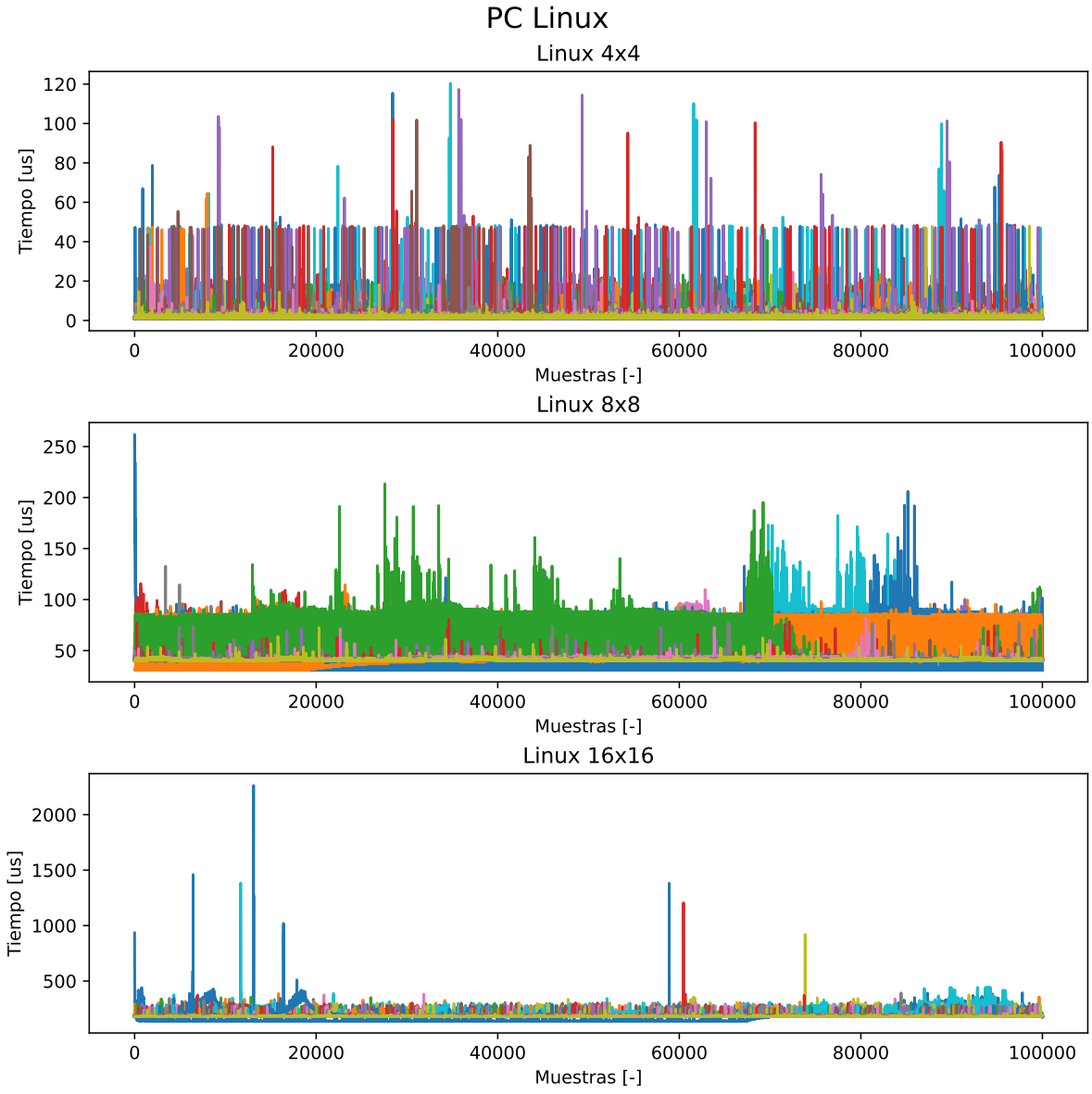


Figura 6.1: Tiempos de ejecución a partir de la ejecución del algoritmo ADMM(4x4) en un Computador de escritorio con Linux.

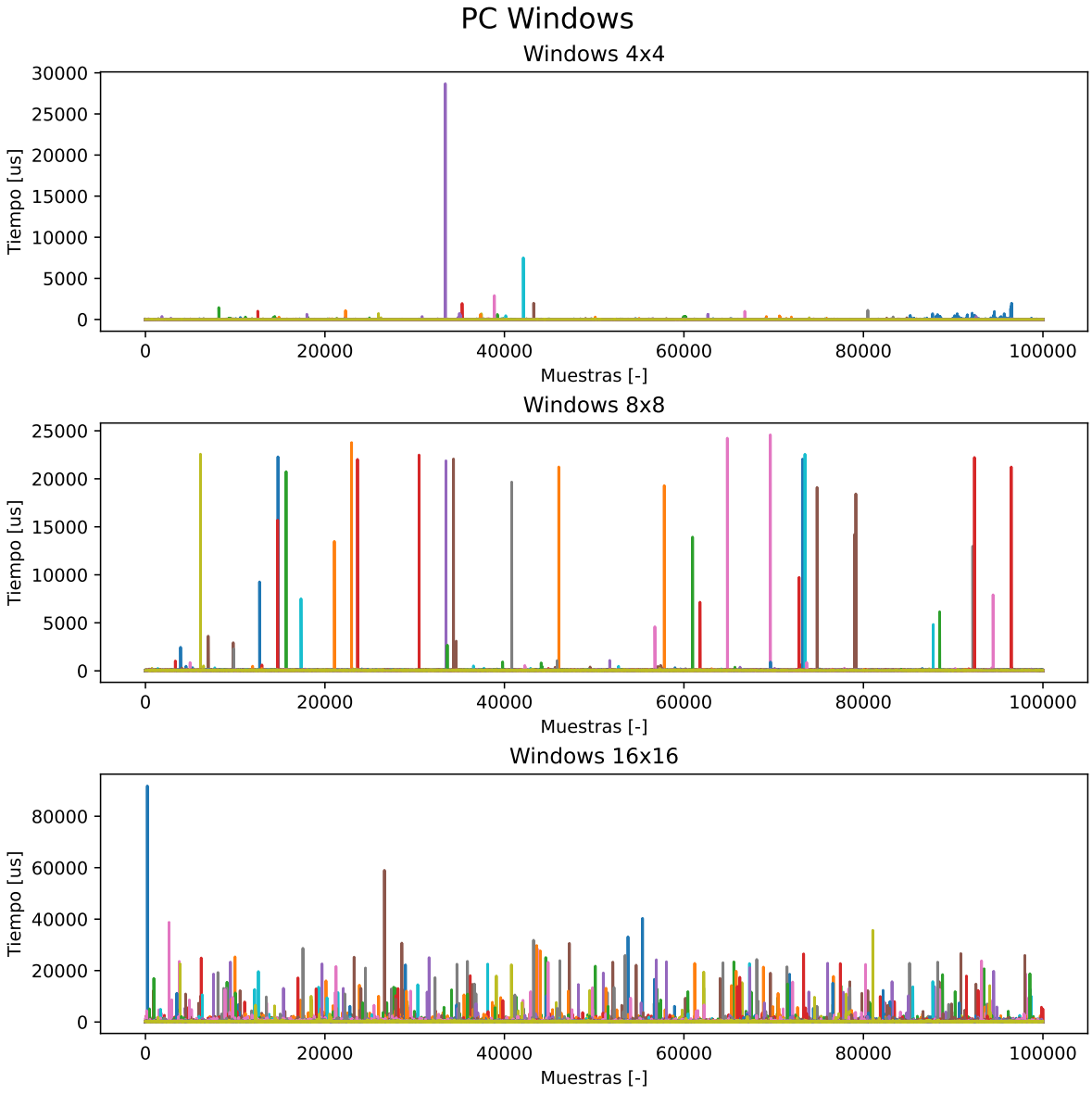


Figura 6.2: Tiempos de ejecución a partir de la ejecución del algoritmo ADMM(4x4) en un Computador de escritorio con Windows.

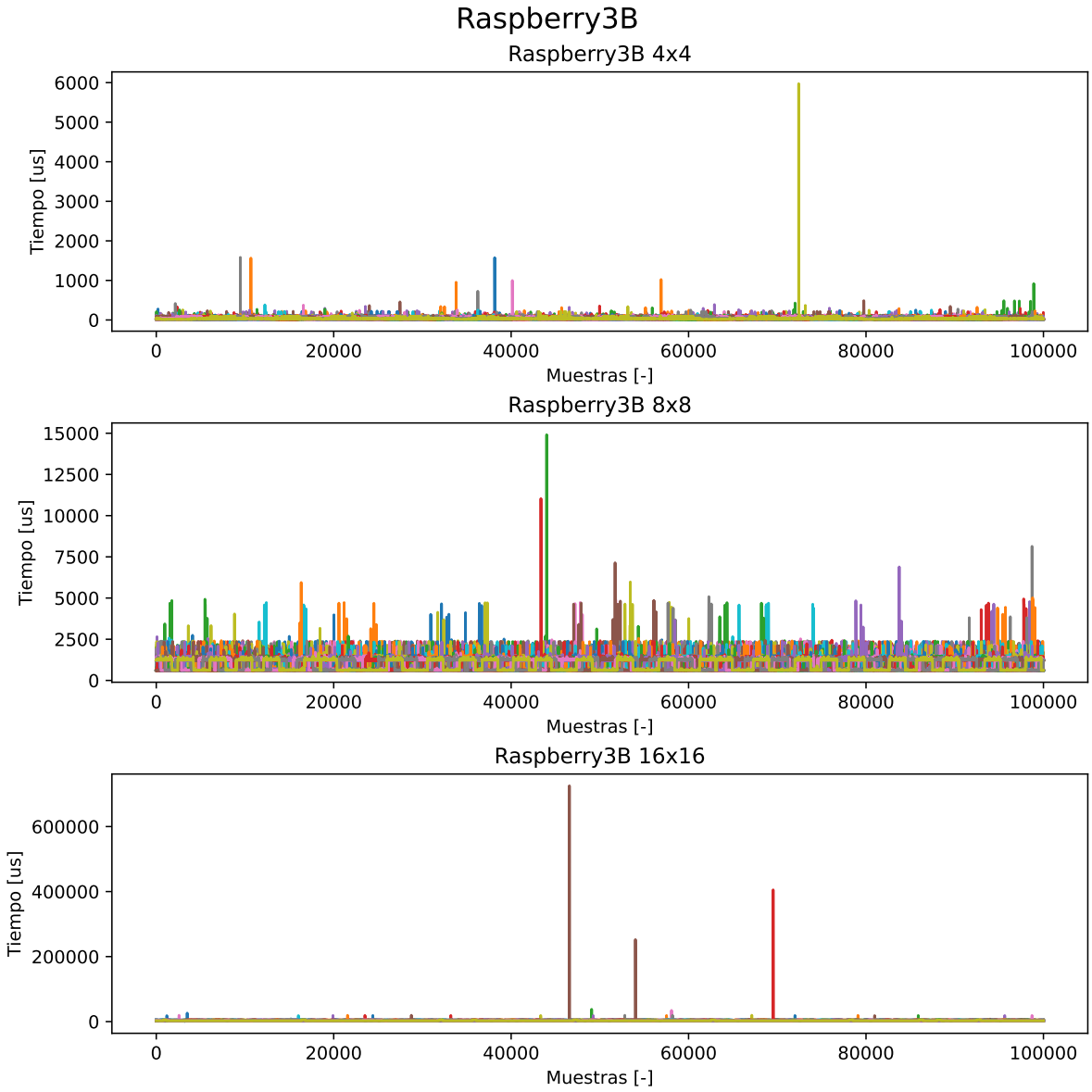


Figura 6.3: Tiempos de ejecución a partir de la ejecución del algoritmo ADMM(4x4) en una Raspberry Pi 3B.

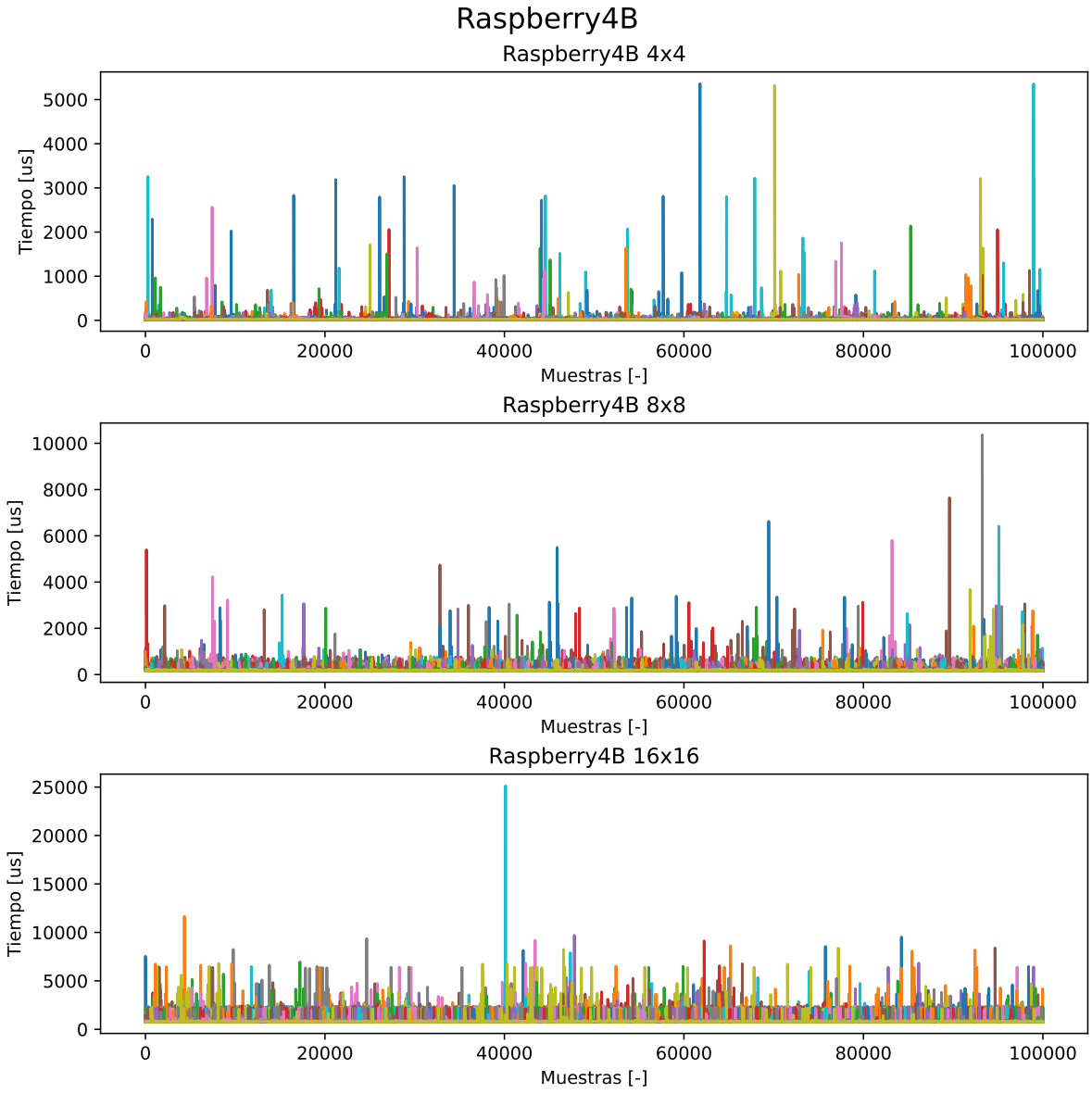


Figura 6.4: Tiempos de ejecución a partir de la ejecución del algoritmo ADMM(4x4) en una Raspberry Pi 4B.

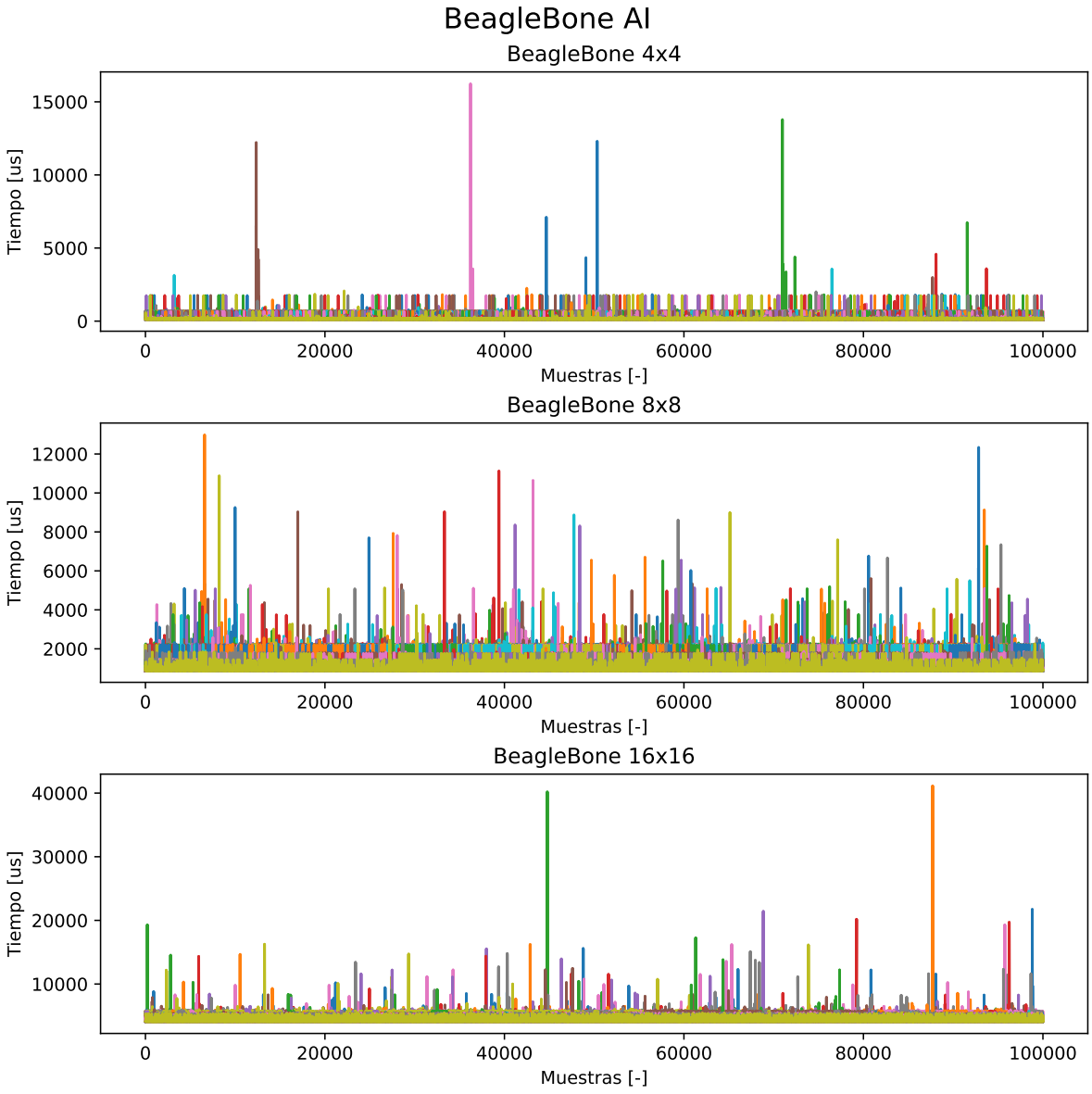


Figura 6.5: Tiempos de ejecución a partir de la ejecución del algoritmo ADMM(4x4) en una BeagleBone AI.

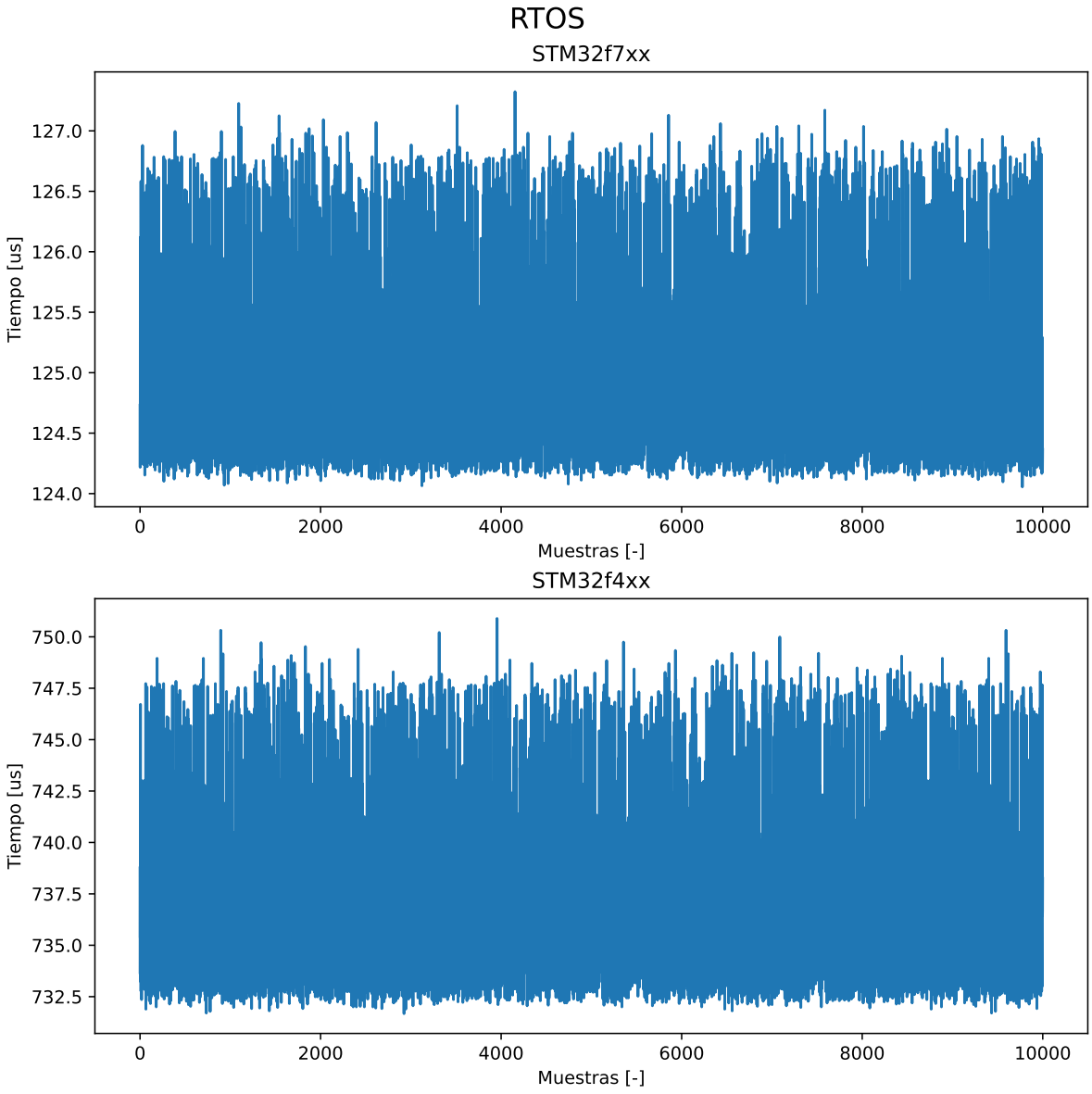


Figura 6.6: Tiempos de ejecución a partir de la ejecución del algoritmo ADMM(4x4) en una STM32f7xx y stm32f4xx con freeRTOS.

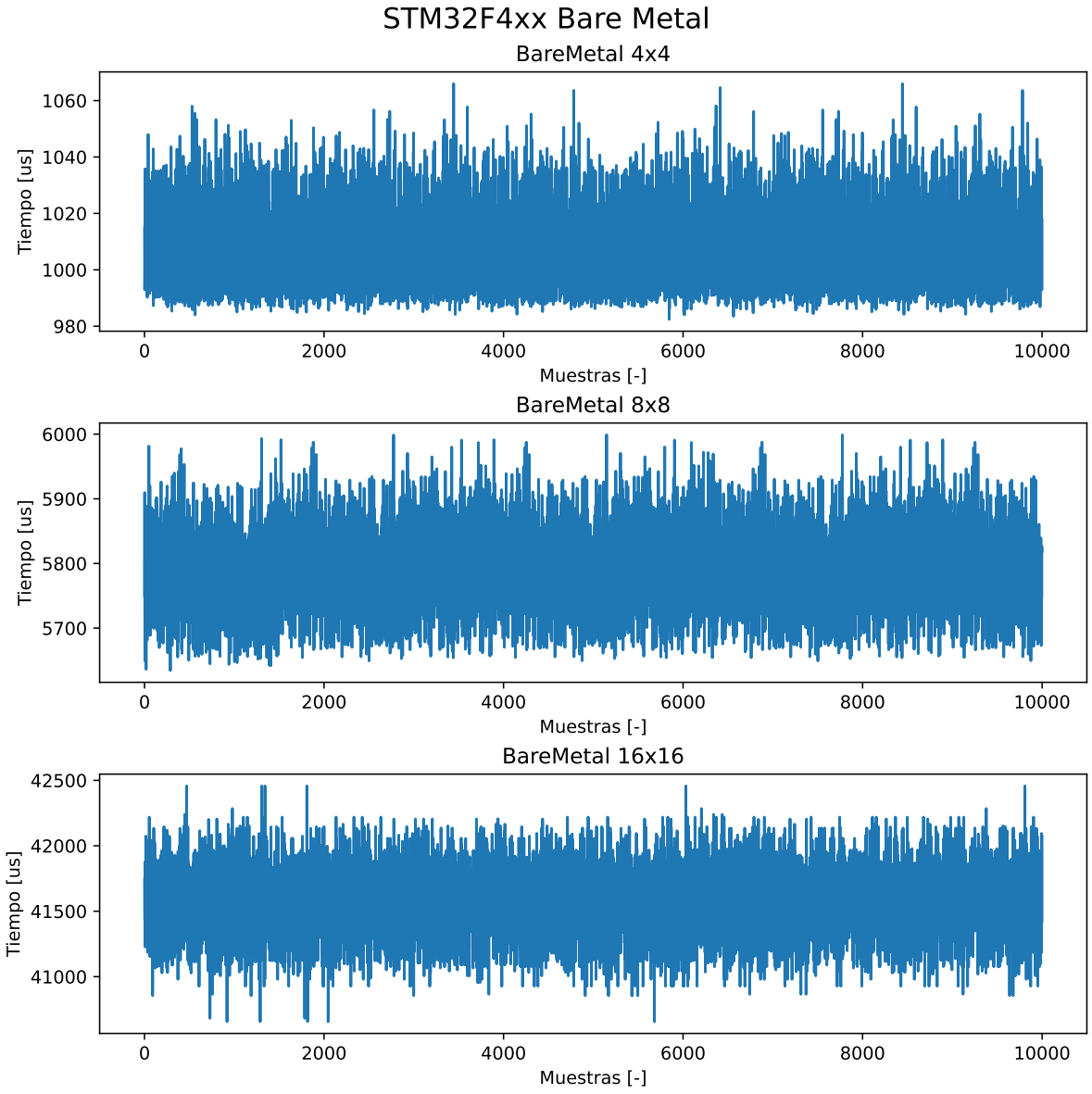


Figura 6.7: Tiempos de ejecución a partir de la ejecución del algoritmo ADMM en una STM32f4xx Bare Metal.

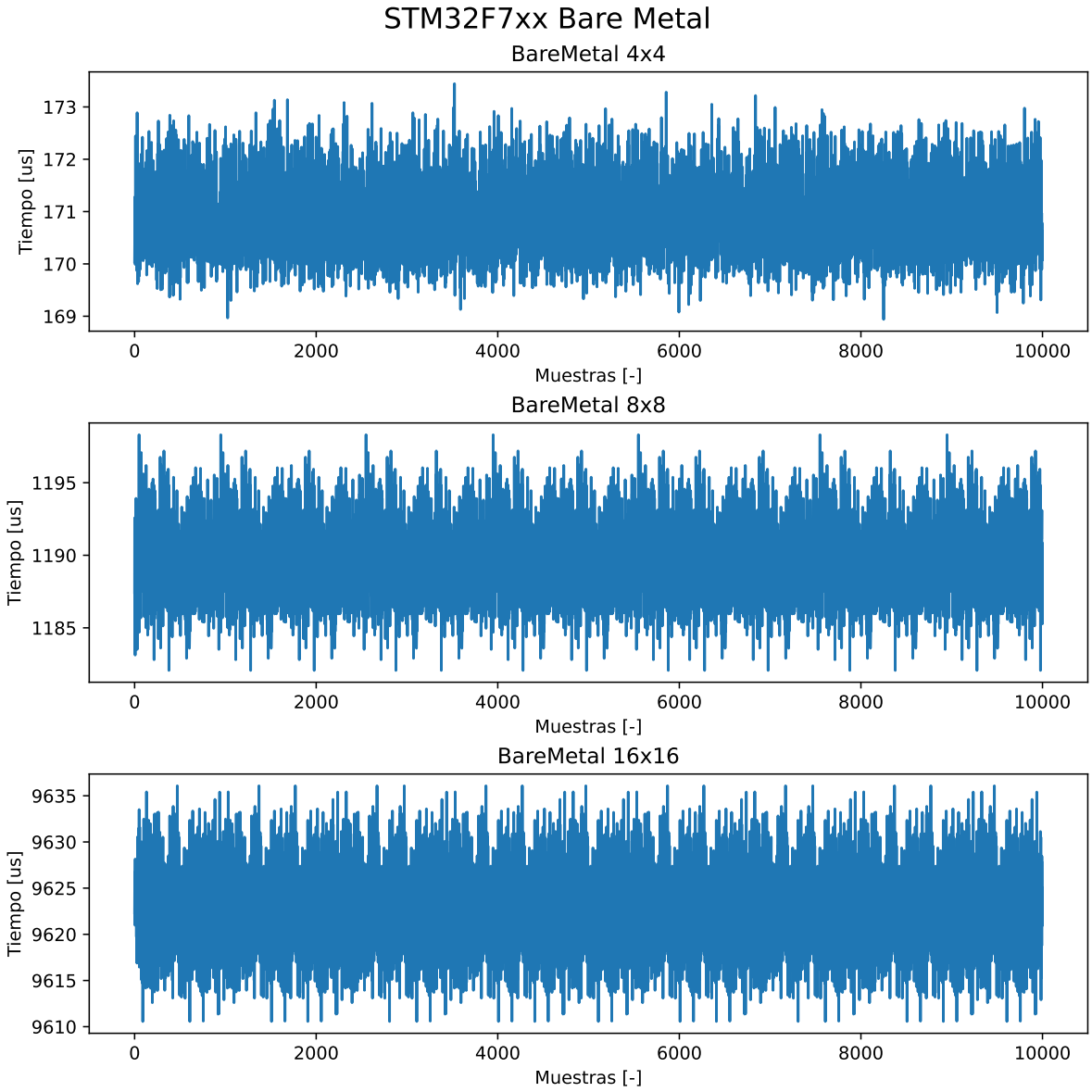


Figura 6.8: Tiempos de ejecución a partir de la ejecución del algoritmo ADMM(4x4) en una STM32f7xx Bare Metal.